Il sistema operativo UNIX/Linux

Gli script di shell

Stefano Quer
Dipartimento di Automatica e Informatica
Politecnico di Torino

Introduzione agli script

- I linguaggi di shell sono linguaggi **interpretati**
  - Non esiste una fase di compilazione esplicita
- Vantaggi & svantaggi
  - Le shell sono disponibili in ogni ambiente UNIX/Linux
  - Ciclo di produzione più veloce
  - Minore efficienza in fase di esecuzione
  - Minori possibilità e ausili di debug
- Utilizzati per scrittura di software
  - "Quick and dirty"
  - Prototipale

Introduzione agli script

- BASH vs Python (e altri)
  - Scelta
    - Il principale punto di forza di BASH nei confronti di altri linguaggi (python, ruby, lua, etc.) è la sua obiquità
    - Se il numero di righe di codice è inferiore a 100, conviene scegliere BASH, altrimenti PYTHON

Introduzione agli script

- BASH vs Python (e altri)
  - Prestazioni
    - Per avere alte prestazioni normalmente si scrive un programma non uno script
    - L'interprete BASH è molto veloce nel partire (starting time)
    - Se occorre manipolare file ASCII, oppure utilizzare pesantemente comandi di shell o filtri tipo sort, uniq, etc., BASH è più adatto e veloce ("will smoke Python performance wise")
    - Se occorre manipolare numeri floating point Python è conveniente ("will win hands down")

Introduzione agli script

- Gli script
  - Sono normalmente memorizzati in file di estensione .sh (.bash)
    - Si ricordi però che in UNIX/Linux le estensioni non vengono utilizzate per determinare il tipo di file
  - Possono essere eseguiti mediante due tecniche
    - Esecuzione diretta
    - Esecuzione indiretta

Esecuzione diretta

```
./scriptname args
```

- Lo script viene eseguito da riga di commando come un normale file eseguibile
  - È necessario il file abbia il permesso di esecuzione
    - chmod +x ./scriptname
  - La prima riga dello script deve specificare il nome dell'interprete dello script
    - #!/bin/bash, #!/bin/sh
  - È possibile eseguire lo script utilizzando una shell particolare
    - /bin/bash ./scriptname args

Esecuzione diretta

- Lo script viene eseguito da una sotto-shell
  - Ovvero, eseguire uno script in maniera diretta implica eseguire un nuovo processo
  - Ambienti (variabili) del processo originario e di quello eseguito non coincidono
  - Le modifiche alle variabili di ambiente effettuate dallo script sono perdute

```
./scriptname args
```

Esecuzione indiretta

```
source ./scriptname args
```

- Si invoca la shell con il nome del programma come parametro
  - È la shell corrente a eseguire lo script
    - "The current shell sources the script"
  - Non è necessario lo script sia eseguibile
  - Le modifiche effettuate dallo script alle variabili di ambiente rimangono valide nella shell corrente

Esempio: esecuzione diretta e indiretta

Esecuzione diretta:
> scriptName.sh<return>
La shell esegue lo script come sottoprocesso. Eseguendo l'exit il sottoprocesso termina. Il processo iniziale riprende il controllo.

```
#!/bin/bash
# NULL Script
exit 0
```

indica un commento

Esecuzione indiretta:
> source scriptName.sh<return>
La shell esegue lo script
Eseguendo l'exit il processo termina ovvero si uccide la shell di partenza

Debug di uno script

- Non esistono tool specifici per effettuare il debug di script bash
  - È ovviamente sempre possibile aggiungere delle "echo" esplicite
- È però possibile eseguire uno script in "debug" in maniera
  - Completa (l'intero script)
    - Si ottiene indicando una opzione di "debug" a livello di intero script
  - Parziale (ovvero solo alcune righe dello script)
    - Si ottiene indicando una ozione di "debug" a livello di alcune righe dello script mediante il comando **set**

Debug di uno script

- Opzioni

Opzioni			
Formato		**Significato**	**Effetto**
Compatto	**Esteso**		
-o noexec -n		Check senza esecuzione	Esegue un controllo sintattico ma non esegue lo script
-o verbose -v		Echo comandi	Visualizza i (fa l'eco dei) comandi eseguiti
-o xtrace –x		Traccia lo script	Visualizza la traccia di esecuzione dell'intero script
-o nounset -u		Check errori	Riporta un errore per variabile non definita

Debug di uno script

- Debug dell'intero script
  - Attivazione da riga di comando
    - /bin/bash –n ./scriptname args
  - Attivazione all'interno dello script
    - #!/bin/bash –v
    - #!/bin/bash –x
    - …
- Debug parziale
    - set –o verbose … set +o verbose
    - set -v … set +v
    - set –x … set +x

Sintassi: regole generali

- I linguaggio bash è relativamente di "alto livello" ovvero è in grado di mischiare
  - Comandi standard di shell
    - ls, wc, find, grep, ...
  - Costrutti "standard" del linguaggio di shell
    - Input e output, variabili e parametri, operatori (aritmetici, logici, etc.), costrutti di controllo (condizionali, iterativi), array, funzioni, etc.
- Istruzioni/comandi sulla stessa riga devono essere separati dal carattere '**;**'
  - Spesso si scrivono istruzioni su righe successive

Sintassi: regole generali

- Commenti
  - Il carattere # indica la presenza di un commento sulla riga
  - Il commento si espande dal carattere # in poi su tutta la riga
- La system call **exit** permette di terminare uno script restituendo un eventuale codice di errore
  - exit
  - exit [0/1]
    - 0 è il valore vero nelle shell

Esempio: uso di comandi di shell

```
#!/bin/bash

# This line is a comment

rm –rf ./..newDir/
mkdir ./../newDir/
cp * ./../newDir/
ls ./../newDir/ ;

# 0 is TRUE in shell programming

exit 0
```

Path assoluto

‘;’ superfluo

Dalla shell chiamante:
echo $?
fornisce 0

Parametri

- I parametri dello script (riga di comando) possono essere individuati mediante **$**
  - Parametri posizionali
    - $0 indica il nome dello script
    - $1, $2, $3, ... indicano i parametri passati allo script sulla riga di comando
  - Parametri speciali
    - $* indica l'intera lista (stringa) dei parametri (non include il nome dello script)
    - $# memorizza il numero di parametri (nome dello script escluso)
    - $$ memorizza il PID del processo

Il comando **shift** effettua uno shift dei parametri verso sinistra ($0 rimane immutato)

Esempio: passaggio parametri

```
#!/bin/bash

# Using command line parameters

echo "Il programma $0 e' in run"
echo "Parametri: $1 $2 $3 …"
echo "Numero dei parametri $#"
echo "Lista dei parametri $*"
shift
echo "Parametri: $1 $2 $3 …"
shift
echo "Parametri: $1 $2 $3 …"

exit 0
```

Le "…" effettuano l'espansione delle variabili

$0, $1, etc. possono anche comparire fuori dalle " "

$0 rimane immutato; quindi $1=$2, $2=$3, etc.

$0 rimane immutato; quindi $1=$2, $2=$3, etc.

Variabili

- Si suddividono in
  - **Locali** (o di shell)
    - Disponibili solo nella shell corrente
  - **Globali** (o d'ambiente)
    - Disponibili in tutte le sotto-shell
    - Ovvero sono esportate dalla shell corrente a tutti i processi da essa eseguiti

Variabili

- Caratteristiche principali delle variabili di shell
  - Non vanno dichiarate
    - Si creano assegnandone un valore
  - Sono case sensitive
    - var != VAR != Var != ...
  - Alcune sono riservate a scopi particolari
- L'elenco di tutte le variabili definite e il relativo valore viene visualizzato con il comando **set**
- Per cancellare il valore di una variabile si utilizza il comando **unset**
  - unset nome

Variabili locali (o di shell)

- Sono caratterizzate da nome e contenuto
  - Il contenuto ne specifica il tipo
    - Constante, stringa, intere, vettoriali o matriciali
  - Di default viene memorizzata una stringa anche quando il valore è numerico
  - Assegnazione
    - nome="valore"
  - Utilizzo
    - $nome

No spazi prima e dopo '='

Virgolette necessarie se il valore assegnato contiene spazi

Esempi

```
➢ var=Hello
➢ echo $var
Hello
➢ var=7+5
➢ echo $var
7+5
➢ i="Hello world!"
➢ echo $i
Hello World!
➢ i=$i" Bye!!!"
➢ echo $i
Hello World! Bye!!!
➢ echo i
i
```

Variabili stringa !

Senza virgolette si ha un errore durante l'assegnazione (a causa dello spazio) !

Concatenazione di stringhe

```
➢ let var=7+5
➢ echo $var
12
```

Variabili intera con espressione numerica (vedere in seguito)

Variabili globali (o di ambiente)

- Per rendere una variabile "globale" ovvero permettere la sua visibilità anche da altri processi si utilizza il comando **export**
  - export nome
- Si osservi che diverse variabili di ambiente
  - Sono riservate e pre-definite
  - Quando una shell viene eseguita tali variabili sono inizializzate automaticamente a partire da valori dell'"environment"
  - In genere tali variabili sono definite con lettere maiuscole per distinguerle da quelle utente
  - Si visualizzano con il comando **printenvb** (o **env**)

Esempio: variabile locale e globale

```
➢ v=prova
➢ echo $v
prova
➢ bash
➢ ps –l
... due bash in run
➢ echo $v

➢ exit
➢ echo $v
prova
```

Variabile locale alla shell corrente

```
➢ v=prova
➢ echo $v
prova
➢ export v
➢ bash
➢ ps –l
... due bash in run
➢ echo $v
prova
➢ exit
➢ echo $v
prova
```

Variabile globale comune alla shell corrente e alla sotto-shell

Esempio: uso di variabili

Clear video

```
#!/bin/bash
clear
echo "Ciao, $USER!"
echo
echo "Elenco utenti connessi"
who
echo "Assegna due variabili locali di shell"
COLORE="nero" ; VALORE="9"
echo "Stringa: $COLORE"
echo "Numero: $VALORE"
echo
echo "Ora restituisco il controllo"

#exit
```

who shows who is logged on

Comandi sulla stessa riga

no exit esplicito ...

Variabili predefinite

Elenco molto parziale

Variabile	Significato
$?	Memorizza il valore di ritorno dell'ultimo processo: 0 in caso di successo, valore diverso da 0 (compreso tra 1 e 255) in caso di errore. Nelle shell il valore 0 corrisponde al valore vero (al contrario del linguaggio C).
$SHELL	Indica la shell in uso corrente
$LOGNAME	Indica lo username utilizzato per il login
$HOME	Indica la home directory dell'utente corrente
$PATH	Memorizza l'elenco dei direttori separati da ':' utilizzato per la ricerca dei comandi (eseguibili)
$PS1 $PS2	Specificano il prompt principale e quello ausiliario (di solito '$' e '>', rispettivamente, # per root)
$IFS	Elenca i caratteri utilizzati per separe le stringhe lette da input (vedere comando read della shell)

Esempi

```
$ PS1= "> "
➢ echo $HOME
...
➢ v=$PS1
➢ echo $PS1
...
➢ PS1="myPrompt > "
myPrompt > echo $v
...
```

Modifica del prompt di shell

```
➢ myExe
myExe: command not found
➢ PATH=$PATH:.
➢ myExe
... myExe running ...
```

Modifica del PATH

```
➢ ls foo
ls: cannot access foo:
No such file or directory
➢ echo $?
2
➢ ls bar*
bar.txt
➢ echo $?
0
```

Valore di ritorno di un comando (0=true)

Lettura (input)

- La funzione **read** permette di eseguire dell'input interattivo, in altre parole permette di leggere una riga da stdin
- Sintassi
  - read [opzioni] var_1 var_2 ... var_n
    - Ogni read può essere seguita o meno da una lista di variabili
    - Ogni variabile specificata conterrà una delle stringhe introdotte in ingresso
    - Eventuali stringhe in eccesso saranno memorizzate tutte nell'ultima variabile
    - Nel caso non siano specificate variabili, tutto l'input viene memorizzato nella variabile REPLY

Lettura (input)

- Opzioni supportate
  - -n nchars
    - Ritorna dalla lettura dopo nchars caratteri senza attendere il new line
  - -t timeout
    - Timeout sulla fase di lettura
    - Restituisce 1 se non si introducono i dati entro timeout secondi
  - etc.

Esempi: lettura da stdin

```
➢ read v
Riga in ingresso
➢ echo $v
Riga in ingresso
```

Testo memorizzato in un'unica variabile v

```
➢ read v1 v2
Riga in ingresso
➢ echo $v1
Riga
➢ echo $v2
in ingresso

➢ Read
Contenuto di reply
➢ echo $REPLY
Contenuto di reply
➢ Read
Contenuto di reply
➢ v=$REPLY
➢ echo $v
Contenuto di reply
```

2 variabili ma 3 stringhe introdotte

Testo memorizzato nella variabile di default REPLAY

Esercizio

- Si scriva una script di shell che a seguito di due messaggi legga da tastiera due valori interi e ne visualizzi la somma e il prodotto

```
#!/bin/bash
# Sum and product

echo -n "Reading n1: "
read n1
echo -n "Reading n2: "
read n2
let s=n1+n2
let p=n1*n2
echo "Sum: $s"
echo "Product: $p"

exit 0
```

-n non a capo

Lettura da tastiera

Espressioni aritmetiche (vedere in seguito)

No spazi prima e dopo =, +, *

Esercizio

- Si scriva uno script che legga da tastiera il nome di un utente e visualizzi quanti login ha effettuato
  - La lista degli utenti connessi è fornita dal comando who oppure w

Uso di comandi di shell, variabili, etc.

```
#!/bin/bash
# Number of login(s) of a specific user

echo -n "User name: "
read user

# who is logged | the user | word count #lines
times=$(who | grep $user | wc -l)

echo "User $user has $times login(s)"

exit 0
```

--lines = -l = #line

Esercizio

- Si scriva uno script che legga da tastiera una stringa e ne visualizzi la lunghezza

```
#!/bin/bash
# String length

echo "Type a word: "
read word

# echoing without newline | word count chars
l=$(echo –n $word | wc –c)

echo "Word $word is $l characters long"

exit 0
```

echo -n = no new line

--chars = -m = #char
--bytes = -c = #bytes

Scrittura (output)

- Le operazioni di visualizzazione possono essere effettuate con le funzioni
  - echo
  - printf
- La funzione **printf** ha una sintassi simile a quella del linguaggio C
  - Utilizza sequenze di escape
  - Non è necessario separare i vari campi con la ","

Scrittura (output)

- La funzione **echo**
  - Visualizza i propri argomenti, separati da spazi e terminati da un carattere di "a capo"
  - Accetta diverse opzioni
    - -e interpreta i caratteri di escape (\...)
      - \b backspace
      - \n newline
      - \t tabulazione
      - \\ barra inversa
      - etc.
    - -n sopprime il carattere di "a capo" finale

Esempi: I/O

```
echo "Printing with a newline"
echo –n "Printing without newline"
echo –e "Deal with \n escape \t\t characters"
printf "Printing without newline"
printf "%s \t%s\n" "Ciao. It's me:" "$HOME"
```

Diverse operazioni di output

I & O insieme in uno script intero

```
#!/bin/bash
# Interactive input/output
echo –n "Insert a sentence: "
read w1 w2 others
echo "Word 1 is: $w1"
echo "Word 2 is: $w2"
echo "The rest of the line is: $others"
exit 0
```

Espressioni aritmetiche

- Per esprimere espressioni aritmetiche è possibile utilizzare diverse notazioni
  - Il comando **let "..."**
  - Le doppie parentesi tonde **((...))**
  - Le parentesi quadre **[...]**
  - Il costrutto **expr**
    - Valuta il valore di una espressione richiamando una nuova shell
    - Meno efficiente
    - Normalmente non utilizzato

Si osservi che una espressione aritmetica viene valutata vera (exit status) SEE è diversa da 0
espressione!=0 → TRUE → exist status=0

Esempi

Espressioni aritmetiche alternative

Uso di ((e))

```
i=1
((v1=i+1))
((v2=$i+1))
v3=$(($i+1))
v4=$((i+1))
echo $i $v1 $v2 $v3 $v4
1 2 2 2 2
```

Uso di let

```
i=1
let "v1=i+1"
let v2=i+1
let v3=$i+1
echo $i $v1 $v2 $v3
1 2 2 2
```

Se non è tra "..." l'espressione **non** deve contenere spazi

Uso di [e]

```
i=1
v1=$[$i+1]
v2=$[i+1]
echo $i $v1 $v2
1 2 2
```

Costrutto condizionale if-then-fi

- Il costrutto condizionale **if-then-fi**
  - Verifica se lo stato di uscita (exit status) di una sequenza di comandi è uguale a 0
    - Ricordare: 0=vero in shell UNIX
  - In caso affermativo esegue uno o più comandi
- Il costrutto può essere esteso
  - Per includere la condizione else
    - if-then-else-fi
  - Per effettuare controlli annidati
    - if-then-…-if-then-…-fi-fi
    - if-then-elif-…-fi

Costrutto condizionale if-then-fi

```
# Syntax 1
if condExpr
then
  statements
fi
```

Formato standard

Costrutto su un'unica riga: necessario il ';'

```
# Syntax 2
if condExpr ; then
  statements
fi
```

Formato con else

```
# Syntax 3
if condExpr
then
  statements
else
  statements
fi
```

if-then-else-fi annidati possono tradursi con if-then-elif-fi

```
# Syntax 4
if condExpr
then
  statements
elif condExpr
then
  statements
else
  statements
fi
```

Costrutto condizionale if-then-fi

- condExpr
  - Le espressioni condizionali possono seguire due sintassi differenti

```
# Syntax 1
test param op param
```

Esistono operatori per
Numeri
Stringhe
Valori logici
File e direttori

```
# Syntax 2
[ param op param ]
```

Prima e dopo le parentesi deve essere inserito uno spazio

Costrutto condizionale if-then-fi

Operatori per numeri	
-eq	==
-ne	!=
-gt	>
-ge	>=
-lt	<
-le	<=
!	! (not)

Operatori per file e direttori	
-d	L'argomento è una directory
-f	L'argomento è una file regolare
-e	L'argomento esiste
-r	L'argomento ha il permesso di lettura
-w	L'argomento ha il permesso di scrittura
-x	L'argomento ha il permesso di esecuzione
-s	L'argomento ha dimensione non nulla

Operatori per stringhe	
=	strcmp
!=	!strcmp
-n string	non NULL string
-z string	NULL (empty) string

Operatori logici	
!	NOT (in condizione singola)
-a	AND (in condizione singola
-o	OR (in condizione singola)
&&	AND (in un elenco di condizioni)
||	OR (in un elenco di condizioni)

Esempi

```
if [ 0 ]   # true
   [ 1 ]   # true
   [ -1 ]  # true
   [ ]     # NULL is false
   [ str ] # a random string, e.g., “abc”
           # or abc is true
```

Valori logici

Formato equivalente:
`if test $v1 –eq $v2`

```
if [ $v1 –eq $v2 ]
then
  echo “v1==v2”
fi
```

Test su interi

```
if [ $v1 -lt 10 ]
then
  echo “$v1 < 10”
else
  echo “$v1 >= 10”
fi
```

Esempi: Check di file

```
if [ “$a” –eq 24 –a “$s” = “str” ]; then
  ...
fi
```

AND di condizioni

Formato equivalente

```
if [ “$a” –eq 24 ] && [ “$s” = “str” ]
 if [[ “$a” –eq 24 && “$s” = “str” ]]
```

```
if [ $recursiveSearch –eq 1 –a –d $2 ]
then
  find $2 –name *.c > $3
else
  find $2 –maxdepth 1 *.c > $3
fi
```

Esempi: Check di stringhe

```
if [ $string = “abc” ]; then
  echo “string \”abc\” found”
fi
```

Test su stringhe

Se $string è vuota (e.g., return da tastiera) la sintassi è errata: e diventa uguale a: **`[ = “abc” ]`**
Utilizzare le virgolette per avere un format “sicuro”:
`if [ “$string” = “abc” ]; then`
che diventerebbe**`[ “” = “abc” ]`**

```
if [ -f foo.c ]; then
  echo “File in the directory”
fi
```

Test su file

Esempi: Script completo

Lettura stringa da stdin
Controllo su stringa
Visualizzazione output

```
#!/bin/sh

echo –n “Is it morning (yes/no)? ”
read string
if [ “$string” = “yes” ]; then
  echo “Good morning”
else
  echo “Good afternoon”
fi

exit 0
```

Esempi: Script completo

```
#!/bin/sh

echo –n “Is it morning (yes/no)? ”
read string
if [ “$string” = “yes” ]; then
  echo “Good morning”
elif [ “$string” = “no” ]; then
  echo “Good afternoon”
else
  echo “Sorry, wrong answer”
fi

exit 0
```

Lettura stringa da stdin
Controllo su stringa
Visualizzazione output
Utilizzando elif

Costrutto iterativo for-in

- Il costrutto **for-in**
  - Esegue i comandi specificati, una volta per ogni valore assunto dalla variabile **var**
  - L'elenco dei valori **[list]** può essere indicato
    - In maniera esplicita (elenco)
    - In maniera implicita (comandi di shell, wild-cards, etc.)

```
# Syntax 1
for var in [list]
do
  statements
done
```

```
# Syntax 2
for var in [list]; do
  statements
done
```

Osservazione: costrutto definito, i.e., itera un numero **predefinito** di volte

Esempi: for con elenco esplicito

```
for foo in 1 2 3 4 5 6 7 8 9 10
do
  echo $foo
done
```

Stampa un elenco di numeri

```
for str in foo bar echo charlie tango
do
  echo $str
done
```

Stampa un elenco di stringhe

```
num="2 4 6 9 2.3 5.9"
for file in $num
do
  echo $file
done
```

Stampa un elenco di numeri utilizzando una variabile (vettoriale, vedere in seguito)

Esempi: for e wild-chars

Iterazione sui parametri dello script

```
n=1
for i in $* ; do
  echo “par #” $n = $i
  let n=n+1
done
```

Visualizza tutti i parametri ricevuti sulla riga di comando

```
for f in $(ls | grep txt); do
  chmod g+x $f
done
```

Cambia i privilegi a file specifici

```
rm –rf number.txt
for i in $(echo {1..50})
do
  echo –n “$i “ >> number.txt
done
```

Genera un file con i numeri da 1 a 50 sulla stessa riga, separati da uno spazio e li scrive in number.txt

Osservazione: la riderezione è a livello di echo; ‘>’ generebbe un nuovo file a ogni iterazione

Costrutto iterativo while-do-done

- Iterazione indefinite (il numero di iterazioni è ignoto)
  - Si itera sino a quando la condizione è vera
  - Si termina il ciclo quando la condizione è falsa

```
# Syntax 1

while [ cond ]
do
  statements
done
```

```
# Syntax 2

while [ cond ] ; do
  statements
done
```

Esempio: Script completo

```
#!/bin/bash

limit=10
var=0
while [ “$var” -lt “$limit” ]
do
  echo “Here we go again $var”
  let var=var+1
done

exit 0
```

Visualizza 10 volte il messaggio indicato

Esempio: Script completo

```
#!/bin/bash

echo “Enter password: “
read myPass

while [ “$myPass” != “secret” ]; do
  echo “Sorry. Try again.”
  read myPass
done

exit 0
```

Visualizza il messaggio indicato sino all’introduzione della stringa corretta

Esempio: Script completo

```
#!/bin/bash

n=1
while read row
do
  echo "Row $n: $row"
  let n=n+1
done < in.txt > out.txt

exit 0
```

Lettura di righe intere in 1 variabile (sino al new-line)

Dato che il costrutto while-do-done è considerato come unico, la redirezione (di I/O) deve essere fatta al termine del costrutto

Scrivere `echo … > out.txt` implicherebbe sovrascrivere il file tutte le volte. Al limite usare `echo … >> file.txt`

Scrivere `while read row < in.txt` implicherebbe rileggere sempre la stessa riga del file

Nomi dei file costanti.
Possibile l'uso di parametri o variabili
: … `< $1 > $var`

Esercizio

- Scrivere uno script bash in grado di
  - Ricevere due interi n1 e n2 sulla riga di commando oppure di leggerli da tastiera se non sono presenti sulla riga di commando
  - Visualizzare una matrice di n1 righe e n2 colonne di valori interi crescenti a partire dal valore 0
  - Esempio

```
> ./myScript 3 4
0  1  2  3
4  5  6  7
8  9  10  11
```

Soluzione

```
#!/bin/bash
if [ $# -lt 2 ] ; then
  echo -n “Values: "
  read n1 n2
else
  n1=$1
  n2=$2
fi
n=0
```

Lettura dati in ingresso

```
r=0
while [ $r -lt $n1 ] ; do
  c=0
  while [ $c -lt $n2 ] ; do
    echo -n "$n "
    let n=n+1
    let c=c+1
  done
  let r=r+1
  echo
done
exit 0
```

Doppio ciclo di visualizzazione

Break, continue e ':'

- I costrutti break e continue hanno comportamenti standard con i cicli for e while
  - Uscita non struttura dal ciclo
  - Passaggio all'iterazione successiva
- Il carattere ':' può essere utilizzato
  - Per creare "istruzioni nulle"

```
if [ -d "$file" ]; then
   :       # Empy instruction
fi
```

Il : è anche utilizzabile come condizione vera
`while :` coincide con **`while [ 0 ]`**

Vettori

- In bash è possibile utilizzare variabili vettoriali mono-dimensionali
  - Ogni variabile può essere definita come vettoriale
    - La dichiarazione esplicita non è necessaria (ma possibile con il costrutto **declare**)
  - Non esiste
    - Limite alla dimensione di un vettore
    - Alcun vincolo sull'utilizzo di indici contigui
  - Gli indici partono usualmente da 0
    - Zero-based indexing, come in C

I vettori in shell **non** sono associative (no hashing)

Vettori

- Si supponga **name** sia il nome di un vettore
  - Definizione
    - Elemento per elemento
      - name[index]="value"
    - Tramite elenco di valori
      - name=(lista di valori separate da spazi)
  - Riferimento
    - Al singolo elemento
      - ${name[index]}
    - A tutti gli elementi
      - ${name[*]}

Un nuovo elemento può essere aggiunto in qualsiasi momento

* Oppure @

L'utilizzo delle parentesi graffe {} è obbligatorio

Vettori

- Numero di elementi
  - ${#name[*]}
- Lunghezza dell'elemento index (numero caratteri)
  - ${#name[index]}

- Il costrutto unset può distruggere vettori o elementi di vettori
  - Eliminazione di un elemento
    - unset name[index]
  - Eliminazione di un intero vettore
    - unset name

Esempi: Uso di vettori

Init come lista e stampa

```
> vet=(1 2 5 ciao)
> echo ${vet[0]}
1
> echo ${vet[*]}
1 2 5 ciao
> echo ${vet[1-2]}
2 5
> vet[4]=bye
> echo ${vet[*]}
1 2 5 ciao bye
```

Eliminazione

```
> unset vet[0]
> echo ${vet[*]}
2 5 ciao bye
> unset vet
> echo ${vet[*]}

> vet[5]=50
> vet[10]=100
> echo ${vet[*]}
50 100
```

Indici non contigui

Esercizio

- Realizzare uno script in grado di
  - Leggere un insieme indefinito di numeri
  - Terminare la fase di lettura quando viene introdotto il valore 0
  - Visualizzare i valori in ordine inverso
  - Esempio

    Input n1: 10

    ...

    Input n10: 100

    Input n11: 0

    Output: 100 ... 10

Soluzione

```
#!/bin/bash
i=0
while [ 0 ] ; do
  echo -n "Input $i: "
  read v
  if [ "$v" -eq "0" ] ; then
    break;
  fi
  vet[$i]=$v
  let i=i+1
done
```

Anche :

Input

echo ${vet[*]} visualizzerebbe gli elementi nelo stesso ordine e separati da uno spazio

```
echo
let i=i-1
while [ "$i" -ge "0" ]
do
  echo "Output $i: ${vet[$i]}"
  let i=i-1
done
exit 0
```

Output in ordine inverso