Laboratorio di sistemi operativi
A.A. 2010/2011
Gruppo 2
Gennaro Oliva
4
Gli script di shell

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

Exit status

- Ogni comando shell, al termine dell'esecuzione, fornisce un exit status ovvero un valore intero compreso tra `0` e 255 che descrive l'esito dell'esecuzione
- Per convenzione:

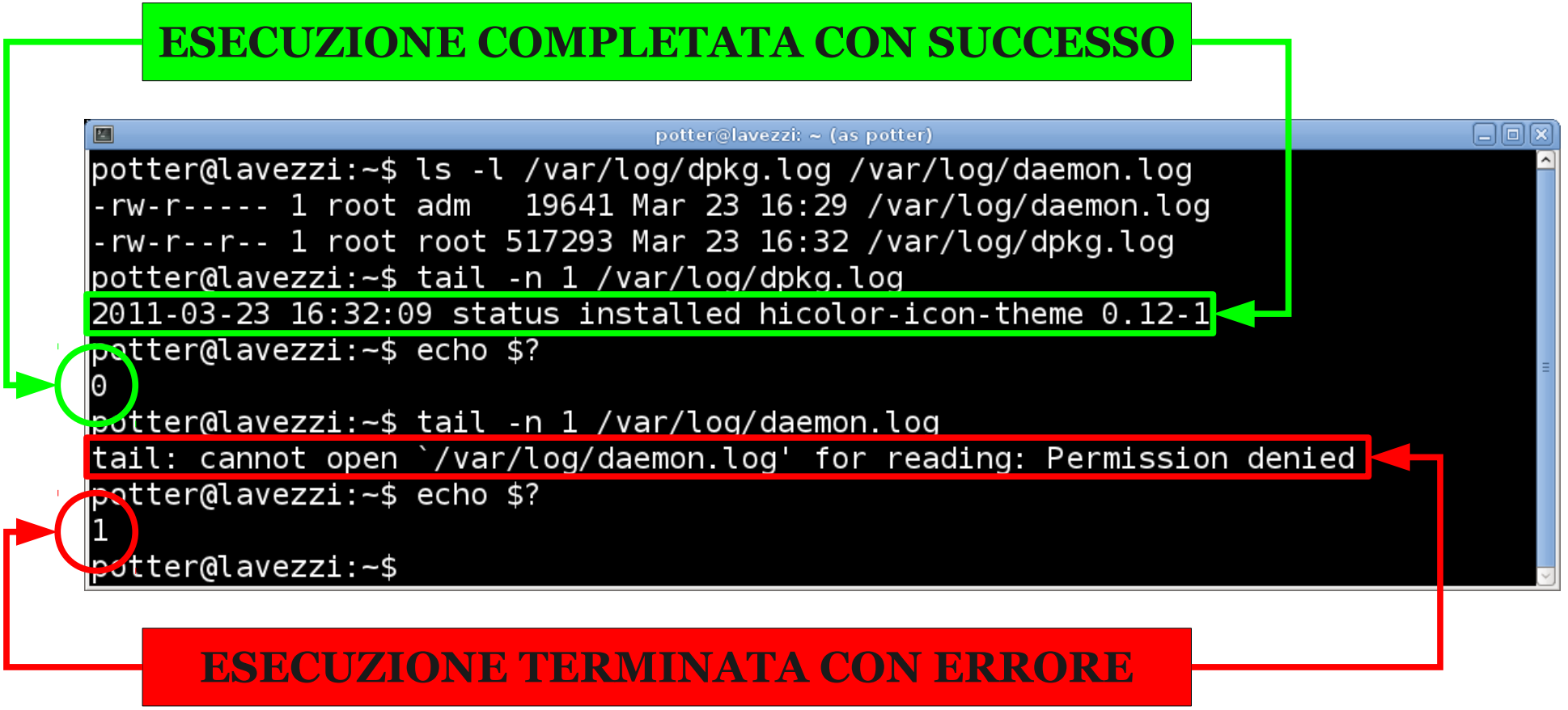
  `0` indica che il programma è stato eseguito con successo

  un valore non nullo indica che il programma non è terminato correttamente
- La shell memorizza l'exit status dell'ultimo comando eseguito nella variabile `$?`, per visualizzarlo possiamo usare echo

```
$ echo $?
```

Exit status

- Cercando di accedere ad un file su cui l'utente non ha permesso di lettura il comando tail ci segnala l'errore



Exit status

- Per alcuni programmi un exit status non nullo non significa necessariamente che si è verificato un errore
- Il comando diff:
  - `0` se i file sono uguali
  - 1 se i file sono diversi
  - 2 se ci sono stati problemi
- Il significato del valore dell'exit status è descritto nella pagina di manuale

Liste di comandi

- Sulla stessa linea di comando è possibile eseguire una lista di comandi separandoli con caratteri speciali ';','|','&','&&','||' ottenendo vari tipi di esecuzione:
- in sequenza (prima 1 poi 2)

```
$ comando1 ; comando2
```

- concorrente (1 in background e 2 in foreground)

```
$ comando1 & comando2
```

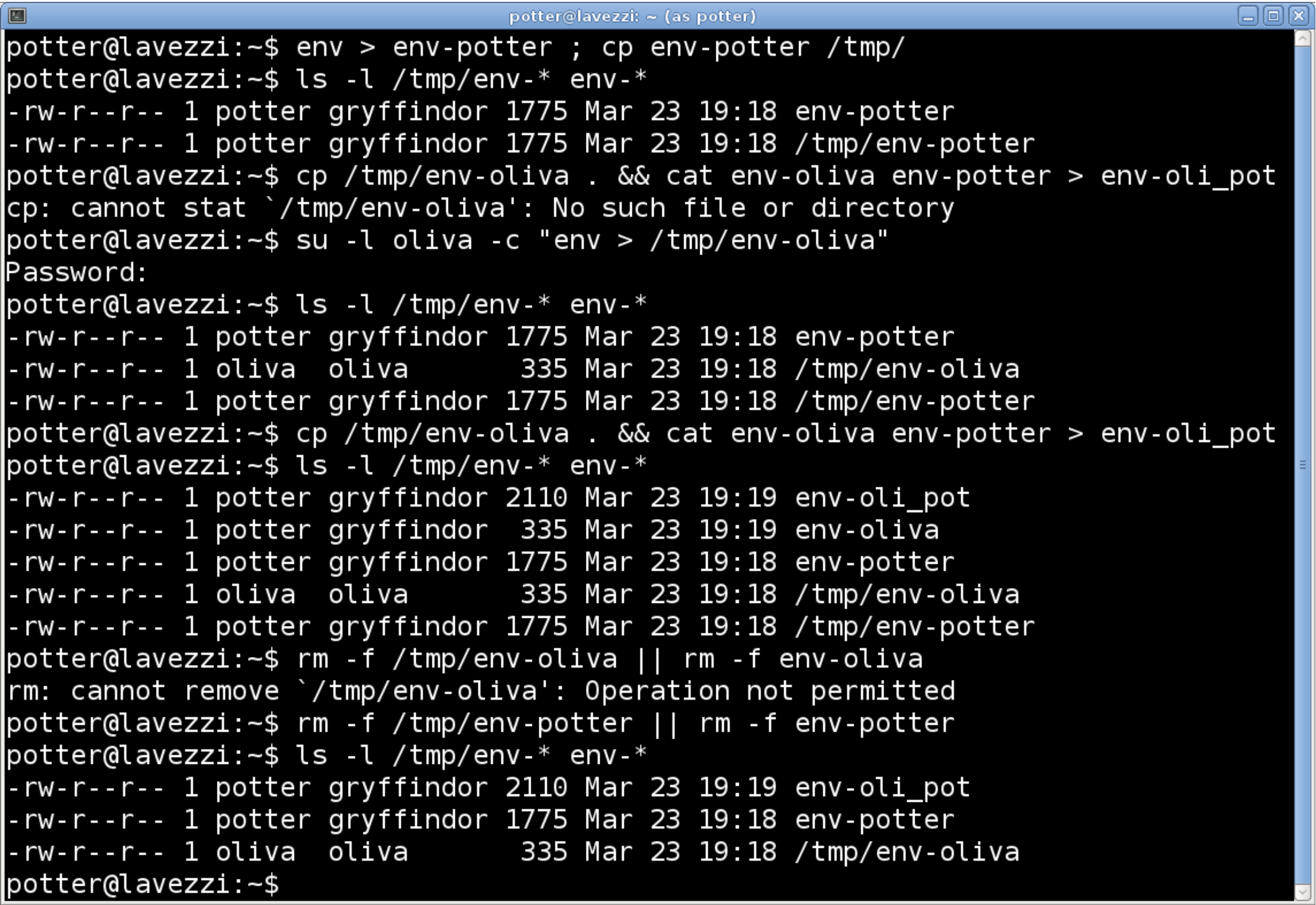
- condizionata (se 1 esce con 0 esegue anche 2)

```
$ comando1 && comando2
```

- esclusiva (se 1 non esce con 0 esegue 2)

```
$ comando1 || comando2
```

Liste di comandi



Liste di comandi ';'

```
potter@lavezzi:~$ env > env-potter ; cp env-potter /tmp/
potter@lavezzi:~$ ls -l /tmp/env-* env-*
-rw-r--r-- 1 potter gryffindor 1775 Mar 23 19:18 env-potter
-rw-r--r-- 1 potter gryffindor 1775 Mar 23 19:18 /tmp/env-potter
```

- Con ';' i due comandi vengono eseguiti in sequenza (prima env poi cp) indipendentemente dal loro exit status

Liste di comandi '&&'

```
potter@lavezzi:~$ ls -l /tmp/env-* env-*
-rw-r--r-- 1 potter gryffindor 1775 Mar 23 19:18 env-potter
-rw-r--r-- 1 potter gryffindor 1775 Mar 23 19:18 /tmp/env-potter
potter@lavezzi:~$ cp /tmp/env-oliva . && cat env-oliva env-potter > env-oli_pot
cp: cannot stat `/tmp/env-oliva': No such file or directory
```

- Con '&&' il primo comando viene eseguito sempre, ma il secondo soltanto se il primo è terminato con exit status `0`
- In questo caso cat non viene eseguito perche' cp non ha trovato il file da copiare

Liste di comandi '&&'

- Con '&&' il primo comando viene eseguito sempre, ma il secondo soltanto se il primo è terminato con exit status `0`

```
potter@lavezzi:~$ su -l oliva -c "env > /tmp/env-oliva"
Password:
potter@lavezzi:~$ ls -l /tmp/env-* env-*
-rw-r--r-- 1 potter gryffindor 1775 Mar 23 19:18 env-potter
-rw-r--r-- 1 oliva  oliva       335 Mar 23 19:18 /tmp/env-oliva
-rw-r--r-- 1 potter gryffindor 1775 Mar 23 19:18 /tmp/env-potter
potter@lavezzi:~$ cp /tmp/env-oliva . && cat env-oliva env-potter > env-oli_pot
potter@lavezzi:~$ ls -l /tmp/env-* env-*
-rw-r--r-- 1 potter gryffindor 2110 Mar 23 19:19 env-oli_pot
-rw-r--r-- 1 potter gryffindor  335 Mar 23 19:19 env-oliva
-rw-r--r-- 1 potter gryffindor 1775 Mar 23 19:18 env-potter
-rw-r--r-- 1 oliva  oliva       335 Mar 23 19:18 /tmp/env-oliva
-rw-r--r-- 1 potter gryffindor 1775 Mar 23 19:18 /tmp/env-potter
```

- Una volta generato il file da copiare, eseguendo lo stesso comando il cat verrà eseguito perché cp è terminato senza errori

Liste di comandi '||'

- Con '||' il primo comando viene sempre eseguito, ed il secondo soltanto se il primo è terminato con exit status diverso da `0`
- In questo caso il secondo rm viene eseguito perché il primo rm non può cancellare il file di un altro utente

```
-rw-r--r-- 1 potter gryffindor  335 Mar 23 19:19 env-oliva
-rw-r--r-- 1 potter gryffindor 1775 Mar 23 19:18 env-potter
-rw-r--r-- 1 oliva  oliva       335 Mar 23 19:18 /tmp/env-oliva
-rw-r--r-- 1 potter gryffindor 1775 Mar 23 19:18 /tmp/env-potter
potter@lavezzi:~$ rm -f /tmp/env-oliva || rm -f env-oliva
rm: cannot remove `/tmp/env-oliva': Operation not permitted
```

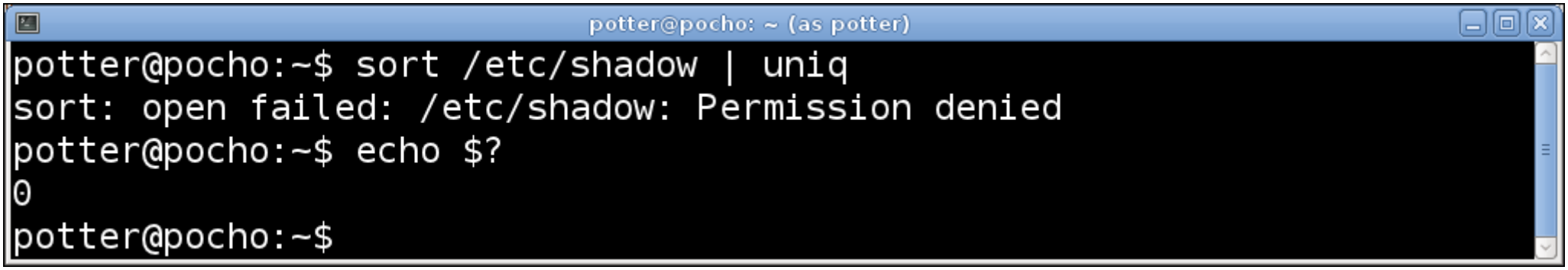
Liste di comandi '||'

- Con '||' il primo comando viene sempre eseguito, ed il secondo soltanto se il primo è terminato con exit status diverso da `0`
- In questo caso il secondo rm non viene eseguito perché il primo rm termina senza errori

```
potter@lavezzi:~$ rm -f /tmp/env-potter || rm -f env-potter
potter@lavezzi:~$ ls -l /tmp/env-* env-*
-rw-r--r-- 1 potter gryffindor 2110 Mar 23 19:19 env-oli_pot
-rw-r--r-- 1 potter gryffindor 1775 Mar 23 19:18 env-potter
-rw-r--r-- 1 oliva  oliva       335 Mar 23 19:18 /tmp/env-oliva
potter@lavezzi:~$
```

Exit status di liste di comandi

- Quando si eseguono più programmi in una sola linea di comando (come nelle pipeline o nelle liste di comandi), l'exit status è relativo all'ultimo comando eseguito
- In questo esempio uniq restituisce un exit status pari a `0` perché ha elaborato con successo lo standard input vuoto (ovvero lo standard output di sort)



Script BASH

- Uno script di shell BASH è un file di testo che:
  - contiene comandi di shell
  - inizia con la stringa “#!/bin/bash”
  - ha permesso di esecuzione
- Non c'è differenza tra quello che si può scrivere utilizzando la shell interattivamente o mediante uno script

#!

- I primi due caratteri #! indicano che il file è uno script o più in generale un programma interpretato
- La stringa successiva è il pathname dell'interprete per il programma
- La shell esegue l'interprete specificato nella prima linea passandogli come argomento il nome dello script

Commenti

- Per inserire commenti negli script di shell è necessario utilizzare il carattere #
- La shell ignora tutto quello che segue # in qualsiasi punto della riga esso si trovi

```
#!/bin/bash
#Il mio primo script saluta
echo ciao mondo #che originale!
```

- L'output di questo script è:

```
ciao mondo
```

Linee lunghe

- Per aumentare la leggibilità dei nostri script in presenza di linee di comando molto lunghe è possibile utilizzare il carattere \
- Tale carattere posto al termine di una linea **senza nessun altro carattere di seguito** segnala alla shell che il comando continua sulla linea successiva

- Esempio:

```
$ echo Questo terminale ha un \
massimo di $COLUMNS caratteri
```

COLUMS è una variabile d'ambiente predefinita

Argomenti di uno script

- Nell'esecuzione di uno script è possibile passare argomenti (o parametri posizionali) sulla linea di comando
- La sintassi da utilizzare:

```
$ myscript argomento1 argomento2 …
```

- In questo modo lo script può operare diversamente a seconda degli argomenti passati
- Gli argomenti possono essere nomi di file, opzioni o anche stringhe qualsiasi

Argomenti di uno script

- Gli argomenti sono ordinati in base alla posizione che occupano sulla linea di comando
- Vengono identificati all'interno dello script con le variabili $N dove N è il numero intero che ne indica la posizione
  - La variabile `$0` è il primo argomento della linea di comando: il nome dello script o il suo pathname a seconda di come viene invocato
  - Le variabili $1, $2... rappresentano, nell'ordine, gli argomenti passati sulla linea di comando allo script
  - La variabile $# contiene il numero di argomenti passati
  - La variabile $@ contiene tutti gli argomenti passati

Argomenti di uno script

- Nel caso dell'esempio:

```
$ myscript argomento1 argomento2
$0=myscript
$1=argomento1
$2=argomento2
$#=2
$@=”argomento1 argomento2”
```

Exit

- Il comando exit può essere utilizzato per terminare uno script e restituire alla shell un exit status
- L'exit status **valido** è un intero nell'intervallo [0, 255] e viene passato come argomento a exit

```
$ exit 1 #restituisce exit status 1
```

- È opportuno seguire la convenzione Unix per cui un exit status pari `0` segnala che lo script è stato eseguito con successo mentre un intero positivo segnala eventuali errori riscontrati
- È utile restituire interi diversi per errori diversi
- Se exit viene specificato senza parametri o lo script viene terminato senza exit, il valore di ritorno dello script quello dell'ultimo comando eseguito al suo interno

if then

- Il costrutto if consente l'esecuzione condizionata di comandi secondo la sintassi:

```
if comando-test
then
  comandi
  …
fi
```

- Il corpo `comandi` che segue il then viene eseguito solo se l'exit value di `comando-test` è `0`
- Il corpo di comandi condizionati termina con la stringa `fi`

if then

- Una sintassi più leggibile del costrutto ugualmente valida:

```
if comando-test ; then
  comandi
  …
fi
```

- Il `comando-test` può essere anche una lista di comandi
- In tal caso l'exit status che abilita l'esecuzione del corpo è quello dell'ultimo comando eseguito nella lista

if then

- L'argomento di if `comando-test` è un comando od una lista di comandi e non un espressione generica da valutare come nel C
- Nell'esempio seguente si utilizza diff come `comando-test`

```
#!/bin/bash
if diff $1 $2
then
  rm $2
  echo Cancello $2 duplicato di $1
fi
```

- La sintassi per l'esecuzione dello script

```
$ cancella_duplicati.sh file1 file2
```

test

- Unix fornisce il comando test per la valutazione delle espressioni
- test ha exit status pari a `0` in caso di espressione vera e pari a 1 in caso di espressione falsa
- Il comando test ha una duplice sintassi, è possibile invocarlo esplicitamente con la sintassi

  test espressione
- oppure per aumentare la leggibilità degli script con la sintassi

  [espressione]

Confronto di valori numerici

- Le espressioni possono specificare confronti tra coppie di valori numerici interi secondo la sintassi:

  `test X -sw Y` oppure `[ X -sw Y ]`

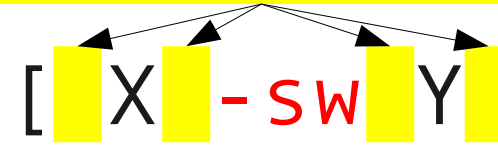
- Dove i valori validi per -sw sono:

-eq oppure =	vero se X = Y (equal)
-ne oppure !=	vero se X ≠ Y (not equal)
-lt	vero se X < Y (less then)
-le	vero se X ≤ Y (less or equal)
-gt	vero se X > Y (greater then)
-ge	vero se X ≥ Y (greater or equal)

Confronto di valori numerici

- Le espressioni possono specificare confronti tra coppie di valori numerici interi secondo la sintassi:

ATTENZIONE AGLI SPAZI!!!



`test X -sw Y` oppure `[ X -sw Y ]`

- Dove i valori validi per -sw sono:

-eq oppure =	vero se X = Y (equal)
-ne oppure !=	vero se X ≠ Y (not equal)
-lt	vero se X < Y (less then)
-le	vero se X ≤ Y (less or equal)
-gt	vero se X > Y (greater then)
-ge	vero se X ≥ Y (greater or equal)

test, stringhe e file

- Le espressioni possono specificare confronti tra coppie di stringhe secondo la sintassi:
  `test “XXX” op “YYY”` oppure
  `[ “XXX” op “YYY” ]`

  dove op può essere = o !=
- Si possono anche verificare le caratteristiche di un file
  `test -sw file_name` oppure
  `[ -sw file_name ]`
- Dove i valori validi per -sw sono:
  -e vero se il file_name esiste
  -x vero se il file_name eseguibile
  -d vero se il file_name è una directory

Confronto tra date di modifica

- Il comando test consente di effettuare confronti sulla data di ultima modifica di coppie di file secondo la sintassi:

  `test file1 -sw file2` oppure
  `[ file1 -sw file2 ]`

- Dove i valori validi per -sw sono:
  -ot vero file1 e più vecchio di file2 (older than)
  -nt vero file1 e più nuovo di file2 (newer than)

test e operatori logici

- Con test è possibile combinare più espressioni nella stessa valutazione mediante la sintassi
  `$ test espressione1 -sw Espressione2`

  oppure:

  `[ espressione1 -sw Espressione2 ]`

  dove -sw può essere:

  -a and
  -o or

- Il carattere ! può essere anteposto ad un'espressione per negarla

  ! not

  `[ ! espressione ]`

Esempio d'uso di test

```
#!/bin/bash
if [ “$#” != “2” ] ; then
  echo Usa $0 file1 file2
  exit 1
fi
if [ ! -e “$1” -o ! -e “$2” ]
  then
  echo Specifica file esistenti!
  exit 1
fi
if diff $1 $2 ; then
  rm $2
  echo Cancello $2 duplicato di $1
fi
```

else ed elif

- La sintassi del comando if consente di specificare ipotesi alternative con elif e con else secondo la sintassi:

```
if comando-test1
then
  comando1-1
  …
elif comando-test2
then
  comando2-1
  …
else
  comando3-1
  …
fi
```

Esempio d'uso di elif ed else

```
#!/bin/bash
if [ “$#” != “2” ] ; then
   echo Usa $0 file1 file2
   exit 1
fi
if [ ! -e “$1” -o ! -e “$2” ] ; then
   echo Specifica file esistenti!
   exit 1
elif [ “$1” == “$2” ] ; then
   echo Specifica file diversi!
   exit 1
else
   if diff $1 $2 ; then
      rm $2
      echo Cancello $2 duplicato di $1
   fi
fi
```

case

- In presenza di numerose possibilità da vagliare è opportuno utilizzare il costrutto case con la sintassi:

```
case word in
  pattern1)
  comandi1
  …
  ;;
  pattern2)
  comandi2
  ;;
  …
esac
```

- La shell confronta il valore dell'espressione word (tipicamente una variabile) con i pattern specificati nell'ordine in cui sono elencati
- Se trova una corrispondenza esegue la lista di comandi associati fino ad incontrare la stringa ';;' e termina la ricerca
- Il corpo del costrutto case si conclude con la stringa `esac`

case

- I pattern possono utilizzare le regole del file globbing
- Specificando come ultimo pattern '*)' i comandi ad esso associati vengono eseguiti quando non viene trovata nessuna corrispondenza
- E' possibile specificare più alternative per un singolo pattern delimitandole con il carattere | (pipe)

```
case word in
  patter1|pattern2|pattern3...)
  comandi
  ;;
  ...
```

Esempio d'uso di case

```
#!/bin/bash
[ "$#" == "1" ] || exit 1
case $1 in
   shell.studenti.unina.it|192.132.34.22)
   LOGIN=N8600000000
   ;;
   192.168.1.1)
   LOGIN=admin
   ;;
   192.168.1.2)
   LOGIN=potter
   ;;
   192.168.1.*)
   LOGIN=root
   ;;
   *)
   LOGIN=$USER
   ;;
esac
ssh $1 -l $LOGIN
```

for do

- Il costrutto for consente di ripetere iterativamente l'esecuzione di un'insieme di comandi con la sintassi:

```
for var in elenco-valori
do
   comando
   …
done
```

- In `elenco-valori` vengono specificati i valori che deve assumere la variabile `var` ad ogni passo del ciclo

Il comando seq

- E' spesso utile eseguire un ciclo for utilizzando un elenco di valori numerici
- Il comando seq genera sequenze di numeri secondo la sintassi

```
$ seq primo incremento ultimo
```

- I parametri primo e incremento sono opzionali
- Due valori vengono considerati primo e ultimo
- Esempi d'uso:

```
$ seq 1  4
1
2
3
4
```

Primo

Ultimo

```
$ seq 1  2  10
1
3
5
7
9
```

Primo

Incremento

Ultimo

Sostituzione di comandi

- La sostituzione di comando è un tipo di espansione della shell che consente di sostituire l'output di un comando con l'espressione:
  - $(comando)

    oppure
  - `comando`
- Gli apici per la sostituzione di comando ` ` differiscono dagli apici singoli ‘ ' del quoting
- Esempi:

```
$ echo Oggi e\' $(date)
Oggi e' gio 24 mar 2011, 08.32.06, CET
```

- Esegue il backup della cartella script aggiungendo la data in formato AAAA-MM-GG al nome dell'archivio

```
$ tar cvfz scripts-$(date +%Y-%m-%d).tgz script/
```

Esempio di utilizzo di seq con for

- Utilizzando seq e la sostituzione di comando possiamo generare una lista di interi per la varibile del nostro ciclo for

```
#!/bin/bash
for n in $(seq 1 10) ; do
  lame track$n.wav track$n.mp3
done
```

- Lo script dell'esempio converte i file track1.wav,track2.wav,... in track1.mp3,track2.mp3,...

for e file globbing

- L'elenco valori può contenere caratteri speciali per il file globbing (*,?,[...]), espressioni con le parentesi graffe ({...,...,...}), variabili
- La shell esegue l'espansione nella creazione dell'elenco e assegna di volta in volta gli elementi alla variabile
- Lo script di esempio compila tutti i file .c della directory corrente e alla fine genera il main dai file oggetto creati

```
#!/bin/bash
for file in *.c ; do
  cc -c $file
done
cc -o main *.o
```

while do

- Il costrutto while consente di ripetere iterativamente un insieme di istruzioni in base all'exit status di un comando test secondo la sintassi:

```
while comando-test
do
   comando
   …
done
```

- Il corpo del ciclo viene eseguito fintanto che il comando-test restituisce un exit value di `0`

while do

- Un sintassi più leggibile del costrutto while:

```
while comando-test ; do
   comando
   …
done
```

- L'insieme di comandi da eseguire è delimitato dalla stringa `done`
- Come per il costrutto if per la valutazione di espressioni generiche è utile usare il comando test

Esempio d'uso di while do

- Lo script che segue verifica periodicamente che la mailbox dell'utente /var/spool/mail/$USER sia più vecchia del file creato all'inzio dello script
- Quando arriva una nuova la data di ultima modifica della mailbox viene aggiornata e lo script esce dal ciclo while

```
#!/bin/bash
touch /tmp/check_mail
while [ “/tmp/checkmail” -ot “/var/spool/mail/$USER” ]
do
   #Attendi 10 secondi
   sleep 10
done
rm -f /tmp/check_mail
echo you have new mail!
```

until do

- Il costrutto until è simile al costrutto while con l'unica differenza che i comandi specificati all'interno del corpo vengono eseguiti soltanto se l'exit status del comando test è diverso da `0`

```
until comando-test
do
   comando
   …
done
```

- Oppure:

```
until comando-test ; do
   comando
   …
done
```

Espressioni aritmetiche

- La shell consente di effettuare operazioni di vario tipo su valori numerici interi mediante la sintassi: `((espressione_aritmetica))`
- All'interno delle espressioni è possibile utilizzare variabili e costanti combinate con operatori simili a quelli utilizzati nel linguaggio C

operatori	operazioni
++ --	incremento e decremento di variabili
+,-,*,/,**	aritmetica elementare
|| && !	logica
>>,<<	shift
<,>,<=,>=	confronto

- Le variabili vengono espanse dalla shell prima della valutazione dell'espressione
- La shell non effettua controlli sull'overflow

Espansione di espressioni aritmetiche

- Preponendo il carattere $ ad un'espressione aritmetica il risultato dell'operazione viene sostituito all'espressione

- ```
  echo Questo terminale ha $COLUMNS \
  colonne e $LINES linee per un \
  totale di $(($COLUMNS*$LINES)) \
  caratteri
  ```

LINES e COLUMS sono variabili d'ambiente predefinite
  ```

Espansione aritmetica

```
#!/bin/bash
[ "$#" != "1" ] || echo usa $0 numero && exit 1
[ "$1" == "1" ] && echo 0 && exit 0
[ "$1" == "2" ] && echo 1 && exit 0
if [ "$1" -ge "3" ] ; then
   A=1; B=1; i=3;
   while [ "$i" -le "$1" ] ; do
        C=$(($A+$B));
        A=$B;
        B=$C;
        ((i++))
   done
   echo $C;
   exit 0
else
   echo Numero $1 non valido
   exit 1
fi
```

Il risultato dell'operazione viene assegnato alla variabile C

il carattere $ si omette perché interessa il risultato
Utilizzarlo provocherebbe un'errore

Comandi true e false

- Per la realizzazione di cicli infiniti sono disponibili i comandi true e false che terminano sempre con exit status di `0` e `1` rispettivamente
- Lo script di esempio esegue il comando who ogni 10 secondi

```
#!/bin/bash
while /bin/true ; do
   who
   # attendi 10 secondi
   sleep 10
   # pulisci lo schermo
   clear
done
```

continue

- L'esecuzione di un'iterazione di cicli for, while ed until si può interrompere con il comando continue
- Il comando termina l'iterazione corrente senza eseguire i comandi che seguono e passa all'iterazione successiva
- Lo script dell'esempio compila un file .c solo se è più nuovo del programma main

```
#!/bin/bash
for file in *.c ; do
   if [ $file -ot main ] ; then
      continue
   fi
   cc -c $file
done
cc -o main *.o
```

break

- L'esecuzione dei comandi contenuti in un ciclo iterativo (for,while,until) può essere interrotta con il comando break
- Utilizzando tale comando il ciclo termina e l'esecuzione dello script continua dalla prima istruzione che segue il costrutto
- Lo script ha lo stesso effetto del precedente ma è più efficiente

```
#!/bin/bash
for file in $(ls -t *.c) ; do
  if [ $file -ot main ] ; then
    break
  fi
  cc -c $file
done
cc -o main *.o
```

Bibliografia

- Programmazione di Shell
  - http://www.pluto.it/files/ildp/guide/abs/index.html