

SISTEMI OPERATIVI

COMANDI

#!/bin/bash	Commento iniziale di un eseguibile bash, indica quale interprete di comandi deve interpretare lo script
pwd	Ottenere la directory corrente
cd	Navigare nelle directory
echo	Serve per stampare a video (si può usare con doppi apici: echo "testo")
echo pippo \; cd /tmp	\ fa in modo che il carattere ; venga letto come carattere e non come separatore di comandi, output: “pippo ; cd /tmp” ci permette di non usare i doppi apici per delimitare la frase
 \${PATH}	Esempio di passaggio di una variabile “PATH”
chown nuovoproprietario nomefile	Comando per modificare il proprietario di un file
chgrp nuovogruppo nomefile	Comando per modificare il gruppo di un file
chmod 764 ./miofile.txt	Esempio di modifica permessi di un file
ls	Mostra il contenuto della directory (opz: -al per i permessi)
./miofile.sh	Esempio di esecuzione di un file creando una subshell (le variabili locali vengono distrutte al termine)
source nomescript oppure . nomescript	Esecuzione di uno script senza creare la subshell (le variabili di ambiente restano alla fine dello script pk siamo sempre nella stessa shell)
env	Elenco variabili di ambiente
export nomevariabile	Rendo una variabile già creata, variabile di ambiente (NB posso anche crearla direttamente così e assegnare un valore)
unset nomevariabile	Elimina completamente una variabile esistente, vuota o no
history	Visualizza un elenco numerato dei comandi precedentemente lanciati dalla shell
!NUMERO	Lancia il corrispondente comando dall'elenco numerato di history
!stringa	Esegue il comando più recentemente lanciato che nella history inizia con “stringa”
set	Senza parametri visualizza le variabili della shell, sia locali che di ambiente.
set +o history	Disabilita la memorizzazione di ulteriori comandi nel file history, il suo opposto è: set -o history , che la ripristina Le <u>variabili</u> create o modificate vengono fatte diventare immediatamente variabili di ambiente e vengono ereditate dalle eventuali shell figlie, il suo opposto è set +a ovvero il modo di default
set -a	Shell figlia lanciata con argomenti: -c percorso file da eseguire
Shell non interattiva	Shell figlia lanciata con argomenti: -c percorso file da eseguire
Shell interattiva NON di login	Shell che vediamo all'inizio nella finestra di terminale lanciata senza nessuno degli argomenti -c -l --login
Shell interattiva di login	È come la shell non di login, ma inizia chiedendo user e password lanciata con argomenti -l oppure --login
mkdir nomecartella	Crea una directory nella posizione logica attuale
rmdir nomecartella	rimuove la directory specificata (solo se vuota)

rm nome	Rimuove la directory o i file specificati. Vedi il relativo manuale. -f -r -d
cat	Molteplici usi, tra i quali leggere file, aggiungere e sovrascriverli
mv file nuovopercorso	Sposta il file specificato nel nuovo percorso
ps oppure top	Mostra i processi attivi (opz: -aux)
sudo comando	Esegue il comando con i privilegi di un altro utente o come amministratore di sistema
du	Visualizza lo spazio occupato su disco dei file e directory (opz: -h -a --exclude="*.txt")
kill PID	Uccide il processo corrispondente al PID specificato. Con -9 -1 è possibile inviare un segnale di terminazione a tutti i processi che il processo bash corrente può far terminare (figli diretti, non nipoti). Vedi altri codici con -L
<i>(Comando) &</i>	Lancia il comando/processo in background
bg PID oppure %indice	Senza uno specifico PID lancia in background l'ultimo processo interrotto, si può specificare anche l'indice %indice
fg PID oppure %indice	Senza uno specifico PID lancia in foreground l'ultimo processo interrotto, si può specificare anche l'indice %indice
jobs	Mostra i processi figli di quella shell, attivi (anche quelli interrotti o in background)
read nomevariabile	Legge input da standard input e lo inserisce nella variabile specificata (simile a echo)
wc percorsofile	Conta le righe (-l), i caratteri (-m) di un file ecc... vedi man
nano oppure pico	Editor interattivo
head percorso_file	Output la prima parte del file, di default 10 righe, altrimenti quelle specificate con -n numero
tail percorso_file	Output l'ultima parte del file, di default 10 righe, altrimenti quelle specificate con -n numero
sed	Molteplici comandi, vedi coreutils
cut [options] [file]	Vedi foto
grep	<i>Comando grep testo</i> filtra l'output del comando
tee [options] [files]	Usata normalmente dopo una pipe , scrive su un file i dati letti, con -a si evita che il file venga sovrascritto. Si può fare contemporaneamente su più files. Si può combinare al comando sudo per avere permessi di root per scrivere certi file.
disown -[ar] jobs	Sgancia il job dalla shell interattiva che lo ha lanciato, i jobs sono specificati con PID o %indice . NB il job corrente si può indicare con %+ , il precedente con %- -r sgancia tutti i jobs running -a sgancia tutti i jobs running e sospesi
\$!	PID dell'ultimo processo lanciato in background NB se metto in background con bg un processo che era stato sospeso con CTRL+Z, la variabile \$! <u>Non</u> viene modificata con il PID di quel processo.

nohup comando arg1 arg2 argN &	Lancia un comando in background già sganciato dalla shell
trap action lista_di segnali	Definisce l'azione da svolgere al ricevimento di un elenco di segnali, esplicitando il nome di una funzione oppure una stringa di codice bash da eseguire.
wait PID oppure \${pid1} \${pid2} o vuoto	Aspetta che il/i processo/i indicato/i dal PID termini. Può essere invocato solo dalla shell che ha lanciato il processo di cui vogliamo attendere la terminazione. Restituisce come exit status l'exit status restituito dal processo figlio specificato dal PID.
tar -xvf Percorso_file	Serve per estrarre i file dall'archivio LOCALE .tgz tramite una shell bash
find nome_file	Serve per cercare dei files, vedi esempi
more percorso_file	Mostra il file specificato un poco alla volta
which nome_file_eseguibile	Visualizza il percorso in cui si trova (solo se nella PATH) l'eseguibile
du percorso_directory	Visualizza l'occupazione del disco
df	Mostra lo spazio libero dei filesystem montati

Cut

Utilizzo base

`cut OPTION [FILE]`

Alcune delle opzioni più comuni sono:

```
-c, --characters=LIST      Seleziona solo questi caratteri
-d, --delimiter=DELIM     usa DELIM invece che TAB come delimitatore dei campi
-f, --fields=LIST          seleziona solo questi campi; inoltre stampa qualsiasi riga che non contiene alcun carattere delimitatore, a meno che l'opzione -s sia specificata
-s, --only-delimited       non stampare le linee che non contengono delimitatori
```

Avendo il file test.txt che contiene

```
12345:Administration:james:smith
67891:Staff:stephan:york
18230:Staff:luke:cutwine
62913:Sales:red:blues
```

Stampare solo i caratteri da 1 a 5 di ogni linea:

```
> cut -c1-5 test.txt
12345
67891
18230
62913
```

Questo può essere utile quando abbiamo campi a larghezza fissa, ma se vogliamo invece stampare la terza colonna (utilizzando come separatore il carattere :), possiamo utilizzare

```
> cut -d: -f3 test.txt
james
stephan
luke
red
```

Capita spesso che si voglia utilizzare lo spazio come delimitatore. Per farlo bisogna metterlo tra apici singoli, come: -d' '.

Infine come ultimo esempio stampiamo il contenuto del file senza la colonna numero due, per far questo utilizzeremo l'opzione --complement che esclude la colonna indicata, quindi:

```
> cut -d: -f2 --complement test.txt
12345:james:smith
67891:stephan:york
18230:luke:cutwine
62913:red:blues
```

Per stampare (o escludere) più colonne bisogna elencarle separate da una virgola nell'opzione -f.

```
> cut -d: -f2,4 test.txt
Administration:smith
Staff:york
Staff:cutwine
Sales:blues
```

cat

Diamo un'occhiata alle opzioni più importanti del comando cat di Linux.

Opzione	Spiegazione
-h, --help	Visualizza l'aiuto del comando cat di Linux
-n	Numera le righe dell'output
-s	Combina più righe vuote in una sola
-b	Numera tutte le righe dell'output tranne le righe vuote
-v	Emette caratteri invisibili
-e	Come -v, incluso il marcatore di fine riga
-t	Come -v, incluso il marcatore di tabulazione
-et	Combinazione di -e e -t; emette tutti i caratteri invisibili

Reindirizzamento	Simbolo	Impiego	Spiegazione
Pipe		cmd1 cmd2	Inoltro dell'output del comando cmd1 all'input del comando cmd2
Input-Redirect	<	cmd < data	Legge l'input al comando cmd proveniente dal file data
Output-Redirect	>	cmd > data	Scrive l'output del comando cmd nel file data; nell'eventualità, il file esistente viene sovrascritto
Output-Redirect	>>	cmd >> data	Scrive l'output del comando cmd nel file data nell'eventualità, il file esistente viene esteso

CHMOD

user			group			others		
R 4	W 2	X 1	R 4	W 2	X 1	R 4	W 2	X 1

Special Permissions

Permessi speciali di file e directory

setuid setgid sticky

4 2 1

setuid - rappresentato da "s" (o da "S") nelle user permissions (settato s con chmod 4***)
File: in esecuzione, il processo associato all'esecuzione del file ottiene anche i diritti dell'owner (l'effective uid diventa quello dell'owner del file).

Tipicamente root. Esempio del comando /usr/bin/passwd

Directory: ignorato

esempio di settaggio setuid per proprietario: chmod u+s ./miofile

altro esempio di settaggio numerico di setuid e altri permessi: il 4 all'inizio e' setuid

vic@vic:~\$ chmod 4761 ./main.exe

vic@vic:~\$ ls -alh main.exe

-rwsrwxr-x 1 vic vic 8,5K dic 17 2015 main.exe

setgid - rappresentato da "s" (o da "S") nelle group permissions (settato con chmod 2***)

File: analogo al setuid ma per il gruppo (è l'effective gid che diventa quello del file)

Directory: implica che i nuovi file e subdirectory create all'interno della directory ereditino il gid della directory stessa (e non quello del gruppo principale dell'utente che lo ha creato). Esempio di una directory condivisa

sticky bit - rappresentato da "t" (o da "T") (settato con chmod 1***)

File: ora ignorato

Directory: i file all'interno di una directory con sticky bit possono essere rinominati o cancellati solo dal proprietario del file, dal proprietario della directory...o da root, ovviamente!

48

Esempi di cambio permessi speciali (setuid)

vic@vic:~\$ ls -alh main.exe

-rwxrwxr-x 1 vic vic 8,5K dic 17 2015 main.exe

vic@vic:~\$ chmod u+s ./main.exe

vic@vic:~\$ ls -alh main.exe

-rwsrwxr-x 1 vic vic 8,5K dic 17 2015 main.exe

vic@vic:~\$ chmod u-s ./main.exe

vic@vic:~\$ ls -alh main.exe

-rwxrwxr-x 1 vic vic 8,5K dic 17 2015 main.exe

NOTARE LE DIFFERENZE TRA s ed S

s setuid e permesso di esecuzione valore 4

S setuid SENZA permesso di esecuzione. Controsenso.

INCONISISTENZA. Non posso eseguire il file.

vic@vic:~\$ chmod 4666 ./main.exe

NB: NON ho dato permessi esecuzione a owner

vic@vic:~\$ ls -alh main.exe

-rwsrw-rw- 1 vic vic 8,5K dic 17 2015 main.exe

NOTARE QUI SOTTO CHE NON POSSO ASSEGNAME S

La visualizzazione di S, è solo una conseguenza dell'inconsistenza tra x ed s

vic@vic:~\$ chmod u+S ./main.exe PROVOCARE ERRORE

chmod: invalid mode: 'u+S'

Try 'chmod --help' for more information.

ANALOGA DIFFERENZA SUSSISTE TRA "t" e "T" per quanto riguarda lo sticky bit

49

NB set -a

Dopo avere eseguito in una bash il comando **set -a**

le variabili create successivamente sono variabili di ambiente e non variabili locali.

Per configurare comunque una variabile affinché sia una variabile locale (non di ambiente) occorre utilizzare il comando **export** con il flag **-n**

Ad esempio, lanciando il seguente script var_callerLocale.sh, in cui la variabile PLUTO viene dichiarata locale mediante il comando export -n PLUTO ottengo che la variabile PLUTO non esista nell'ambiente di esecuzione dello script var_calledLocale.sh chiamato dal primo script. La variabile PLUTO non era di ambiente.

var_callerLocale.sh

```
#!/bin/bash
set -a
PIPPO=pippo
PLUTO=pluto
echo "PIPPO = ${PIPPO}"
echo "PLUTO = ${PLUTO}"
export -n PLUTO
./var_calledLocal.sh
```

var_calledLocale.sh

```
#!/bin/bash
echo dentro
var_calledLocal.sh
echo "PIPPO = ${PIPPO}"
echo "PLUTO = ${PLUTO}"
```

L'output dell'esecuzione è quello qui a destra:

PIPPO = pippo
PLUTO = pluto
PIPPO = pippo
PLUTO = 64

Esempi echo

NUM=MERDA	definisco una variabile di nome NUM e valore MERDA
echo \${NUM}	stampo a video la variabile NUM, si vedrà MERDA
echo \${NUM}X	stampo a video la variabile NUM, seguita dal carattere X si vedrà MERDAX
echo \$NUM	stampo a video la variabile NUM si vedrà MERDA
echo \$NUMX	vorrei stampare a video la variabile NUM, ma non metto le parentesi graffe, così la shell non capisce dove finisce il nome della variabile e non sostituisce il valore al nome. <u>Non viene visualizzato nulla</u>
echo \$NUM X	come prima, ma ora c'è uno spazio tra NUM e il carattere V così la shell capisce che il nome della variabile finisce dove comincia lo spazio e sostituisce il valore al nome. Viene visualizzato MERDA X

CARATTERI SPECIALI

> >> <	Redirezione I/O
	Pipe
* ? [...]	Wildcards
`command`	Comand substitution
;	Esecuzione sequenziale
&&	Esecuzione condizionale
(...)	Raggruppamento comandi
&	Esecuzione in background
“ ” ‘ ’	Quoting
#	Commento (tranne un caso speciale)
\$	Espansione di variabile
\	Carattere di escape
<<	“here document”
<<<	“here string”

ESPANSIONI 1

Le espansioni principali, elencate in ordine di effettuazione, sono:

• history expansion	!123
• brace expansion	a{damn,czk,bubu}
• tilde expansion	~/nomedirectory
• parameter and variable expansion	\$1 \$? \$! \${var}
• arithmetic expansion	\$(())
• command substitution (effettuata da sinistra verso destra)	\$()
• word splitting	* ? [...]
• pathname expansion	
• quote removal	rimuove unquoted ' " non generate dalle precedenti espansioni

(In alcuni sistemi sono possibili process substitution).

Brace expansion: **echo a{bb,cc,ddd}**

Diventa: abb acc addd

NB all'interno della brace expansion non ci devono essere spazi, nel caso bisogna farli precedere dal carattere \ o delimitare la porzione con i doppi apici, esempio
a{bb,cc,"d d"}ee

Brace expansion: **echo a{b..k}m**

abm acm adm aem afm agm ahm aim ajm akm

Brace expansion: **echo a{4..7}m**

a4m a5m a6m a7m

echo \${A}\${B}\${C},\${C},\${A}\${B}}a

Brace expansion e variable expansion

WILDCARDS ? * [...]

Con cosa sono sostituiti?

* puo' essere sostituito da una qualunque sequenza di caratteri, anche vuota.

? puo' essere sostituito da esattamente un singolo carattere.

[elenco] puo' essere sostituito da un solo carattere tra quelli specificati in elenco.

Wildcards [...]

Cosa posso mettere dentro le parentesi quadre? E con cosa viene sostituito ?
[abk] puo' essere sostituito da un solo carattere tra a b oppure k.

[1-7] puo' essere sostituito da un solo carattere tra 1 2 3 4 5 6 oppure 7.

[c-f] puo' essere sostituito da un solo carattere tra c d e oppure f.

[[:digit:]] puo' essere sostituito da un solo carattere numerico (una cifra).

[[:upper:]] puo' essere sostituito da un solo carattere maiuscolo.

[[:lower:]] puo' essere sostituito da un solo carattere minuscolo.

Notare che le parentesi quadre selezionano uno solo tra i caratteri elencati dentro.

Nell'elenco possono comparire diverse sequenze di parentesi quadre.

Le sequenze speciali [:digit:] [:upper:] [:lower:] **devono stare dentro altre parentesi quadre esterne.**

Es: Supponiamo che in una directory ci siano i file: aB a1B a2B akB akmB akmtB
Allora:

```
ls a[[:digit:]]B  
visualizza a1B a2B
```

```
ls a[[:lower:]][[[:lower:]][[[:lower:]]]B  
visualizza akmtB
```

Notare la differenza tra i seguenti comandi (occhio alle parentesi quadre annidate)

ls a[[:lower:]][[[:lower:]]B	[] []
visualizza akmB	
ls a[[:lower:]][[[:lower:]]B	[] []
visualizza akB	

ESPANSIONI 2

Parametri a riga di comando passati al programma (4)

Parameter Expansion

Come utilizzare in uno script gli argomenti a riga di comando passati allo script

Esistono variabili d'ambiente che contengono gli argomenti passati allo script

\$# il numero di argomenti passati allo script
\$0 il nome del processo in esecuzione
\$1 primo argomento, \$2 secondo argomento,
\$* tutti gli argomenti passati a riga di comando concatenati e separati da spazi
\$@ come \$* ma se quotato gli argomenti vengono quotati separatamente

Esempio: All'interno di uno script posso usarle così:

```
file esempio_script.sh
echo "ho passato $# argomenti alla shell"
echo "tutti gli argomenti sono $*"
```

NOTA BENE: I parametri NON POSSONO essere modificati

NOTA BENE: Il programma vede i parametri COSÌ SONO DIVENTATI DOPO LA EVENTUALE SOSTITUZIONE DEI METACARATTERI * e ?

Ad esempio, se nella directory corrente ci sono i seguenti file x.c, y.c e z.c, ed io lancio lo script, che contiene le 2 righe sopra riportate, in due modi diversi, vengono stampati in output i seguenti argomenti:

SENZA metacaratteri

```
/esempio_script.sh pippo
ho passato 1 argomenti alla shell
tutti gli argomenti sono pippo
```

CON metacaratteri

```
/esempio_script.sh *.c
ho passato 3 argomenti alla shell
tutti gli argomenti sono x.c y.c z.c
```

83

Parametri a riga di comando passati al programma (5)

File esempio_args.sh

```
#!/bin/bash
echo "ho passato $# argomenti alla shell"
echo "il nome del processo in esecuzione e' $0"
echo "gli argomenti passati a riga di comando sono $*"
```

```
for name in $* ; do
    echo "argomento e' ${name}";
done
```

eseguitelo chiamandolo con diversi argomenti e delimitatori

```
/esempio_args.sh
./esempio_args.sh alfa beta gamma
./esempio_args.sh "alfa beta gamma"
./esempio_args.sh "alfa beta" gamma
```

NB: la variabile name dichiarata nel for rimane utilizzabile anche dopo l'uscita dal for poiché il for viene eseguito nella bash stessa, non in una bash figlia.

La variabile name fuori dal for avrà l'ultimo valore che le era stato assegnato

84

Parametri a riga di comando passati al programma (6)

Differenze tra \$* e \$@

I parametri \$* e \$@ sono uguali quando non quotati dai ", cioè sono la concatenazione separata da blank dei singoli argomenti.

```
$* == $@ --> $1 $2 $3 ... $n
```

I parametri \$* e \$@ si comportano diversamente quando sono quotati con " in particolare:

```
"$*" --> "$1 $2 $3 ... $n"      quotati tutti gli argomenti assieme
"$@" --> "$1" "$2" "$3" ... "$n"  quotato singolarmente ogni argomento
```

Parametri a riga di comando passati al programma (6bis)

Il parametro \$@ e' simile a \$* cioè contiene tutti gli argomenti passati allo script, ma quando quotato con i " mantiene quotati i singoli argomenti e permette di non spezzare gli argomenti che contengono degli spazi.

File esempio_\\$@.sh.

Eseguitelo con diversi argomenti e delimitatori

```
esempio_\$@.sh alfa beta gamma
esempio_\$@.sh "alfa beta" gamma
```

```
echo 'for con $* non quotato'
for name in $* ; do
    echo "argomento ${name}";
done
echo 'for con $$ quotato'
for name in $$* ; do
    echo "argomento ${name}";
done
echo 'for con $@ non quotato'
for name in $@ ; do
    echo "argomento ${name}";
done
echo 'for con $$@ quotato'
for name in $$@ ; do
    echo "argomento ${name}";
done
```

NON SOLO PER I MANIACI

NB: \$@ si usa anche quando uno script deve eseguire un altro comando passandogli tutti gli argomenti che ha ricevuto a riga di comando

85

ESEMPI FIND

find /usr/

cerca tutte le directory dallo user in giù (quelle che da qualche parte nel loro percorso abbiano usr)

find /usr/ -type d o f

Filtrà solo le directory (**d**) o i file (**f**)

find /usr/ -type d -iname "std"

solo le directory nel cui nome ci sia std (sia maiuscolo che minuscolo (la i di iname dice di ignorare il case))

find /usr/ -type d -name "std"

solo le directory nel cui nome ci sia std (case sensitive)

find /usr/ -maxdepth 2

solo entro i primi 2 sottolivelli di usr

find /usr/ -mindepth 4

solo dal 4 sottolivello di usr

find /usr/ -exec

comando per ognuno dei file che trova esegue il comando exec

es: **find /usr/ -exec head -n 1 '{}'; ('{}'; → serve per passare il nome del file che ha trovato) trova tutti i file a partire da usr e di ognuno mostra la prima riga a partire dalla testa**

Arithmetic expansion

Valutazione Aritmetica di espressioni con soli interi (1):

operatori `(())` e `$(())`

E' possibile valutare una stringa come se fosse una espressione costituita da operazioni aritmetiche tra soli numeri interi

L'operatore `(())` racchiude TUTTA una riga di comando semplice, che deve essere una espressione (più un eventuale assegnamento).

L'operatore `(())` esegue la riga di comando che racchiude valutando aritmeticamente gli operandi.

Ad esempio

```
(( NUM=3+2 ))
```

assegna 5 alla variabile NUM

L'operatore `$(())` invece serve se dovete racchiudere SOLO UNA PARTE di una riga di comando, che deve essere una espressione (più un eventuale assegnamento).

La bash

- prima valuta aritmeticamente l'espressione racchiusa dall'operatore `$(())`

- poi, nella riga di comando, mette al posto del `$((...))` il risultato calcolato dall'operatore `$(())`

- infine, esegue la riga di comando modificata

Ad esempio, nella riga di comando

```
echo stampa pippo$((3+2))
```

la bash

prima valuta l'espressione `$((3+2))` e ne calcola il risultato che è 5

poi nella riga di comando originale sostituisce 5 al posto di `$((3+2))`

ottenendo la nuova riga di comando

```
echo stampa pippo5
```

infine esegue quest'ultima riga di comando visualizzando a video

```
stampo pippo5
```

Valutazione Aritmetica di espressioni con soli interi (2):

operatori `(())` e `$(())`

Errori sintattici nell'espressione provocano errori.

```
NUM=1  
NUM=$((NUM)+3  
echo  
$NUM  
stampa a video la stringa 1+3
```

NUM=1

```
((NUM=$((NUM)+3))
```

echo \${NAMEFILE}

uso corretto di operatore (())

stampa a video la stringa 4

NUM=1

```
(( $(NUM)+3))
```

echo \${NAMEFILE}

uso SBAGLIATO di operatore (())

stampa a video

```
"syntax error near unexpected token ("
```

```
NUM=1  
NAMEFILE=pioppo${NUM}+3  
echo ${NAMEFILE}  
stampa a video la stringa pioppo1+3
```

NUM=1

```
(( NAMEFILE=pioppo${NUM}+3 ))
```

echo \${NAMEFILE}

uso sbagliato di operatore (())

impossibile valutare aritmeticamente pioppo

stampa a video la stringa 3

NUM=1

```
NAMEFILE=pioppo$(( $(NUM)+3 ))
```

echo \${NAMEFILE}

uso corretto di operatore \$(())

stampa a video la stringa pioppo4

87

88

RIFERIMENTI INDIRETTI

Riferimenti Indiretti a Variabili (1) Indirect References to Variables operatore `${!}` solo in bash versione 2

Supponiamo di avere una prima variabile varA che contiene un valore qualunque.
Supponiamo di avere una altra variabile il cui valore è proprio il nome della prima variabile.

Voglio usare il valore della prima variabile sfruttando solo il nome della seconda variabile il cui valore è proprio il nome della prima variabile. Si dice che la seconda variabile è un riferimento indiretto alla prima variabile.

Accedere al valore di una prima variabile il cui nome è il valore di una seconda variabile è possibile nella bash a partire dalla versione 2.

Si sfrutta un operatore `${!}`

Esempio:

```
varA=pioppo  
nomevar=varA  
echo ${!nomevar} stampa a video pioppo
```

94

Riferimenti Indiretti a Variabili (2) operatore `${!}` solo in bash versione 2

File `esempio_while.sh` permette di riferirsi alle var \$1 \$2 \$3
da provare chiamandolo con diversi argomenti e delimitatori
esempio_while.sh
esempio_while.sh alfa beta gamma
esempio_while.sh "alfa beta gamma"
esempio_while.sh "alfa beta" gamma

```
#!/bin/bash  
echo "ho passato $# argomenti alla shell"  
echo "il nome del processo in esecuzione e' $0"  
echo "gli argomenti passati a riga di comando sono $*"
```

```
NUM=1  
while (( "$NUM" <= "$#" ))  
do  
    # notare il ! davanti al NUM  
    echo "arg ${NUM} is ${!NUM}"  
    ((NUM=$((NUM)+1))  
done
```

95

EXIT STATUS

Exit Status

Valore restituito da un programma al chiamante

Differenza tra valore restituito e output

Ogni programma o comando restituisce un valore numerico compreso tra **0 e 255** per indicare se c'è stato un errore durante l'esecuzione oppure se tutto è andato bene. Un risultato **0 indica tutto bene** mentre un risultato **diverso da zero indica errore**.

Tale risultato non viene visualizzato sullo schermo bensì viene passato alla shell che ha chiamato l'esecuzione del programma stesso. In tal modo il chiamante può controllare in modo automatizzato il buon andamento dell'esecuzione dei programmi.

Il risultato non è l'output fatto a video, che invece serve per far vedere all'utente delle informazioni.

Come restituire il risultato in un programma C – l'istruzione return;

vedi esempio primo.c

Come restituire il risultato in uno script bash – il comando exit
esempio: **exit 9** fa terminare lo script e restituisce 9 come risultato

Come catturare il risultato di un programma chiamato da uno script

Si usa una variabile d'ambiente predefinita **\$?** che viene modificata ogni volta che un programma o un comando termina e in cui viene messo il risultato numerico restituito dal comando o programma

```
./primo.exe
echo "il processo chiamato ha restituito come valore di uscita $?"
```

96

Exit Status riservati

Per convenzione, gli eseguibili restituiscono un valore intero compreso tra **0 e 255** per indicare se è accaduto un errore e quale errore è accaduto.

Il valore 0 indica che l'esecuzione si è svolta correttamente.

Exit Code Number	Meaning	Example	Comments
1	catchall for general errors	let "var1 = 1/0"	miscellaneous errors, such as "divide by zero"
2	misuse of shell builtins, according to Bash documentation		Seldom seen, usually defaults to exit code 1
126	command invoked cannot execute		permission problem or command is not an executable
127	"command not found"		possible problem with \$PATH or a typo
128	invalid argument to exit	exit 3.14159	exit takes only integer args in the range 0 – 255
128+n	fatal error signal "n"	kill -9 \$PPID of script	\$? returns 137 (128 + 9)
130	script terminated by Control-C		Control-C is fatal error signal 2, (130 = 128 + 2, see above)
255*	exit status out of range	exit -1	exit takes only integer args in the range 0 – 255

Advanced Bash-Scripting Guide - Mendel Cooper

97

Exit Status di Espressione valutata aritmeticamente

La **valutazione aritmetica** effettuata tramite gli operatori **\$(())**, oppure anche **(())** se comprende tutta la riga di comando, **restituisce un valore di Exit Status che sarà:**

Diverso da zero se durante la valutazione aritmetica è **accaduto un errore**. Il valore restituito indica il tipo di errore restituito.

```
(( ls 5 ))      non valutabile aritmeticamente. $? conterrà 1
```

Uguale a zero se la valutazione aritmetica fornisce un **risultato logico true**
((5 >= 2)) \$? conterrà 0

Diverso da zero se la valutazione aritmetica fornisce un **risultato logico false**
((5 <= 2)) \$? conterrà 1

Uguale a zero se la valutazione aritmetica fornisce un **risultato intero diverso da zero**
((4)) \$? conterrà 0
((VAR=5+3)) \$? conterrà 0

Diverso da zero se la valutazione aritmetica fornisce un risultato intero uguale a zero
((0)) \$? conterrà 1
((VAR=6-2*3)) \$? conterrà 1

Se l'assegnamento è interno all'espressione (cioè non è eseguito per ultimo), allora il risultato dell'espressione è quello del confronto eseguito per ultimo.

```
(( (VAR=6-2*3) != 1 ))      assegna 0 a VAR ma $? conterrà 0
perché la condizione è vera
```

OCCHIO: Errori con la valutazione aritmetica

```
$ NUM=13
$ echo $?
0
$ echo ${NUM}
13
$ (( NUM=${NUM}+4 ))
$ echo $?
0
$ echo ${NUM}
17
$ (( NUM=ALFA ))      <----- causa errore
$ echo $?
1      <----- valore di errore
$ echo ${NUM}
0      <----- contenuto errato
```

99

Il risultato Exit Status restituito da una lista di comandi è l'Exit Status restituito dall'ultimo comando che è stato lanciato dalla lista di comandi stessa.

NB: poiché le sequenze condizionale e le espressioni condizionali contengono dei comandi semplici e possono essere terminate senza avere eseguito tutti i comandi, in funzione dei risultati restituiti dai comandi eseguiti in precedenza, può capitare che l'ultimo comando eseguito non sia l'ultimo a destra elencato nella lista.

Sequenze di comandi e comandi composti

Compound Commands – Comandi composti Costrutti per controllo di flusso di comandi (1)

Versione semplificata

```
for varname in elencoword ; do list ; done
for (( expr1 ; expr2 ; expr3 )) ; do list ; done
if listA ; then listB ; [ elif listC ; then listD ; ] ... [ else listZ ; ] fi
while list; do list; done
```

NOTA BENE:

Le espressioni expr* dentro il for (()) sono **sempre** valutate aritmeticamente, quindi devono essere espressioni interpretabili aritmeticamente.
Invece, i comandi dentro list* sono valutati aritmeticamente solo se racchiudo i singoli comandi dentro doppie parentesi tonde.

Il risultato restituito da list* e' il risultato dell'ultimo comando eseguito della lista oppure zero se nessun comando viene eseguito.

101

Compound Commands – Comandi composti Costrutti per controllo di flusso di comandi (2)

Esempi

```
for name in a b c ; do echo ${name} ; rm ${name} ; done
for (( i=0; $i<13; i=$i+2 )) ; do echo ciao$i ; done
if ls ./main.c && gcc -o main.exe main.c && ./main.exe ; then
    echo main returns $?
fi
if [[ -e main.c && -e Makefile ]] && make && ./main.exe ; then
    echo compiled program returns $?
fi
while OUT=`./script1.sh` ; script2.sh $OUT ; do echo ancora done
i=0; while (( $i < 30 )) ; do echo $i ; (( i=$i+1 )) ; done
```

102

Sequenze di comandi condizionali e non

Sequenze non condizionali (vedi una slide precedente)

Il metacarattere ";" viene utilizzato per eseguire due o più comandi in sequenza ed indica la fine degli argomenti passati a ciascun comando riga di comando
date ; ls /usr/vittorio/ ; pwd

Sequenze condizionali

"||" viene utilizzato per eseguire due comandi in sequenza, ma il secondo viene eseguito solo se il primo termina con un exit code **diverso da 0 (failure)**

"&&" viene utilizzato per eseguire due comandi in sequenza, ma il secondo viene eseguito solo se il primo termina con un exit code **uguale a 0 (success)**

Esempi

Eseguire il secondo comando in caso di successo del primo
\$ gcc prog.c -o prog && prog

Eseguire il secondo comando in caso di fallimento del primo
\$ gcc prog.c || echo Compilazione fallita

103

STREAM I/O PREDEFINITI

Stream di I/O predefiniti dei processi (2): command substitution

Scopo: sostituire a run time, in uno script, la riga di comando di un programma con l'output su stdout prodotto dall'esecuzione del programma stesso.

Esempio: come catturare, in una variabile, l'output di un programma chiamato da uno script. (fa eseguire primo.exe)

```
OUT= `./primo.exe`  
echo "l'output del processo e' ${OUT}"
```

Nota Bene: L'apice giusto da usare è ` quello che in alto tende a sinistra e in basso tende a destra. Viene chiamato backticks o backquotes.

Nelle tastiere americane si trova nel primo tasto in alto a sinistra, accoppiato e sotto alla tilde. Non è l'apostrofo italiano ' (detto single quote) , il tasto del backquotes nelle tastiere italiane non esiste. Usare tastiera americana.

Sintassi alternativa per la command substitution : \$(rigadicomando)

esempio: OUT= \$(./primo.exe) *NB: sono parentesi TONDE*

In entrambe le sintassi, le eventuali wildcards ? * [] vengono interpretate dalla shell e sostituite. Per disabilitare (escape) la sostituzione delle wildcards, si circonda la riga di comando con dei caratteri opportuni (quotes).

111

Stream di I/O predefiniti dei processi (2): command substitution (2)

supponiamo

1. Che nella home directory dell'utente esista un file di nome prova-6.txt contenente una sola riga di testo "asdrubale non ragiona"
2. Che ci troviamo in una directory diversa dalla home directory.
3. Che in quella directory esista uno script esempio_command_substitution.sh contenente quello che segue

```
NUM=5  
echo "output dello script: " ` cat ~/${NUM}(( ${NUM} +1 )) .txt | wc -l`
```

4. Infine supponiamo di eseguire quello script invocandolo con i seguenti argomenti:

```
./esempio_command_substitution.sh prova 5
```

L'output dello script sarà il seguente
output dello script: 1

Ciò dimostra che la bash, prima sostituisce in quest'ordine tilde, metacaratteri, variabili, e poi esegue la command substitution

112

Quoting di stringhe: command substitution & escape

In una riga di comando, le eventuali wildcards ? * [] variabili etc etc vengono interpretate dalla shell e sostituite. Per disabilitare (escape) la sostituzione delle wildcards, si circonda la riga di comando con dei caratteri opportuni (quotes).

Esempio di command substitution e sostituzione di wildcards e variabili

```
echo Dear ${USER} the files are a[[:digit:]]B and the date is: `date`  
Dear vittorio the files are a1B a2B and the date is: Tue, Sep 15, 2015 12:34:52 PM
```

" " Double quote per escape wildcards e spazi (quoting parziale di stringhe)

Impedisce di usare il ; come separatore di comandi,

Impedisce la sostituzione di wildcards

ma permette di sostituire le variabili con il loro contenuto

e permette l'esecuzione di comandi (command substitution).

```
echo " Dear ${USER} the files are a[[:digit:]]B and the date is: `date` "
```

```
Dear vittorio the files are a1B a2B and the date is: Tue, Sep 15, 2015 12:35:44 PM
```

' ' Single quote escape wildcards, command substitution, variable substitution, spazi

Impedisce di usare il ; come separatore di comandi,

Impedisce la sostituzione di wildcards

e Impedisce di sostituire le variabili con il loro contenuto

e Impedisce l'esecuzione di comandi (command substitution).

```
echo ' Dear ${USER} the files are a[[:digit:]]B and the date is: `date` '
```

```
Dear ${USER} the files are a1B a2B and the date is: `date`
```

113

ATTENZIONE: la bash processa l'output prodotto dalla command substitution (1)

supponiamo che nella directory corrente esistano solo i due script seguenti, outputconasterisco.sh e chiama_outputconasterisco.sh.

```
outputconasterisco.sh
```

```
echo **
```

```
chiama_outputconasterisco.sh
```

```
./outputconasterisco.sh
```

```
echo "riga di separazione"
```

```
echo `./outputconasterisco.sh`
```

Eseguendo lo script ./chiama_outputconasterisco.sh ottengo il seguente output

```
*
```

```
riga di separazione
```

```
chiama_outputconasterisco.sh outputconasterisco.sh
```

Notare che nel comando contenente echo `./outputconasterisco.sh` l'asterisco buttato sullo standard output è stato interpretato dalla bash e sostituito con i nomi dei file presenti nella directory corrente.

115

ESPRESSIONI CONDIZIONALI

Espressioni CONDIZIONALI (1)

- Le espressioni condizionali sono dei particolari comandi che valutano alcune condizioni e restituiscono un exit status di valore 0 per indicare la verità dell'espressione o di valore diverso da zero per indicarne la falsità.
 - Ciascuna espressione condizionale si scrive mettendo le condizioni da valutare tra doppie parentesi quadre [[...]]
 - Le condizioni che possono essere inserite in una espressione condizionale sono condizioni create appositamente, e **NON POSSONO ESSERE COMANDI**.
 - Le condizioni all'interno delle [[]] possono essere composte tra loro mediante degli operatori logici ! (not), && (and), || (or) e mediante parentesi tonde per stabilire l'ordine di valutazione. ad esempio [[((condA)&&(condB))|| condC]]
 - All'interno delle espressioni condizionali [[]] la bash **effettua solo** le interpretazioni **variable expansion, arithmetic expansion con \$()** ma non ((())), **command substitution**, process substitution, and quote removal, ma solo negli operandi.
 - Sono invece **disabilitate** le interpretazioni **Word splitting, brace expansion e pathname expansion** e **NON POSSONO COMPARIRE DEI COMANDI**
 - Gli operatori di condizione (quelli con il - davanti, ad esempio -e nomefile) per essere riconosciuti e valutati correttamente devono essere **NON quotati**. e non possono essere generati da command substitution.
- Quindi:
- non posso eseguire comandi ma posso eseguire command substitution (i backtick)** ma solo negli operandi, non per generare operatori.
 - le doppie parentesi tonde senza \$ sono interpretate non come valutazione aritmetica ma come accorpamento logico di comandi per definire l'ordine di valutazione.

116

Espressioni CONDIZIONALI (2)

COMPORRE CONDIZIONI DENTRO ESPRESSIONI CONDIZIONALI

Notare che in una stessa espressione condizionale [[]] io posso effettuare sia confronti aritmetici (**specificando opportuni operatori -eq -ne -lt -gt -ge**), ma anche confronti non aritmetici (specificando altri operatori).

Penso infatti verificare **condizioni non aritmetiche che riguardano stringhe**, usando gli **operatori di confronto lessicografico tra stringhe == != < <= > >=**.

Penso anche verificare se delle stringhe sono vuote o no (con gli operatori -z -f -h -r -s -t -w -x -O -G -L).

Penso verificare **condizioni sui file** (operatori -d -e -f -h -r -s -t -w -x -O -G -L).

Penso **confrontare le date di ultima modifica di due files** (operatori -nt -ot).

Quindi, con le espressioni condizionali posso verificare espressioni aritmetiche ma **NON POSSO eseguire assegnamenti a variabili**.

NOTA BENE: NEGLI OPERANDI dei confronti aritmetici possono comparire **valutazioni aritmetiche** effettuate mediante \$(()) ma **NON POSSO USARE VALUTAZIONI ARITMETICHE COME CONDIZIONI ISOLATE** perché vengono fraintese.

SI	[[\$NUM -lt \$((\$VAR * (3 + \$MUL)))]]	
NO	[[\$((\$VAR < 3 + \$MUL))]]	non errore sintattico ma fraintende
NO	[[pippo]]	Fraintende , restituisce exit status 0
NO	[[3]]	Fraintende , restituisce exit status 0
NO	[[0]]	Fraintende , restituisce exit status 0

117

Espressioni CONDIZIONALI (3)

DIFFERENZE TRA ESPRESSIONI CONDIZIONALI E VALUTAZIONI ARITMETICHE

Gli operatori di valutazione aritmetica (()) o \$(()) valutano aritmeticamente tutto il comando che specifico all'interno, quindi , l'operatore di valutazione aritmetica :

- puo' contenere solo espressioni valutabili aritmeticamente
- puo' comporre espressioni aritmetiche mediante operatori logici && (AND) || (OR) e ! (NOT).
- non puo' contenere espressioni condizionali
- non puo' contenere comandi

SI ((\$VAR * (3 + \$MUL) < 4 && 7 > \$VAR))

NO ((\$VAR < 3 + \$MUL && [[\$NUM -lt 5]]) errore sintattico

118

Espressioni CONDIZIONALI (4)

Esempi di espressioni corrette o sbagliate

[[ls /]]	syntax error
[[\$(("11" > "2"))]]	Fraintende , produce exit status 0 (true) sempre
[[(("11" > "2"))]]	Fraintende , produce exit status 1 (false) perché le tonde parentesi non valutano aritmeticamente e la valutazione è lessicografica (primo 1 a sinistra precede primo 2 a destra).
[[-e /usr/include/stdio.h]]	OK. Esiste il file /usr/include/stdio.h ? true va bene
[["-e /usr/include/stdio.h"]]	Fraintende , restituisce true ma per sbaglio, infatti...
[["-e /usr/include/stdio"]]	Fraintende , restituisce true ma stdio non esiste
[["-e" /usr/include/stdio.h]]	syntax error

Per effettuare confronti aritmetici, quindi, non posso usare la coppia di doppie parentesi tonde senza \$ ma devo utilizzare gli appositi operatori di confronto aritmetico, che sono -eq, -ne, -lt, -le, -gt, or -ge

NUM=11; [[\${NUM} < 3]]; **Sembra Strano** ma va bene. Exit status 0 (true) perché il confronto è eseguito lessicograficamente tra stringhe.

NUM=11; [[\${NUM} -le 3]]; OK. ha exit status 1 (false) perché confronta aritmeticamente le stringhe.

Non posso annidare espressioni condizionali perché sono anch'esse dei comandi.

[[-e /usr/include/stdio.h && [[-e /usr/include/stdio.h]]] **syntax error**

Non posso usare command substitution per generare operatori

[[`echo -e /usr/include/stdio.h`]]

Fraintende

[[-e `echo /usr/include/stdio.h`]]

OK

119

Espressioni CONDIZIONALI (5) - versioni

enhanced brackets

[[condiz]]

single brackets

[condiz]

test

test condiz

Esistono 3 operatori analoghi per le espressioni condizionali, al cui interno specificare gli operatori per Condizioni di file, Confronto tra stringhe, Confronto aritmetico. L'operatore [[]] è più recente e non esiste nelle bash molto vecchie.

L'operatore [[]] permette di **comporre tra loro piu' condizioni**, utilizzando gli stessi operatori logici usati in C ovvero ! (negazione) && (and) e || (or).

Inoltre, **all'interno del blocco [[]] posso usare parentesi tonde** per raggruppare espressioni e modificare la precedenza degli operatori e **posso andare a capo** proseguendo l'espressione

Diversamente, i due operatori [] e test permettono di comporre tra loro piu' condizioni, utilizzando però **operatori logici diversi**, e precisamente ! (negazione) -a (and) e -o (or).

Inoltre, con questi due operatori, **NON POSSO usare parentesi tonde** per raggruppare espressioni e modificare la precedenza degli operatori e **NON posso andare a capo** se non usando il \ a fine riga .

120

Espressioni CONDIZIONALI (6)

Le espressioni condizionali [[expression]] restituiscono un exit status 0 or 1 in dipendenza della valutazione della condizione esplicitata nella espressione. Tale risultato si ritrova come al solito nella variabile \$?

All'interno delle espressioni condizionali [[]]

- sono **disabilitate** le interpretazioni di tipo **Word splitting** e **pathname expansion**
- sono **abilitate** invece variable expansion, arithmetic expansion (solo quelle con \$(()) ma non con (()) senza \$), command substitution, process substitution, e quote removal.

- Gli operatori di condizione (quelli con il - davanti, ad esempio -e nomefile) per essere riconosciuti e valutati correttamente devono essere **NON quotati** in alcun modo.

Usando l'operatore doppia parentesi quadra [[]] le espressioni condizionali possono essere collegati con gli operatori logici usati anche in C, quali ! && || e raggruppati con parentesi tonde.

Le precedenze degli operatori logici decrescono in questo ordine: ! && ||

NOTA BENE per le prossime pagine:

Se tra gli argomenti degli operatori compare un file allora:

- se per questo argomento viene specificato /dev/fd/n allora l'operatore controlla il file descriptor n.

- se per questo argomento viene specificato uno dei file /dev/stdin /dev/stdout /dev/stderr allora l'operatore controlla il file descriptor di valore 0 1 e 2.

121

Continua sotto...

ESPRESSIONI CONDIZIONALI

Espressioni CONDIZIONALI (7)

Alcuni operatori per verificare **condizioni su file**:

```

-d file      True if file exists and is a directory.
-e file      True if file exists.
-f file      True if file exists and is a regular file.
-h file      True if file exists and is a symbolic link.
-r file      True if file exists and is readable.
-s file      True if file exists and has a size greater than zero.
-t fd        True if file descriptor fd is open and refers to a terminal.
-w file      True if file exists and is writable.
-x file      True if file exists and is executable.
-O file      True if file exists and is owned by the effective user id.
-G file      True if file exists and is owned by the effective group id.
-L file      True if file exists and is a symbolic link. (deprecated, see -h)

file1 -nt file2    True if file1 is newer (according to modification date) than file2,
                   or if file1 exists and file2 does not.

file1 -ot file2    True if file1 is older than file2, or if file2 exists and file1 does not.

```

Nota Bene: Ne esistono altre, per vedere quali guardare man bash nella parte intitolata
CONDITIONAL EXPRESSIONS.

122

Espressioni CONDIZIONALI (8)

Alcuni operatori per verificare **condizioni su stringhe, aritmetiche e altre varie**:

-o optname	True if shell option optname is enabled. See the list of options under the description of the -o option to the set command.
Operatori su stringhe	
-z string	True if the length of string is zero.
-n string	True if the length of string is non-zero.
string1 == string2	True if the strings are equal
string1 != string2	True if the strings are not equal.
string1 < string2	True if string1 sorts before string2 lexicographically.
string1 > string2	True if string1 sorts after string2 lexicographically.

Operatori aritmetici su stringhe

arg1 OP arg2	OP is one of -eq , -ne , -lt , -le , -gt , or -ge . These arithmetic binary operators return true if arg1 is equal to, not equal to, less than, less than or equal to, greater than, or greater than or equal to arg2, respectively. Arg1 and arg2 may be positive or negative integers.
---------------------	--

123

Esempio d'uso di Espressioni CONDIZIONALI (1)

Due porzioni di codice bash che fanno la stessa cosa usando in un caso le espressioni condizionali ed una valutazione aritmetica come condizione dell'if, e nel secondo caso invece usando direttamente l'espressione condizionale come lista di comandi dell'if che ne valuta l'exit status:

```

[[ -e ${name} ]]
if (( $? == 0 )) ; then
  echo "esiste ${name}, lo elimino";
  rm -f ${name}
fi;

```

analogamente

```

if [[ -e ${name} ]] ; then
  echo "esiste ${name}, lo elimino";
  rm -f ${name}
fi;

```

124

Esempio d'uso di Espressioni CONDIZIONALI (1b)

- 1) var="prova"
[[-n \${var}]] ; echo \$?
output 0
- 2) var=""
[[-n \${var}]] ; echo \$?
output 1
- 3) [[-n ""]] ; echo \$?
output 1
- 4) unset var
[[-n \${var}]] ; echo \$?
output
bash: unexpected argument ']' to conditional unary operator
bash: syntax error near ']'0
- 5) [[-n]] ; echo \$?
output
bash: unexpected argument ']' to conditional unary operator
bash: syntax error near ']'0

NOTARE LA DIFFERENZA TRA I CASI 4 e 5*: nel caso 4 la bash capisce che si tratta di espr condizionale, riconosce l'operatore -n e, prima di tentare di fare l'espansione della variabile var, correttamente considera 0 la lunghezza della variabile che non esiste. Nel caso 5, invece, la bash dice che manca l'argomento.

125

Esempio d'uso di Espressioni CONDIZIONALI (2)

Due porzioni di codice bash che fanno la stessa cosa usando due diversi tipi di espressioni condizionali: le single brackets [] e le extended brackets [[]]

```

if [ -d ${name} -a ${#name} -lt 10 ] ; then
  echo "ok ${name}";
else
  echo "no ${name}";
fi;

```

Analogamente

```

if [[ -d ${name} && ${#name} -lt 10 ]] ; then
  echo "ok ${name}";
else
  echo "no ${name}";
fi;

```

Attenzione all'interpretazione di &&
può essere
operatore AND logico in Espressioni CONDIZIONALI
o operatore in SEQUENZE di COMANDI condizionali

QUI && è AND LOGICO

```

if [[ -e ${name} && ${#name} -lt 10 ]] ; then
  echo "ok ${name}";
fi

```

QUI && è operatore in sequenza di comandi condizionale

```

if [[ -e prova.c ]] && gcc -o prova.exe prova.c ; then
  echo "posso eseguire prova.exe"
fi

```

126

127

Word splitting

Quoting '\$stringa'

Usare caratteri non stampabili in una stringa

Parole aventi forma \$'charsequence' sono trattate in modo speciale.
La sequenza charsequence puo' contenere backslash-escaped characters.

- 1) La parola viene espansa in una stringa single-quoted, cioe' **perde il \$ all'inizio ma mantiene i due ' all'inizio e alla fine** (come se \$ non esistesse).
- 2) le backslash-escaped characters, se presenti, sono sostituite come specificato nello standard ANSI C:

\a alert (bell)	\b backspace
\e	\E an escape character
\f form feed	\n new line
\r carriage return	\t horizontal tab
\v vertical tab	\\\ backslash
\' single quote	\\" double quote

\nnn the eight-bit character whose value is the octal value nnn (one to three digits)
\xHH the eight-bit character whose value is the hexadecimal value HH
(one or two hex digits)
\cx a control-x character, as if the dollar sign had not been present.

Sta cosa tornera' utile per specificare il contenuto della variabile IFS poiche' questa contiene anche caratteri non stampabili.

133

Word Splitting: CARATTERI SEPARATORI IN ELENCHI

La variabile IFS contiene i caratteri che fungono da separatori delle parole negli elenchi, IFS=\$' \t\n'

Notare che IFS di default contiene uno spazio bianco, un tab e un newline (a capo).

Se devo lanciare dei comandi in cui devo trattare dei nomi di file che contengono degli spazi bianchi, se non posso fare diversamente devo i) usare degli elenchi separati da newline o tab, e ii) togliere dai separatori lo spazio bianco.

IFS=\$' \t\n'

Poi eseguirò il comando che dovevo lanciare e dopo rimetterò lo IFS come era prima.

Ese: directory che contiene due files, "aa bb.txt" e "aa cc.txt"

Vedere che succede se lancio

```
for name in `ls aa*` ; do echo ${name} ; done
```

Visualizzo

```
aa  
bb.txt  
aa  
cc.txt
```

TRUCCO OSCENO MA FUNZIONA

OLDIFS=\${IFS}

IFS=\$'\t\n'

```
for name in `ls -1 aa*` ; do echo ${name} ; done
```

IFS=\${OLDIFS}

Word Splitting: CARATTERI SEPARATORI IN ELENCHI (2)

Precisazione per TarloI (sconosciuto)
perche', se tolgo spazio da IFS, bash riconosce grammatica?

perché i separatori contenuti in IFS sono utilizzati per individuare in modo specifico le separazioni tra gli elenchi di nomi.

I separatori usati per individuare le separazioni tra parole chiave del linguaggio sono invece non modificabili e sono sempre gli spazi bianchi, i tab e le andate a capo

134

135

Read

read - Lettura da standard input (tastiera o file) (1)

Uno script può leggere dallo standard input delle sequenze di caratteri usando un comando chiamato **read**. Il comando **read** riceve la sequenza di caratteri digitate da tastiera fino alla pressione del tasto INVIO (RETURN) e mette i caratteri ricevuti in una variabile che viene passata come argomento alla **read** stessa. Se invece lo standard input è stato ridiretto da un file, allora la **read** legge una riga di quel file ed una eventuale **read** successiva legge la riga successiva.

La **read** restituisce un risultato che indica se la lettura è andata a buon fine, cioè restituisce:

- 0 se non si arriva a fine file e viene letto qualcosa,
- >0 se si arriva a fine file

```
while true ; do
    read RIGA
    if (( "$?" != 0 ))
    then
        echo "eof reached"
        break
    else
        echo "read \"${RIGA}\""
    fi
done
```

va bene anche while ((1)) ;

136

read - Precisazione su raggiungimento di fine file (3) procedure di lettura corrette e tra loro equivalenti

Occorre una nozione aggiuntiva: la stringa \${#VAR} viene interpretata dalla bash sostituendola con la stringa di cifra che rappresenta il numero di caratteri di cui la variabile VAR, se esiste, è formata (la lunghezza della variabile VAR).

Se la variabile non esiste allora la stringa viene sostituita dalla stringa vuota.

Esempio: VAR="ciao"; echo \${#VAR}; produce in output 4

Quando si fa una lettura con la **read**, se la **read** dice di essere arrivata a fine file occorre comunque controllare se nella variabile letta c'è qualcosa dentro.

Il controlli seguenti sono equivalenti:

- accettano in input anche righe vuote (riga con la sola andata a capo) che lasciano la variabile RIGA vuota nel caso che non si sia ancora raggiunta la fine del file.

```
while read RIGA; if (( $(?==0)); then true; elif (( $(#RIGA) != 0)); then true; else false; fi ;
do
echo read "${RIGA}"; done
OR Logico dentro espressione condizionale
while read RIGA ; [[ $? == 0 || ${RIGA} != "" ]]; do echo "read ${RIGA}"; done
while read RIGA ; [[ $? -eq 0 || ${#RIGA} > 0 ]]; do echo "read ${RIGA}"; done
while read RIGA ; [[ $? == 0 ]] || [[ -n ${RIGA} ]]; do echo "read ${RIGA}"; done
```

Sequenza di comandi condizionale, prosegue se exit status != 0 138

read - Precisazione su lettura di riga con spazi iniziali (5)

Può accadere che una riga digitata da tastiera oppure anche una riga di un file, abbiano all'inizio degli spazi banchi oppure delle tabulazioni.

Ad esempio, un file denominato file_con_spazio_iniziale_nella_riga.txt potrebbe contenere una riga così fatta: " K 23F G2"

In tal caso, una **read** (a cui viene passata UNA variabile) che legge quella riga NON mette nella variabile di lettura gli spazi iniziali ma solo i caratteri a partire dal primo carattere non bianco.

Perciò, una lettura fatta come qui sotto indicato

```
read RIGA < file_con_spazio_iniziale_nella_riga.txt
echo "${RIGA}"
```

causerebbe l'output seguente:

" K 23F G2"

in cui, evidentemente, non sono stati letti i caratteri bianchi e le tabulazioni iniziali.

Ciò è dovuto al fatto che la **read** tenta di leggere le parole in una riga, non la riga.

Per leggere correttamente anche gli spazi bianchi, occorre dire alla bash che gli spazi bianchi e le tabulazioni SONO delimitatori delle parole.

Occorre a tal scopo settare preventivamente la variabile IFS come VUOTA

```
IFS=""; read RIGA < file_con_spazio_iniziale_nella_riga.txt
echo "${RIGA}"
```

causa l'output corretto seguente:

" K 23F G2"

140

read - Lettura di un numero specificato di caratteri (7)

L'opzione **-N** della **read** permette di specificare il numero esatto di caratteri che devono essere letti.

Esempio: Legge 4 caratteri dallo standard input e li mette nella variabile STRINGA

```
read -N 4 STRINGA
```

Notare che non vengono più separate le word, anche gli spazi e i tab sono considerati caratteri qualsiasi.

L'andata a capo \n NON viene considerata un terminatore che interrompe la read bensì un carattere letto.

Se durante la lettura viene raggiunta la fine riga,

la read non termina e aspetta altri caratteri per raggiungere il numero di caratteri richiesto e mette anche il \n nella variabile

Se una read chiede meno caratteri di quelli presenti nella riga (del file o digitata dall'utente), che fine fanno i caratteri residui?

Ciascuna read comincia la lettura proprio dai caratteri residui (cioè non letti) dalla read che l'ha preceduta.

Vedere l'esempio qui di lato e commentarlo.

Notare la terza lettura che legge l'andata a capo

Esempi di comandi e i loro output

```
vic@vic:~$ cat mioinput.txt
messaggio1
messaggio2
vic@vic:~$ exec 103< mioinput.txt
vic@vic:~$ read -n 4 -u 103 RIGA
vic@vic:~$ echo $RIGA
mess
vic@vic:~$ read -n 4 -u 103 RIGA
vic@vic:~$ echo $RIGA
aggi
vic@vic:~$ read -n 4 -u 103 RIGA
vic@vic:~$ echo $RIGA
o1
vic@vic:~$ read -n 4 -u 103 RIGA
vic@vic:~$ echo $RIGA
mess
vic@vic:~$ read -n 4 -u 103 RIGA
vic@vic:~$ echo $RIGA
essa
vic@vic:~$ read -n 4 -u 103 RIGA
vic@vic:~$ echo $RIGA
aggi
vic@vic:~$ read -n 4 -u 103 RIGA
vic@vic:~$ echo $RIGA
o2
vic@vic:~$ read -n 4 -u 103 RIGA
vic@vic:~$ echo $RIGA
1
vic@vic:~$ echo $RIGA
2
vic@vic:~$ exec 103<-
vic@vic:~$
```

142

read - Precisazione su raggiungimento di fine file (2)

Può accadere che la lettura incontri la fine del file senza incontrare l'andata a capo. Capita se leggo da file e nel file l'ultima riga **non ha** l'andata a capo \n alla fine.

In tal caso la **read** mette, nella variabile che gli viene passata, tutti i caratteri non ancora letti che precedono la fine del file, e poi restituisce un valore >0 per indicare che il file è stato usato fino alla fine.

Quindi, quando la **read** dice che è arrivata alla fine del file di input, possono essere accaduti due eventi diversi:

- 1) la **read** ha incontrato subito la fine del file e quindi nella variabile non è stato messo nulla, e quindi nella variabile la **read** mette la stringa vuota "".
- 2) la **read** ha letto dei caratteri e poi ha incontrato la fine del file, e quindi nella variabile troverà i caratteri letti ma la **read** restituisce comunque un valore >0.

Perciò, quando si fa una lettura con la **read**, se la **read** dice di essere arrivata a fine file occorre comunque controllare se nella variabile letta c'è qualcosa dentro.

Vedere la pagina successiva per esempi di controllo.

137

read - Lettura da standard input (tastiera o file) (4)

Se al comando **read** vengono passati come argomenti una o più variabili, allora il contenuto della variabile IFS viene usato per separare la linea letta in parole e per assegnare alle variabili passate le parole lette, in particolare:

• Se la riga letta contiene più parole del numero delle variabili passate alla **read**, allora ciascuna variabile viene riempita con una parola estratta dalla linea letta, tranne l'ultima variabile che riceve tutto quello che resta della linea.

• Se invece la riga letta contiene meno parole del numero di variabili passate alla **read**, allora solo le prime variabili ricevono una parola, alle altre è assegnato il valore vuoto. Esistono alcune opzioni interessanti: -o -N -r ed altre meno necessarie -e -n -t . Il risultato restituito da **read** è sempre 0 tranne che in caso di eof (o fd non valido o timeout scaduto, se specificati).

Se eseguo il comando

read varA varB varC	
e scrivo a tastiera la frase	prima seconda terza
le variabili assumono valore	varA="prima" varB="seconda" varC="terza"

se invece scrivo a tastiera la frase

prima seconda terza quarta	
le variabili assumono valore	varA="prima" varB="seconda" varC="terza quarta"

se invece scrivo a tastiera la frase

prima seconda	
le variabili assumono valore	varA="prima" varB="seconda" varC=""

139

read - Lettura fino a un numero massimo di caratteri (6)

L'opzione **-n** della **read** permette di specificare il numero massimo di caratteri che devono essere letti.

Esempio: Legge fino a 4 caratteri dallo standard input e li mette nella variabile STRINGA

```
read -n 4 STRINGA
```

Notare che non vengono più separate le word, anche gli spazi e i tab sono considerati caratteri qualsiasi.

Invece l'andata a capo \n viene ancora considerata un terminatore che interrompe la **read**.

Se durante la lettura viene raggiunta la fine riga, la read termina anche se non ho letto tutti i caratteri richiesti. In tal caso, nella variabile

trovo meno caratteri di quelli che avevo richiesto. La read successiva partira dalla riga successiva.

Se una read chiede K caratteri, alla fine della digitazione dei K caratteri richiesti la **read** termina anche se non è stato premuto l'invio (return) cioè anche se non è stata raggiunta la fine riga.

Che fine fanno i caratteri residui?

Ciascuna read comincia la lettura proprio dai caratteri residui (cioè non letti) dalla read che l'ha preceduta.

Esempi di comandi e i loro output

```
vic@vic:~$ cat mioinput.txt
messaggio1
messaggio2
vic@vic:~$ exec 103< mioinput.txt
vic@vic:~$ read -n 4 -u 103 RIGA
vic@vic:~$ echo $RIGA
mess
vic@vic:~$ read -n 4 -u 103 RIGA
vic@vic:~$ echo $RIGA
aggi
vic@vic:~$ read -n 4 -u 103 RIGA
vic@vic:~$ echo $RIGA
o1
vic@vic:~$ read -n 4 -u 103 RIGA
vic@vic:~$ echo $RIGA
mess
vic@vic:~$ read -n 4 -u 103 RIGA
vic@vic:~$ echo $RIGA
essa
vic@vic:~$ read -n 4 -u 103 RIGA
vic@vic:~$ echo $RIGA
aggi
vic@vic:~$ read -n 4 -u 103 RIGA
vic@vic:~$ echo $RIGA
o2
vic@vic:~$ read -n 4 -u 103 RIGA
vic@vic:~$ echo $RIGA
1
vic@vic:~$ echo $RIGA
2
vic@vic:~$ exec 103<-
vic@vic:~$
```

141

read - confronto tra opzioni -n e -N (7)

Esempi di comandi e i loro output

Esempi di comandi e loro output

```
vic@vic:~$ cat mioinput.txt
messaggio1
messaggio2
vic@vic:~$ exec 103< mioinput.txt
vic@vic:~$ read -N 4 -u 103 RIGA
vic@vic:~$ echo $RIGA
mess
vic@vic:~$ read -N 4 -u 103 RIGA
vic@vic:~$ echo $RIGA
aggi
vic@vic:~$ read -N 4 -u 103 RIGA
vic@vic:~$ echo $RIGA
o1
vic@vic:~$ read -N 4 -u 103 RIGA
vic@vic:~$ echo $RIGA
mess
vic@vic:~$ read -N 4 -u 103 RIGA
vic@vic:~$ echo $RIGA
essa
vic@vic:~$ read -N 4 -u 103 RIGA
vic@vic:~$ echo $RIGA
aggi
vic@vic:~$ read -N 4 -u 103 RIGA
vic@vic:~$ echo $RIGA
o2
vic@vic:~$ read -N 4 -u 103 RIGA
vic@vic:~$ echo $RIGA
2
vic@vic:~$ read -N 4 -u 103 RIGA
vic@vic:~$ echo $RIGA
1
vic@vic:~$ echo $RIGA
2
vic@vic:~$ exec 103<-
vic@vic:~$
```

143

read - Lettura di un numero specificato di caratteri (7)

L'opzione **-N** della **read** permette di specificare

il numero esatto di caratteri che devono essere letti.

Esempio: Legge 4 caratteri dallo standard input

```
read -N 4 STRINGA
```

Notare che non vengono più separate le word, anche gli spazi e i tab sono considerati caratteri qualsiasi.

L'andata a capo \n NON viene considerata un terminatore che interrompe la read bensì un carattere letto.

Se durante la lettura viene raggiunta la fine riga,

la read non termina e aspetta altri caratteri per raggiungere il numero di caratteri richiesto e mette anche il \n nella variabile

Se una read chiede meno caratteri di quelli presenti nella riga (del file o digitata dall'utente), che fine fanno i caratteri residui?

Ciascuna read comincia la lettura proprio dai caratteri residui (cioè non letti) dalla read che l'ha preceduta.

Vedere l'esempio qui di lato e commentarlo.

Notare la terza lettura che legge l'andata a capo

STREAM DI I/O NON PREDEFINITI

Stream di I/O Non Predefiniti (1) Apertura, File Descriptor, accesso

Uno script bash può avere necessità di usare dei file su disco per fare I/O anche se non vengono passati mediante stdin ed stdout.

Ricordiamo che ad stdin, stdout e stderr sono associati dei file descriptor (rispettivamente 0, 1, 2) che permettono di accedere a quegli stream.

E' possibile aprire un altro file da disco, ottenere un altro file descriptor che lo rappresenta ed utilizzare quel nuovo file descriptor per accedere al file aperto.

All'apertura del file, l'utente può decidere il file descriptor (il numero) che rappresenterà il file aperto, ma se tale fd è già usato avvengono problemi. In alternativa (procedura vivamente consigliata) l'utente può chiedere al sistema operativo di scegliere un fd libero.

Gli operatori sono indicati nella seconda e terza colonna della seguente tabella:

Modo Apertura	Utente sceglie fd (n è il numero scelto dall'utente)	Sistema sceglie fd libero e lo inserisce in variabile
Solo Lettura	<code>exec n< PercorsoFile</code>	<code>exec \${NomeVar}< PercorsoFile</code>
Scrittura	<code>exec n> PercorsoFile</code>	<code>exec \${NomeVar}> PercorsoFile</code>
Aggiunta in coda	<code>exec n>> PercorsoFile</code>	<code>exec \${NomeVar}>> PercorsoFile</code>
Lettura e Scrittura	<code>exec n<> PercorsoFile</code>	<code>exec \${NomeVar}<> PercorsoFile</code>

NB: i simboli < > >> <> devono essere attaccati (senza spazi) al numero o alla }

144

Stream di I/O Non Predefiniti (2) Esempi: Apertura di file in Lettura e in Scrittura

uso una variabile che chiamo FD in cui verrà messo il file descriptor del file aperto. Nel comando **read** specifico l'opzione **-u** (seguita dal file descriptor del file aperto) per indicare al comando read da quale file aperto deve essere effettuata la lettura.

esempio:

effettuo le letture dal file mioinput.txt aprendolo in lettura

```
exec ${FD}< /home/vittorio/mioinput.txt
while read -u ${FD} StringaLetta ;
do
echo "ho letto: ${StringaLetta}"
done
```

esempio:

scrivo l'output dei comandi echo sul file miooutput.txt aprendolo in scrittura

```
exec ${FD}> /home/vittorio/miooutput.txt
for name in pippo pippa pippi ; do
echo "inserisco ${name}" 1>&${FD}
done
```

145

Stream di I/O Predefiniti e Non (1) Dove trovo i file descriptor aperti da una bash?

Suppongo di avere una bash interattiva aperta.

La variabile \$\$ mi dice il PID process identifier della shell corrente.

Supponiamo che il PID della mia shell sia 1231.

Nella directory /proc/ esiste una sotto-directory per ciascun processo in esecuzione del processo stesso.

Quindi il seguente comando visualizza il contenuto della directory corrispondente alla shell corrente

```
ls /proc/$$/
```

Nella sotto-directory propria di ciascun processo, esiste una sotto-directory fd in cui sono presenti dei file speciali che sono i file aperti da quel processo.

Guarda caso, trovo sempre (tranne casi speciali) i file aperti aventi nome 0 1 2

Se nella mia bash apro un altro file, ad esempio col comando

```
exec {FD}< /tmp/caz.txt
```

e scopro che il file aperto ha file descriptor 7

```
echo ${FD}
```

7

se guardo nella directory /proc/1231/fd/ vedo che è stato aggiunto un file speciale di nome 7.

```
ls /proc/$$/fd/
```

146

Stream di I/O Predefiniti e Non (2) Chiusura di file mediante il suo file descriptor

Qualunque sia il modo di apertura con cui ho aperto un file (lettura scrittura o entrambi), la chiusura di un file può essere effettuata utilizzando il comando exec con il seguente strano operatore

```
exec n>&-
```

dove n e' il file descriptor del file da chiudere.

Analogamente, se la variabile FD contiene il valore del file descriptor da chiudere, posso chiudere quel file utilizzando la seguente strana sintassi:

```
exec {FD}>&-
```

Nota bene, se dopo la chiusura del file utilizzo il file descriptor, la bash produce un errore.

```
exec 103> /home/vittorio/mioinput.txt
```

```
echo "messaggio1" 1>&103
```

```
# chiudo
```

```
exec 103>&-
```

```
echo "messaggio2" 1>&103
```

bash: 103: Bad file descriptor ← produce un messaggio di errore

147

RIDIREZIONAMENTI DI STREAM I/O

Vedi dispense complete pag 138-163 (148-173)

RAGGRUPPAMENTO DI COMANDI

Raggruppamento di comandi (1)

Se si racchiude una sequenza di comandi tra parentesi tonde, allora in esecuzione viene creata una subshell per eseguire quella sequenza dei comandi.

Il risultato restituito dalla subshell (restituito dalla parentesi tonda) è il risultato dell'ultimo comando eseguito nella subshell, cioè l'ultimo comando eseguito tra quelli dentro le parentesi tonde.

Tutti i comandi condividono gli stessi stdin/stdout/stderr **utilizzandoli in sequenza**. Quindi, all'output del primo comando dentro le parentesi viene concatenato l'output del secondo comando poi quello del terzo etc etc etc

Ciò permette di trattare / ridirigere input ed output di tutti i comandi dentro le parentesi tonde come se fossero un solo comando.

Esempio: con il comando:

```
ls; pwd; whoami > out.txt
```

visualizzo

nomi files in directory corrente

/home/vittorio

E dentro il file out.txt trovo

vittorio

Esempio: con il comando tra parentesi

```
( ls; pwd; whoami ) > out.txt
```

non visualizzo nulla

e dentro il file out.txt trovo

```
a1B a2B aB akB akmB akmtB
```

/home/vittorio

vittorio

174

Raggruppamento di comandi (2)

stdin stdout e stderr dei singoli comandi dentro le parentesi tonde vengono concatenati

concatenazione stdout

```
( cat file1.txt ; cat file2.txt ) | grep stringa
```

il comando grep legge, come se provenissero da tastiera, le righe di entrambi i file prima le righe di file1.txt e poi le righe di file2.txt

concatenazione stdout e stderr

```
( cat file1.txt ; cat file2.txt ) |& grep stringa
```

il comando grep legge, come se provenissero da tastiera, le righe di entrambi i file prima le righe di file1.txt e poi le righe di file2.txt

concatenazione stdin

```
cat file1.txt | ( read RIGA1 ; usa RIGA1 ; read RIGA2 ; usa RIGA2 )
```

i due comandi read leggono una la prima e l'altro la seconda riga prodotte da cat

175

Raggruppamento di comandi (3)

la bash è un po' stronza e cerca di ottimizzare, nonostante la nostra volontà

Esempio per far vedere che :

- **se dentro la parentesi tonda metto un solo comando, allora la bash padre non crea una altra shell figlia in cui far eseguire il singolo comando**

```
ps ; ( ps )
```

vedere che l'output dei due ps mostra una sola bash in entrambi i casi

lanciare poi

```
ps ; ( ps ; ps )
```

e vedere che nell'output dei due ps interni compaiono due bash, quella padre e una figlia

176

GNU Coreutils (info coreutils)

Esempio di uso di ridirezionamenti e GNU Coreutils

Cosa si puo' fare con sequenze di comandi, raggruppamenti, ridirezionamenti, ...?
Ne approfitto per farvi vedere alcune utilities che manipolano file di testo, appartenenti al progetto GNU Coreutils (unificazione dei vecchi pacchetti Fileutils, Shellutils e Textutils).
Vedi <http://www.gnu.org/software/coreutils/coreutils.html>

Dato un file di testo, di nome dati.txt, utilizzare programmi disponibili comunemente in un sistema linux per ottenere il seguente risultato:

Prendere le prime 2 righe del file e le ultime 3 righe del file stesso.
Di queste righe selezionare solo quelle che contengono la sequenza AL
Nelle righe rimaste, sostituire le lettere AL con le lettere CUF
Nelle righe cosi' modificate eliminare i primi 3 caratteri (mantenere dal 4° in poi)

Scrivere le righe modificate nel file output.txt

dati.txt	
pappapero	
caracavAlo	
piripiccone	
ammappete	
uffaufa	
pedalandeo	
avvAllare	
remare	

output.txt
acavCUFo
CUFlare

Soluzione:
(head -n 2 dati.txt ; tail -n 3 dati.txt) | grep AL | sed 's/AL/CUF/g' | cut -b 4- > output.txt

178

Qualche dettaglio: coreutils e codifica dei caratteri

Alcune delle coreutils, in particolare **cut** e **wc**, consentono all'utente di chiedere di lavorare sui bytes oppure sui caratteri. Qual è la differenza?

I caratteri possono essere codificati con **8 bit** (Latin-1 (iso-8859-1) cioè codifica ascii estesa) oppure con **due bytes** (UNICODE UTF-16) oppure con **un numero variabile di bytes** (codifiche multibyte, UNICODE UTF-8, compatibile con ASCII).

Se chiediamo a **cut** o **wc** di lavorare con i caratteri, i processi analizzano il flusso di byte e determinano come sono codificati i caratteri e poi li contano o rimuovono o lasciano a seconda del comando richiesto.

Se in un file sono contenuti due caratteri codificati con due bytes ciascuno e due caratteri codificati con 1 byte ciascuno, allora il numero di caratteri sarà 4 mentre il numero di bytes di quel file sarà 6.

Normalmente **wc** usa la variabile **LC_CTYPE** (tipo locale dei caratteri) per decidere che tipo di codifica dei caratteri deve usare, normalmente è UTF-8. Ad esempio, **LC_CTYPE=en_US.UTF-8**. Se la codifica è UTF-8 allora caratteri e bytes sono due cose diverse.

Se la codifica è C allora si considerano char di 1 byte ed allora byte e caratteri sono la stessa cosa:

Esempio in cui uso carattere è codificato con i due caratteri 0xc3 0xa9.
comando output
printf 'xc3xa9' | LC_CTYPE=en_US.UTF-8 wc -c 2
printf 'xc3xa9' | LC_CTYPE=en_US.UTF-8 wc -m 1
printf 'xc3xa9' | LC_CTYPE=C wc -c 2
printf 'xc3xa9' | LC_CTYPE=C wc -m 2

180

Esempi di uso di utilities in Coreutils

cat dati.txt | tail -n 3

tail -f /var/log/messages

cat dati.txt | tee a.txt b.txt

ESEMPIO di creazione e riempimento di file in una directory in cui devo assumere i permessi di root.

- Comando che provoca errore, no permessi di scrittura
echo "contenuto del file" > /root/prova.txt

- Ancora errore, sudo viene eseguito dopo ridirezionamento:
sudo echo "contenuto del file" > /root/prova.txt

- OK
sudo tee /root/prova.txt <<END
"contenuto del file"
END

179

sed - stream editor (coreutils) (1)

l'editor di stream, **sed** prevede una quantita' assurda di possibili comandi, vediamone alcuni come esempi. Lanciare man sed per ulteriori dettagli.

Nell'esempio precedente, **sed 's/AL/CUF/g'** Il carattere **g** serve a far sostituire, in ciascuna linea processata, **tutte** le occorrenze delle stringhe **AL**. In assenza del carattere **g**, in ciascuna riga verrebbe modificata solo la prima stringa **AL** incontrata.

Esempi con sed

Sostituisce la **prima occorrenza** di togli con metti in ciascuna riga del file nomefile.
sed 's/togli/metti/' nomefile

Rimuovere il **primo tra i caratteri** a che trova in ciascuna riga del file nomefile
sed 's/a/' nomefile

Rimuovere il carattere in prima posizione di ogni linea che si riceve dallo standard input.

^ significa inizio linea, **.** significa un carattere qualunque
sed 's/^./'

Rimuovere l'**ultimo carattere** di ogni linea ricevuta dallo stdin. **\$** significa fine linea, il **.** significa un carattere qualunque
sed 's/.\$/'

Esegue **due rimozioni insieme** (**:**) Rimuovere il primo e l'ultimo carattere in ogni linea.
sed 's/..\$/../'

Rimuove I primi 3 caratteri ad inizio linea.
sed 's/.../../'

Rimuove I primi 4 caratteri ad inizio linea.
sed -r 's/{4}/'

181

sed - stream editor (coreutils) (2)

Continua esempi con sed

To remove last n characters of every line (nell'esempio 3 caratteri)
sed -r 's/.{3}\$//'

To remove everything except the 1st n characters in every line
sed -r 's/.{1}.{3}.*\$//'

To remove everything except the last n characters in a file
sed -r 's/.{1}.{3}.*\$//1'

To remove multiple characters present in a file (**g** means all occurrences):
sed 's/[aoe]/g'

To remove all occurrences of a pattern
sed 's/lari/g'

To delete only nth occurrences of a character in every line (2 occurrences in the example)
sed 's/u//2'

To delete everything in a line followed by a character
sed 's/a.*\$//'

To remove all alpha-numeric characters present in every line
sed 's/[a-zA-Z0-9]//g'

182

Process Identifier di Shell bash: \$\$ e \$BASHPID

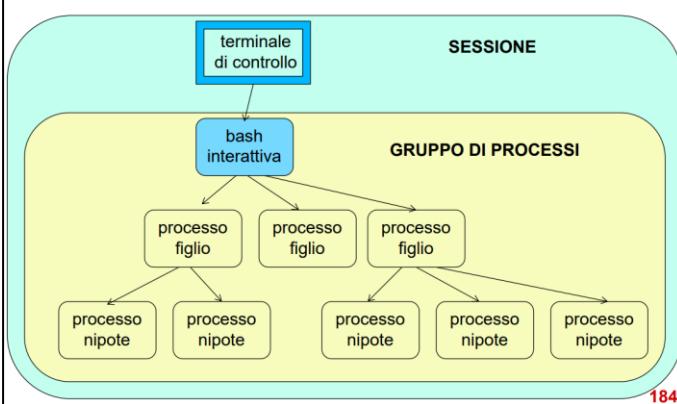
- Ogni Processo è identificato da un identificatore numerico univoco, il PID.
- Anche le shell in esecuzione posseggono un loro PID.
- Se occorre conoscere il PID di una shell in esecuzione, si usa la variabile **\$\$** che viene espansa con il PID della shell corrente.
Esempio:
echo il pid della shell è \$\$
il pid della shell è 14726
- Se uso la variabile **\$\$** all'interno di uno script, vedo il PID della bash che esegue lo script.
- Esiste una eccezione di comportamento.** Se raggruppo dei comandi facendoli eseguire in una subshell (), allora dentro le () la variabile **\$\$** mi fa vedere il PID della bash più esterna, non della subshell ().
Esempio:
echo -n pid fuori \$\$; (echo -n pid dentro \$\$; pwd)
pid fuori 14726 pid dentro 14726 /home/studente
- Se occorre usare il PID di una subshell specificata con (), occorre usare la variabile d'ambiente **BASHPID**. La variabile **BASHPID** contiene il PID della bash corrente, anche se questa è una bash creata con ().
Esempio:
echo -n pid fuori \$\$; (echo -n pid dentro \$BASHPID ; pwd)
pid fuori 14726 pid dentro 22399 /home/studente
- Posso usare sempre la variabile **BASHPID** al posto della **\$\$**, però la var **BASHPID** è definita solo in bash e solo in bash delle versioni > 4. **BASHPID** è meno portabile

183

CONCETTI

Concetti di Processo, Sessione, Gruppo di Processi, Terminale di controllo del gruppo di processi (0)

- Scenario



Concetti di Processo, Gruppo di Processi, Terminale di controllo del gruppo di processi (1)

- Processi:** Un processo è la più piccola unità di elaborazione completa ed autonoma che può essere eseguita in un computer.
 - Un **processo** è un insieme di thread di esecuzione operanti all'interno di un contesto, il quale comprende uno spazio di indirizzamento in memoria ed una tabella dei descrittori di file aperti per quel processo.
 - Gruppo di processi (process group):** Un processo può lanciare l'esecuzione di altri processi: questi altri processi appartengono allo stesso gruppo di processi del processo padre, a meno che non si svolgano azioni che modifichino il gruppo di appartenenza.
 - Terminale di controllo:** Un processo lanciato in esecuzione può avere un "controlling terminal" (un terminale da cui è controllato, che è l'astrazione di terminale (console=video+tastiera) da cui prende gli standard input output ed error).
 - Terminale di controllo del gruppo di processi** Un processo lanciato in esecuzione eredita lo stesso "controlling terminal" dal padre che lo ha lanciato, a meno che non si svolgano azioni che sganciano il processo dal terminale di controllo. Quindi tutti i processi di uno stesso gruppo di processi condividono lo stesso terminale di controllo.
- 185

Creazione della Sessione

- Per gestire l'insieme di processi, il kernel necessita di raggruppare i processi in modo più complesso che non la semplice relazione padre-figlio.
- Perciò viene usata l'astrazione di "**sessione**" e di "**gruppo di processi**".
- Un utente normalmente apre una **shell interattiva** (di login o no) che viene inserita dal sistema operativo in un terminale (la window che rappresenta la console monitor/video+tastiera) e da questa shell eseguirà operazioni varie, tra cui l'esecuzione di comandi, eseguibili binari e script.
- Il terminale in cui la shell interattiva esegue è detto "**terminale di controllo**" (controlling terminal).
- La shell interattiva all'inizio **crea una a sessione**, specificando un nuovo sessionId, e **lega la sessione al terminale di controllo**. Questa shell interattiva diventa così il leader della sessione.
- I processi discendenti di quella shell apparterranno alla stessa sessione, a meno che non si eseguano operazioni di distacco.
- La shell interattiva prende come stdin stdout ed stderr quelli che il terminale gli fornisce.
 - I comandi, eseguibili binari e script lanciati dalla shell interattiva ereditano i file aperti del padre, quindi ereditano anche gli stdin, stdout ed stderr del terminale.

186

Motivazione della Sessione

- Quando un utente si disconnette da un sistema, il kernel deve terminare tutti i processi che l'utente sta ancora eseguendo
 - altrimenti, gli utenti lascerebbero un gran numero di vecchi processi in attesa di input che non potranno mai arrivare.
- Come determinare quali processi terminare?
- Per semplificare questa attività, i processi sono organizzati in sessioni.
- Il leader della sessione è il processo che ha creato la sessione.
- L'ID della sessione è uguale al pid del processo che ha creato la sessione tramite la chiamata di sistema setsid().
 - La funzione setsid() non accetta argomenti e restituisce il nuovo ID di sessione, cioè il PID del processo leader.
- Tutti i discendenti di quel processo sono quindi membri di quella sessione a meno che non si rimuovano specificamente da essa.

187

USO DI STRINGHE

Parameter Expansion – Uso di stringhe (1) Estrazione di sottostringhe da variabili

La bash fornisce alcuni operatori che vengono specificati all'interno dei simboli che vengono utilizzati per ottenere il contenuto di una variabile, la variable expansion.

Tali operatori forniscono una stringa che e' una parte del contenuto della variabile oppure una parte modificata del contenuto della variabile.

Il contenuto originale della variabile non viene modificato, a meno che io non effettui un assegnamento alla variabile originale.

Se l'estrazione di parte del contenuto o la sostituzione non sono possibili, gli operatori restituiscono l'intero contenuto della variabile, senza modifiche.

Gli operatori che effettuano una sostituzione, normalmente ne effettuano una soltanto, anche se fossero possibili piu' sostituzioni. Però qualche operatore consente di effettuare più sostituzioni.

Per effettuare piu' sostituzioni complesse si può però invocare piu' volte il comando salvando di volta in volta il risultato ottenuto.

Anticipiamo alcune definizioni:
definiamo suffisso=che sta alla fine prefisso=che sta all'inizio.

Parameter Expansion – Uso di stringhe (2) Estrazione di sottostringhe da variabili

Esempio di uso: estrarre numero tra [] in VAR="13] qualcosa con [o] fine"

Definiamo suffisso=che sta alla fine prefisso=che sta all'inizio.

pattern puo' contenere wildcard che cercano di matchare con sottostringhe in VAR pattern puo' contenere anche variabili

```
 ${VAR%%pattern}
$ echo ${VAR%%]*}      Rimuovo il piu' lungo suffisso che fa match con stringa orig
[13]                   Rimuovo fino alla fine a destra
                        la sottostringa piu' lunga che inizia con ]
${VAR%pattern}
$ echo ${VAR%]*}      Rimuovo il piu' corto suffisso che fa match con stringa orig
[13] qualcosa con [ o   Rimuovo fino alla fine della stringa originale
                        la sottostringa piu' corta che inizia con ]
${VAR##pattern}
$ echo ${VAR##*}      Rimuovo il piu' lungo prefisso che fa match con stringa orig
o ] fine               Rimuovo, dall'inizio dell'originale, la sottostringa piu' lunga
                        che finisce con [
${VAR#pattern}
$ echo ${VAR#*}       Rimuovo il piu' corto prefisso che fa match con stringa orig
[13] qualcosa con [ o fine Rimuovo, dall'inizio dell'originale, la sottostringa piu' corta
                        che finisce con [
```

222

Parameter Expansion – Uso di stringhe (3) sottostringhe da variabili

es: VAR="alfabetagamma"

\${#VAR} viene espansa nella stringa che esprime la lunghezza in byte del contenuto di VAR. Con la variabile VAR di esempio produce la stringa "17". Produce la stringa "0" se la variabile VAR non esiste o se è vuota.

\${VAR/pattern/string} cerca nel contenuto di VAR la sottostringa piu' lunga che fa match con il pattern specificato (con wildcard) e lo sostituisce con string. Se sono possibili piu' sostituzioni viene sostituita solo la sottostringa piu' vicina all'inizio.
ALTRA="va"; echo "\${VAR/b/a/k\\$}{ALTRA}p"
produce in output: alfakvafp

\${VAR:offset} sottostringa che parte dal offset-esimo carattere del contenuto di VAR L'offset del primo carattere e' zero. Gli argomenti vengono valutati aritmeticamente. Lo stesso per il prossimo operatore

\${VAR:offset:length} sottostringa lunga length che parte dal offset-esimo carattere del contenuto di VAR
es: VAR="alfabetagamma" ; DA="2" ; FINOA="5" ;
echo "\${VAR:\$DA}:{\$FINOA}+3" ;
produce in output: fabetaga

Parameter Expansion – Uso di stringhe (4) estrazione con sostituzione da variabili

esempio: come effettuare piu' sostituzioni nel contenuto di una variabile

```
#!/bin/bash
VAR="a1BDaxxx2BaDxxx3BbDxx4BcDxxx5BDxxx6BdD"
PREVIOUS=${VAR}
while true ; do
    VAR=${VAR/B?D/ZZZ}
    if [ ${VAR} == ${PREVIOUS} ] ; then
        break ;
    else
        PREVIOUS=${VAR}
    fi
done
echo "VAR=${VAR}"
```

Lo script visualizza
VAR=a1BDaxxx2ZZZxxx3ZZZxx4ZZZxxx5BDxxx6ZZZ

NB: più semplicemente si poteva usare: VAR=\${VAR//B?D/ZZZ} (vedi dopo)

224

Parameter Expansion – Uso di stringhe (4bis) ulteriori dettagli su sostituzione in variabili

L'operatore di sostituzione varia il suo comportamento se inserisco uno dei 3 caratteri speciali subito dopo il primo / dopo il nome di variabile, in particolare:

\${VAR//pattern/string} quando dopo il primo / c'e' un altro / allora TUTTE le sottostringhe che fanno match con il pattern specificato vengono sostituite, non solo la prima.
esempio: sostituisce tutti i cane con gatto {VAR//cane/gatto}
esempio: sostituisce tutti gli asterischi * con * per impedire che il contenuto della variabile possa essere interpretato.
 \${VAR//*/*}

\${VAR/#pattern/string} quando dopo il primo / c'e' un # allora il pattern viene sostituito SOLO SE si trova all' INIZIO della variabile.

\${VAR/%pattern/string} quando dopo il primo / c'e' un % allora il pattern viene sostituito SOLO SE si trova alla FINE della variabile.

Parameter Expansion – Uso di stringhe (5) operatori vari su variabili

Expansione verso nomi di variabili corrispondenti ad un prefisso pattern

\${!VarNamePrefix*} restituisce un elenco con tutti i nomi delle variabili il cui nome inizia con il prefisso specificato VarNamePrefix

Esempio: se esistono le seguenti variabili

```
BASH=/bin/bash
BASH_ALIASES=()
BASH_ARGC=()
BASH_ARGV=()
BASH_CMDS=()
BASH_LINENO=()
BASH_SOURCE=()
BASH_VERSION='4.1.17(9)-release'
CYG_SYS_BASHRC=1
ed eseguo il comando
echo ${BASH_AR}
vedro' in output
BASH_ARGC BASH_ARGV
```

226

225