

L'interprete dei comandi in alcuni s.o.

- ✿ Unix, Linux:
 - ✿ Diverse Shell di comandi, tra le più utilizzate **bash** (bourne again shell).
- Windows:
 - Shell DOS
 - Emulazione di interfaccia testuale Linux: bash realizzata dalla piattaforma cygwin. Utile per far eseguire, in ambiente windows, applicazioni e servizi implementati per Linux.
 - In Windows 10 è stata aggiunta una bash (non completissima)
- Mac:
 - Diverse Shell di comandi, compresa **bash**

- Nel corso di sistemi operativi utilizzeremo compilatori con sola interfaccia testuale, in particolare il GNU C Compiler (gcc).
- Gli ordini al compilatore, così come l'ordine di eseguire un programma, verranno impartiti utilizzando una shell bash.
 - In Linux e su Mac esiste già un terminale con shell bash.
 - In Windows è perciò necessario installare l'emulatore cygwin oppure installare una macchina virtuale Linux.

Interfaccia utente GUI

- ✿ Interfaccia user-friendly che realizza la metafora della scrivania (**desktop**)
 - ✖ Interazione semplice via mouse
 - ✖ Le **icone** rappresentano file, directory, programmi, azioni, etc.
 - ✖ I diversi tasti del mouse, posizionato su oggetti differenti, provocano diversi tipi di azione (forniscono informazioni sull'oggetto in questione, eseguono funzioni tipiche dell'oggetto, aprono directory – **folder**, o **cartelle**, nel gergo GUI)

Interpretazione dei comandi bash: Espansioni

La shell legge ciascuna riga di comando, e la interpreta eseguendo alcune operazioni. Alcune di queste servono a capire dove finisce un comando e dove inizia il successivo. Altre servono ad individuare le parole (word splitting) e a identificare le parole riservate del linguaggio e a identificare la presenza di costrutti linguistici for while if. Altre ancora sostituiscono (espandono) alcune parti della riga di comando con altre, interpretando caratteri speciali. Alcuni caratteri speciali però, disabilitano alcune espansioni. In generale, la shell comincia riconoscendo i caratteri speciali. Successivamente la shell opera alcune sostituzioni nella riga di comando, effettuando le cosiddette espansioni (expansion).

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

- **history expansion** !123
- **brace expansion** a{damn,czk,bubu}e
- **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).

Vedremo i dettagli di queste espansioni man mano andando avanti.

Nozioni per uso del Terminale: comando echo

anticipazione: Il comando echo permette di visualizzare a video la sequenza dei caratteri scritti subito dopo la parola echo e **fino al primo carattere di andata a capolinea** (che è inserito digitando il tasto <INVIO> o <RETURN>).

Il comando

```
echo pippo pippa pippi
```

visualizza

```
    pippo pippa pippi
```

Se ho bisogno di far stampare a video anche caratteri speciali come **punti e virgola, andate a capo (per visualizzare su più righe)**, e altri, devo inserire sia prima che dopo la stringa da stampare il separatore “ doppio apice (non sono due caratteri apici ma il doppio apice, quello sopra il tasto col simbolo 2.

Esempio:

```
echo "pippo ; pippa pippi"
```

Esercizio: All'interno di una shell far stampare a video su due linee diverse la stringa

```
pappa
```

```
ppero
```

Impossibile se non si usano i doppi apici

Soluzione: digitare (notando i doppi apici)

```
echo "pappa<INVIO>
```

```
ppero" <INVIO>
```

Nozioni per uso del Terminale: Variabili e Variable expansion (1)

La shell dei comandi permette di usare delle **Variabili** che sono dei simboli dotati di un **nome** (che identifica la variabile) e di un **valore (che può essere cambiato)**.

Il nome è una sequenza di caratteri anche numerici, ad es: PATH, PS1, USER PPID.

Attenzione, maiuscole e minuscole sono considerate diverse. Ad es PaTH !=PATH

Il valore è anch'esso una sequenza di caratteri compresi caratteri numerici e simboli di punteggiatura, ad es: /usr:/usr/bin:/usr/sbin 17 i686-pc-cygwin

Si noti che anche se il valore è composto solo da cifre (caratteri numerici), si tratta sempre di una stringa di caratteri.

Le variabili di una shell possono essere stabilite e modificate sia dal sistema operativo sia dall'utente che usa quella shell.

Alcune variabili (dette d'ambiente) vengono impostate subito dal sistema operativo non appena viene iniziata l'esecuzione della shell (ad es la variabile PATH).

Altre variabili possono essere create ex-novo dall'utente in un qualunque momento dell'esecuzione della shell.

Le variabili possono essere usate quando si digitano degli ordini per la shell. La shell **riconosce i nomi delle variabili contenuti negli ordini digitati, e cambia il contenuto dell'ordine sostituendo al nome della variabile il valore della variabile.**

Affinchè la shell **distingua il nome di una variabile**, questa deve essere **preceduta** dalla coppia di caratteri \${ e seguita dal carattere } Se, all'interno dell'ordine digitato, il nome della variabile è seguito da spazi (caratteri "bianchi") non c'è pericolo di confondere la variabile con il resto dell'ordine e si possono omettere le parentesi graffe.

Nozioni per uso del Terminale: Variabili (2)

assegnamenti e **variable** (e parameter) **expansion**

Ricordando che il comando echo permette di visualizzare a video la sequenza dei caratteri scritti subito dopo la parola echo e fino al primo carattere di andata a capolinea (che è inserito digitando il tasto <INVIO> o <RETURN>).

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. Non viene visualizzato nulla

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

Nozioni per uso del Terminale: Variabili (2bis)

Nota Bene: Assegnazione di valore ad una variabile

Quando si assegna un valore ad una variabile, **NON SONO CONSENTITI SPAZI NE' PRIMA NE' DOPO IL SIMBOLO =**

Assegnamento corretto senza spazi prima o dopo il =

VARIABILE=contenuto

Assegnamento sbagliato a causa di spazio PRIMA del simbolo =

VARIABILE =contenuto

In questo caso la shell cerca di eseguire il comando avente nome VARIABILE passandogli come argomento la stringa =contenuto

Avrei lo stesso errore se lasciassi uno spazio ANCHE dopo l'uguale

VARIABILE = contenuto

Assegnamento sbagliato a causa di spazio DOPO il simbolo =

VARIABILE= contenuto

In questo caso la shell cerca di eseguire il comando avente nome **contenuto** costruendo per tale comando un nuovo ambiente di esecuzione in cui colloca una variabile vuota di nome VARIABILE (vedere slide piu' avanti).

Nozioni per uso del Terminale: Variabili (3)

Esiste una **variabile d'ambiente** particolare e importantissima, detta **PATH**

Viene impostata dal sistema operativo già all'inizio dell'esecuzione della shell.

L'utente può cambiare il valore di questa variabile.

La variabile PATH contiene una sequenza di percorsi assoluti nel filesystem di alcune directory in cui sono contenuti gli eseguibili. I diversi percorsi sono separati dal carattere :

Esempio di valore di PATH /bin:/sbin:/usr/bin:/usr/local/bin:/home/vittorio

Questa PATH definisce i percorsi che portano alle directory seguenti

- /bin
- /sbin
- /usr/bin
- /usr/local/bin
- /home/vittorio

Quando io ordino alla shell di eseguire un certo file binario, chiamandolo per nome, ma senza specificare il percorso completo (assoluto o relativo) per raggiungere quel file, allora **la shell cerca quel file binario all'interno delle directory specificate dai percorsi che formano la variabile PATH, nell'ordine con cui i percorsi sono contenuti nella variabile PATH**, cioè nell'esempio prima in /bin poi in /sbin etc.etc.

- Quando la shell trova il file eseguibile lo esegue.
- Se il file eseguibile non viene trovato nelle directory specificate, la shell visualizza un errore e non esegue il file.

Usare il Terminale a linea di comando (bash)

Per aprire il terminale a riga di comando Linux-like in ambiente grafico:

In Windows cercare e cliccare l'icona di “cygwin bash shell” o “cygwin terminal”.

In Linux e Mac cliccare sul menù “terminal” o “terminal emulator” o “console”.

Si aprirà una “finestra” grafica e comparirà una piccola segnalazione **lampeggiante (cursore)** che indica che la shell è pronta ad accettare dei caratteri da tastiera.

Ad ogni istante, la shell opera stando in una posizione (directory) del filesystem denominata **directory corrente**. All'inizio dell'esecuzione della shell, la directory corrente è la home directory dell'utente che esegue la shell stessa.

L'utente può cambiare la directory corrente utilizzando il comando **cd**.

Usando l'interfaccia utente a linea di comando possono essere eseguiti

- **comandi** (piu' precisamente **comandi built-in**). Sono implementati e inclusi nella shell stessa e quindi non esistono come file separati. Sono forniti dal sistema operativo. Ad esempio cd, if, for.
- **file binari eseguibili**. Sono file che contengono codice macchina e che si trovano nel filesystem. Possono essere forniti dal sistema operativo o dagli utenti. Ad esempio, ls, vi, tar, gcc, primo.exe.
- **script**. Sono file di testo che contengono una sequenza di nomi di comandi, binari e altri script che verranno eseguiti uno dopo l'altro. Possono essere forniti dal sistema operativo o dagli utenti. Ad esempio exemplo_script.sh

Per essere eseguito, un file binario o uno script deve avere i **permessi di esecuzione**.

Un utente può impostare il permesso di esecuzione di un proprio file usando il comando chmod come segue: **chmod u+x esempio_script.sh**

Come eseguire da Terminale a linea di comando

Come specificare il nome del comando o dell'eseguibile?

I **comandi** possono essere eseguiti semplicemente invocandone il nome. (es: cd ../)

I **file eseguibili (binari o script)** devono invece essere invocati specificandone

- o **il percorso assoluto** (a partire dalla root)
ad es per eseguire primo.exe digitò /home/vittorio/primo.exe
- oppure **il percorso relativo** (a partire dalla directory corrente)
se la directory corrente è /home/ada allora digitò ../../vittorio/primo.exe
- oppure **il solo nome, a condizione che quel file sia contenuto in una directory specificata nella variabile d'ambiente PATH**

Ad esempio, se la variabile PATH è /bin:/usr/bin:/home/vittorio:/usr/local/bin
allora qualunque sia la directory corrente, per eseguire il file /home/vittorio/primo.exe
basta digitare il solo nome del file primo.exe

Esercizio:

Se la directory corrente è /home/vittorio ma quella directory non si trova nella variabile PATH, allora come posso eseguire il file primo.exe, che si trova in quella directory, specificando il percorso relativo?

./primo.exe

Subshell

Una subshell e' una shell (shell figlia) creata da un'altra shell (shell padre).

IMPORTANTE: Una subshell viene creata in caso di:

- Esecuzione di comandi **raggruppati** (vedi dopo).
- Esecuzione di **script**.
- Esecuzione di processo in **background** (vedi dopo).
- **NB:** l'esecuzione di un comando built-in avviene nella stessa shell padre

Ogni shell ha una propria directory corrente (ereditata dalla shell padre).

Ogni shell ha delle proprie variabili.

Ogni subshell eredita dalla shell padre una copia delle variabili d'ambiente (vedi dopo).

Ogni subshell non eredita le variabili locali della shell padre.

Quando una shell deve eseguire uno script esegue queste operazioni:

- Legge la prima riga dello script in cui è indicato quale interprete di comandi deve eseguire lo script
es: `#!/bin/bash`
- **Crea una subshell**
- Il nome dello script viene passato come argomento (opzione `-c`) alla nuova subshell
- La nuova subshell esegue lo script.
- Alla fine dell'esecuzione dello script la subshell termina e restituisce il controllo alla shell padre, restituendo un valore intero che indica il risultato.

Nozioni per uso del Terminale: Variabili (4)

Ogni shell supporta due tipi di variabili

Variabili locali

Non “trasmesse” da una shell alle subshell da essa create

Utilizzate per computazioni locali all'interno di uno script

Variabili di ambiente

“Trasmesse” dalla shell alle subshell.

Viene creata una copia della variabile per la subshell

Se la subshell modifica la sua copia della variabile,
la variabile originale nella shell non cambia.

Soltamente utilizzate per la comunicazione fra parent e child shell

es: variabili \$HOME \$PATH \$USER %SHELL \$TERM

Per visualizzare l'**elenco delle variabili di ambiente**, utilizzare il comando **env** (ENVironment)

Quando dichiaro una variabile con la sintassi già vista dichiaro una variabile LOCALE.

nomevariabile=ValoreVariabile

Per trasformare una variabile locale già dichiarata in una variabile di ambiente, devo usare il comando **export** (notare che non uso il \$)

export nomevariabile

Posso anche creare una variabile dichiarandola subito di ambiente

export nomevariabile=ValoreVariabile

Nozioni per uso del Terminale: Variabili (5)

Esempio per esplicitare differenza tra var locali e var d'ambiente: ./var_caller.sh

var_caller.sh

```
echo "caller"  
# setto la var locale PIPPO  
# la var d'ambiente PATH esiste già
```

```
PIPPO=ALFA  
echo "PATH = ${PATH}"  
echo "PIPPO = ${PIPPO}"
```

```
echo "calling subshell"
```

./var_called.sh

```
echo "ancora dentro caller"  
echo "variabili sono state modificate ?"  
  
echo "PATH = ${PATH}"  
echo "PIPPO = ${PIPPO}"
```

var_called.sh

```
echo "called"  
echo "le variabili sono state passate ? "
```

```
echo "PATH = ${PATH}"  
echo "PIPPO = ${PIPPO}"
```

```
echo "modifico variabili "
```

```
PIPPO="${PIPPO}:MODIFICATO"  
PATH="${PATH}:MODIFICATO"
```

```
echo "PATH = ${PATH}"  
echo "PIPPO = ${PIPPO}"
```

```
echo "termina called"
```

Nozioni per uso del Terminale: Variabili (6)

Quando invoco uno script,
viene eseguita una nuova shell
al termine dell'esecuzione quella shell viene eliminata

Per ogni eseguibile invocato,
viene collocata in memoria una copia dell'ambiente di esecuzione,
l'eseguibile può modificare il proprio ambiente,
al termine dell'esecuzione l'ambiente viene eliminato dalla memoria.

Script execution senza creazione di subshell

Normalmente, se una shell lancia uno script, questo viene eseguito in una subshell

- con l'interprete di comandi indicato nella prima riga speciale dello script `#!/bin/bash`, se esiste quella riga,
- o con lo stesso interprete della shell chiamante, se quella prima riga non esiste.

Se lo script modifica le proprie variabili, la shell padre non si accorge delle modifiche.

Però, una shell puo' eseguire uno script senza creare la subshell in cui lo script dovrebbe essere eseguito.

Come ?

source nomescript

Oppure

▪ nomescript

In tal modo non viene considerata la prima riga speciale dello script.

A cosa serve? A modificare le variabili della shell padre mediante uno script.

Le variabili modificate dallo script sono proprio quelle della shell (padre, o meglio unica) , quindi la shell vede modificate le proprie variabili.

source e **▪** sono due comandi built-in ovvero implementati all'interno della shell

Script execution senza creazione di subshell (2)

Attenzione che il comando source va usato solo se siamo sicuri che all'interno dello script da eseguire ci sono dei comandi che possono essere correttamente interpretati dalla shell bash chiamante.

Se invochiamo con source uno script che necessita di un interprete diverso provochiamo dei problemi, poiché **non verrà lanciato l'interprete corretto indicato nella prima riga dello script**.

Ad esempio, se lo script perl **myperl.pl** contiene le seguenti 3 righe:

```
#!/usr/bin/perl
@famiglia = ("padre", "madre", "figlio", "figlia");
for ($i=0; $i<=#famiglia; $i++) { print "$famiglia[$i]\n"; }
```

ed io, all'interno di una bash, eseguo lo script con source

```
source ./myperl.pl
```

provoco degli errori perché non viene lanciato l'interprete /usr/bin/perl ed è la bash chiamante che cerca di interpretare i comandi perl

```
bash: ./myperl.pl: line 3: syntax error near unexpected token `('
bash: ./myperl.pl: line 3: `@famiglia = ("padre", "madre", "figlio", "figlia");'
```

Nozioni per uso del Terminale: Variabili (7)

Esempio per esplicitare differenza tra var locali e var d'ambiente: ./var_caller.sh

var_caller2.sh

```
echo "caller"  
# setto la var locale PIPPO  
# la var d'ambiente PATH esiste già
```

```
PIPPO=ALFA  
echo "PATH = ${PATH}"  
echo "PIPPO = ${PIPPO}"
```

```
echo "calling subshell"
```

```
source ./var_called2.sh
```

```
echo "ancora dentro caller"  
echo "variabili sono state modificate ?"
```

```
echo "PATH = ${PATH}"  
echo "PIPPO = ${PIPPO}"  
echo "NUOVA= ${NUOVA}"
```

var_called2.sh

```
echo "called"  
echo "le variabili sono state passate ? "
```

```
echo "PATH = ${PATH}"  
echo "PIPPO = ${PIPPO}"
```

```
echo "modifico variabili "
```

```
PIPPO="${PIPPO}:MODIFICATO"  
PATH="${PATH}:MODIFICATO"
```

```
echo "creo nuova variabile"  
NUOVA=NUOVOCONTENUTO
```

```
echo "PATH = ${PATH}"  
echo "PIPPO = ${PIPPO}"  
echo "NUOVA= ${NUOVA}"
```

```
echo "termina called"
```

```
# che accade se DEcommento qui sotto?  
# exit 13
```

Nozioni per uso del Terminale: Variabili (8)

Creazione di variabili d'ambiente da passare a subshell e Campo di visibilità di variabili

È possibile definire una o più variabili nell'ambiente (nella subshell) di un eseguibile che si sta per fare eseguire, senza farle ereditare a quelli successivi, semplicemente scrivendo le assegnazioni prima del nome del comando.

Per esempio:

```
var="stringa" comando          # comando vede e puo' usare var  
echo ${var}                      #echo non vede la variabile var
```

Attenzione, se esiste già una variabile d'ambiente con quello stesso nome, il comando lanciato vede solo la variabile nuova, alla fine del comando torna visibile la variabile vecchia.

Per esempio:

```
export var="contenutoiniziale"  
var="stringa" comando          # comando vede var che vale "stringa"  
echo ${var}                      #echo vede la variabile che vale "contenutoiniziale"
```

Nozioni per uso del Terminale: Variabili (9)

Esempio per esplicitare creazione di variabili da passare a subshell

Notare che tali variabili diventano automaticamente **di ambiente per la subshell**

```
# init_var_caller3.sh
#!/bin/bash

VAR="contenutoiniziale"
# non cambia se lascio VAR locale
# o se la imposto come variab. di ambiente
# export VAR

echo "VAR = ${VAR}"

VAR="stringa" ./init_var_called.sh

# non cambia se lo chiamo con source
# provarlo anche cosi'
# VAR="stringa" source /init_var_called.sh

echo "VAR = ${VAR}"
```

```
# init_var_called.sh
#!/bin/bash

echo "la variabile e' stata passata ? "
echo "VAR = ${VAR}"

# Verifico se nuova VAR e' di ambiente
echo "env | grep VAR"
env | grep VAR
```

Nozioni per uso del Terminale: Variabili (10)

Variabili Vuote vs. Variabili non esistenti

C'e' differenza tra
una variabile che esiste ma e' vuota
ed una variabile che invece non esiste.

- Se una variabile non e' mai stata dichiarata allora non esiste.
 - Se a una variabile e' stato assegnato come valore la stringa vuota "" allora esiste ma e' vuota.

Posso eliminare una variabile esistente (vuota o no) col comando **unset**
unset nomevariabile

Dopo averla eliminata, quella variabile non esiste.

Attenzione, che una variabile vuota o non esistente puo' provocare errori sintattici a runtime negli script in cui si confronta il valore delle variabili.

[\$var = ""] errore sintattico se \$var e' vuota oppure se non esiste
 e' come se ci fosse scritto [= ""]

Occorre sempre quotare il nome della variabile

["\$var" = ""] corretto sintatticamente se \$var e' vuota e anche se var non esiste
[\$var = ""] SBAGLIATO se var vuota o non esiste, come se fosse scritto [= ""

```
[ -n "$var" ]      # corretto , restituisce true se var non vuota, false se var vuota  
[ -n $var ]        # SBAGLIATO perche' restituisce vero se var non esiste
```

Nozioni per uso del Terminale: funzionalita' **history** **history expansion**

La funzionalita' **history** memorizza i comandi lanciati dalla shell e permette di visualizzarli ed eventualmente rilanciarli. La storia dei comandi viene mantenuta anche dopo la chiusura della shell.

Il comando built-in **history** visualizza un elenco numerato dei comandi precedentemente lanciati dalla shell, con numero crescente dal comando più vecchio verso il più recente. Il numero assegnato ad un comando quindi non cambia quando lancio altri comandi. Il numero serve a rilanciare quel comando.

<i>più vecchio</i>	180	ls
	181	./sizeof_vettore.exe
	182	clear
	183	cat sizeof_pached.c
	184	gcc -ansi -o sizeof_packed.exe sizeof_packed.d
<i>più recente</i>	185	./sizeof_packed.exe

Se lancio il comando **!NUMERO** eseguo il comando che nella history è indicato col numero NUMERO. Se ad esempio lancio

!181

allora eseguo **./sizeof_vettore.exe**

Se lancio il comando **!stringa** allora eseguo il comando più recentemente lanciato (cioè quello col numero più grande) che nella history inizia con stringa. Se ad esempio lancio

!c

allora eseguo **cat sizeof_pached.c**

Nozioni per uso del Terminale: Comando **set**

Il comando built-in **set** svolge diversi compiti:

set lanciato senza nessun parametro visualizza tutte le variabili della shell, sia quelle locali che quelle d'ambiente. Inoltre visualizza le funzioni implementate nella shell (le vedremo più avanti (forse)).

set lanciato con dei parametri serve a settare o resettare una opzione di comportamento della shell in cui viene lanciato.

Ad esempio

set +o history disabilita la memorizzazione di ulteriori comandi eseguiti nel file di history. I comandi lanciati prima della disabilitazione rimangono nel file di history.

set -o history abilita la memorizzazione dei comandi eseguiti mettendoli nel file di history.

set -a causa il fatto che le variabili create o modificate vengono immediatamente fatte diventare variabili d'ambiente e vengono ereditate dalle eventuali shell figlie.

set +a causa il fatto che le variabili create sono variabili locali. È il modo di default.

Vedere il comando **man set** per leggere tutte le possibili opzioni di comportamento della shell che possono essere attivate o disattivate col comando **set**.

Se il man non trova le info su set, installate il pacchetto manpages-posix-dev

Nozioni per uso del Terminale: Comando **set**

Risposta per Tarlo6 - Creare variabili locali dopo 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**

Parametri a riga di comando passati al programma (1)

Cosa sono gli argomenti o parametri a riga di comando di un programma

Sono un insieme ordinato di caratteri, separati da spazi, che vengono passati al programma che viene eseguito in una shell, nel momento iniziale in cui il programma viene lanciato.

Quando da una shell digito un comando da eseguire, gli argomenti devono perciò essere digitati assieme (e di seguito) al nome del programma per costituire l'ordine da dare alla shell. Ciascun argomento è separato dagli altri mediante uno o più spazi (carattere blank, la barra spaziatrice della tastiera).

Esempio: chmod u+x /home/vittorio/primo.exe

L'intera stringa di caratteri **chmod u+x /home/vittorio/primo.exe** si chiama **riga di comando**

Il primo pezzo **chmod** è il nome dell'eseguibile (o anche **argomento di indice zero**). Ciascuno degli altri pezzi è un argomento o parametro.

Nell'esempio, il pezzo **u+x** è **l'argomento di indice 1**.

Nell'esempio, il pezzo **/home/vittorio/primo.exe** è **l'argomento di indice 2**.

Nell'esempio, si dice che il numero degli argomenti passati è **2**.

Il programma, una volta cominciata la propria esecuzione, ha modo di conoscere il numero degli argomenti che gli sono stati passati e di ottenere questi argomenti ed utilizzarli.

Avvio della shell bash

La shell bash si comporta in maniera diversa a seconda di quali argomenti a riga di comando le vengono passati nel momento in cui ne viene lanciata l'esecuzione.

Distinguiamo 3 modi in cui può essere eseguita la bash:

shell non interattiva

shell figlia che esegue script

lanciata con argomenti **-c** percorso_script_da_eseguire

shell interattiva NON di login

quella 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**

shell bash interattiva "di login" o "non di login"

La shell interattiva bash si comporta inizialmente in due modi differenti a seconda di come viene lanciata cioe' a seconda di quali argomenti a riga di comando le vengono passati.

Shell bash interattiva di login: quando viene lanciata **con** una delle opzioni **-l** oppure **--login**.

La shell interattiva **di login**, nel momento in cui comincia la sua esecuzione, cerca di eseguire (se esistono) i seguenti files:

1. il file "/etc/profile"
2. poi uno solo (il primo che trova) tra i file ".bash_profile", ".bash_login" e ".profile" collocati nella home directory dell'utente e che risulti essere disponibile;
3. poi il file ".bashrc" collocato nella *home directory* dell'utente;

Nel momento in cui termina, la shell di login esegue il file .bash_logout collocato nella home dell'utente (se il file esiste).

Shell bash interattiva non di login: quando viene lanciata **senza** alcuna delle due opzioni **-l** oppure **--login**.

La shell interattiva **non di login**, nel momento in cui comincia la sua esecuzione, cerca di eseguire (se esiste) il solo file ".bashrc" collocato nella *home directory* dell'utente.

Per entrambe le shell, l'utente puo' modificare i file di configurazione nella propria home directory, per customizzare il comportamento della shell.

shell bash "non interattiva"

La shell non interattiva bash è la shell lanciata per eseguire uno script.

Viene specificato un argomento –c nomescriptdaeseguire.

A differenza della shell interattiva, la shell non interattiva NON esegue i comandi contenuti in nessuno dei file (di sistema o dell'utente) che customizzano il comportamento della shell interattiva (/etc/profile .bash_profile .bash_login .profile .bashrc).

Parametri a riga di comando passati al programma (2)

Separatore di comandi ; e Delimitatore di argomenti "

Problema: Comandi Multipli su una stessa riga di comando

La bash permette che **in una stessa riga di comando possano essere scritti più comandi**, che verranno eseguiti uno dopo l'altro, **non appena termina un comando viene eseguito il successivo**.

Il carattere **;** è il **separatore tra comandi**, che stabilisce cioè dove finisce un comando e dove inizia il successivo.

Ad es: se voglio stampare a video la parola pippo e poi voglio cambiare la directory corrente andando in /tmp potrei scrivere in una sola riga i due seguenti comandi:

```
echo pippo ; cd /tmp
```

Diventa però impossibile stampare a video frasi strane tipo: pippo ; cd /bin che contiene purtroppo il carattere di separazione ;

Per farlo si include la frase completa tra doppi apici. Infatti, **il doppio apice " (double quote) serve a delimitare l'inizio e la fine di un argomento, così che il punto e virgola viene visto non come un delimitatore di comando ma come parte di un argomento.**

```
echo "pippo ; cd /tmp "
```

In questo modo si **visualizza** la frase **pippo ; cd /tmp** e non si cambia directory

Parametri a riga di comando passati al programma (3)

Quoting di stringhe " " e Quoting di singoli caratteri \

Consideriamo ancora il comando appena visto **echo " pippo ; cd /tmp "**

NOTA BENE 1:

Oltre al delimitatore di argomenti “ esiste anche il delimitatore ‘
Vedremo la differenza piu’ avanti parlando di substitution.

NOTA BENE 2:

Delimitandola con un delimitatore, la stringa " pippo ; cd /tmp" appare al comando echo come un unico argomento.

NOTA BENE 3:

E' possibile disabilitare l'interpretazione del separatore ; facendolo precedere da un ulteriore carattere speciale detto carattere di escape \

In tal modo, il comando `echo pippo \\; cd /tmp`

visualizzera' pippo ; cd /tmp

NOTA BENE 4: ESCAPE per QUOTING di singoli caratteri speciali

In generale, precedendo un carattere speciale con il carattere \ (quoting di singolo carattere) si ottiene che il carattere speciale verrà utilizzato come un carattere qualunque, senza la sua speciale interpretazione da parte della shell.

Brace Expansion - generazione stringhe (1)

concetto di base

Quando passo alla bash una riga di comando da eseguire, la bash per prima cosa cerca di capire se nella riga sono presenti delle coppie di parentesi graffe che rappresentano un ordine di generare delle stringhe secondo delle regole.

- Un ordine di brace expansion è **una stringa di testo**,
 - racchiusa tra separatori quali spazi, tab o andate a capo
 - e al cui interno compaiono una coppia di graffe non precedute da un \$,
 - e al cui interno non ci sono spazi bianchi o tab.
- Questa stringa che ordina una brace expansion è formato da tre parti, di cui la prima (preambolo) e l'ultima (postscritto) possono non essere presenti.
- La parte di mezzo è quella racchiusa all'interno di una coppia di parentesi graffe.
- Dentro le graffe sono presenti una o più stringhe separate da virgole.
- **Ciascuna stringa rappresenta una possibile scelta di stringhe che possono essere aggiunte al preambolo e seguite dal postscritto per formare delle nuove stringhe di testo.**

Ad esempio, se la riga di comando è fatta così:

```
echo va{acaga,ffancu,ammori,catihafat}lo
```

il comando viene espando in questo modo

```
echo vaacagalo vaffanculo vaammorilo vacatihofatlo
```

ottenendo il gentile output

vaacagalo vaffanculo vaammorilo vacatihofatlo **71**

Brace Expansion - generazione stringhe (2)

spazi bianchi nelle stringhe

Possono mancare preambolo o postscritto o entrambi

a{bb,cc,ddd} diventa abb acc addd

Non possono essere presenti spazi bianchi o tab altrimenti la brace expansion non viene riconosciuta e non viene applicata l'espansione.

Il comando: echo a{bb,cc, dd}ee

ottiene questo output: a{bb,cc, dd}ee

Analogamente, il comando echo a{bb,cc,d d}ee

ottiene questo output : a{bb,cc,d d}ee

Se voglio far generare stringhe con spazi bianchi, devo proteggere i singoli spazi con dei caratteri backslash

Ad esempio, echo a{bb,cc,d\ d}ee

ottiene questo output: abbee accee ad dee

Ad esempio, echo aa\ a{bb,cc,dd}ee

ottiene questo output: aa abbee aa accee aa addee

Non posso proteggere tutto l'ordine di brace expansion con doppi apici perché disabilitano l'interpretazione dell'espansione, ma posso proteggere ciascuna singola stringa (quella tra due virgole)

Il comando echo "a{bb,cc,d d}ee" ottiene in output

a{bb,cc,d d}ee

mentre il comando echo a{bb,cc,"d d"}ee ottiene in output

abbee accee ad dee

Brace Expansion - generazione stringhe (3) annidamento e variabili

Annidamento delle brace expansion

E' possibile inserire degli ordini di brace expansion all'interno di altri ordini di brace expansion.

Il comando

```
echo /usr/{ucb/{ex,edit},lib/{bin,sbin}}
```

ottiene questo output

```
/usr/ucb/ex /usr/ucb/edit /usr/lib/bin /usr/lib/sbin
```

E' possibile inserire più livelli di annidamento delle brace expansion

Variabili nelle brace expansion

E' possibile inserire delle variabili negli ordini di brace expansion
all'interno di altri ordini di brace

A=bin

B=log

C=boot

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

visualizza

```
binlogboota binboota binbinloga
```

Brace Expansion - generazione stringhe (4)

brace expansion con Sequence Expression

Esiste una particolare forma di brace expansion che consente di generare stringhe elencando un intervallo di caratteri che possono essere usati come possibili scelte. L'intervallo viene indicato mediante i due estremi connessi con due punti

Il comando
ottiene questo output

```
echo a{b..k}m  
abm acm adm aem afm agm ahm aim ajm akm
```

Il comando
ottiene questo output

```
echo a{4..7}m  
a4m a5m a6m a7m
```

E' possibile annidare queste sequence expression all'interno di brace expansion
annidate

Il comando
ottiene questo output

```
echo a{b,c{4..7}}m  
abm ac4m ac5m ac6m ac7m
```

NB: Esiste una forma più complessa di sequence expression, in cui si può stabilire un incremento non unitario nell'intervallo di caratteri, ma questa forma non è supportata in tutte le versioni della bash, quindi non ne parliamo.

Tilde Expansion ~

La tilde expansion riguarda 5 casi essenziali:

1. un carattere tilde isolato
 2. un carattere tilde seguito da slash non quotato
 3. un carattere tilde seguito da slash non quotato seguito da altri caratteri
- In questi casi la tilde viene sostituita dal percorso assoluto della home directory dell'utente che sta eseguendo la riga di comando, l'effective user.

Supponendo che l'utente che esegue i comandi sia l'utente vittorio

cd ~	cambia la directroy corrente in	/home/vittorio
echo ~/	visualizza	/home/vittorio/
echo ~/vaff	visualizza	/home/vittorio/vaff

4. una parola che inizia con la tilde seguita da un nome utente
 5. una parola che inizia con la tilde seguita da un nome utente, seguita da slash non quotato a cui possono seguire altri caratteri
- In questi casi la tilde più nome utente viene sostituita dal percorso assoluto della home directory dell'utente specificato dal nome.

Supponendo che esista un utente panzieri e che a eseguire sia vittorio

echo ~panzieri	visualizza	/home/panzieri
echo ~panzieri/ciao	visualizza	/home/panzieri/ciao

Se l'utente specificato non esiste, l'espansione non viene fatta, stringa invariata

echo ~panzieriNonEsiste	visualizza	~panzieriNonEsiste
-------------------------	------------	--------------------

Wildcards * ? [...] Pathname substitution

I Metacaratteri * e ? sono caratteri che vengono inseriti dall'utente nei comandi digitati e che la shell interpreta cercando di sostituirli con una sequenza di caratteri per ottenere i nomi di files nel filesystem

Con cosa sono sostituiti?

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

? può essere sostituito da esattamente un singolo carattere.

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

Esempi: usiamo il comando **ls** che visualizza i nomi dei file nella directory specificata.

Nessuna sostituzione

ls /home/vittorio visualizza i nomi di tutti i file della directory

ls /home/vittorio/primo.c visualizza il nome del solo file primo.c

Sostituzione di * con una qualunque sequenza di caratteri, anche vuota, che permetta di ottenere il nome di uno o più file

`ls /home/vittorio/*.exe` visualizza il nome di quei file della directory vittorio

il cui nome termina per .exe (cioè primo.exe)

`ls /home/vittorio/primo*` visualizza i nomi di quei file della directory vittorio il cui nome inizia per primo, cioè primo.c primo.ex

Sostituzione di ? con un singolo carattere (NON vuoto),

ls /home/vittorio/pri?o.c visualizza il nome del file primo.c di directory vittorio

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
```

Wildcards [...] - esempio di uso sbagliato

Non posso annidare delle parentesi quadre se non per contenere le parole riservate

[:digit:] [:upper:] [:lower:]

in una directory che contiene i file akB akmB a1B a2B

se scrivo

```
ls a[[:lower:][:digit:]]B  
visualizzo akB a1B a2B
```

se scrivo

```
ls a[[:lower:][12]]B
```

le parentesi quadre [12] sono interne ad altre parentesi quadre, quindi non vengono sostituite con niente, ed il comando risponde:

```
ls: cannot access a[[:lower:][12]]B: No such file or director
```

Comandi della bash ed eseguibili utili

Comandi

Comandi ed eseguibili

pwd cd mkdir rmdir ls rm echo cat set mv ps sudo du kill bg fg
read wc killall

Istruzioni di controllo di flusso

for do done while do done if then elif then else fi

Espressione condizionale (su file o su variabile)

[condizione di un file]

Valutazione di espressione matematica applicata a variabili d'ambiente ((istruzione con espressione))

Eseguibili binari forniti dal sistema operativo

editor interattivi
vi nano (pico)

utilità

man more less grep find tail head cut tee ed tar gzip diff patch gcc make

Comandi della bash (in ordine di importanza)

pwd	mostra directory di lavoro corrente .
cd <u>percorso directory</u>	cambia la directory di lavoro corrente .
mkdir <u>percorso directory</u>	crea una nuova directory nel percorso specificato
rmdir <u>percorso directory</u>	elimina la directory specificata, se è vuota
ls -alh <u>percorso</u>	stampa informazioni su tutti i files contenuti nel percorso
rm <u>percorso file</u>	elimina il file specificato
echo <u>sequenza di caratteri</u>	visualizza in output la sequenza di caratteri specificata
cat <u>percorso file</u>	visualizza in output il contenuto del file specificato
env	visualizza le variabili ed il loro valore
which <u>nomefileeseguibile</u>	visualizza il percorso in cui si trova (solo se nella PATH) l'eseguibile
mv <u>percorso file</u> <u>percorso nuovo</u>	sposta il file specificato in una nuova posizione
ps aux	stampa informazioni sui processi in esecuzione
du <u>percorso directory</u>	visualizza l'occupazione del disco.
kill -9 <u>pid processo</u>	elimina processo avente identificativo pid_processo
killall <u>nome processo</u>	elimina tutti i processi con quel nome nome_processo
bg	ripristina un job fermato e messo in sottofondo
fg	porta il job più recente in primo piano
df	mostra spazio libero dei filesystem montati
touch <u>percorso file</u>	crea il file specificato se non esiste, oppure ne aggiorna data.
more <u>percorso file</u>	mostra il file specificato un poco alla volta
head <u>percorso file</u>	mostra le prime 10 linee del file specificato
tail <u>percorso file</u>	mostra le ultime 10 linee del file specificato
man <u>nomecomando</u>	è il manuale, fornisce informazioni sul comando specificato
find	cercare dei files
grep	cerca tra le righe di file quelle che contengono alcune parole
read <u>nomevariabile</u>	legge input da standard input e lo inserisce nella variabile specificata
wc	conta il numero di parole o di caratteri di un file
true	restituisce exit status 0 (vero)
false	restituisce exit status 1 (non vero)

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 a[:digit:]B 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 a[:digit:]B and the date is: `date`
```

Quotes: command substitution & escape (2)

esempi

```
echo Dear ${USER} the date is: `date` ; echo sei in ritardo  
Dear vittorio the date is: Tue, Sep 15, 2015 12:58:21 PM  
sei in ritardo
```

```
echo " Dear ${USER} the date is: `date` ; echo sei in ritardo "  
Dear vittorio the date is: Tue, Sep 15, 2015 12:58:07 PM ; echo sei in ritardo
```

```
echo ' Dear ${USER} the date is: `date` ; echo sei in ritardo '  
      Dear ${USER} the date is: `date` ; echo sei in ritardo
```

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.

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 condizionali 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.

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 -le -lt -ge -gt**), ma anche confronti non aritmetici (specificando altri operatori).

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

Posso anche verificare se delle stringhe sono vuote o no (con gli operatori **-z -n**).

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

Posso **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

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
- può comporre espressioni aritmetiche mediante operatori logici && (AND) || (OR) e ! (NOT).
- non può contenere espressioni condizionali
- non può contenere comandi

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

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

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

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 .

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.

Espressioni CONDIZIONALI (7)

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

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

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

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.

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 ;
```

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 1
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.

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
```

Evitare Errori Più Comuni (1)

**METTETE SPAZI PRIMA E DOPO le [[e le]]
ed anche PRIMA E DOPO gli operatori (es. -)**

Ad esempio, l'espressione condizionale seguente

```
if [[-d /usr/include/stdio.h ]] ; then echo esiste ; fi
```

provoca errore, perché la bash crede che [[-d sia un comando da eseguire, infatti avvisa:

[-d: command not found

perché MANCA LO SPAZIO TRA le [[e il -d

Infatti, **il parser della bash riconosce le espressioni condizionali prima separando la riga in parole delimitate da spazi. Se mancano gli spazi le espressioni condizionali non vengono riconosciute.**

Analogamente per gli spazi prima e dopo qualunque operatore, aritmetico, lessicografico, su file, insomma tutti.

Il comando

```
if [[ -dFILECHENONESISTE ]] ; then echo esiste ; fi
```

esegue e mette in output esiste.

Sembra dire che FILECHENONESISTE esiste perché senza spazi viene valutata (E NON ESEGUITA) la stringa -dFILECHENON ESISTE che essendo non vuota restituisce vero.

Evitare Errori Più Comuni (2)

Cosa posso mettere dentro un if ?
Liste di Comandi o Espressioni condizionali

OK if ls prova ; then echo ciao ; else echo fanculo ; fi
OK, ma strano if `ls prova` ; then echo ciao ; else echo fanculo ; fi
OK if [[-e prova]] ; then echo ciao ; else echo fanculo ; fi

Cosa posso mettere dentro le [[]] ?
solo gli operatori delle espressioni condizionali,
NON dei comandi,
se non come command substitution negli operandi

OK if [["aaa" == `cat prova`]] ; then echo ciao ; else echo fanculo ; fi
OK if [[-e prova]] ; then echo ciao ; else echo fanculo ; fi
NOOO if [[ls prova]] ; then echo ciao ; else echo fanculo ; fi
OK if [[-e `ls pro?a.c`]] ; then echo esiste prova.c ; fi

Evitare Errori Più Comuni (3)

NO parentesi tonde tra le [] semplici

OK

```
if [[ ( $var ne 13 || $var -ne 17 ) && -d /tmp ]] then echo ciao ; fi
```

Syntax error! NO! if [(\$var ne 13 -o \$var -ne 17) -a -d /tmp] then echo ciao; fi

**SI composizioni logiche dentro
a valutazioni aritmetiche (())**

OK

```
if (( $var < 13 || $var > 17 )) ; then echo ciao ; fi
```

**SI composizioni logiche dentro
a espressioni condizionali [[]]**

ma con operatori -a -o senza tonde () in []

OK

```
if [[ $var -lt 13 || $var > 17 ]] then echo ciao ; fi
```

Espressioni CONDIZIONALI (9)

NOTA BENE:

RIBADISCO CHE NON POSSO eseguire dei COMANDI all'interno dell'operatore espressione condizionale. Posso mettere solo delle espressioni con operatori di confronto o di condizioni su file oppure delle command substitution.

Cioe' **non posso eseguire** un comando (ad esempio come quello che segue) perche' il comando specificato tra le parentesi quadre semplicemente NON VIENE ESEGUITO poiche' **all'interno delle espressioni condizionali non e' previsto che venga eseguito un comando**. Diverso è il caso delle command substitution.

NO !!!!!!! ERROR !!!!!!

```
if [[ ls nomefile.txt ]] ; then echo "il file esiste" ; else echo "il file non esiste" ; fi
```

Se occorre eseguire un comando e valutarne il risultato, NON DEVO METTERE il comando all'interno di una espressione condizionale ma devo metterlo semplicemente come condizione di un if, cosi':

SI

```
if ls nomefile.txt ; then echo "il file esiste" ; else echo "il file non esiste" ; fi
```

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)

\e

\f form feed

\r carriage return

\v vertical tab

\' single 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.

\b backspace

\E an escape character

\n new line

\t horizontal tab

\\" backslash

\\" double quote

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

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.

Es: 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

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)) ;

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 troverò 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.

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 cifre 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
```

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 piu' 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 e' assegnato il valore vuoto. Esistono alcune opzioni interessanti **-u -N -r** ed altre meno necessarie **-e -n -t**. Il risultato restituito da read e' 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=""
```

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 bianchi 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 NON 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**"

read - Lettura di un numero specificato di caratteri (6)

L'opzione **-N** della **read** permette di specificare il numero 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.

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 partirà dalla riga successiva.

Se una read chiede meno caratteri di quelli presenti nella riga 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.

Esempi di comandi e i loro output

```
vic@vic:~$ cat miooutput.txt
messaggio1
messaggio2
vic@vic:~$ exec 103< miooutput.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 $RIGA
aggi
vic@vic:~$ read -N 4 -u 103 RIGA
vic@vic:~$ echo $RIGA
o2
vic@vic:~$ read -N 4 -u 103 RIGA
vic@vic:~$ echo $?
1
vic@vic:~$ echo $RIGA

vic@vic:~$ exec 103<&
vic@vic:~$
```

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 utilizzarne 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	exec n< PercorsoFile	exec {NomeVar}< PercorsoFile
Scrittura	exec n> PercorsoFile	exec {NomeVar}> PercorsoFile
Aggiunta in coda	exec n>> PercorsoFile	exec {NomeVar}>> PercorsoFile
Lettura e Scrittura	exec n<> PercorsoFile	exec {NomeVar}<> PercorsoFile

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

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
```

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/
```

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 puo' 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

Ridirezionamenti di Stream di I/O (0)

1) RIDIREZIONAMENTO DI FILE DESCRIPTOR DI PROCESSI FIGLI

Quando lanciamo un comando o processo o script all'interno di una bash il processo figlio ottiene una copia di tutti i file descriptor del padre, quindi lavora con gli stessi stream aperti del padre.

Però il padre, nel momento in cui lancia il processo figlio, può decidere di cambiare gli stream da far usare al figlio, associando, a ciascun file descriptor da passare al figlio, uno stream diverso.

In tal caso, i file descriptor passati al figlio avranno lo stesso identificatore numerico di quelli del padre, ma saranno associati a stream diversi.

Il padre continuerà ad usare i suoi file descriptor associati ai vecchi stream.

2) AUTO-RIDIREZIONAMENTO , ovvero RIDIREZIONAMENTO DI PROPRI FILE DESCRIPTOR

Un processo può decidere di aprire nuovi file, chiudere file aperti o di associare un proprio file descriptor ad un altro stream (ridirezionamento).

In quest'ultimo caso, i file descriptor ridirezionati mantengono il loro valore ma sono associati a stream diversi. Da quel momento in avanti, il processo usando i vecchi file descriptor accederà ai nuovi stream.

Apertura chiusura e ridirezionamento di propri file sono effettuati mediante il comando exec

Ridirezionamenti di Stream di I/O (1)

Ridirezionamenti:

- < ricevere input da file.
- > mandare std output verso file eliminando il vecchio contenuto del file
- >> mandare std output verso file aggiungendolo al vecchio contenuto del file
- | ridirigere output di un programma nell' input di un altro programma

Ricevere input da file.

L'utente puo' utilizzare lo standard input di un programma per dare input non solo da tastiera ma anche da file, **ridirezionando il contenuto di un file sullo standard input del programma, al momento dell'ordine di esecuzione del programma**:

program < nome_file_input

il programma vedrà il contenuto del file nome_file_input come se venisse digitato da tastiera.

Emulare la fine del file di input da tastiera: **Ctrl+D**

Notare che usando il ridirezionamento dell'input, c'è un momento in cui tutto il contenuto del file di input è stato passato al programma ed il programma si accorge di essere arrivato alla fine del file.

Se invece non faccio il ridirezionamento dell'input e fornisco l'input da tastiera, non incontro mai una fine del file di input. Per produrre da tastiera lo stesso effetto della terminazione del file di input devo digitare contemporaneamente i tasti CTRL e D che inviano un carattere speciale che indica la fine del file.

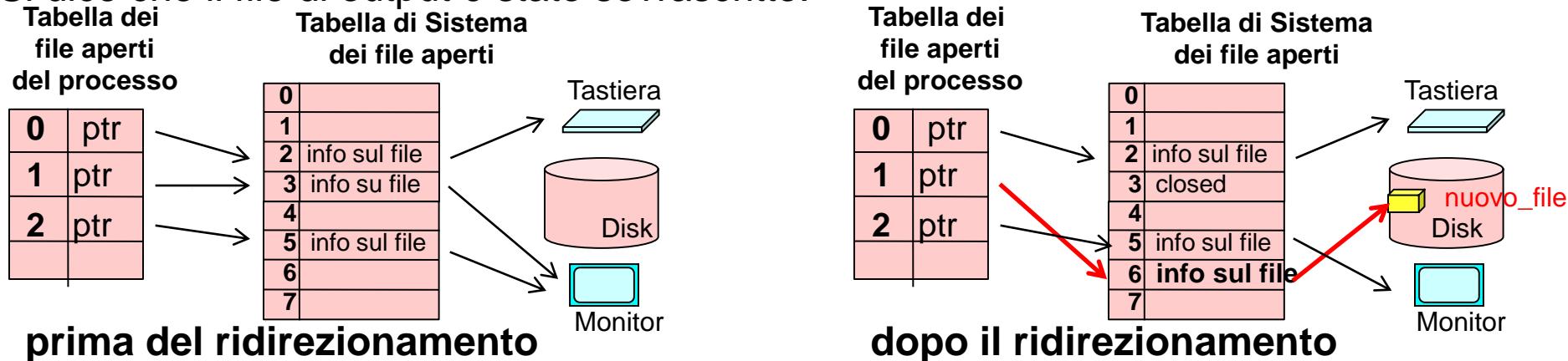
Ridirezionamenti di Stream di I/O (2)

Analogamente **lo standard output** di un programma può essere **ridirezionato su un file**, su cui sarà scritto tutto l'output del programma invece che su video

program > nome_file_output

Nel file nome_file_output troveremo i caratteriche il programma voleva mandare a video.
Il precedente contenuto del file verrà perso ed **alla fine dell'esecuzione del programma nel file troveremo solo l'output generato dal programma stesso.**

Si dice che il file di output è stato sovrascritto.



Il ridirezionamento dello standard output può essere fatto senza eliminare il vecchio contenuto del file di output bensì mantenendo il vecchio contenuto ed **aggiungendo il coda al file l'output del programma**

program >> nome_file_output

Ridirezionamenti di Stream di I/O (3)

Supponiamo che un programma *program* mandi in output le due seguenti righe:

 pippo

 pappa

Supponiamo che il file *nome_file_output* contenga queste tre righe:

 uno

 due

 tre

Se eseguiamo il programma ordinando, con il ridirezionamento, di aggiungere l'output in coda al file *nome_file_output*, così:

program >> nome_file_output

alla fine dell'esecuzione il contenuto del file *nome_file_output* sarà:

 uno

 due

 tre

 pippo

 pappa

Se eseguiamo il programma ridirezionando l'output sul file *nome_file_output*, così:

program > nome_file_output

alla fine dell'esecuzione il contenuto del file *nome_file_output* sarà solo:

 pippo

 pappa

Ridirezionamenti di Stream di I/O (4)

I due ridirezionamenti (input ed output) possono essere fatti **contemporaneamente**

program < nome_file_input > nome_file_output

O analogamente

program > nome_file_output < nome_file_input

In questo modo, il contenuto del nome_file_input viene usato come se fosse l'input da tastiera per il programma program, e l'output del programma viene scritto nel file nome_file_output.

Inoltre, in bash si può **ridirezionare assieme standard output e standard error su uno stesso file**, sovrascrivendo il vecchio contenuto:

program &> nome_file_error_and_output

Infine, in bash si può **ridirezionare standard ouput e standard error su due diversi file**, sovrascrivendo il vecchio contenuto:

program 2> nome_file_error > nome_file_output

Ridirezionamenti di Stream di I/O (5)

Si noti che i **ridirezionamenti non modificano il valore dei file descriptor che vengono ridiretti** e nemmeno il numero di file descriptor dello script che li effettua. Questo accade perché il sistema operativo usa proprio il valore intero del file descriptor da ridirigere per indicare il file su cui viene ridiretto.

Ad esempio, se una bash lancia uno script `fd_di_script.sh` ridirezionando entrambi gli stream di output predefiniti su uno stesso file, quello script continua a vedere i due file descriptor 1 e 2 anche se questi fanno scrivere su uno stesso file.

ad esempio, lo script `fd_di_script.sh` visualizza l'elenco dei suoi file descriptor, così:

```
#!/bin/bash  
ls /proc/$$/fd/
```

Eseguiamo lo script ridirezionando entrambi gli stream di output predefiniti su uno stesso file:

```
./fd_di_script.sh &> out.txt
```

Nel file `out.txt` vedremo che sono indicati i tre file descriptor 0 1 2 come sempre.

Ridirezionamenti di Stream di I/O (6)

Generalizzazione operatori > <

In generale,

se una bash ha uno stream (un file aperto)
e quello stream è identificato da un file descriptor di valore **N**,
allora,
è possibile ridirigere su un altro file
lo stream indicato da quel file descriptor,
usando una sintassi che specifica
il valore intero N di quel file descriptor.

N> NomeFileTarget

ridireziona il file descriptor N sul file **con nome** NomeFileTarget
come già visto, se si omette N si intende standard output.
usare **>>** per append

<N NomeFileSource

ridireziona il file con nome NomeFileSource sul file descriptor N del
programma specificato alla sinistra dell'operatore.
Come già visto, se si omette N si intende standard input.

Ridirezionamenti di Stream di I/O (7)

PIPE | Connettere due processi mediante Ridirezione di stdout su stdin

E' possibile far eseguire contemporaneamente due processi mandando lo standard output del primo nello standard input del secondo, mediante l'operatore pipe |.

```
program1 | program2
```

I due processi eseguono in contemporanea.

Quando il primo processo termina o chiude il suo standard output,
il secondo processo vede chiudersi il proprio standard input.

E' possibile connettere più processi in una sequenza di pipe

```
program1 | program2 | program3
```

Ridirezionamenti di Stream di I/O (8)

nuova bash per eseguire comandi composti in PIPE

Attenzione! In uno script dove compare una pipe, SE un comando a destra o a sinistra della pipe è un comando composto (dei loop) oppure è uno script, ALLORA QUEL COMANDO ESEGUE IN UNA SHELL BASH FIGLIA.

Guardiamo il seguente script `pipe.sh`

```
MIAVAR="iniziale"
echo prima ${MIAVAR}
ps
echo "poffarre" | while read MIAVAR ; do ps; echo dentro ${MIAVAR}; done
echo dopo ${MIAVAR}
```

eseguendo lo script `pipe.sh`, ottengo il seguente output

prima iniziale

PID TTY	TIME	CMD
1830 pts/0	00:00:00	bash
3990 pts/0	00:00:00	bash
3991 pts/0	00:00:00	ps

<- bash di shell interattiva
<- bash che esegue pipe.sh

PID TTY	TIME	CMD
1830 pts/0	00:00:00	bash
3990 pts/0	00:00:00	bash
3993 pts/0	00:00:00	bash
3994 pts/0	00:00:00	ps

<- bash di shell interattiva
<- bash che esegue pipe.sh
<- bash che esegue while

dentro poffarre

la variabile MIAVAR creata in terza bash ha assunto valore "poffarre"
MIAVAR ha mantenuto valore originale, non ha contenuto "poffarre"

dopo iniziale

Ridirezionamenti di Stream di I/O (9)

Operatore >& per ridirezionamento tra file descriptor

Se una bash ha due stream (due file aperti) entrambi di output (o entrambi di input), e ciascuno stream è identificato da un file descriptor di valore **N** ed **M** rispettivamente, allora

e' possibile ridirigere lo stream **N** sullo stream **M** mediante l'operatore

N>&M

Dopo di questo, ciò che scrivo su N viene scritto su M. Lo stream N mantiene il valore N del suo file descriptor ma, nella tabella dei file aperti del processo, il suo puntatore alla tabella di sistema viene sostituito da una copia del puntatore dello stream M

Esisteranno da quel momento due file descriptor N ed M di valore diverso ma che puntano allo stesso file aperto.

ls pippo.txt 2>&1

ridireziona lo stderr di ls sullo stdout di ls

Tabella dei
file aperti
del processo

0	ptr
N	ptr
M	ptr

Tabella di Sistema
dei file aperti

0	
1	info sui file
2	info sul file
3	
4	
5	info sul file
6	
7	

prima del ridirezionamento

dopo il ridirezionamento

0	ptr
N	ptr
M	ptr

0	
1	info sui file
2	info sul file
3	
4	
5	info sul file
6	
7	

Ridirezionamenti di Stream di I/O (10)

Attenti alla differenza tra > e >&

Attenzione a non confondersi

```
cat out.txt 2>1
```

qui 1 è visto come il nome di un file
si ridirige lo stderr di cat sul file 1
non sullo stdout di cat

```
cat out.txt 2>&1
```

qui 1 è visto come file descriptor di un file aperto
si ridirige lo stderr di cat sullo stdout di cat

Ridirezionamenti di Stream di I/O (11)

Ordine dei Ridirezionamenti con file descriptor (1)

E' importante l'ordine con cui compongo gli operatori di ridirezionamento di file descriptor. Sono eseguiti da sinistra verso destra.

ls pippo.txt 2> error.txt 1>&2

Il primo ridirezionamento 2> error.txt

apre il file error.txt e mette le info sul file aperto nella nuova posizione 6 della tabella di sistema.

Poi copia nello spazio occupato dal puntatore del file descriptor 2 l'indirizzo della riga nuova della tabella di sistema.

Quindi ora fd 2 punta al file error. txt mentre fd 1 scrive ancora su video

Il secondo ridirezionamento 1>&2

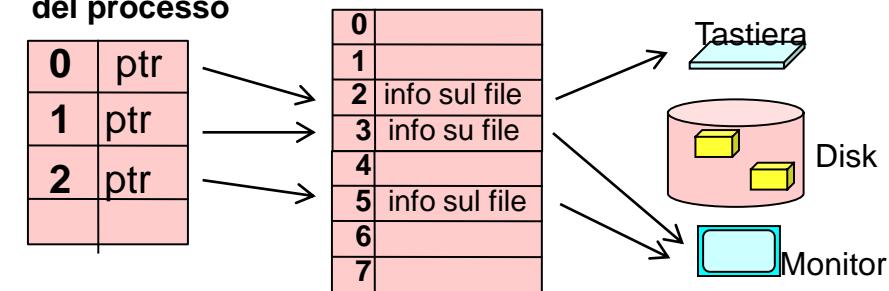
copia il puntatore del file descriptor 2 nello spazio occupato dal puntatore del file descriptor 1

Quindi ora sia fd 1 che fd 2 puntano alla stessa riga della tabella di sistema ed entrambi scrivono sul file error.txt

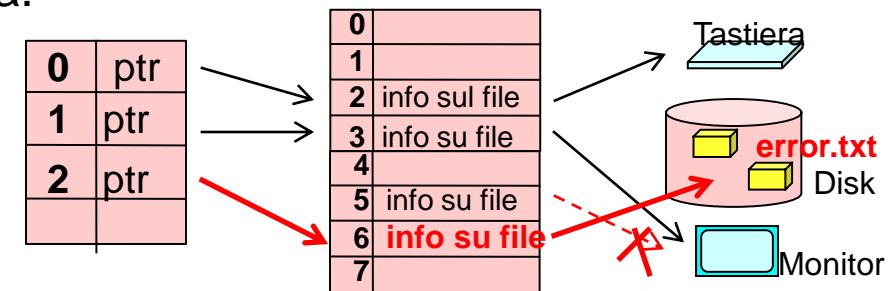
Tabella dei file aperti del processo

0	ptr
1	ptr
2	ptr

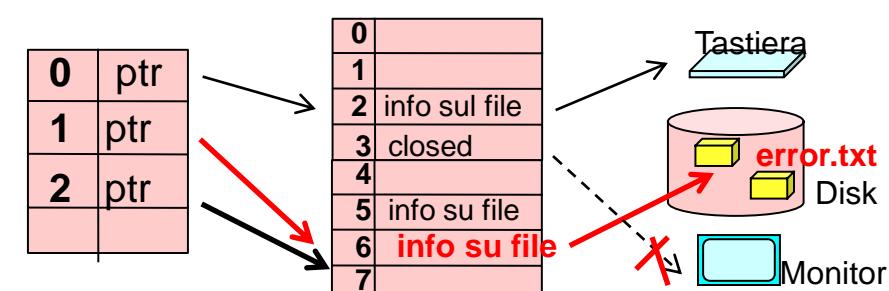
Tabella di Sistema dei file aperti



0	ptr
1	ptr
2	ptr



0	ptr
1	ptr
2	ptr



Ridirezionamenti di Stream di I/O (12)

Ordine dei Ridirezionamenti con file descriptor (2)

Scambiamo l'ordine degli operatori di ridirezionamento dell'esempio precedente.

```
ls pippo.txt 1>&2 2> error.txt
```

Il primo ridirezionamento 1>&2

copia il puntatore del file descriptor 2
nello spazio occupato dal puntatore
del file descriptor 1

Quindi ora sia fd 1 che fd 2 puntano
alla stessa riga della tabella di sistema
ed entrambi scrivono su video

Il secondo ridirezionamento 2>error.txt

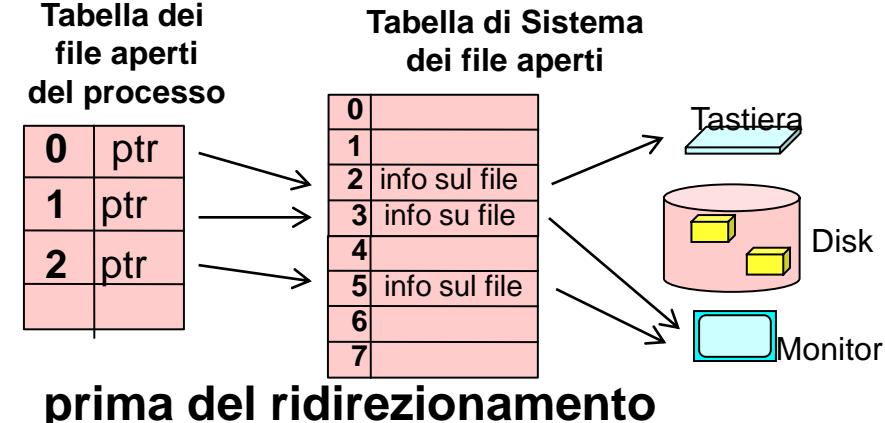
apre il file error.txt e mette le info sul file aperto
nella nuova posizione 6 della tabella di sistema.

Poi copia nello spazio occupato dal puntatore
del file descriptor 2 l'indirizzo della riga nuova
della tabella di sistema.

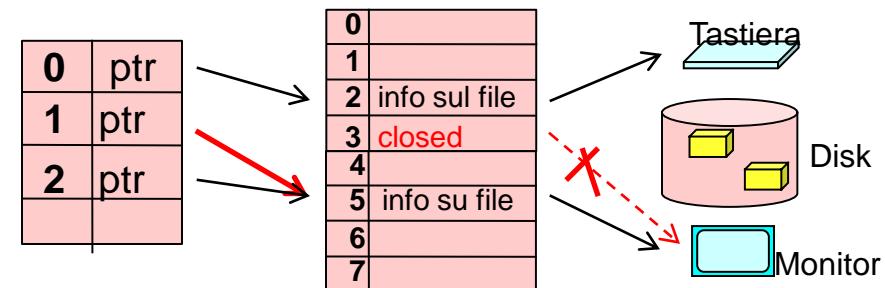
Quindi ora fd 1 scrive su video mentre
fd 2 scrive sul file error.txt

Tabella dei
file aperti
del processo

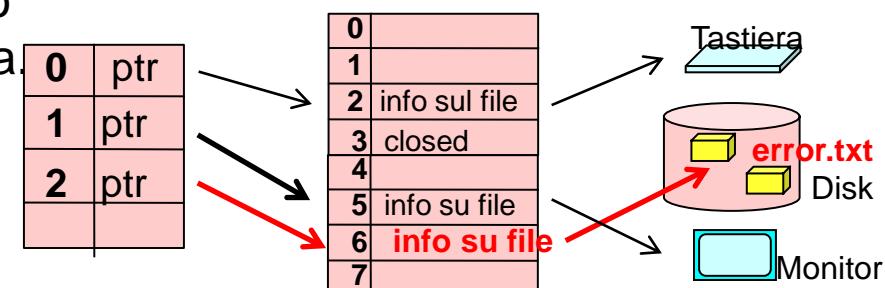
0	ptr
1	ptr
2	ptr



0	ptr
1	ptr
2	ptr



0	ptr
1	ptr
2	ptr



Ridirezionamenti di Stream di I/O (13)

Ordine dei Ridirezionamenti con file descriptor (3)

Confrontiamo i risultati dei due precedenti ridirezionamenti in cui cambia solo l'ordine. Si esegue da sinistra a destra.

ls pippo.txt 2> error.txt 1>&2

ridireziona stderr di ls sul file error.txt e poi ridireziona stdout di ls su stderr di ls che attualmente punta su error.txt,
quindi **entrambi stdout e stderr di ls finiscono nel file error.txt**

invece

ls pippo.txt 1>&2 2> error.txt

ridireziona stdout di ls su stderr di ls che attualmente punta a video
ridireziona stderr di ls su error.txt
quindi alla fine **stderr punta ad error.txt**
mentre stdout scrive su video

Tabella dei file aperti del processo

0	ptr
1	ptr
2	ptr

Tabella di Sistema dei file aperti

0	
1	
2	info sul file
3	info su file
4	
5	info sul file
6	
7	

Tastiera



Disk



Monitor

prima del ridirezionamento

0	ptr
1	ptr
2	ptr

0	
1	
2	info sul file
3	closed
4	
5	closed
6	info su file
7	

Tastiera



Disk



Monitor

dopo il ridirezionamento

0	ptr
1	ptr
2	ptr

0	
1	
2	info sul file
3	closed
4	
5	info su file
6	info su file
7	

Tastiera



Disk



Monitor

dopo il ridirezionamento

Ridirezionamenti di Stream di I/O (14)

Esempio: ridirigere il solo stderr sullo stdin di un altro programma

Voglio ridirigere il solo stderr del comando ls, usandolo come stdin del comando grep e buttando via lo stdout di ls stesso.

```
ls nonesiste.txt 2>&1 > /dev/null | grep such
```

ridireziona stderr di ls su stdout di ls,
ridireziona stdout di ls su /dev/null facendo buttare via l'output
passa il nuovo stdout di ls come stdin di grep
quindi il vecchio stderr di ls finisce nello stdin di grep

Notare le differenze tra il precedente ridirezionamento e quello qui sotto, in cui l'ordine dei due ridirezionamenti è invertito:

```
ls nonesiste.txt > /dev/null 2>&1 | grep such
```

ridireziona stdout di ls su /dev/null
ridireziona stderr di ls su stdout di ls che ora punta a /dev/null
quindi entrambi stdout e stderr di ls finiscono nel device /dev/null
Al comando grep non arriva niente di niente.

Ridirezionamenti di Stream di I/O (15)

Scorciatoia |& invece di composizione 2>&1 |

Come visto nella pagina precedente, il Ridirezionamento di entrambi gli stream di output ed error verso lo stdin di altro programma, puo' essere fatto utilizzando la combinazione di operatori

```
2>&1 |
```

Provarlo eseguendo i due seguenti comandi in una directory in cui non esiste un file che si chiama pippo.txt, in modo da generare un messaggio di errore "no such file or directory", ridirezionarlo sullo standard output, collegarlo allo standard input di grep, selezionare la sola riga che contiene such e mandarla a video.

```
ls pippo.txt 2>&1 | grep such
```

```
ls * 2>&1 | grep such
```

Nota bene: **esiste un operatore apposito** che rappresenta una scorciatoia **dell'operatore composito 2>&1 |** che fa la stessa cosa, ed e' |&

```
ls pippo.txt |& grep such
```

Ridirezionamenti di Stream di I/O (16)

E' importante notare **la differenza nelle tempistiche di esecuzione** dei programmi quando questi sono collegati dal separatore ; o dalla |

program1 ; program2 ; program3

Con il separatore ; io faccio eseguire i diversi programmi uno dopo l'altro, cioè prima che parta il secondo programma deve finire il primo e così via.

Inoltre l'output di un programma non viene ridirezionato nell'input del programma successivo.

program1 | program2 | program3

Invece, con la | i programmi specificati partono assieme e l'output di un programma viene ridirezionato nell'input del programma successivo.

Ridirezionamenti di Stream di I/O (17)

E' possibile ridirigere contemporaneamente standard output e standard error di un programma program1 ridirigidolo verso lo standard input di un programma program2, utilizzando l'operatore |&

```
program1 |& program2
```

Con l'operatore composito |& i due programmi specificati partono assieme e l'output (sia stdout che stderr) del primo un programma sono ridirezionati insieme nell'input del secondo programma.

Auto-Ridirezionamento (18)

Una bash può ridirigere un proprio stream aperto verso un nuovo file descriptor. Da quel momento la bash vedrà il nuovo stream mediante il vecchio file descriptor. Gli autoridirezionamenti sono effettuati con i consueti operatori applicati ai file descriptor ma usati come argomenti della funzione exec.

Esempio:

Supponiamo di avere un file righedicomando.txt così fatto:

```
echo ciao
echo ${BASHPID}
ls /proc/${BASHPID}/fd/
sleep 10
```

Proviamo ad eseguire i seguenti comandi in una shell interattiva

```
exec {NUOVOFD}< righedicomando.txt
exec 0>&${NUOVOFD}
```

La bash usa come stdin il file righedicomando.txt , quindi visualizzerà i fd della bash stessa, poi aspetterà 10 secondi e infine terminerà chiudendo la finestra interattiva.

Ridirezionamenti di Stream di I/O (19)

Ridirezionamento per blocchi di comandi

Per i costrutti linguistici bash che eseguono blocchi di comandi (for, while, if then else) è possibile applicare un ridirezionamento unico per tutti i comandi del blocco di comandi, compresi i comandi eseguiti nella condizione.

Ad esempio, eseguendo il seguente script, applico un ridirezionamento dello standard output a tutti i comandi all'interno del loop del while :

(file riduzione1BloccoWhile.sh)

```
NUM=1
echo "${NUM}"
while (( ${NUM} <= "3" )) ; do
    echo "${NUM}"
    ((NUM=${NUM}+1))
done > pippo.txt
echo "${NUM}"
```

Dopo l'esecuzione dello script, nel file pippo.txt troveremo le 3 righe:

```
1
2
3
```

mentre a video vediamo le 2 righe

```
1
4
```

Ridirezionamenti di Stream di I/O (20)

Ridirezionamento per blocchi di comandi

E' possibile applicare un **ridirezionamento** anche per flussi di input.

Supponiamo di avere un file `Assoluzionilnaspettate.txt` strutturato come nell'esempio:
una riga con il reo, seguita da una riga con l'accusa, e così via:

<i>read dentro condizione while</i>	vittorio	ghini	17/07/1994
	blasfemia		assolto per insufficienza di prove
	giovanni	pau	15/07/1995
	ubriachezza		denuncia ritirata
	luca	andreucci	03/01/1992
	vilipendio		assolto per scomparsa del denunciante

Eseguiamo lo script `ridirezionelInputBloccoFor.sh` che segue

```
while read nome cognome data ; do
    if read accusa verdetto ; then
        echo $cognome errore $verdetto
    else
        echo terminazione inaspettata del file di input
        exit 1
    fi
done < Assoluzionilnaspettate.txt
```

Ridirezionamenti di Stream di I/O (21)

Esempi Ridirezionamento per blocchi di comandi

riduzione1BlocchifElse.sh

Ridirezionamento per blocchi di comandi

```
NUM=1
echo "${NUM}"
if (( "${NUM}" <= "3" )) ; then
    ((NUM=${NUM}+1))
    echo "${NUM}"
else
    ((NUM=${NUM}+2))
    echo "${NUM}"
fi > pippo.txt
echo "${NUM}"
```

L'output del ramo if e del ramo else finiscono entrambi nel file pippo.txt. Il primo e l'ultimo echo vanno in stdout

riduzione1Annidata.sh

Annidamento dei ridirezionamenti

```
NUM=1
echo "${NUM}"
if (( "${NUM}" <= "3" )) ; then
    ((NUM=${NUM}+1))
    echo "${NUM}"
    echo "vaffa" > cacchio.txt
else
    ((NUM=${NUM}+2))
    echo "${NUM}"
fi > pippo.txt
echo "${NUM}"
```

L'output in pippo.txt rimane quello dell'esempio di sinistra, mentre nel file cacchio.txt mi ritroco la riga con vaffa

Ridirezionamenti di Stream di I/O (22)

Esempi Ridirezionamento per blocchi di comandi

ridirezioneBlocchIfElse.sh

NB: Anche i comandi dentro la condizione dell' if vengono ridirezionati assieme a quelli del blocco if then else

```
if  ls ./out1.txt ; then
    echo "esiste"
else
    echo "non esiste"
fi &> pippo.txt
```

Se il file ./out1.txt esiste, allora nel file pippo.txt trovo,
./out1.txt
esiste

Se invece il file ./out1.txt non esiste,
allora nel file pippo.txt trovo,
ls: cannot access './out1.txt':
 No such file or directory
non esiste

ridirezioneBloccoWhile.sh

NB: Anche i comandi dentro la condizione di for e while vengono ridirezionati come quelli del blocco do done

```
while  ls ./out1.txt ; do
    echo "esiste ma ora ... "
    rm ./out1.txt
done &> pippo.txt
echo "non esiste "
```

Se il file ./out1.txt esiste, allora nel file pippo.txt trovo,
./out1.txt
esiste ma ora ...

ls: ./out1.txt': No such file or directory
Se invece il file ./out1.txt non esiste,
allora nel file pippo.txt trovo solo,
ls: ./out1.txt': No such file or directory

Ridirezione dell'input passato a riga di comando

Here Documents <<

La word che segue il metacarattere << viene utilizzata come delimitatore finale dell'input da passare al comando a sinistra del metacarattere.

Viene ridirezionato nell'input tutto quello che compare tra la word esplicitata dopo il metacarattere << e fino a dove quella word compare ancora all'inizio di una riga.

NON vengono espansi eventuali metacaratteri presenti nella word.

Utilità del ridirezionamento: poter passare input come se provenisse da un file ma senza dover scrivere un file.

Esempio:

```
while read A B C ; do echo $B ; done <<FINE
```

```
uno due tre quattro
```

```
alfa beta gamma
```

```
gatto cane
```

```
FINE
```

```
echo ciao
```

```
produce in output
```

```
due
```

```
beta
```

```
cane
```

```
ciao
```

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

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

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 stderr

```
( cat file1.txt ; cat file2.txt ) 2| 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

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

Commenti Aggiuntivi da fare sulla PROSSIMA SLIDE

Quella su uso di ridirezionamenti e GNU Coreutils

- Far vedere il comando **man man** ed evidenziare il concetto **di sezione del man** e le due **sezioni 1** (executable programs and shell commands) e **sezione 3** (library calls).
- Far capire che conviene (se possibile) usare in cascata più eseguibili già esistenti invece che scrivere un programma apposta da zero.
- Citare il pacchetto **coreutils** e ricordare che proviene dall'unione di più pacchetti precedenti (fileutils, shellutils, textutils) e suggerire di guardare cosa altro c'è .
- Descrivere l'esercizio proposto e la soluzione
- Descrivere le utilities **head tail sed cut cat grep** e poi **tee**.
- Occhio al parametro 4- passato a cut.
- Evidenziare che la maggior parte delle utilities prevedono un doppio comportamento, ovvero:
 - possono accettare dopo le opzioni uno o piu' argomenti che specificano il nome del/dei file da cui leggere input.
 - possono leggere input da stdin se non c'e' l'argomento nomefile o se c'e' un – come argomento finale.
- Fare un esempio in cui uso tail senza specificare un argomento nomefile per fargli usare input proveniente da stdin ed evidenziare il comportamento differente rispetto a come l'ho usato nell'esercizio
 - `cat dati.txt | tail -n 3`
- Aggiungere un esempio con l'utility **tee**
- Far vedere **tail -f** e mostrare che non viene gestita mutua esclusione sui file

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

pappappero
caracavALo
piripiccione
ammappete
uffauffa
pedalando
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

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
```

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//'
```

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/>.//'
```

Eseguo **due rimozioni insieme** ; Rimuovere il primo e l'ultimo carattere in ogni linea.

```
sed 's/..;s/>.//'
```

Rimuove I primi 3 caratteri ad inizio linea.

```
sed 's/...//'
```

Rimuove I primi 4 caratteri ad inizio linea.

```
sed -r 's/.{4}//'
```

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/(.{3}).*\^1/'
```

To remove everything except the last n characters in a file

```
sed -r 's/.*(.{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'
```

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 [

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="vaf"; 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)

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

Espansione 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