

Approfondiamo la struttura della shell Bash

Struttura interna, comandi e builtin,
variabili, ...

La Bash ...

- È un normale programma eseguibile

bash [opt] [scriptfile] [args]

- interpreta i comandi forniti *interattivamente* o letti (uno per riga nello **scriptfile**)
- alcune opzioni:
 - c prende i comandi da stringa
 - i interattiva
 - l *login*, cambia la fase di inizializzazione (vedi poi)
 - v *verbose*, stampa informazioni su quello che sta facendo
- **args** argomenti aggiuntivi accessibili secondo le modalità che discuteremo in seguito (*parametri posizionali e speciali*)

La bash ...(2)

- Può lavorare in due modalità:

bash [opt] [scriptfile] [args]

- *non interattiva*: se **scriptfile** è presente cerca di eseguire i comandi presenti nel file uno alla volta
- *interattiva*: interagisce con l'utente mostrando il prompt. Ci sono ancora due modalità:
 - Shell interattiva *di login*
 - Shell interattiva *normale*

Bash interattiva di login

- È quella che si ha di fronte appena completata la procedura di *login* (opz **-l**)
 - se non è stata specificata l'opzione **-noprofile** tenta di leggere ed eseguire il contenuto dei file
 - `/etc/profile`
 - `~/.bash_profile` oppure `~/.bash_login` oppure `~/.profile` (in quest'ordine)
 - interagisce con l'utente mostrando il prompt etc...
 - Al termine della sessione di lavoro, esegue, se esiste, il file `~/.bash_logout`

Bash interattiva normale

- È quella che si ha di fronte ogni volta che apriamo un terminale
 - se non è stata specificata l'opzione `-norc` (oppure `--rcfile`) tenta di leggere ed eseguire il contenuto dei file
 - `/etc/bash.bashrc` oppure `~/.bashrc` (in quest'ordine)
 - interagisce con l'utente mostrando il prompt etc...
 - Al termine della sessione di lavoro, non fa niente di particolare

Bash non interattiva

- La shell viene invocata con :

bash [opt] scriptfile [args]

- se **scriptfile** è presente esegue i comandi presenti nel file uno alla volta e poi termina
 - il separatore fra i comandi è newline o ‘;’, vedi poi
- *Parametri posizionali*: **\$1 \$2 \$3 ...** usati nello script per riferire i vari argomenti. Il parametro **\$0** indica il nome dello script.

Bash non interattiva (2)

- Inizializzazione ... :
 - la bash non interattiva non esegue file di configurazione
 - ma prima di eseguire lo **scriptfile** :
 - controlla la variabile di ambiente **BASH_ENV** (di default non presente o vuota)
 - espande il suo valore (vedi poi)
 - usa il risultato come il nome di un file di comandi da leggere ed eseguire

Bash non interattiva (3)

- Un piccolo esempio di script:

```
echo "Script $0"
```

```
echo "Primo Parametro $1"
```

```
echo "Secondo Parametro $2"
```

- Come procedere per l'esecuzione:

- salvare i comandi sopra in un file (es. **scriptfile**)

- attenti al separatore (newline)

- assicurarsi che su **scriptfile** sia permessa l'esecuzione

- lanciare

```
bash:~$ bash scriptfile arg1 arg2
```

Bash non interattiva (4)

```
bash:~$ bash scriptfile arg1 arg2
```

```
Script ./scriptfile
```

```
Primo Parametro arg1
```

```
Secondo Parametro arg2
```

```
bash:~$
```

Bash non interattiva (5)

- È possibile (ed è buona norma) specificare direttamente all'interno dello script la shell che deve interpretarlo:

```
bash:~$ cat scriptfile1
```

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

```
echo "Script $0"
```

```
echo "Primo Parametro $1"
```

```
echo "Secondo Parametro $2"
```

```
bash:~$ ls -l scriptfile1
```

```
-rwxr-xr-x 1 susanna ... Feb 6 2005 scriptfile1
```

```
bash:~$
```

Bash non interattiva (6)

- Il risultato è lo stesso di prima, ma non è necessario invocare la bash esplicitamente

```
bash:~$ scriptfile1 gg ff dd
Script ./scriptfile1
Primo Parametro gg
Secondo Parametro ff
bash:~$
```

File eseguibili e builtin

- Un comando richiesto alla shell può
 - corrispondere a un *file eseguibile* (localizzato da qualche parte nel file system) oppure
 - può corrispondere ad una funzionalità implementata internamente alla shell (detta *builtin*)

File eseguibili e builtin (2)

- file eseguibili

```
bash:~$ ./a.out
```

```
bash:~$ ls
```

- il file eseguibile viene ricercato in tutte le directory contenute nella variabile di ambiente PATH
- se il file esiste : la shell crea un nuovo processo shell (usando opportune SC) che cura l'esecuzione del programma contenuto nel file eseguibile. La shell padre si mette in attesa della terminazione del figlio e poi rinvia il prompt

File eseguibili e builtin (3)

- builtin

- la shell esegue direttamente il builtin al suo interno senza attivare altri processi
- es, cambio della working directory

```
bash:~$ cd
```

```
bash:~$
```

- es, scrittura di una stringa su stdout

```
bash:~$ echo ciao
```

```
ciao
```

```
bash:~$
```

Personalizzazione di Bash

Meccanismo di ridenominazione dei comandi (*aliasing*)

builtin **alias** **unalias**

alias <nome>=<comando>

unalias <nome>

- **alias** permette di definire dei legami fra una parola (<nome>) e una stringa (<comando>)
- **unalias** elimina il legame per il la parola **nome**
- la shell prima di eseguire un comando richiesto esamina la prima parola e va a controllare nella lista degli alias
 - se è presente un alias per quella parola lo espande con la stringa corrispondente (il comando)
- attenzione: prima e dopo il segno '=' non devono comparire spazi!

alias unalias (2)

- A che serve:

- permette di definire nomi mnemonici semplici per comandi anche complessi:

```
bash:~$ alias cerca=grep
```

```
bash:~$ alias trovac="find . -name *.c"
```

```
bash:~$ alias printall="lpr *.ps"
```

- rendere più sicuri comandi pericolosi:

```
bash:~$ alias rm="rm -i"
```

```
bash:~$ rm pippo
```

```
rm: remove regular file pippo? n
```

```
bash:~$
```

alias unalias (3)

- A che serve (cont.):

- permette anche di conoscere tutti gli alias

```
bash:~$ alias
```

```
alias cp="cp -i"
```

```
alias rm="rm -i"
```

```
alias bye="exit"
```

```
bash:~$
```

- o se è definito un alias per un nome

```
bash:~$ alias printall
```

```
alias printall="lpr *.ps"
```

```
bash:~$
```

alias unalias (4)

- Si può effettuare l'alias di un alias:
 - bash quando espande un alias considera la prima parola del testo di rimpiazzo e verifica se sono possibili ulteriori espansioni

```
bash:~$ alias listall
```

```
alias listall="ls -F *.sh"
```

```
bash:~$ alias llss=listall
```

```
bash:~$ llss
```

```
prova.sh* l.sh* d.sh*
```

```
bash:~$
```

alias unalias (5)

- Si può effettuare l'alias di un alias (cont):
 - per non andare in ciclo l'espansione si ferma quando la prima parola corrisponde ad un alias già espanso

```
bash:~$ alias rm="rm -i"
```

```
bash:~$ rm pippo
```

```
rm: remove regular file pippo? n
```

```
bash:~$
```

alias unalias (6)

- Sostituzioni successive:

```
alias <nome>=<comando>
```

- se l'ultimo carattere del testo del comando è uno spazio o una tabulazione, allora anche la parola successiva viene controllata per una possibile sostituzione attraverso alias:

```
bash:~/ciccio$ alias mydir="~/TEMP"
```

```
bash:~/ciccio$ cd mydir
```

```
bash: cd: mydir: No such file or directory
```

```
bash:~/ciccio$ alias cd="cd "
```

```
bash:~/ciccio$ cd mydir
```

```
bash:~/TEMP$
```

alias unalias (7)

- In bash, funzioni meglio di alias!
 - A differenza della C shell, bash non consente di definire alias con argomenti.
 - Se necessario si possono utilizzare le funzioni (vedi poi).
 - In generale in bash l'uso di alias è superato dall'uso delle funzioni
- Gli alias come altre personalizzazione della shell sono di solito inseriti nel file `.bashrc` o `.profile`

Settare le opzioni: builtin `set`

- Le opzioni sono una sorta di ‘switch’, con valore booleano che influenzano il comportamento della shell:

`set +o <option>` disattiva l’opzione

`set -o <option>` attiva l’opzione

attenzione al significato *controintuitivo* di più e meno

- quasi ogni opzione ha una forma breve

`set -o noglob` equivale a

`set -f`

Settare le opzioni: **set** (2)

- per vedere tutte le opzioni settate

```
bash:~$ set -o
```

– alcuni esempi

```
emacs on           -- emacs editing of command line
```

```
noglob on         -- non espande pathname
```

```
-- '*' e '?'
```

```
ignoreeof off    -- accetta Ctrl-D per logout
```

```
history on       -- history abilitata
```

Variabili di shell

- Le variabili della shell:
 - una variabile è un *nome* cui è associato un *valore*
 - nome: stringa alfanumerica che comincia per lettera
 - valore: stringa di caratteri
 - per assegnare un valore ad una variabile
<varname>=[<value>]
 - se **varname** non esiste viene creata altrimenti il valore precedente viene sovrascritto
 - attenzione: prima e dopo il segno '=' non devono comparire spazi

Variabili di shell (2)

- Una variabile si dice *definita* quando contiene un valore
 - anche la stringa vuota!
- Può essere cancellata con
 - `bash:~$unset varname`
- Per riferire il valore si usa la notazione
 - `$<varname>` oppure `${<varname>}`

Variabili di shell (3)

- Esempi:

```
bash:~$ Y=pippo          -- definisce Y
bash:~$ echo $Y ${Y}7    -- stampa il valore
pippo pippo7
bash:~$ X=$Y             -- assegna il valore di Y a X
bash:~$ echo $X
pippo
bash:~$ X=Y              -- assegna 'Y' a X
bash:~$ echo $X
Y
bash:~$ unset Y         -- cancella Y
```

Variabili di shell predefinite

- Alcuni variabili sono assegnate da Bash, es:

SHELL -- *shell di login*

HOSTTYPE -- *tipo di host, es i386-linux*

HISTSIZE -- *numero cmd nella history*

HISTFILE -- *file dove salvare la history*

– Per vederle tutte : **set**

- esempi:

```
bash:~$ echo $HISTSIZE
```

```
500
```

```
bash:~$ echo $HISTFILE
```

```
/home/s/susanna/.bash_history
```

```
bash:~$
```

Variabili di shell: PS1

- Controllare il prompt:
 - **PS1** controlla il *prompt primario*, quello della shell interattiva. Alcune stringhe hanno un significato particolare
 - **\u** nome dell'utente
 - **\s** nome della shell
 - **\v** versione della shell
 - **\w** working directory
 - **\h** hostname
 - esempio:

```
bash:~$ PS1='\u@\h:\w$'
```

```
susanna@fujih1:~$ PS1='\s:\w$'
```

```
bash:~$
```

Variabili di shell: **PATH**

- *Search path*: alcune variabili sono legate ai path dove cercare comandi e directory
 - **PATH** serie di directory in cui viene cercato il comando da eseguire, es:

```
bash:~$ echo $PATH
/usr/local/bin:/usr/local/bin/X11:/bin:/usr/bin:/usr/bin/X11:.
```
 - normalmente è predefinita

Variabili di shell: **PATH** (2)

- *E se non contiene il punto (.) ?*

```
bash:~$ echo $PATH
```

```
/local/bin:/usr/local/bin/X11:/bin:/usr/bin:/usr/bin/X11
```

```
bash:~$ ls -F
```

```
myscript*
```

```
bash:~$ myscript
```

```
bash: myscript: command not found
```

```
bash:~$ ./myscript
```

```
./myscript : ciao mondo!
```

```
bash:~$
```

Variabili di ambiente

- Le variabili di shell fanno parte dell'ambiente *locale* della shell stessa
 - quindi non sono visibili a processi o sottoshell attivate
 - una classe speciale di variabili, dette variabili di ambiente, sono invece visibili anche ai sottoprocessi
 - una qualsiasi variabile può essere resa una variabile d'ambiente esportandola:

```
export <varnames>          --esporta
```

```
export <varname>=<value>  --defin e esporta
```

```
export                    --lista variabili esportate
```

Variabili di ambiente (2)

- Alcune variabili locali sono esportate di default:
 - es **HOME**, **PATH**, **PWD**
 - le definizioni in **.bashrc** sono valide in ogni shell interattiva

- **Esempi:**

```
bash:~$ export PATH=$PATH:.
```

```
bash:~$ bash      -- creo una sottoshell
```

```
bash:~$ echo $PATH
```

```
/local/bin:/usr/local/bin/X11:/bin:/usr/bin:/usr/bin/X11:.
```

```
bash:~$      -- PATH è stata ereditata
```

Parametri builtin

- La shell prevede alcune variabili builtin, i *parametri posizionali* e *speciali*, che risultano utili per la programmazione di script.
- *Parametri posizionali* :
 - `$n` o `${n}` valore dell' *n*-esimo argomento (*n*=1,2 ...)
 - esempio

```
#!/bin/bash
echo Primo Parametro $1
echo Secondo Parametro $2
```

Parametri speciali (alcuni)

- \$0** Nome dello script
- \$*** Insieme di tutti i parametri posizionali a partire dal primo. Tra apici doppi rappresenta un'unica parola composta dal contenuto dei parametri posizionali .
- \$@** Insieme di tutti i parametri posizionali a partire dal primo. Tra apici doppi rappresenta una serie di parole, ognuna composta dal contenuto del rispettivo parametro posizionale.
Quindi "\$@" equivale a "\$1" "\$2" "\$3" ...
- \$\$** PID (*process identifier*) della shell

Parametri speciali (alcuni) (2)

- Esempio

```
bash:~$ more scriptArg.sh
```

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

```
echo Sono lo script $0
```

```
echo Mi sono stati passati $# argomenti
```

```
echo Eccoli: $*
```

```
bash:~$ scriptArg.sh ll kk
```

```
Sono lo script ./scriptArg
```

```
Mi sono stati passati 2 argomenti
```

```
Eccoli: ll kk
```

```
bash:~$
```

Controllo del flusso

If, while etc...

Strutture di controllo

- Permettono di
 - condizionare l'esecuzione di porzioni di codice al verificarsi di certi eventi
 - eseguire ripetutamente alcune parti etc.
- Bash fornisce tutte le strutture di controllo tipiche dei programmi imperativi
 - vengono usate soprattutto negli script ma si possono usare anche nella linea di comando

Strutture di controllo (2)

– **if-then-else**

- esegue una lista di comandi se una condizione è / non è vera

– **for**

- ripete una lista di comandi un numero prefissato di volte

– **while, until**

- ripete una lista di comandi finchè una certa condizione è vera / falsa

– **case**

- esegue una lista di comandi scelta in base al valore di una variabile

– **select**

- permette all'utente di scegliere fra una lista di opzioni

Costrutto **if**

- esegue liste di comandi differenti, in funzione di condizioni espresse anch'esse da liste di comandi
- sintassi (usando ‘;’ come terminatore della condizione)

```
if <condition>; then  
    <command-list>  
[elif <condition>; then  
    <command-list>] ...  
[else  
    <command-list>]  
fi
```

Costrutto **if** (2)

– sintassi (usando ‘newline’ come terminatore)

```
if <condition>  
then  
    <command-list>  
[elif <condition>  
then  
    <command-list>] ...  
[else  
    <command-list>]  
fi
```

Costrutto **if** (3)

- Semantica:

- esegue la lista di comandi `<condition>` che segue `if`
- se l'*exit status* è 0 (*vero*) esegue `<command-list>` che segue `then` e termina
- altrimenti esegue le condizioni degli `elif` in sequenza fino a trovarne una verificata
- se nessuna condizione è verificata esegue la `<command-list>` che segue `else`, se esiste, e termina
- l'*exit status* è quello dell'ultimo comando eseguito (0 se non ha eseguito niente)

Costrutto **if** (4)

- **Usò tipico**

- siccome 0 significa esecuzione non anomala:

```
if <esecuzione regolare del comando>; then  
    <elaborazione normale>  
else  
    <gestione errore>  
fi
```

Costrutto **if** : esempi

- Esempio: eseguiamo cd e poi ls

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

```
if cd $1; then
```

```
    echo "$0: File listing:"
```

```
    ls
```

```
else
```

```
    echo "$0: Error" 1>&2
```

```
fi
```

Costrutto **if** : esempi (3)

- Esempio: eseguiamo cd e poi ls

```
bash:~$ ./ifscript .  
./ifscript: File listing:  
pippo.c a.out mio.txt  
bash:~$ ./ifscript gigi  
./ifscript: Error  
bash:~$
```

Condizione: combinare *exit status*

- **&&**, **||**, **!** (and, or, negazione) si possono usare per combinare gli exit status nelle condizioni
- Es: verifichiamo che un file contenga una di due parole date:

```
file=$1; wrd1=$2; wrd2=$3;
if grep $wrd1 $file || grep $wrd2 $file; then
    echo "$wrd1 o $wrd2 sono in $file"
fi
```

- analogamente se ci sono entrambe ...

Test

- La condizione dell'**if** è un comando (possibilmente composto) ma questo non significa che si può testare solo la terminazione di un comando
- con la seguente sintassi

`test <condition> oppure [<condition>]`

- si può controllare:
 - proprietà dei file (presenza, assenza, permessi...)
 - confronti tra stringhe e interi
 - combinazioni logiche di condizioni

Test - stringhe

- Alcuni confronti fra stringhe:
 - con la condizione di verità

<code>str1 = str2</code>	<i>se str1 e str2 sono uguali</i>
<code>str1 != str2</code>	<i>se str1 e str2 sono diverse</i>
<code>-n str1</code>	<i>se str1 non è nulla</i>
<code>-z str1</code>	<i>se str1 è nulla</i>

Costrutto **if** : esempi (2)

- Esempio: eseguiamo `cd` e poi `ls`

```
#!/bin/bash
cd $1;
if [ $? = 0 ]; then
    echo "$0: File listing:"
    ls
else
    echo "$0: Error" 1>&2
fi
```

Test - attributi file

- `-e file` *se file esiste*
- `-d file` *se file esiste ed è directory*
- `-f file` *se file esiste e non è speciale
(dir, dev)*
- `-s file` *se file esiste e non è vuoto*
- `-x -r -w file` *controlla diritti
esecuzione, lettura e scrittura*
- `-O file` *se sei l'owner del file*
- `-G file` *se un tuo gruppo è gruppo di file*
- `file1 -nt file2`
- `file1 -ot file2`
*se file1 è più nuovo (vecchio) di file2
(data ultima modifica)*

Costrutto **if** : esempi (3)

- Esempio: inseriamo controlli nel precedente script

```
#!/bin/bash
if [ $# = 0 ]; then
    echo "Usage: $0 dirname" 1>&2
elif ! [ -d $1 ]; then
    echo "$0 : $1: Not a directory" 1>&2
elif cd $1; then
    echo "$0: File listing:"
    ls
else
    echo "$0: Error cannot cd to $1" 1>&2
fi
```

Costrutto **for**

- Permette di eseguire un blocco di istruzioni un numero prefissato di volte. Una variabile, detta *variabile di loop*, assume un valore diverso ad ogni iterazione
- diversamente dai costrutti **for** dei linguaggi convenzionali non permette di specificare *quante* iterazioni fare, ma una *lista di valori assunti dalla variabile di loop*. Sintassi

```
for <var> [ in <list> ]; do  
  <command-list>  
done
```

- se **<list>** è omessa si assume la lista degli argomenti dello script (**\$@**)

Costrutto `for` (2)

- Semantica:
 - Espande l'elenco `<list>` generando la lista degli elementi
 - Eseguce una scansione degli elementi nella lista (separatore il primo carattere in `$IFS`)
 - Alla variabile `<var>` ad ogni iterazione viene assegnato un nuovo elemento della lista e quindi si esegue il blocco `<command-list>` (che tipicamente riferisce la variabile di loop)
 - L'*exit status* è quello dell'ultimo comando eseguito all'interno della lista `do` oppure 0 se nessun comando è stato eseguito

Costrutto **for** : esempi

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

```
#Applica more a tutti gli argomenti
```

```
if [ $# = 0 ]; then
```

```
    echo "Usage: $0 file1 ... fileN" 1>&2
```

```
fi
```

```
for FILE in $@ ; do
```

```
    more $FILE
```

```
done
```

Costrutto **for** : esempi (2)

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

```
#Applica more a tutti gli argomenti
```

```
if [ $# = 0 ]; then
```

```
    echo "Usage: $0 file1 ... fileN" 1>&2
```

```
fi
```

```
#non importa specificare la lista
```

```
for FILE do
```

```
    more $FILE
```

```
done
```

Costrutto **for** : esempio C-like

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

```
#stampa i numeri pari fino a 20
```

```
for ((i=0; i<=20; i+=2)); do
```

```
    echo $i
```

```
fi
```

Costrutto **for** : esempio C-like (2)

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

```
#calcola i numeri di Fibonacci minori di 200
```

```
echo Ecco i numeri di Fibonacci ...
```

```
for (( i=1, j=1; j<=200; k=i, i=j, j=i+k ))
```

```
do
```

```
    echo $j
```

```
done
```

Costrutto **case**

- Permette di confrontare una stringa con una lista di pattern, e di eseguire di conseguenza diversi blocchi di istruzioni (simile a switch in C, Java)
- Sintassi:

```
case <expr> in  
    <pattern>  
        <command-list> ;;  
    <pattern>  
        <command-list> ;;  
...  
esac
```

Costrutto case (2)

– Sintassi alternativa:

```
case <expr> in
  (<pattern>)
    <command-list> ;;
  (<pattern>)
    <command-list> ;;
...
esac
```

Costrutto case (2)

- Semantica:

- L'espressione **<expr>** (in genere una variabile) viene espansa e poi confrontata con ognuno dei **<pattern>**
 - stesse regole dell'espansione di percorso
 - il confronto avviene in sequenza
- Se un pattern viene verificato si esegue la lista di comandi corrispondente e si esce
- Ogni pattern può in realtà essere l'or di più pattern
<pattern1> | ... | <patternN>
- L'exit status è quello dell'ultimo comando eseguito oppure 0 se nessun comando è stato eseguito

Costrutto **case** : esempio

- Lo script

mycd dir file.tar

- che con 1 o 0 parametri stampa la variabile **\$PWD**
- mentre con 2 parametri copia in **dir**, il file **file.tar** e lo decompime (con **tar**)
- con più di 2 parametri da errore

Costrutto **case** : esempio (2)

```
#!/bin/bash
case "$#" in
    ( 0|1 ) echo $PWD;;
    ( 2 ) if [ -d -x -w $1 ]; then
            cp $2 $1
            cd $1
            tar xvf $2
        fi ;;
    ( * ) echo "$0: too many args" 1>&2 ;;
esac
```

Costrutto `select`

- Permette di generare un menu e gestire la scelta da tastiera dell'utente

- Sintassi:

```
select <var> [ in <list> ]; do
```

```
  <command-list>
```

```
done
```

- **Semantica:**

- il comando `<list>` viene espanso generando una lista di elementi (se è assente si usa "\$@")

Costrutto `select` (2)

- **Semantica (cont):**
 - ogni elemento della lista viene proposto sullo standard error (ognuno preceduto da un numero) .
 - Quindi viene mostrato il prompt di `$PS3` (di default `$`) e chiesto il numero all'utente
 - la scelta fatta viene memorizzata nella variabile **REPLY** e l'elemento corrispondente della lista in `<var>`
 - con una scelta non valida il menu viene riproposto
 - se è valida si esegue `<command-list>` e si ripete tutto
 - si esce con il builtin **break**
 - L'exit status è quello dell'ultimo comando eseguito oppure 0 se nessun comando è stato eseguito

Costrutto `select` : esempio

- Lo script

`icd`

- che elenca le directory presenti in quella corrente
- e a scelta dell'utente si sposta in una di queste ed effettua il listing dei file presenti

Costrutto `select`: esempio (2)

```
#!/bin/bash
PS3="Scelta?"
select dest in $(command ls -aF | grep "/"); do
    if [ -d -x -r $dest ]; then
        cd $dest;
        echo "icd: Changed to $dest"
        ls
        break
    else
        echo "icd: wrong choice" 1>&2
    fi
done
```

Costrutti **while** **until**

- Permettono di ripetere l'esecuzione di un blocco di istruzioni fino al verificarsi (**while**) o al falsificarsi (**until**) di una condizione

- Sintassi:

```
while <condition>; do           until <condition>; do  
  <command-list>                <command-list>  
done                             done
```

- **<condition>** è analogo a quello dell'**if**
- al solito vera (0), falsa (!=0)
- L'exit status è quello dell'ultimo comando di **<command-list>** oppure 0 se non si entra nel ciclo

Consigli per il debugging ...

Prima di tutto ...

- **ATTENZIONE:**
 - gli script possono essere pericolosi, proteggete file e directory ed eseguiteli in ambienti non danneggiabili finchè non siete ragionevolmente sicuri della loro correttezza!
 - Attenzione a lasciare gli spazi dove servono ed agli effetti delle espansioni!

Opzioni per il debugging

- Alcune opzioni utili per il debugging:
 - settabili con comando **set** [-/+o]
 - **noexec -n** : non esegue, verifica solo la correttezza sintattica
 - **verbose -v** : stampa ogni comando prima di eseguirlo

Opzioni per il debugging (2)

- Alcune opzioni utili per il debugging (cont):

- **xtrace -x** : mostra il risultato dell'espansione prima di eseguire il comando

```
bash:~$ ls *.c
```

```
pippo.c pluto.c
```

```
bash:~$ set -x
```

```
bash:~$ ls *.c
```

```
+ ls -F pippo.c pluto.c
```

```
pippo.c pluto.c
```

```
bash:~$
```

Funzioni

Funzioni

- Bash offre la possibilità di definire *funzioni*
 - un funzione associa un *nome* ad un *programma di shell* che viene mantenuto in memoria e che può essere richiamato come un comando interno (builtin)

```
[function] <nome> () {  
    <lista di comandi>  
}
```
- Le funzioni sono eseguite nella shell corrente
 - e non in una sottoshell come gli script

Funzioni (2)

- Parametri posizionali e speciali sono utilizzabili come negli script
 - es. possono essere usate per definire alias con parametri

```
rmall () {  
    find . -name "$1" -exec rm -i {} \; ;  
}
```
- Le funzioni si possono cancellare con

```
unset -f funct_name
```

Funzioni (3)

- Per vedere le funzioni definite in fase di inizializzazione della shell ...

```
bash:~$ declare -f
```

fornisce tutte le funzioni ed il loro codice sullo standard output

```
bash:~$ declare -F
```

fornisce i nomi di tutte le funzioni (senza il codice)

```
bash:~$ type -all name_function
```

fornisce tutte le informazioni ed il codice della funzione di nome

```
name_function
```

- Vediamo alcuni esempi

Funzioni (4)

```
bash:~$ rmall () { find . -name "$1" -exec \
rm -i {} \; ; }
```

```
bash:~$ type -all rmall
```

```
rmall is a function
```

```
rmall ()
```

```
{
find . -name "$1" -exec rm -i {} \; ;
}
```

```
bash:~$ rmall kk
```

```
rm: remove regular file './kk'? y
```

```
bash:~$
```

Funzioni (5)

- Definire funzioni da file (modificare la shell corrente)

```
bash:~$ more myfunctions
```

```
function rmall () {
```

```
    find . -name "$1" -exec rm -i {} \; ; }
```

```
bash:~$ . ./myfunctions -- o source
```

```
bash:~$ type -all rmall
```

```
rmall is a function
```

```
rmall ()
```

```
{
```

```
find . -name "$1" -exec rm -i {} \; ;
```

```
}
```

```
bash:~$
```

Builtin `.'` e `source`

- Comandi interni (builtin) della bash
 - equivalenti
 - sintassi
 - `. filename [arguments]`
 - `source filename [arguments]`
 - entrambi leggono ed eseguono i comandi contenuti in `filename` nell'ambiente della shell corrente

Funzioni (6)

```
bash:~$ rmall () { find . -name "$1" -exec \
rm -i {} \; ; }
```

- Attenzione a mettere i giusti meccanismi di quoting (escape) per inibire o permettere l'espansione dei metacaratteri da parte della shell !!!!

– " " oppure ' ' oppure \

- Ne parliamo nella prossima sezione ...

Funzioni (7)

- **Attenzione:**
 - le variabili definite dentro una funzioni sono globali ed accessibili al di fuori della funzione
 - questo spesso crea problemi: vediamo un esempio un po' artificioso

Funzioni (8)

```
#!/bin/bash
function esempiofun ()
{
    echo in function: $0 $1 $2
    var1="in function"
    echo var1: $var1
}
var1 ="outside function"
echo var1: $var1
echo $0 $1 $2
esempiofun funarg1 funarg2
echo var1: $var1
echo $0 $1 $2
```

Funzioni (9)

```
bash:~$ ./escript arg1 arg2
var1: outside function
./escript arg1 arg2
in function ./escript funarg1 funarg2
var1: in function
var1: in function
./escript arg1 arg2
bash:~$
```

Funzioni (10)

- **Attenzione:**
 - le variabili definite dentro una funzioni sono globali ed accessibili al di fuori della funzione
 - questo spesso crea problemi: vediamo un esempio un po' artificioso
 - conviene sempre limitare lo scope delle variabili nelle funzioni con **local**

Funzioni (11)

```
#!/bin/bash
function esempiofun ()
{ local var1
  echo in function: $0 $1 $2
  var1="in function"
  echo var1: $var1
}
var1 ="outside function"
echo var1: $var1
echo $0 $1 $2
esempiofun funarg1 funarg2
echo var1: $var1
echo $0 $1 $2
```

Funzioni (12)

```
bash:~$ ./escript arg1 arg2
var1: outside function
./escript arg1 arg2
in function ./escript funarg1 funarg2
var1: in function
var1: outside function
./escript arg1 arg2
bash:~$
```

Espansione e Quoting ...

Espansione e quoting

- *Espansione:*
 - la shell, prima di eseguire la linea di comando interpreta le variabili ed i simboli speciali sostituendoli (espandendoli) con quanto ‘rappresentato’
- *Quoting:*
 - inibizione della espansione per mezzo di simboli che impongono alla shell l’interpretazione ‘letterale’ di altri simboli che altrimenti avrebbero un significato speciale
 - alla fine dell’espansione i simboli di quoting vengono rimossi, in modo che un eventuale programma che riceva il risultato dell’espansione come argomenti non ne trovi traccia

Vari tipi di espansione

- La bash, prima di eseguire un comando opera diverse espansioni, nel seguente ordine:
 1. Espansione degli *alias* e dell'*history*
 2. Espansione delle *parentesi graffe* (C)
 3. Espansione della *tilde* (~) (C)
 4. Espansione delle *variabili* (Korn)
 5. Sostituzione dei *comandi* (Bourne e Korn)
 6. Espansione delle *espressioni aritmetiche*
 7. Suddivisione in *parole*
 8. Espansione di percorso o *globbing*

Espansione di *alias* ed *history*

- Se la prima parola di una linea di comando è un alias la shell lo espande (ricorsivamente) come già visto
 - L'espansione si applica anche alla parola successiva se l'alias termina con spazio o tab
- Se la prima parola inizia con il metacarattere "!" allora la shell interpreta la parola come riferimento alla history come già visto
 - es.
 - !n** *n-esima riga nella history*
 - !!** *riga di comando precedente*

Espansione delle *parentesi graffe*

- Meccanismo che permette la generazione di stringhe arbitrarie usando pattern del tipo:
 - `<prefisso>{<elenco>}<uffisso>`
 - l'elenco è dato da una serie di parole separate da virgole
 - ...
 - es:
 - `sal{v,d,modi}are` si espande a `salvare`,
`saldare`, `salmodiare`
 - `c{{er,as}c,ucin}are` si espande a `cercare`,
`cascare`, `cucinare`
 - introdotto nella C shell

Espansione delle *parentesi graffe* (2)

- Ancora es:

```
bash:~$ mkdir m{i,ia}o
```

```
bash:~$ ls -F m*
```

```
miao/ mio/
```

```
bash:~$ rm -f miao/{lib.{?,??},*~,??}.log}
```

```
bash:~$
```

- Nota:

- in questo caso le stringhe che risultano dall'espansione non sono necessariamente nomi di file (come accade invece nell'espansione di percorso)

Espansione della *tilde* (~)

- Se una parola inizia con il simbolo *tilde* (~)
 - la shell interpreta quanto segue (fino al primo ‘/’), come un username e lo sostituisce con il nome della sua home directory
 - `~username` → home directory di `username`
 - ‘~/’ e ‘~’ si espandono nella home directory dell’utente loggato (ovvero nel contenuto della variabile **HOME**)
 - `~/,~` → `$HOME`
 - es.
 - `bash:~$ cd ~besseghi`
 - `bash:/home/personale/besseghi$`

Espansione delle *variabili*

- In ogni parola del tipo

`$stringa` oppure `${stringa}`

`stringa` viene interpretato come il nome di una variabile e viene espanso dalla shell con il suo valore

es.

```
bash:~$ PARTE=Dani
```

```
bash:~$ echo $PARTEele
```

```
bash:~$ echo ${PARTE}ele
```

```
Daniele
```

```
bash:~$
```

Sostituzione dei *comandi*

- Consente di espandere un comando con il suo (standard) output:

`$ (<comando>)`

— es.

```
bash:~$ ELENCO=$(ls)
```

```
bash:~$ echo $ELENCO
```

```
pippe pluto paperone main.c
```

```
bash:~$ ELENCO=$(ls *.c)
```

```
bash:~$ echo $ELENCO
```

```
main.c
```

```
bash:~$
```

Sostituzione dei *comandi* (2)

- Ancora esempi:

-- rimuove i file che terminano per '~'

-- nel sottoalbero con radice in '.'

```
bash:~$ rm $(find . -name "*~")
```

-- si può usare una diversa sintassi

-- attenzione alla direzione degli apici!!!

-- vanno da sin a ds

```
bash:~$ rm `find . -name "*~" `
```

-- questa seconda è obsoleta e mantenuta solo per compatibilità ma può spiegare alcuni strani comportamenti

Espressioni *aritmetiche*

- Trattamento delle espressioni aritmetiche intere:

`$((<espressione>))` o `$(<espressione>)`

— es.

```
bash:~$ echo 12+23
```

```
12+23
```

```
bash:~$ echo $((12+23))
```

```
35
```

-- dich di variabile intera

```
bash:~$ let VALORE=$((12+23))
```

```
bash:~$ echo $VALORE + 1
```

```
35 + 1
```

```
bash:~$
```

Suddivisione in *parole*

- Una parola è una sequenza di caratteri che non sia un operatore o una entità da valutare
 - è una entità atomica (es. arg. fornito ad un programma)
 - I delimitatori di parole sono contenuti nella variabile IFS (*Internal Field Separator*) che per default contiene spazio, tab e newline (‘ ’, ‘\t’, ‘\n’)
 - La suddivisione di parole non avviene per stringhe delimitate da apici singoli e doppi
 - es.

```
bash:~$ ls "un file con spazi nel nome"  
un file con spazi nel nome  
bash :~$
```

Suddivisione in *parole* (2)

– es. perché?

```
bash:~$ echo mm${IFS}mm
```

```
mm mm
```

```
bash:~$ echo "mm${IFS}mm"
```

```
mm
```

```
mm
```

```
bash:~$ ls un\ file\ con\ spazi\ nel\ nome
```

```
un file con spazi nel nome
```

```
bash:~$
```

Espansione di percorso o *globbing*

- Se una parola contiene uno dei simboli speciali ‘*’, ‘?’ o ‘[...]’
 - viene interpretata come modello ed espansa con l’elenco, ordinato alfabeticamente, dei percorsi (pathname) corrispondenti al modello (lo abbiamo visto)
 - Nota:
 - l’espansione non riguarda i file nascosti, a meno che il punto ‘.’ non faccia parte del modello:

```
bash:~$ ls .bash*  
.bashrc .bash_profile  
bash:~$
```

Quoting

- Deriva dal verbo inglese *to quote* (citare) ed indica i meccanismi che inibiscono l'espansione
 - in particolare viene rimosso il significato speciale di alcuni simboli, che nel quoting vengono interpretati *letteralmente*
 - ci sono tre meccanismi di quoting:
 - carattere di escape (backslash) \
 - apici semplici `
 - apici doppi " o virgolette.

Escape e continuazione

- Il carattere di escape (backslash) \
 - indica che il carattere successivo non deve essere considerato un carattere speciale
 - es:

```
bash:~$ ls .bash\*  
ls: .bash*: No such file or directory  
bash:~$
```

Il modello non é stato espanso e l'asterisco è considerato un carattere normale parte del nome del file da listare
 - *Continuazione*: Se \ è seguito subito dal newline indica che il comando continua sulla linea successiva

Apici singoli

- Una stringa racchiusa fra apici singoli non è soggetta a *nessuna* espansione

' testo '

– attenzione al verso degli apici: l'apice inclinato in modo opposto è legato alla sintassi obsoleta delle sostituzioni dei comandi (`)

– es:

```
bash:~$ A=prova
```

```
bash:~$ echo 'nessuna espansione di $A o *'
```

```
nessuna espansione di $A o *
```

```
bash:~$
```

Apici doppi

- Inibiscono solo l'espansione di percorso:

"testo"

– in questo caso \$ e \ vengono valutati normalmente

– es:

```
bash:~$ A=prova
```

```
bash:~$ echo "nessuna espansione di $A \ $A  
o *"
```

```
nessuna espansione di prova $A o *
```

```
bash:~$
```

Combinare comandi

Una panoramica completa

Terminazione ed Exit status

- Ogni comando Unix al termine della sua esecuzione restituisce un valore numerico (detto *exit status*)
 - tipicamente *zero* significa esecuzione regolare e ogni altro valore terminazione anomala
 - gli exit status si possono usare nelle espressioni booleane all'interno dei comandi condizionali di shell.
 - in questo caso zero viene assimilato a true e tutto il resto a false.
 - la variabile predefinita `$?` da l'exit status dell'ultimo comando eseguito

Bash: comandi semplici

`[var assign] <command> <args> <redirs>`

– es: `A=1 B=2 myscript pippo < pluto`

- In pratica:

- è una sequenza (opzionale) di assegnamenti a variabili,
- seguita da una lista di parole di cui la prima (`command`) è interpretata come il comando da eseguire
- seguita da eventuali ridirezioni (`redirs`)
- terminato da un carattere di controllo (newline o ‘;’)
- L’ *exit status* è quello del comando (se la terminazione è normale) oppure lo stabilisce la shell ...

Bash: comandi semplici (2)

Codici di terminazione ‘anomala’:

- comando non trovato 127
- file non eseguibile 126
- comando terminato da segnale n : $128 + n$
- esempi di evento/segnale / n
 - CTRL-C SIGINT 2
 - **kill** SIGTERM 15
 - **kill -9** SIGKILL 9

Bash: pipelining

```
[ ! ] <command1> [ | <command2> ]
```

- sequenza di comandi separata dal carattere di pipe ‘|’
- In questo caso lo **stdout** di **command1** viene connesso attraverso una pipe allo **stdin** di **command2** etc
- ogni comando è eseguito in un processo differente (sottoshell)
- il suo exit status è quello dell’ultimo comando nella pipeline (o la sua negazione logica se è stato specificato !)

Liste

- Una lista è una sequenza di una o più pipeline
 - separata da uno degli operatori: `;` `&` `&&` `||`
 - terminata da `;` `&` o *newline*
 - una lista può essere raggruppata da parentesi (tonde o graffe) per controllarne l'esecuzione
 - L'exit status della lista è l'exit status dell'ultimo comando eseguito dalla lista stessa

Liste: sequenze non condizionali

- Sintassi

<command1> ; <command2>

- viene eseguito **command1**
- quando termina **command1** si esegue **command2**
- l'*exit status* è quello di **command2**

- ; sostituisce logicamente il *newline*

```
bash:$ sleep 40; echo done
```

```
-- attende 40 sec
```

```
done
```

```
bash:$
```

Liste: comando in background

- Lo abbiamo già visto

<command> &

- la shell esegue **command** in una sottoshell, senza attenderne la terminazione e ripresenta subito il prompt
- l'exit status è 0
- es.

```
bash:$ sleep 40 &
```

```
bash:$
```

Liste: operatore di controllo & &

- Sintassi:

<command1> && <command2>

- la shell esegue **command1**
- se l'exit value di **command1** è 0 (true) esegue anche **command2**
- l'exit value è l'AND logico dell'exit value dei due comandi (lazy)
- serve per eseguire il secondo comando solo se il primo ha avuto successo. Es:

```
bash:$ mkdir prova && echo prova creata!
```

(segue)

Liste: operatore di controllo && (2)

```
bash:$ mkdir prova && echo prova creata!  
prova creata!
```

```
bash:$ mkdir prova && echo prova creata!  
mkdir: cannot create directory `prova': File  
exists  
bash:$
```

Liste: operatore di controllo | |

- Sintassi:

<command1> | | <command2>

- la shell esegue **command1**
- se l'exit value di **command1** è diverso da 0 (false) esegue anche **command2**
- l'exit value è l'OR logico dell'exit value dei due comandi (lazy)
- serve per eseguire il secondo comando solo se il primo *non* ha avuto successo. Es:

```
bash:$ mkdir prova | | echo prova NON creata!  
(segue)
```

Liste: operatore di controllo || (2)

```
bash:$ mkdir prova && echo prova creata!  
prova creata!
```

```
bash:$ mkdir prova && echo prova creata!  
mkdir: cannot create directory `prova': File  
exists
```

```
bash:$ mkdir prova || echo prova NON creata!  
mkdir: cannot create directory `prova': File  
exists
```

```
prova NON creata!
```

```
bash:$
```

Delimitatori di lista { ... }

- Sintassi:

```
{ <list>; }
```

- la lista `list` viene eseguita nella shell corrente, senza creare alcuna sottoshell
- L'effetto è quello di raggruppare più comandi in un unico blocco (exit status quello di *list*)
- ATTENZIONE: il `;` finale è necessario come pure lo spazio fra lista e parentesi graffe

```
bash:$ { date; pwd; } > out
```

-- scrive in 'out' sia l'stdout di date che di pwd

```
bash:$
```

Delimitatori di lista (...)

- Sintassi:

(**<list>**)

– la lista **list** viene eseguita in una sottoshell

- assegnamenti di variabili e comandi interni che influenzano l'ambiente di shell non lasciano traccia dopo l'esecuzione
- l'exit status è quello di list

```
bash:$ ( cd Work; mkdir pippo ) && echo OK
```

-- tenta di spostarsi nella directory Work e di creare la directory pippo, se ci riesce scrive un messaggio di conferma

```
bash:$
```