

Sistemi Operativi
A.A. 2009-10

Shell Scripting

Alberto Montresor
Renzo Davoli

Copyright © 2001-2005 Alberto Montresor, Renzo Davoli

Permission is granted to copy, distribute and/or modify this document under the terms of the GNU Free Documentation License, Version 1.2 or any later version published by the Free Software Foundation; with no Invariant Sections, no Front-Cover Texts, and no Back-Cover Texts. A copy of the license can be found at:

<http://www.gnu.org/licenses/fdl.html#TOC1>

Introduzione

- ♦ **Contenuto di questo modulo:**
 - ♦ Introduzione alla programmazione di un S.O. tramite *shell scripting*, creando sequenze di comandi che verranno interpretate dalla shell del S.O
- ♦ **Perchè imparare un linguaggio di shell scripting?**
 - ♦ Permette di automatizzare attività ripetute
 - ♦ Esempio: inizializzazione del sistema, verifiche periodiche di sicurezza
 - ♦ E' fondamentale per l'amministrazione di un sistema
- ♦ **Alla fine di questo modulo**
 - ♦ Dovete essere in grado di utilizzare un S.O. come utenti sofisticati
 - ♦ Dovete essere in grado di capire la sintassi di script complessi
- ♦ **To be continued...**
 - ♦ Modulo di amministrazione di sistema

Scripting: esempio

/etc/rc.d/init.d/network (estratto)

```
#!/bin/bash
# network          Bring up/down networking
# If IPv6 is explicitly configured, make sure it's available.
if [ "$NETWORKING_IPV6" = "yes" ]; then
    alias=`modprobe -c | awk '/^alias net-pf-10 / { print $3 }'`
    if [ "$alias" != "ipv6" -a ! -f /proc/net/if_inet6 ]; then
        echo "alias net-pf-10 ipv6" >> /etc/modules.conf
    fi
fi
# find all the interfaces besides loopback.
interfaces=`ls ifcfg* | LANG=C egrep -v '(ifcfg-lo|:|rpmsave|rpmorig|
rpmnew)' | \
    LANG=C egrep -v '(~|\.\bak)$' | \
    LANG=C egrep 'ifcfg-[A-Za-z0-9_-]+$' | \
    sed 's/^ifcfg-//g'`
```

Sommario

- ♦ **Unix: nozioni base**
 - ♦ File system, diritti di accesso, comandi base per interagire con il file system, stampa, ...
- ♦ **Shell: nozioni base**
 - ♦ Tipi di shell, bash, redirectione, ...
- ♦ **Shell: scripting avanzato**
 - ♦ Variabili, strutture di controllo, funzioni, ...
- ♦ **Text processing e utility avanzate**
 - ♦ Espressioni regolari, comandi find awk sed, ...
- ♦ **Lectures consigliate**
 - ♦ *“Introduction to Unix”*, di Frank G. Fiamingo, Linda DeBula, Linda Condron – Vedi sito web del corso
 - ♦ *“Advanced Bash-Scripting Guide”*, di Mendel Cooper
Vedi sito web del corso
 - ♦ Manuali in linea
 - ♦ Comando **man**
 - ♦ Comando **info**
 - ♦ Glass-Ables, *“UNIX for Programmers and Users”*, cap. 1-7

1. Unix: nozioni base

Lecture consigliate:

- Fiamingo and Debula, *Introduction to Unix*, sezioni 1-4
- Glass and Ables, *Unix for Programmers and Users*, sezioni 1-2
- Per la sintassi completa dei comandi: **man** e **info**

Introduzione

- ◆ **Un po' di storia**

- ◆ '60 Multics
- ◆ '70 AT&T Bell Labs
- ◆ '70-'80 UC Berkeley
- ◆ '80 Unix commerciali, frammentazione
- ◆ '90 Linux

- ◆ **Filosofia di Unix**

- ◆ Multiutente / multitasking
- ◆ Composizione di tool (filtri)
- ◆ Progettato da programmatori per programmatori
- ◆ “Everything is a file”

Shell

- ♦ **Una shell:**
 - ♦ E' un programma che si frappone fra l'utente e il S.O.
 - ♦ Vi presenta il prompt che utilizzate per inserire comandi
 - ♦ E' programmabile: vi permette di definire script
 - ♦ Programmi in formato testuale che racchiudono comandi
 - ♦ E' comunque un programma come tutti gli altri

Shell

- ◆ **Comandi**

- ◆ rappresentano una richiesta di esecuzione in UNIX
- ◆ forma generale: ***comando opzioni argomenti***

- ◆ **Note:**

- ◆ gli argomenti indicano l'oggetto su cui eseguire il comando
- ◆ le opzioni (generalmente!)
 - ◆ iniziano con -, --
 - ◆ modificano l'azione del comando

- ◆ **Esempi di comandi:**

- ◆ comandi built-in della shell
- ◆ script interpretati
- ◆ codice oggetto compilato

Tasti di controllo nella shell

^S sospende la visualizzazione

^Q riattiva la visualizzazione

^C cancella un'operazione

^D end-of-line

^V tratta il carattere di controllo seguente come se fosse un carattere normale

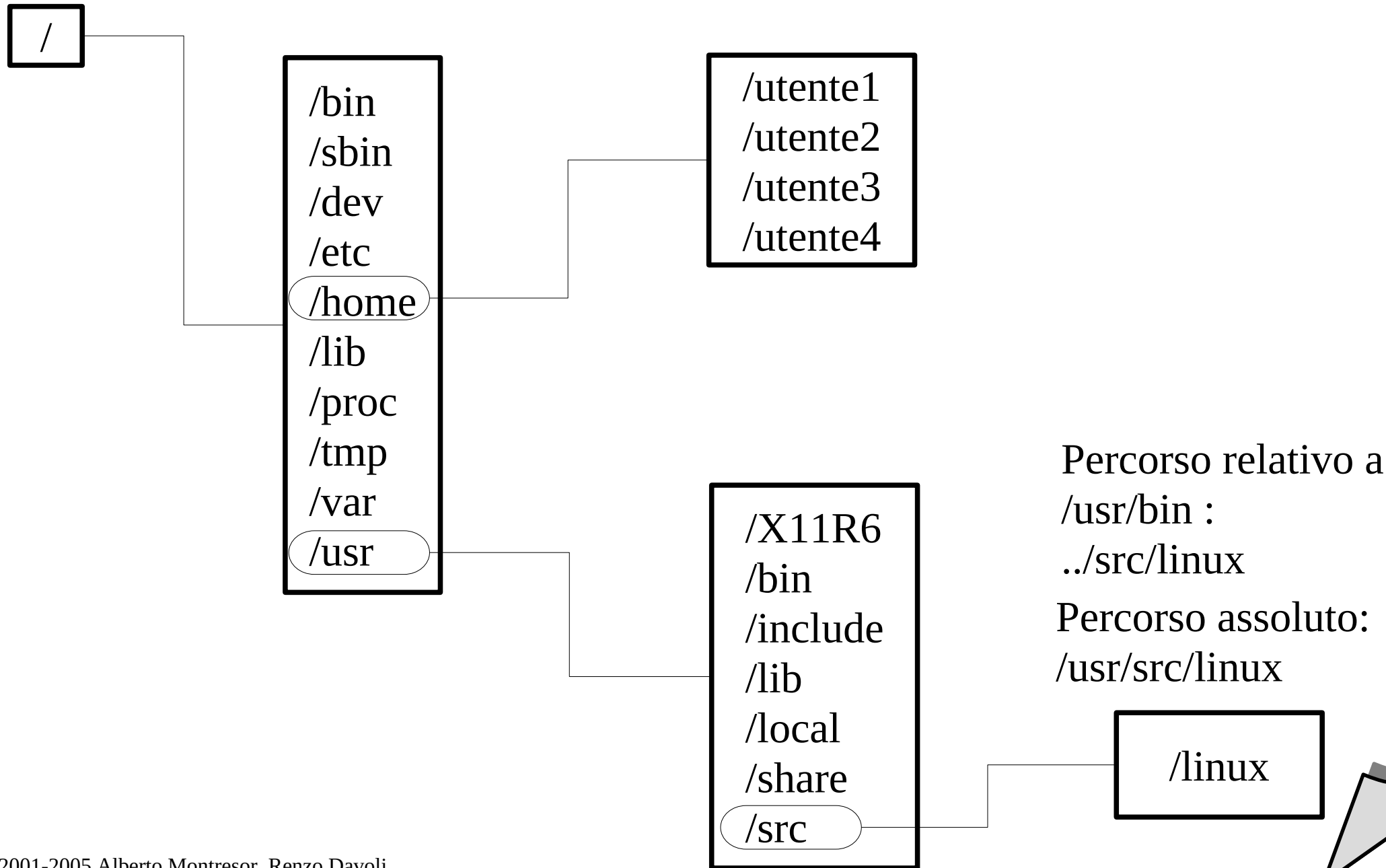
Help

- ♦ **Leggere la documentazione on-line**
 - ♦ `man command`
 - ♦ `info command`
 - ♦ `apropos keyword`
 - ♦ `man man`
 - ♦ `info`

File system

- ♦ **Il termine file system assume vari significati**
 - ♦ è l'insieme dei file e delle directory che sono accessibili ad una macchina Linux
 - ♦ è l'organizzazione logica utilizzata da un s.o. per gestire un insieme di file in memoria secondaria
 - ♦ il termine viene anche utilizzato per indicare una singola unità di memoria secondaria
- ♦ **I file system sono organizzati ad albero (quasi)**
 - ♦ directory e sottodirectory costituiscono i nodi dell'albero
 - ♦ i file costituiscono le foglie
 - ♦ la directory root / costituisce la radice dell'albero

Gerarchia standard del file system



Filesystem Hierarchy Standard

- ♦ **/bin** Comandi base per utenti "comuni"
- ♦ **/sbin** Comandi per la gestione del sistema, non destinati ad utenti comuni
- ♦ **/dev** Device file per accedere a periferiche o sistemi di memorizzazione
- ♦ **/etc** File di configurazione del sistema
- ♦ **/home** "Home directory" degli utenti
- ♦ **/lib** Librerie condivise dai programmi e utili per il loro funzionamento
- ♦ **/tmp** Directory dei file temporanei
- ♦ **/var** Dati variabili, code di stampa
- ♦ **/usr** Contiene gran parte dei programmi esistenti nel sistema
- ♦ **/proc** file system virtuale senza reale allocazione su disco. Viene utilizzato per fornire informazioni di sistema relative in modo particolare al kernel.

File system - "Navigazione"

- ♦ **pwd**
 - ♦ Print Working Directory
- ♦ **cd *directory***
 - ♦ Change working Directory ("go to" directory)
- ♦ **mkdir *directory***
 - ♦ MaKe a directory
- ♦ **rmdir *directory***
 - ♦ ReMove directory
- ♦ **ls *directory***
 - ♦ LiSt directory content
 - ♦ -a all files
 - ♦ -l long listing
 - ♦ -g group information

Attributi

- ◆ **Per ottenere informazioni complete su un file:**

```
% ls -lgsF prova.txt
```

```
1 -rw-r--r-- 1 montreso staff 213 Oct 2 00:12 prova.txt
```

- ◆ **Descrizione:**

- ◆ Numero di blocchi utilizzati dal file (1 blocco = 1024 bytes)
 - ◆ Tipo e permessi del file
 - ◆ Il conteggio di hard-link
 - ◆ Lo username, groupname possessori del file
 - ◆ Dimensione in byte
 - ◆ Data di ultima modifica
 - ◆ Nome del file
- ◆ **Questi sono gran parte dei dati contenuti in un i-node**

Concetto di owner / gruppo

- ◆ **Ogni file è associato a:**
 - ◆ un utente proprietario del file
 - ◆ un gruppo (i.e., insieme di utenti) con speciali diritti sul file
- ◆ **Come identificare utenti e gruppi:**
 - ◆ *user id* (valore intero, Unix internals); *username* (stringa)
 - ◆ *group id* (valore intero, Unix internals); *groupname* (stringa)
- ◆ **Come associare un utente ad un file:**
 - ◆ quando create un file, viene associato al vostro user id
 - ◆ potete modificare il proprietario tramite **chown newUserId file(s)**
 - ◆ normalmente non disponibile in un sistema in cui vengono gestite system quotas

Gestione dei gruppi

- ◆ **Come ottenere la lista dei gruppi a cui appartenete**

groups [*username*]

invocata senza argomenti, elenca i gruppi a cui appartenete;

indicando **username**, ritorna i gruppi associati a **username**

- ◆ **Come associare un gruppo ad un file**

- ◆ quando create un file, viene associato al vostro gruppo corrente

- ◆ il vostro gruppo corrente iniziale è scelto dall'amministratore

- ◆ potete cambiare il vostro gruppo corrente (aprendo una nuova shell) tramite **newgrp** <groupname>

- ◆ Potete modificare il gruppo associato ad un file tramite il comando **chgrp** <groupname> <file(s)>

Permessi dei file

- Ogni file è associato a 9 flag chiamati “Permission”

<i>User</i>			<i>Group</i>			<i>Others</i>		
<i>R</i>	<i>W</i>	<i>X</i>	<i>R</i>	<i>W</i>	<i>X</i>	<i>R</i>	<i>W</i>	<i>X</i>

- *Read*:
 - file regolari: possibilità di leggere il contenuto
 - directory: leggere l'elenco dei file contenuti in una directory
- *Write*:
 - file regolari: possibilità di modificare il contenuto
 - directory: possibilità di aggiungere, rimuovere, rinominare file
- *Execute*:
 - file regolari: possibilità di eseguire il file (se ha senso)
 - directory: possibilità di fare cd nella directory o accedervi tramite path

Permessi dei file

- ♦ **In realtà:**

- ♦ la gestione dei permessi è leggermente più complessa di quanto presentato
- ♦ la vedremo (in versione "completa") nel modulo di sicurezza

- ♦ **Quando un processo è in esecuzione, possiede:**

- ♦ Un user ID / group ID reale (usato per accounting)
- ♦ Un user ID / group ID effettivo (usato per accesso)

- ♦ **Quali permission vengono utilizzati?**

- ♦ Se l'user ID effettivo corrisponde a quello del possessore del file, si applicano le User permission
- ♦ Altrimenti, se il group ID effettivo corrisponde a quello del file, si applicano le Group permission
- ♦ Altrimenti, si applicano le Others permission

Come cambiare i permessi

- ◆ **Relativo: `chmod [ugo][+ -][rwxX] file(s)`**

- ◆ Esempi:

- ◆ **`chmod u+x script.sh`**

Aggiunge il diritto di esecuzione per il proprietario per il file `script.sh`

- ◆ **`chmod -R ug+rwX src/*`**

Aggiunge il diritto di scrittura, lettura per il proprietario e il gruppo per i file e contenuti in `src/`, ricorsivamente. Inoltre aggiunge il diritto di esecuzione per le directory

- ◆ **`chmod -R o-rwx $HOME`**

Toglie tutti i diritti a tutti gli utenti che non sono il proprietario e non appartengono al gruppo, ricorsivamente

- ◆ Nota:

Consultate **`info chmod`** per maggiori dettagli

Come cambiare i permessi

- ◆ **Assoluto: `chmod octal-number file(s)`**

<i>User</i>			<i>Group</i>			<i>Others</i>		
<i>R</i>	<i>W</i>	<i>X</i>	<i>R</i>	<i>W</i>	<i>X</i>	<i>R</i>	<i>W</i>	<i>X</i>
4	2	1	4	2	1	4	2	1

- ◆ Esempi:
 - ◆ **`chmod 755 public_html`**
Assegna diritti di scrittura, lettura e esecuzione all'utente, diritti di lettura e esecuzione al gruppo e agli utenti
 - ◆ **`chmod 644 .procmailrc`**
Assegna diritti di scrittura, lettura all'utente, diritti di lettura al gruppo e agli altri

Gestione dei file

- **rm**
 - ReMove (delete) files
- **cp**
 - CoPy files
- **mv**
 - MoVe (or rename) file
- **ln**
 - LiNk creation (symbolic or not)
- **more, less**
 - page through a text file
- **df [options] [directory]**
 - mostra lo spazio libero nei dischi

```
% df -Tm
```
- **du [options] [directory]**

```
% du
% du directory
% du -s directory
% du -k directory
```

Link

- **`ln file hlink`**

- E' un hard-link
- Crea una entry nella directory corrente chiamata *hlink* con lo stesso inode number di *file*
- Il link number dell'inode di *file* viene incrementato di 1

- **`ln -s file slink`**

- E' un link simbolico
- Crea un file speciale nella directory corrente chiamato *slink* che "punta" alla directory entry *file*
- Il link number dell'inode di *file* non viene incrementato

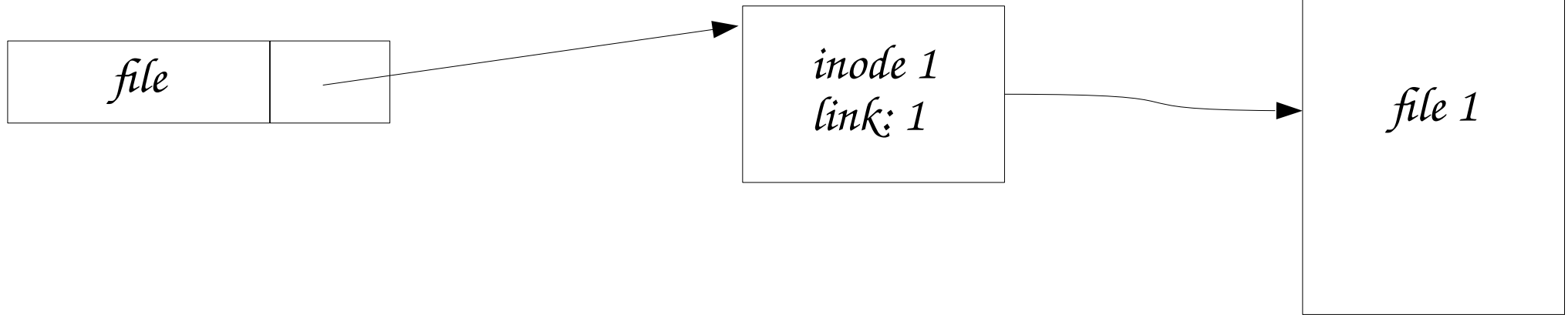
- **Se cancello *file*:**

- hard-link: il link number dell'inode viene decrementato
- link simbolico: il link diviene "stale", ovvero punta ad un file non esistente

Esempio - Situazione iniziale

Directory:

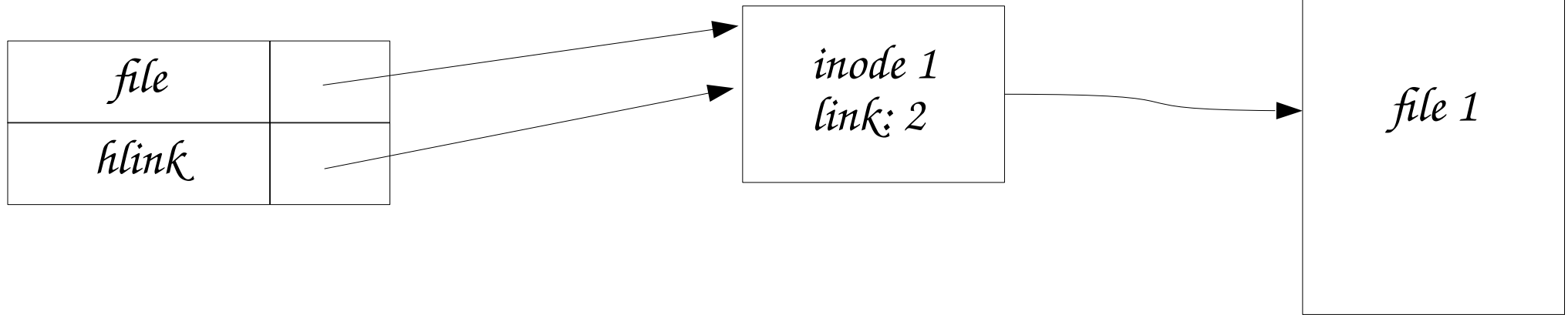
name inode n.



Esempio - Creazione di un hard-link

Directory:

name *inode n.*



Esempio - Creazione di un link simbolico

Directory:

name *inode n.*

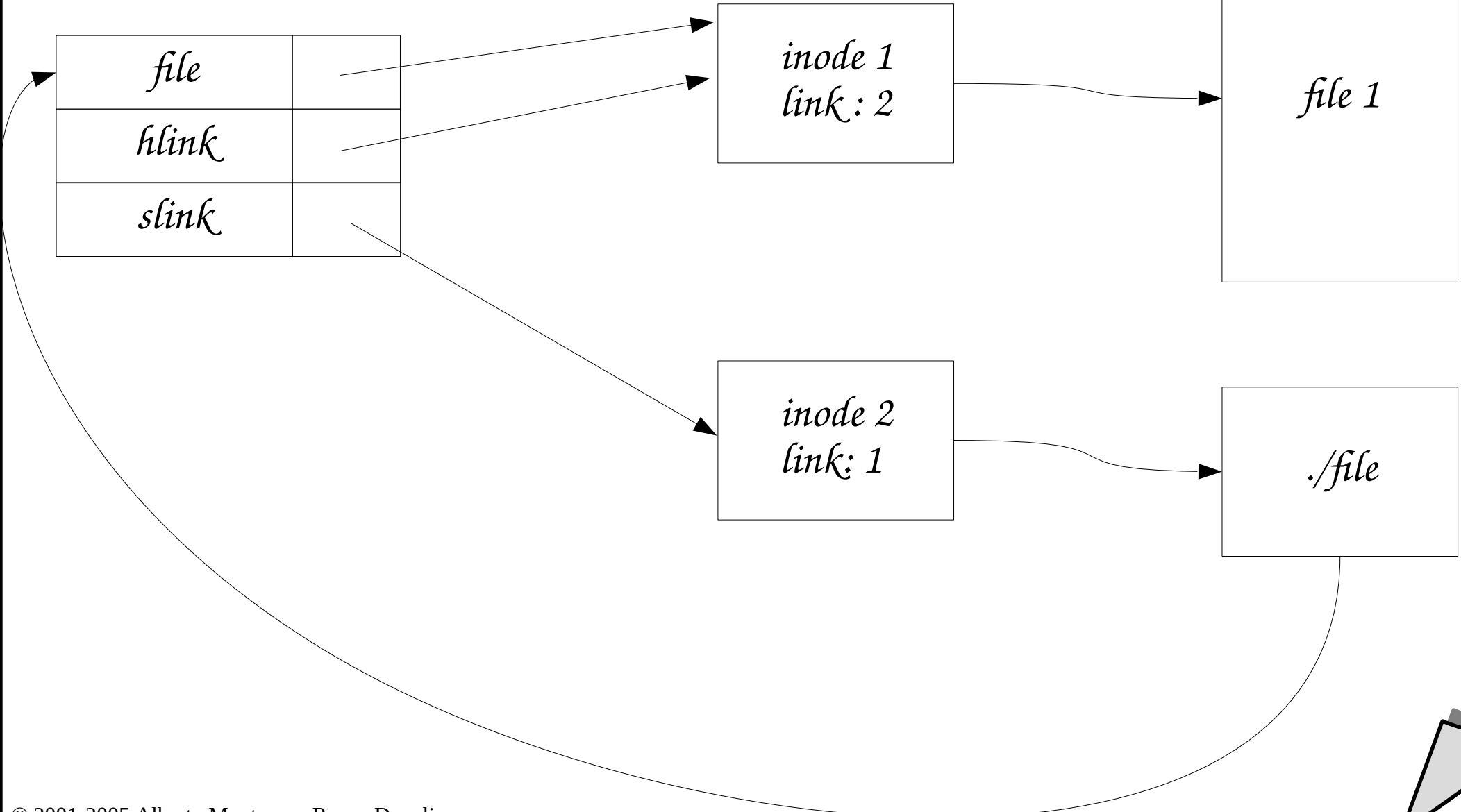
<i>file</i>	
<i>hlink</i>	
<i>slink</i>	

inode 1
link: 2

file 1

inode 2
link: 1

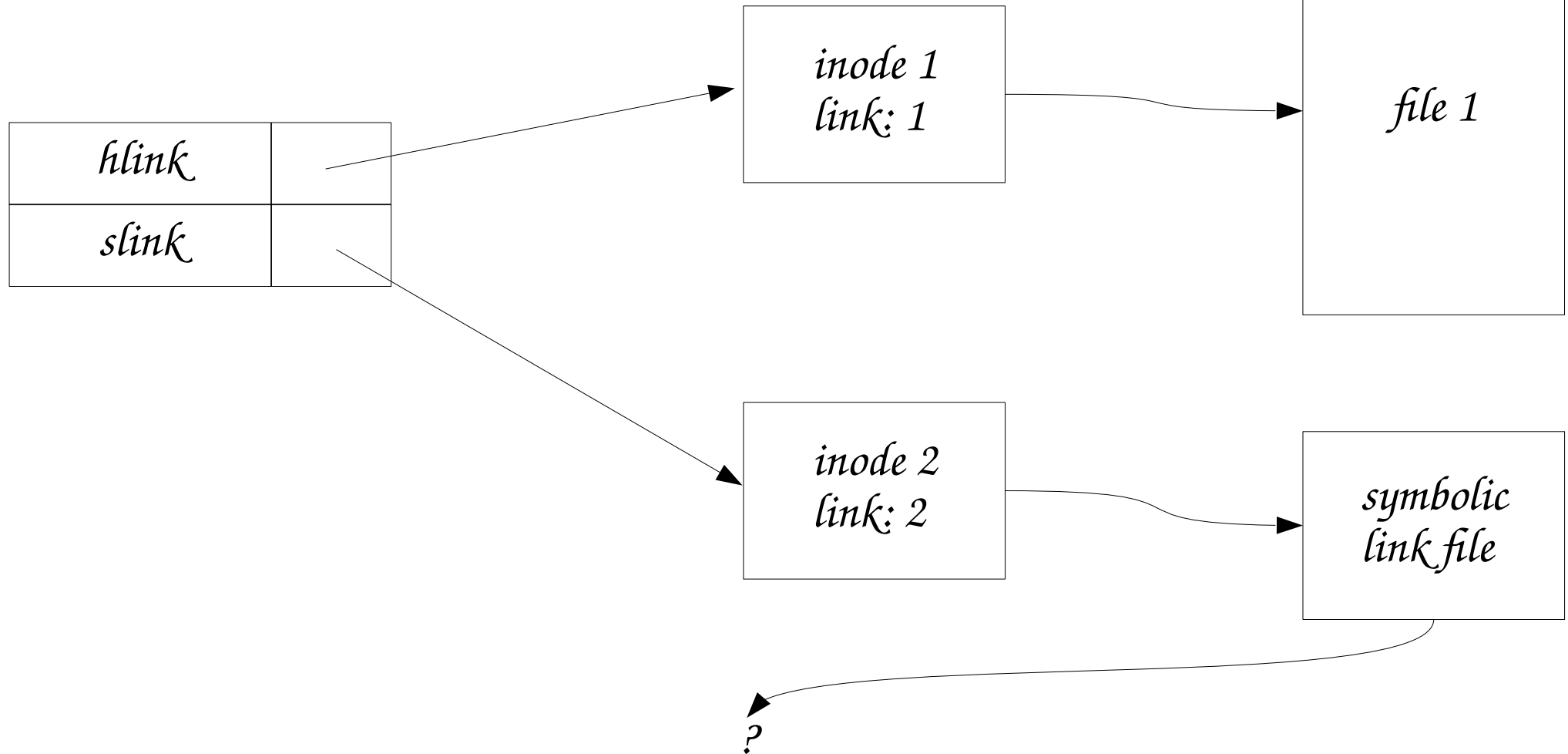
./file



Esempio - Rimozione del file originale

Directory:

name *inode n.*



Comandi per gestione file

- **df [options] [directory]**

- mostra lo spazio libero nei dischi

```
% df -Tm
```

- **du [options] [directory]**

```
% du
```

```
% du directory
```

```
% du -s directory
```

```
% du -k directory
```

Gestione dei processi

▪ **Attributi associati ai processi**

- **pid**
Identificatore del processo
- **ppid**
Parent pid (identificatore del processo padre)
- **nice number**
Priorità statica del processo; può essere cambiata con il comando **nice**
- **TTY**
Terminal device associata al processo
- **real, effective user id**
real, effective group id
Identificatore del owner e del group owner del processo
- **altro:**
memoria utilizzata, cpu utilizzata, etc.

Gestione dei processi

▪ Comando ps

- Riporta lo stato dei processi attivi nel sistema

```
$ ps
```

```
PID TTY TIME CMD
```

```
648 pts/2 00:00:00 bash
```

```
$ ps alx
```

F	UID	PID	PPID	PRI	NI	VSZ	RSS	WCHAN	STAT	TTY	TIME	COMMAND
0	0	1	0	15	0	500	244	1207b7	S	?	0:05	init
0	0	2	1	15	0	0	0	124a05	SW	?	0:00	[keventd]
0	0	3	1	34	19	0	0	11d0be	SWN	?	0:00	[ksoftirqd_CPU0]
0	0	4	1	25	0	0	0	135409	SW	?	0:00	[kswapd]
0	0	5	1	25	0	0	0	140f23	SW	?	0:00	[bdflush]
0	0	6	1	15	0	0	0	1207b7	SW	?	0:00	[kupdated]
0	0	7	1	25	0	0	0	15115f	SW	?	0:00	[kinoded]
0	0	9	1	19	0	0	0	23469f	SW	?	0:00	[mdrecoveryd]
0	0	12	1	15	0	0	0	1207b7	SW	?	0:00	[kreiserfsd]
0	0	150	1	0	-20	0	0	107713	SW<	?	0:00	[lvm-mpd]

Gestione dei processi

▪ **Comando nice**

- esegue un comando con una priorità statica diversa

`nice -n 19 command`

▪ **Comando renice**

- cambia la priorità di un processo in esecuzione

`renice [+ -]value -p pid`

▪ **Comando kill**

- termina un processo

`kill pid`

`kill -9 pid`

Gestione dei processi

- **Processi in foreground:**
 - Processi che "controllano" il terminale da cui sono stati lanciati
 - In ogni istante, un solo processo è in foreground
- **Processi in background**
 - Vengono eseguiti senza "controllare" il terminale a cui sono "attaccati"
- **Job control**
 - Permette di portare i processi da background a foreground e viceversa
- **&** Lancia un processo direttamente in background
Esempio: **long_cmd &**
- **^Z** Ferma (stop) il processo in foreground
- **jobs** Lista i processi in background
- **%n** Si riferisce al processo in background n
Esempio: **kill %1**
- **fg** Porta un processo in background in foreground
Esempio: **fg %1**
- **bg** Fa ripartire in background i processi fermati

2. Shell: nozioni base

Lecture consigliate:

- Fiamingo and Debula, *Introduction to Unix*, sezioni 5-6
- Glass and Ables, *Unix for Programmers and Users*, sezioni 3-6
- Per la sintassi completa dei comandi: **man** e **info**

Tipi di shell

■ Esistono un certo numero di shell

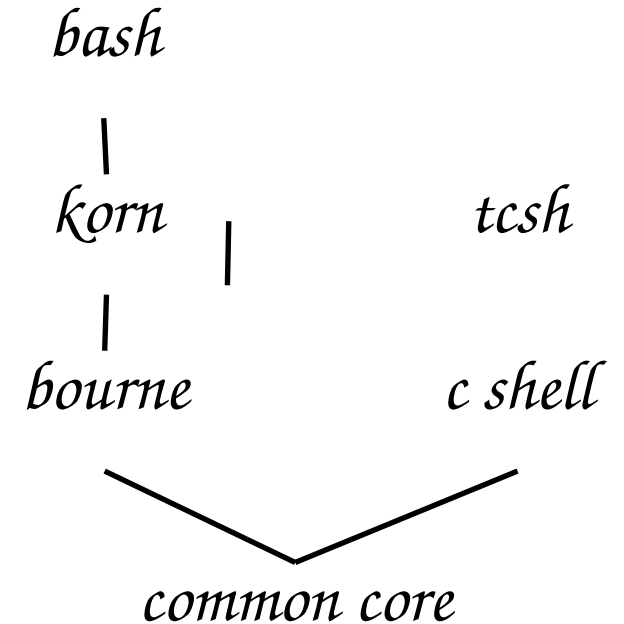
- Bourne shell (*sh*)
- Korn shell (*ksh*)
- C shell (*cs*, *tcsh*)
- **Bash (Bourne-again shell) (*bash*)**

■ Quale shell scegliere?

- Questione di gusti...
- Bash è la più diffusa in ambiente Linux

■ Piano di attacco

- Oggi vedremo le caratteristiche comuni (common core)
- Poi ci concentreremo sulla bash



Selezionare una shell

- **Quando vi viene fornito un account UNIX**

- Una shell viene selezionata per voi

- **Per vedere che shell state utilizzando:**

```
% echo $SHELL
# display the content of variable SHELL
```

- **Per cambiare shell:**

```
% chsh [<username>]
# ask for the full pathname of the new shell
```

- **Esempio:**

```
% echo $SHELL
/bin/bash
% chsh
New shell [/bin/bash]: /bin/csh
```

Caratteristiche delle Shell

- ◆ **Sommario:**

- ◆ Comandi built-in
- ◆ Meta caratteri
- ◆ Redirezione dell'I/O
- ◆ Pipes
- ◆ Wildcards
- ◆ Command substitution
- ◆ Sequenze
 - ◆ Condizionali
 - ◆ Non condizionali
- Raggruppamento di comandi
- Esecuzione in background
- Quoting
- Sub-shell
- Variabili
 - Locali
 - Di ambiente
- Script
- Here document

Comandi built-in

- **Comandi:**

- Esterni

Quando richiedete l'esecuzione di un comando esterno, il corrispondente file eseguibile viene cercato, caricato in memoria ed eseguito

Esempio: il comando **ls** si trova in **/bin/ls**

- Interni, o built-in

Il comando viene riconosciuto ed eseguito direttamente dalla shell

Esempi:

echo # displays all of its arguments

 # to standard output

cd # change directory

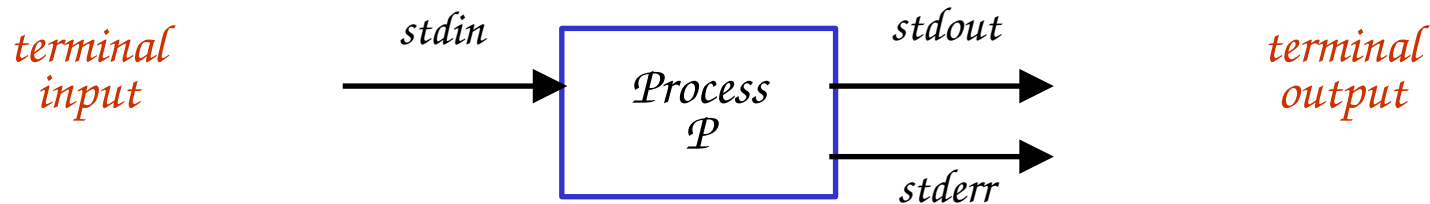
Metacaratteri

■ Caratteri speciali

- `>`, `>>`, `<` redirezione I/O
- `|` pipe
- `*`, `?`, `[...]` wildcards
- `'command'` command substitution
- `;` esecuzione sequenziale
- `||`, `&&` esecuzione condizionale
- `(...)` raggruppamento comandi
- `&` esecuzione in background
- `""`, `' '` quoting
- `#` commento *
- `$` espansione di variabile
- `\` carattere di escape *
- `<<` “here documents”

Redirezione dell'input/output, pipes

- **Ogni processo è associato a tre “stream”**
 - Standard output (stdout)
 - Standard input (stdin)
 - Standard error (stderr)
- **Redirezione dell'I/O e pipe permettono:**
 - “Slegare” questi stream dalle loro sorgenti/destinazioni abituali
 - “Legarli” ad altri sorgenti / destinazioni



Redirezione dell'input/output

▪ **Redirection**

- Salvare l'output di un processo su un file (output redirection)
- Usare il contenuto di un file come input di un processo

▪ **Esempi:**

- Salva l'output di `ls` in `list.txt`

```
ls > list.txt
```

- Aggiunge l'output di `ls` a `list.txt`

```
ls >> list.txt
```

- Spedisce il contenuto di `list.txt` a montreso@phd.cs.unibo.it

```
mail montreso@phd.cs.unibo.it < list.txt
```

- Redireziona `stdout` e `stderr` del comando `rm` al file `/dev/null`

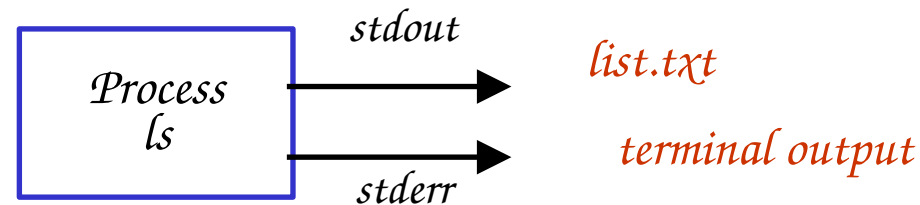
```
rm file >& /dev/null
```

Redirezione dell'input/output

```
mail montreso@phd.cs.unibo.it < list.txt
```



```
ls > list.txt
```



Nota:

esistono esempi di redirection più complessi; consultate `info bash` per i dettagli

Pipes

■ Pipe, o catena di montaggio:

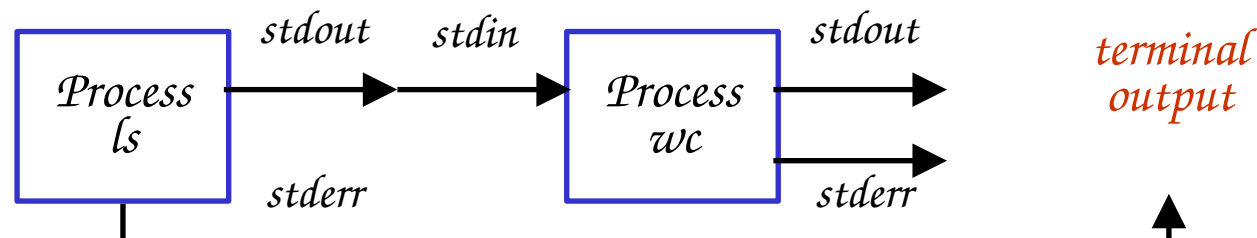
- La shell vi permette di usare lo standard output di un processo come standard input di un altro processo

- Esempio:

```
% ls  
a b c d f1 f2
```

```
% ls | wc -w
```

```
6
```



■ Usage: tee -ia <pathname>

- Copia il contenuto dello standard input sul file specificato e sullo standard output; -a esegue l'append
- il nome **tee** deriva dai "giunti a T" usati dagli idraulici
- Esempio:

```
% who | tee who.capture | sort
```

Wildcards

- **Utilizzati per specificare file pattern**

- la stringa contenente wildcards viene sostituita con l'elenco dei file che soddisfano la condizione
- Caratteri speciali:
 - * matching di qualsiasi stringa
 - ? matching di qualsiasi carattere singolo
 - [...] matching di qualsiasi carattere inserito nelle parentesi
- Esempi:
 - *.c
 - prova00?.c
 - prova[0-9][0-9][0-9].txt

Command substitution

- **Gli apici ' ' sono utilizzati per fare *command substitution***
 - Il comando racchiuso fra apici viene eseguito, e il suo standard output viene sostituito al posto del comando
 - Esempi:
 - % echo Data odierna: 'date'
 - % echo Utenti collegati: 'who | wc -l'
 - % tar zcvf src-'date'.tgz src/

Sequenze (condizionali e non)

■ Sequenze non condizionali

- Il metacarattere ; viene utilizzato per eseguire due comandi in sequenza
- Esempio:

```
% date ; pwd ; ls
```

■ Sequenze condizionali

- || viene utilizzato per eseguire due comandi in sequenza, solo se il primo ha un exit code uguale a **1** (*failure*)
- && viene utilizzato per eseguire due comandi in sequenza, solo se il primo ha un exit code uguale a **0** (*success*)
- Esempi:

```
% gcc prog.c && a.out  
% gcc prog.c || echo Compilazione fallita
```

Raggruppamento di comandi

▪ Raggruppamenti

- E' possibile raggruppare comandi racchiudendoli dentro delle parentesi
- Vengono eseguiti in una subshell
- Condividono gli stessi stdin, stdout e stderr
- Esempi:

```
% date ; ls ; pwd > out.txt
```

```
% (date ; ls ; pwd) > out.txt
```

Esecuzione in background

- **Se un comando è seguito dal metacarattere &:**

- Viene creata una subshell
- il comando viene eseguito in background, in concorrenza con la shell che state utilizzando
- non prende il controllo della tastiera
- utile per eseguire attività lunghe che non necessitano di input dall'utente
- Esempi:

```
% find / -name passwd -print &
```

```
20123
```

```
% /etc/passwd
```

```
% find / -name passwd -print &> results.txt &
```

```
20124
```

```
%
```

Quoting

- ◆ **Esiste la possibilità di disabilitare wildcard / command substitution / variable substitution**

- ◆ Single quotes ' inibiscono wildcard, command substitution, variable substitution

- ◆ Esempio:

```
% echo 3 * 4 = 12
```

```
% echo '3*4 = 12'
```

- ◆ Double quotes " inibiscono wildcard e basta

- ◆ Esempio:

```
% echo " my name is $name - date is 'date' "
```

```
% echo ' my name is $name - date is 'date' '
```

Subshells

- ♦ **Quando aprite un terminale, viene eseguita una shell**
- ♦ **Viene creata una child shell (o subshell)**
 - ♦ Nel caso di comandi raggruppati, come (**date; ls ; pwd**)
 - ♦ Quando viene eseguito un script
 - ♦ Quando viene eseguito un processo in background
- ♦ **Caratteristiche delle subshell:**
 - ♦ Hanno la propria directory corrente:
% (**cd / ; pwd**)
/
% **pwd**
/home/montreso
 - ♦ Due distinte aree di variabili vengono gestite differientemente (vedi dopo)

Variabili

- ◆ **Ogni shell supporta due tipi di variabili:**
 - ◆ *Variabili locali*: non ereditate da una shell alle subshell create da essa
 - ◆ Utilizzate per computazioni locali all'interno di uno script
 - ◆ *Variabili di ambiente*: ereditate da una shell alle subshell create da essa
 - ◆ Utilizzate per comunicazioni fra parent child shell
- ◆ **Entrambe le variabili contengono dati di tipo stringa**
- ◆ **Ogni shell ha alcune variabili di ambiente inizializzate da file di startup o dalla shell stessa**
 - ◆ `$HOME`, `$PATH`, `$MAIL`, `$USER`, `$SHELL`, `$TERM`, etc.
 - ◆ Per visualizzare l'elenco completo, usate il comando `env`

Utilizzo delle variabili

- **Per accedere al contenuto di una variabile:**

- Utilizzate il metacarattere \$
- **\$name** è la versione abbreviata di **\${name}** (a volte è necessario)

- **Per assegnare un valore ad una variabile:**

- Sintassi diversa a seconda della shell
- Nel caso di bash

```
nome=va lo re           # problem with spaces  
nome="va lo re"       # no problem with spaces
```

- Variabili dichiarate in questo modo sono locali
- Per trasformare una variabile locale in una d'ambiente, usate il comando export

```
% export nome
```

- Nota: nel caso di csh
% **setenv name value**

Esempio

- **Uso di variabili locali e d'ambiente:**

```
% firstname="Alberto"  
% lastname="Montresor"  
% echo $firstname $lastname  
% export firstname  
% bash  
% echo $firstname $lastname  
% exit  
% echo $firstname $lastname
```

3. Shell scripting

Lecture consigliate:

- “*Advanced Bash-Scripting Guide*”, di Mendel Cooper
- Per la sintassi completa dei comandi: **man** e **info**

Script

- **Cos'è uno script?**
 - Qualunque sequenza di comandi può essere registrata in un file di testo per poi essere eseguita
 - Utili per eseguire sequenze di comandi ripetitive / automatiche
- **Per eseguire uno script:**
 - Scrivete il vostro script in un file
 - Lo rendete eseguibile tramite il comando **chmod**
 - Digitate il nome del file (previa raggiungibilità via path)
- **Azioni della shell per eseguire uno script**
 - Determina quale shell deve essere utilizzate per eseguire lo script
 - Crea una sottoshell che esegue lo script
 - Il file dello script viene passato come argomento alla sottoshell
 - Il contenuto del file viene utilizzato come input della shell

Selezione della shell

- **La selezione di quale shell eseguirà lo script è basata sulla prima riga dello script stesso:**
 - se contiene solo un simbolo #, viene utilizzata la shell da cui lo script è stato lanciato
 - se contiene solo un simbolo **#!*pathname***, viene utilizzata la shell identificata da *pathname*
 - Esempio: per eseguire uno script con la Korn shell
#!/bin/ksh
 - Forma raccomandata (completamente non ambigua)
 - altrimenti, viene interpretato da una Bourne shell
- **Alcuni comportamenti interessanti:**
 - #!/bin/rm**
 - #!/bin/cat**

Variabili built-in per lo scripting

- **\$\$** l'identificatore di processo della shell (*sh*)
- **\$0** il nome dello shell script (*sh, csh*)
- **\$1 - \$9** \$n è l'n-esimo argomento della linea di comando (*sh, csh*)
- **\${n}** \${n} è l'n-esimo argomento della linea di comando (*sh, csh*)
- **\$*** Lista tutti gli argomenti di command line (*sh, csh*)
- **\$#** Numero di argomenti sulla command line (*sh*)
- **\$?** Valore di uscita dell'ultimo comando eseguito (*sh*)

Esempio di script

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

```
a=23          # Simple case
```

```
echo $a
```

```
b=$a
```

```
echo $b
```

```
# Now, getting a little bit fancier...
```

```
a=`echo Hello!` # Assigns result of 'echo' command to 'a'
```

```
echo $a
```

```
a=`ls -l`      # Assigns result of 'ls -l' command to 'a'
```

```
echo $a
```

```
exit 0
```


Esempio di script

```
#!/bin/bash
# This is a simple script that removes blank lines from a file.
# No argument checking.
# Same as
#   sed -e '/^$/d' filename
# invoked from the command line.
sed -e /^$/d "$1"
# The '-e' means an "editing" command follows (optional here).
# '^' is the beginning of line, '$' is the end.
# This match lines with nothing between the beginning and the end,
# blank lines.
# The 'd' is the delete command.
# Quoting the command-line arg permits
# whitespace and special characters in the filename.
exit 0
```

Here Document

- **<command> << <word>**
<command> <</ <word>
 - copia il proprio standard input fino alla linea che inizia con la parola <word> (esclusa) e quindi esegue <command> utilizzando questi dati copiati come standard input
 - << esegue variable substitution
 - <</ non esegue variable substitution
 - Esempio:

```
#!/bin/bash
mail $1 << ENDOFTEXT
Ciao,
    puoi passare da me alle $2?
Alberto
ENDOFTEXT
echo Mail sent to $1
```

Espressioni

▪ Valutazione espressioni

- La shell non supporta direttamente la valutazione di espressioni
- Esiste l'utilità **`expr`** *expression*
 - Valuta l'espressione e spedisce il risultato allo standard output
 - Tutti i componenti dell'espressione devono essere separati da spazi
 - Tutti i metacaratteri (ad es. *) devono essere prefissi da un backslash (ad es. *)
 - Il risultato può essere numerico oppure stringa
 - Il risultato può essere assegnato ad una variabile facendo uso opportuno della command substitution

Espressioni

▪ Operatori

- Aritmetici: + - * / %
- Confronto: = != > < >= <=
- Logici: &, |, !
- Parentesi: ()
 - Nota: devono essere prefissate dallo backslash
- Stringa:
 - *match string regularExpression*
 - *substr string start length*
 - *length string*
 - *index string charList*
- Nota: vedi *espressioni regolari* nelle slide successive

Espressioni: esempi

```
% x=1
```

```
% x='expr $x + 1'
```

```
% echo $x
```

```
2
```

```
% echo 'expr $x \* $x'
```

```
4
```

```
% echo 'expr length "cat"'
```

```
3
```

```
% echo 'expr index "donkey" "key"'
```

```
4
```

```
% echo 'expr substr "donkey" 4 3'
```

```
key
```

Exit status

- **Quando un processo termina, ritorna un exit status**
- **Convenzioni UNIX:**
 - 0 *success* (TRUE)
 - non-zero *failure* (FALSE)
- **Comando `exit nn`**
 - Lo script termina con exit status **nn**
- **Built-in variable `$?`**
 - Ritorna l'exit status dell'ultimo comando eseguito

Exit status

```
% cat script.sh
```

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

```
echo hello
```

```
echo $?      # Exit status 0 returned (success).
```

```
lskdf       # Unrecognized command.
```

```
echo $?      # Non-zero exit status returned.
```

```
exit 113    # Will return 113 to shell.
```

```
% ./script.sh
```

```
hello
```

```
0
```

```
./prova: lskdf: command not found
```

```
127
```

Condizioni

- **Condizione = exit status**

- Nelle strutture di controllo delle shell, gli exit status di un comando vengono utilizzati come condizioni nei test

- **Esempio:**

```
if cmp a b > /dev/null # Suppress output.  
then echo "Files a and b are identical."  
else echo "Files a and b differ."  
fi
```


Condizioni

▪ **Utility test <expression>**

- ritorna un exit code 0 se l'espressione viene valutata vera
- ritorna un exit code 1 se l'espressione viene valutata falsa

▪ **Espressioni valutate da test**

- **int1 -eq int2** true if integer **int1** == integer **int2**
- **int1 -ne int2** true if integer **int1** != integer **int2**
- **int1 -ge int2** true if integer **int1** >= integer **int2**
- **int1 -gt int2** true if integer **int1** > integer **int2**
- **int1 -le int2** true if integer **int1** <= integer **int2**
- **int1 -lt int2** true if integer **int1** < integer **int2**

Condizioni

▪ **Espressioni booleane**

- `!expr` true if `expr` is false
- `expr1 -a expr2` true if `expr1` and `expr2`
- `expr1 -o expr2` true if `expr1` or `expr2`
- `\(\)` grouping expressions

▪ **Confronti fra stringhe**

- `str1 = str2` true if strings `str1`, `str2` are equal
- `str1 != str2` true if strings `str1`, `str2` are not equal
- `-z string` true if string `string` is empty
- `-n string` true if string `string` contains at least one character

Condizioni

▪ **Espressioni su file**

- **-d filename** true if **filename** exists as directory
- **-f filename** true if **filename** exists as regular file
- **-r filename** true if **filename** exists and is readable
- **-w filename** true if **filename** exists and is writable
- **-x filename** true if **filename** exists and is executable
- **-s filename** true if **filename** exists and is not empty
- etc.

Valutazione di condizioni

- **Sinonimo per test: []**

- Esegue la condizione contenuta tra parentesi quadre
- [è un comando built-in

- **Sinonimo per test: [[]]**

- Extended test command
- Esegue la condizione contenuta tra doppie parentesi quadre
- Non viene effettuato filename expansion tra [[e]]
- Preferibile a [], in quanto operatori come && || > < vengono interpretati correttamente in [[]]
- [[]]

Esempi

- **Controlla l'esistenza di un argomento**

```
if [ -n "$1" ]  
then  
    lines=$1  
fi
```

- **Esegue un change dir e verifica la directory corrente**

```
cd $LOG_DIR  
if [ 'pwd' != "$LOG_DIR" ]  
then  
    echo "Can't change to $LOG_DIR."  
    exit $ERROR_XCD  
fi
```

Strutture di controllo: **if** - **then** - **elif** - **else**

▪ **Significato**

- I comandi in **list1** vengono eseguiti
- Se l'ultimo comando in **list1** ha successo, **list2** viene eseguito
- Altrimenti, **list3** viene eseguito
- Se l'ultimo comando in **list3** ha successo, **list4** viene eseguito
- Altrimenti viene eseguito **list5**

▪ **Nota:**

- Un costrutto **if** può contenere zero o più sezioni **elif**
- La sezione **else** è opzionale

▪ **Struttura**

```
if list1  
then  
    list2  
elif list3  
then  
    list4  
else  
    list5  
fi
```

Esempio: `if - then - elif - else`

```
#!/bin/bash

stop=0

while [[ $stop -eq 0 ]]
do
    cat << ENDOFMENU

    1:      administrator
    2:      student
    3:      teacher

    ENDOFMENU
```

Esempio: `if` - `then` - `elif` - `else`

```
echo "Your choice?"  
read reply  
if [[ "$reply" = "1" ]]; then  
    administrator  
elif [[ "$reply" = "2" ]]; then  
    student  
elif [[ "$reply" = "3" ]]; then  
    teacher  
else  
    echo "Error"  
endif
```


Strutture di controllo: **case - in - esac**

▪ **Formato:**

- *expression* è una espressione di stringa
- *pattern* può contenere wildcard

▪ **Significato:**

- *espressione* viene valutata e confrontata con ognuno dei pattern (dal primo all'ultimo)
- quando il primo matching *pattern* viene trovato, la lista di comandi associati viene eseguita
- dopo di che si salta al **esac** corrispondente

▪ **Struttura**

```
case expression in  
pattern1 )  
list1  
;;  
pattern2 )  
list2  
;;  
esac
```

Esempio: case - in - esac

```
#!/bin/bash
# stampa utenti che consumano piu' spazio su HD
case "$1" in
    "") lines=50
        ;;
    *[^0-9]*) echo "Usage: `basename $0` usersnum";
        exit 1
        ;;
    *) lines=$1
        ;;
esac
du -s /home/* | sort -gr | head -$lines
```

Strutture di controllo: **while** - **do** - **done**

▪ **Formato:**

- *list1* è una lista di comandi
- *list2* è una lista di comandi

▪ **Significato:**

- Il comando **while** esegue i comandi in *list1* e termina se l'ultimo comando in *list1* fallisce
- altrimenti, i comandi in *list2* sono eseguiti e il processo viene ripetuto
- un comando **break** forza l'uscita istantanea dal loop
- un comando **continue** forza il loop a riprendere dalla prossima iterazione

Struttura

```
while list1  
do  
    list2  
done
```

Esempio: while - do - done

```
#!/bin/bash
# tabelline

if [ "$1" -eq "" ]; then
    echo "Usage: $0 max"
    exit
fi

x=1

while [ $x -le $1 ]
do
    y=1
    while [ $y -le $1 ]
    do
        echo 'expr $x \* $y' " "
        y='expr $y + 1'
    done
    echo
    x='expr $x + 1'
done
```

Strutture di controllo: **until** - **do** - **done**

▪ **Formato:**

- *list1* è una lista di comandi
- *list2* è una lista di comandi

▪ **Significato:**

- Il comando **until** esegue i comandi in *list1* e termina se l'ultimo comando in *list1* ha successo
- altrimenti, i comandi in *list2* sono eseguiti e il processo viene ripetuto
- un comando **break** forza l'uscita istantanea dal loop
- un comando **continue** forza il loop a riprendere dalla prossima iterazione

▪ **Struttura**

```
until list1  
do  
    list2  
done
```

Strutture di controllo: **for** - **in** - **do** - **done**

▪ **Formato:**

- *name* è un identificatore di variabile
- *words* è una lista di parole
- *list* è una lista di comandi

▪ **Significato:**

- Il comando **for** esegue un ciclo variando il valore della variabile *name* per tutte le parole nella lista *words*
- I comandi in *list* vengono valutati ad ogni iterazione.
- Se la clausola **in** è omessa, **\$*** viene utilizzato al suo posto.
- è possibile utilizzare **break** e **continue**

▪ **Struttura**

```
for name [in words]  
do  
    list  
done
```

Esempio: **for** - **in** - **do** - **done**

```
#!/bin/bash
for color in red yellow green blue
do
    echo one color is $color
done
```


Output:

```
one color is red
one color is yellow
one color is green
one color is blue
```

Esempio: for - in - do - done

```
#!/bin/bash
# Cancella i file di backup di emacs, previo consenso
for file in $(ls *~)
do
    echo -n "Sei sicuro di voler rimuovere $file ?"
    read reply
    if [ $reply = "Y" -o $reply = "y" ]; then
        rm -f $file
        echo File $file removed
    fi
done
```

*Forma alternativa di command substitution; equivalente a 'ls *~'*



Esempio: `for - in - do - done`

```
#!/bin/bash
# bin-grep.sh: Locates matching strings in a binary file.
# A "grep" replacement for binary files. Similar to grep -a
EBADARGS=65
ENOFIELD=66
if [ $# -ne 2 ]; then
    echo "Usage: `basename $0` string filename"
    exit $E_BADARGS
fi
if [ ! -f "$2" ]; then
    echo "File \"$2\" does not exist."
    exit $E_NOFILE
fi
```

Esempio: **for - in - do - done**

```
# The "strings" command lists strings in binary files.  
for word in $( strings "$2" | grep "$1" )  
    # Output then piped to "grep", which tests for desired  
    # string.  
do  
    echo $word  
done
```

Esercizio 1: saferm

■ Descrizione

- Scrivere uno script che abbia le funzionalità di **rm**, ma che invece di cancellare definitivamente i file li sposti in una directory **.trash** nella vostra home
- Usage:
 - **saferm -L** elenca il contenuto del cestino
 - **saferm -P** svuota (“purge”) il cestino
 - **saferm -R files** ripristina il file *file*
 - **saferm files** rimuove i file spostandoli nel cestino
- Nota: le varie opzioni sono esclusive; ovvero, non si può lanciare un comando **saferm -L -P** ; generate un errore e stampate l’usage dello script nel caso

Esercizio 1: saferm

■ Gestione di file con lo stesso nome:

- se un nome di file da inserire nel cestino esiste già, rinominare il file esistente concatenando la sua data
- suggerimento: utilizzate **date -r +%s**
- Esempio:
 - Se nel cestino c'è un file **prova.sh**, e volete aggiungere un altro file prova.sh, rinominate il primo come "**prova.sh.1030606290**" e poi spostate il secondo nel cestino

■ Estensioni

- tenete conto della possibilità di ripristinare file precedenti

Comandi built-in: I/O

▪ **echo**

- Stampa i suoi argomenti su stdout
- **echo -n** evita che sia aggiunto un newline
- **echo**, utilizzato con command substitution, può essere utilizzato per settare una variabile
 - Es: **a=`echo "HELLO" | tr A-Z a-z`**

▪ **printf**

- Versione migliorata di **echo**, con sintassi simile a quella di printf in C
- Per dettagli, vedere manuale (**info printf**)
- Esempio:
PI=3.14159265358979
PI5=\$(printf "%.5f" \$PI)

Comandi built-in: I/O

▪ **read**

- **read var** legge l'input da una tastiera e lo copia nella variabile **var**
- Il comando **read** ha qualche opzione interessante che permette di ottenere keystroke senza premere ENTER
- Esempio:

```
read -s -n1 -p "Hit a key " keypress  
echo; echo "Keypress was \"$keypress\"."
```

- opzione **-s** significa no echo input
- opzione **-nN** significa accetta solo N caratteri di input
- opzione **-p** significa stampa il prompt seguente

Comandi built-in: variabili

▪ **declare o typeset (sinonimi)**

- Permettono di restringere le proprietà di una variabile
- Una forma “debole” di gestione dei tipi
- Opzioni:
 - **declare -r var** Dichiarare una variabile come read-only
 - **declare -i var** Dichiarare la variabile come intera
 - **declare -a var** Dichiarare la variabile come array
 - **declare -x var** Esportare la variabile

▪ **Sinonimi**

- **export** è equivalente a **declare -x**
- **readonly** è equivalente a **declare -r**

Comandi built-in: variabili

▪ **let**

- Il comando **let** esegue operazioni aritmetiche sulle variabili. In molti casi, funziona come una versione più semplice di **expr**
- Esempi:

let a=11 # Assegna 11 ad a

let "a <<= 3" # Shift a sinistra di tre posizioni

let "a /= 4" # Divisione per 4 di a

let "a -= 5" # Sottrazione per 5

▪ **unset**

- Cancella il contenuto di una variabile

Comandi built-in: script execution

- **source (dot command, .)**

- Esegue uno script, senza aprire una subshell
- Corrisponde alla direttiva **#include** del preprocessore c
- Esempio:

```
#!/bin/bash
# cr.sh (change root)
cd /
```

- Dalla linea di comando:

```
% pwd
/home/montreso
% ./cr.sh
% pwd
/home/montreso
```

```
% pwd
/home/montreso
% source cr.sh
% pwd
/
```

Comandi built-in: comandi vari

- **true**
 - Ritorna sempre un exit status 0 (successo)
- **false**
 - ritorna sempre un exit status 1 (fallimento)
- **type [cmd]**
 - stampa un messaggio che descrive se il comando *cmd* è una keyword, è un comando built-in oppure un comando esterno
- **shopt [options]**
 - setta alcune opzioni della shell
 - Es: **shopt -s cdspell** permette misspelling in **cd**

Comandi built-in: comandi vari

▪ **alias, unalias**

- Un **alias** Bash non è altro che una scorciatoia per abbreviare lunghe sequenze di comandi
- Esempio:
 - `alias dir="ls -l"`
 - `alias rd="rm -r"`
 - `unalias dir`
 - `alias h="history -r 10"`

▪ **history**

- permette di visualizzare l'elenco degli ultimi comandi eseguiti

Comandi built-in: directory stack

▪ Descrizione

- La shell Bash (funzionalità ereditata dalla C shell) permette di creare e manipolare una pila di directory, utile per visitare un albero di directory.

▪ Comandi:

- Comando **dirs**
Stampa il contenuto del directory stack
- Comando **pushd *dirname***
Push della directory *dirname* nello stack; si sposta nella directory *dirname*; è automaticamente seguito dal comando **dirs**
- Comando **popd**
Pop dallo stack (rimuove il top element); si sposta sull'attuale top element; è automaticamente seguito dal comando **dirs**
- Variabile **\$DIRSTACK**
Contiene il top element dello stack

Esempio: directory stack

```
% pushd /dir1
/dir1 /home/montreso
% pwd
/dir1
% pushd /dir2
/dir2 /dir1 /home/montreso
% pwd
/dir2
% popd
/dir1 / home/montreso
% pwd
/dir1
% popd
/home/montreso
```

File di configurazione: inizializzazione

- **/etc/profile**
 - system wide defaults, mostly setting the environment (all Bourne-type shells, not just Bash [1])
- **/etc/bashrc**
 - system wide functions and aliases for Bash
- **\$HOME/.bash_profile**
 - user-specific Bash environmental default settings, found in each user's home directory (the local counterpart to **/etc/profile**)
- **\$HOME/.bashrc**
 - user-specific Bash init file, found in each user's home directory (the local counterpart to **/etc/bashrc**). Only interactive shells and user scripts read this file.

File di configurazione: terminazione

- **\$HOME/.bash_logout**

- user-specific instruction file, found in each user's home directory. Upon exit from a login (Bash) shell, the commands in this file execute.

- **Esempio: file di configurazione (.bashrc)**

```
alias rm="rm -i"  
alias cp="cp -i"  
alias mv="mv -i"  
alias mnt="mount /mnt/zip"
```

Esempio: file di configurazione (/etc/profile)

```
if ! echo $PATH | /bin/grep -q "/usr/X11R6/bin" ; then
    PATH="$PATH:/usr/X11R6/bin"
fi
USER='id -un'
HOSTNAME='/bin/hostname'
for i in /etc/profile.d/*.sh; do
    if [ -x $i ]; then
        . $i
    fi
done
unset i # not needed; just an example
```


Comandi: cat – tac - rev

- **Comando cat (concatenate):**

- Stampa i file su stdout. Utilizzato con redirection, server a concatenare file:

```
cat file.1 file.2 file.3 > file.123
```

- **Comando tac (inverso di cat):**

- Stampa le linee dei file in ordine inverso, partendo dall'ultima linea

- **Comando rev (reverse):**

- Stampa (invertite) le linee dei file, partendo dalla prima linea

Comando: find

- **Comando: find *pathlist expression***

- L'utility **find** analizza ricorsivamente i path contenuti in ***pathlist*** e applica ***expression*** ad ogni file

- **Sintassi di expression:**

- ***-name pattern***

True se il nome del file fa matching con pattern, che può includere i metacaratteri di shell: * [] ?

- ***-perm permission***

True se permission corrisponde ai permessi del file

- ***-print***

stampa il pathname del file e ritorna true

Comando: find

- **Sintassi di expression:**

- **-ls**

Stampa gli attributi del file e ritorna true

- **-user *username*, -uid *userId***

True se il possessore del file è *username / userId*

- **-group *groupname*, -gid *groupId***

True se il gruppo del file è *groupname / groupId*

- **-atime | -mtime | -ctime -count**

True se il file è stato “acceduto | modificato | modificato oppure cambiati gli attributi” negli ultimi *count* giorni

Comando: find

- **Sintassi di expression:**

- **-type b|c|d|p|f|l|s**

True se il file è di tipo a blocchi | a caratteri | una directory | un named pipe | un file regolare | un link | un socket

- **-exec *command***

True se l'exit status di *command* è 0. *command* deve essere terminato da un "escaped ;" (\;). Se si specifica il simbolo {} come argomento di *command*, esso viene sostituito con il pathname del file corrente

- **-not, !, -a, -and, -o, -or**

Operatori logici (not, and, or)

Comando find: esempi

- **find . -name "*.c" -print**
 - Stampa il nome dei file sorgente C nella directory corrente e in tutte le sottodirectory
- **find / -mtime -14 -ls**
 - Elenca tutti i file che sono stati modificati negli ultimi 14 giorni
- **find . -name "*.bak" -ls -exec rm {} \;**
 - Cancella tutti i file che hanno estensione .bak
- **find . \(-name "*.c" -o -name "*.txt" \) **
-print
 - Elenca i nomi di tutti i file che hanno estensione .c e .txt

Comando: xargs

■ Comando: `xargs command`

- `xargs` legge una lista di argomenti dallo standard input, delimitati da blank oppure da newline
- esegue ***command*** passandogli la suddetta lista di argomenti

■ Esempio: concatena tutti i file *.c

```
% find -name "*.c" -print
```

```
./a.c
```

```
./b.c
```

```
% find -name "*.c" -print | xargs cat > prova
```

```
%
```

Esempi: find e xargs

- `emacs $(find . -name "*.java" | xargs grep -l \ "Alfa")`
 - Utilizza emacs per visualizzare tutti i file nella directory corrente e nelle sottodirectory che hanno estensione java e contengono la parola "Alfa".
- `find ~ \(-name "*~" -o "#*#" \) -print | \
xargs --no-run-if-empty rm -vf`
 - Rimuove tutti i file di backup o temporanei di emacs dalla home directory (ricorsivamente)
- `find . -type d -not -perm ug=w | xargs \
chmod ug+w`
 - Aggiunge il diritto di scrittura a tutti le directory nella directory corrente e nel suo sottoalbero

Spazi e caratteri speciali

- `find ~ \(-name "*~" -o "#*#" \) -print0 | \`
`xargs --no-run-if-empty --null rm -vf`
- **Attenzione agli spazi e ai caratteri speciali:**
 - `xargs` separa i nomi dei file tramite spazi o new line
 - il file *relazione 1.txt* viene trattato come due file
 - l'opzione `-print0` di `find` stampa stringhe null-terminated
 - l'opzione `--null` di `xargs` prende stringhe null-terminated
 - questa versione gestisce correttamente qualunque tipo di carattere speciale (come ")

Esempio

```
#!/bin/bash

# Move (verbose) all files in current directory
# to directory specified on command line.

if [ -z "$1" ]; then
    echo "Usage: `basename $0` directory-to-copy-to"
    exit 65
fi

ls . | xargs -i -t mv {} $1

# This is the exact equivalent of mv * $1
# unless any of the filenames has "whitespace" characters.

exit 0
```

Comandi per la gestione del testo

- **Comando: head [-n] file**
 - Lista le prime n linee di un file (10 default)
- **Comando: tail [-n] file**
 - Lista le ultime n linee di un file (10 default)
- **Comando: cut**
 - Un tool per estrarre campi dai file. Nonostante esistano altri tool (più sofisticati), cut è utile per la sua semplicità.
 - Due opzioni particolarmente importanti:
 - **-d delimiter**: specifica il carattere di delimitazione (tab default)
 - **-f fields**: specifica quali campi stampare

Comandi per la gestione del testo

- **Comandi: expand, unexpand**

- L'utility expand converte i tab in spazi. L'utility unexpand converte gli spazi in tab

- **Comando: uniq**

- Questa utility rimuove linee duplicate (consecutive) dallo standard input. Viene usato spesso nei pipe con sort

- **Comando: sort**

- Ordina lo standard input linea per linea. E' in grado di eseguire ordinamenti lessicografici sulle linee, o di gestire l'ordinamento dei vari campi.
- Ad esempio: l'opzione -g ordina in modo numerico il primo campo dell'input

Esempio

```
% du -s /home/*
```

```
10000 /home/montreso
```

```
500 /home/rossi
```

```
2345 /home/schena
```

```
26758 /home/bompani ...
```

```
% du -s /home/* | sort -gr
```

```
26758 /home/bompani
```

```
10000 /home/montreso
```

```
2345 /home/schena
```

```
500 /home/rossi ...
```

Esempio (cont.)

```
% du -s /home/* | sort -gr | head -2
```

```
26758 /home/bompani
```

```
10000 /home/montreso
```

```
% du -s /home/* | sort -gr | head \  
-2 | cut -f2
```

```
/home/bompani
```

```
/home/montreso
```

```
% for homedir in $(du -s /home/* | sort -gr | \  
head -2 | cut -f2); do echo $(basename $homedir) ; \ done
```

```
bompani
```

```
montreso
```

Comandi per la gestione del testo

■ Comando: **wc (word count)**

- Conta linee, parole, caratteri
- **-l** | **-w** | **-c** conta solo le linee | le parole | i caratteri
- Certi comandi includono le funzionalità di **wc** come opzioni:
 - ... | **grep foo** | **wc -l** è equivalente a
 - ... | **grep --count foo**

■ Comando: **tr**

- Utility per la conversione di caratteri, seguendo un insieme di regole definite dall'utente:
- Esempio: **tr A-Z a-z < filename**
 - Stampa filename trasformando tutti i caratteri in minuscoli
- Esempio: **tr -d 0-9 < filename**
 - Stampa filename eliminando tutte i caratteri numerici

Esempio: filenames-to-lowercase

```
#!/bin/bash

# Changes every filename in working directory to all
# lowercase.

for filename in * ; do
    fname=`basename $filename`
    n=`echo $fname | tr A-Z a-z`
    # Change name to lowercase.
    if [ "$fname" != "$n" ] ; then
        # Rename only files not lowercase.
        mv "$fname" "$n"
    fi
done

exit 0
```

Esempio: dos2unix

```
#!/bin/bash

# dos2unix.sh: DOS to UNIX text file converter.

E_WRONGARGS=65

if [ -z "$1" -a -z "$2" ]; then
    echo "Usage: `basename $0` file-source file-dest"
    exit $E_WRONGARGS
fi

CR='\015' # Lines in DOS text files end in a CR-LF.

tr -d $CR < $1 > $2 # Delete CR and write to $2

exit 0
```


Comandi per il confronto di file

▪ **Comando: `cmp file1 file2`**

- Ritorna true (exit status 0) se due file sono uguali, ritorna false (exit status != 0) altrimenti
- Stampa la prima linea con differenze
- Con l'opzione `-s` non stampa nulla (utile per script)

▪ **Comando: `diff file1 file2`**

- Ritorna true (exit status 0) se due file sono uguali, ritorna false (exit status != 0) altrimenti
- Stampa un elenco di differenze tra i due file (linee aggiunte, linee modificate, linee cancellate)

▪ **Comando: `diff dir1 dir2`**

- Confronta due directory e mostra le differenze (file presenti in una sola delle due)

Comandi per la gestione di file

- **Comando: locate, slocate, updatedb**

- I comandi locate e slocate (secure version di locate) cercano file utilizzando un database apposito. Il database riflette il contenuto del file system, ma va aggiornato con updatedb (solo root)

- **Comando: file {file}***

- Identifica il tipo di file a partire dal suo magic number (quando disponibile). L'elenco dei magic number si trova in /usr/share/magic (o altre posizioni, consultate info file)

- Esempio:

```
% file prova.sh
```

```
prova.sh: Bourne-Again shell script text executable
```

Comandi per la gestione dei file

- **Comando: `basename file`**
 - rimuove l'informazione del path da un pathname
- **Comando: `dirname file`**
 - rimuove l'informazione del nome di file da un pathname
- **Nota: sono funzioni di stringa, non agiscono su un file effettivo**
- **Esempi:**

```
% basename /home/montreso/index.html  
index.html  
% dirname /home/montreso/index.html  
/home/montreso  
  
echo "Usage: `basename $0` arg1 arg2 ... argn"
```

Archiviazione e compressione

■ **Compressione:**

- Comandi: **compress**, **uncompress**
- Comandi: **gzip**, **gunzip**
- Comandi: **bzip2**, **bunzip2**
- Comprimono e decomprimono file. **compress** è ormai in disuso (originario dei primi sistemi Unix); **bzip2** è meno diffuso, ma è in alcuni casi più efficiente di **gzip**

■ **Archiviazione: tar**

- Archiviazione: **tar zcvf *archive-name* {file}***
 - z=comprimi, c=crea, v=verbose, f=su file
- Estrazione: **tar zxvf *archive-name***
 - z=espandi, x=estrai, v=verbose, f=da file

Comandi per la gestione del tempo

▪ **Comando: touch *{file}****

- utility per modificare il tempo di accesso/modifica di un file, portandolo ad un tempo specificato (**touch -d**)
- può essere utilizzata anche per creare un nuovo file
- creare file vuoti può essere utile per memorizzare informazioni di data

▪ **Comando: date**

- Stampa informazioni sulla data corrente, in vari formati

▪ **Comando: time *command***

- esegue il comando *command* e stampa una serie di statistiche sul tempo impiegato
- Esempio: **time find / -name "*.bak" -print**

Espressioni regolari

- **Un espressione regolare:**
 - è una stringa di caratteri e metacaratteri
 - i metacaratteri sono caratteri speciali che vengono interpretati in modo "non letterale" dal meccanismo delle espressioni regolari
- **Le espressioni regolari (Regular Expression, RE) sono utilizzate per ricerche e manipolazioni di stringhe**
 - grep, awk, sed, etc.
- **Definizione: match**
 - Diciamo che una RE fa match con una particolare stringa se è possibile generare la stringa a partire dalla RE
- **Nota: Unix wildcard e RE hanno metacaratteri (e significati) differenti; non vanno confuse**

Espressioni regolari

▪ Sintassi:

- L'asterisco * fa match con qualsiasi numero di ripetizioni del carattere che lo precede, incluso 0
 - Esempio: “**11*33**” fa match con 1133, 11133, 111133, etc
- Il punto . fa match con qualsiasi carattere, a parte newline
 - Esempio: “**13.3**” fa match con 13a3, 1303, ma non con 13\n3
- Il caret ^ fa match con l'inizio di una linea (ma ha anche significati addizionali)
 - Esempio: “**^Subject:.***” fa match con una linea di subject di posta elettronica
- Il dollaro \$ fa match con la fine di una linea:
 - Esempio: “**^\$**” fa match con una linea vuota

Espressioni regolari

■ Sintassi:

- Le parentesi quadre [...] sono utilizzati per fare match un sottoinsieme (o un range) di caratteri
 - “[xyz]” fa match con i caratteri x, y, z
 - “[c-n]” fa match con qualsiasi carattere fra c ed n
 - “[a-zA-Z0-9]” fa match con qualsiasi carattere alfanumerico
 - “[0-9]*” fa match con qualsiasi stringa decimale
 - “[^0-9]” fa match con qualsiasi carattere non numerico
 - “[Yy][Ee][Ss]” fa match con yes, Yes, YES, yEs, etc.
- Il backslash \ è usato come escape per i metacaratteri; il carattere viene interpretato letteralmente; es. “\\$\$”
- I metacaratteri perdono il loro significato speciale dentro []

Espressioni regolari

- **Le espressioni regolari hanno un concetto di "parola":**
 - una parola è un pattern contenente solo lettere, numeri e underscore _
- **E' possibile fare matching**
 - con l'inizio di una parola: `\<`
 - Esempio: `\<Fo`
fa match con tutte le parole che iniziano con Fo
 - con la fine di una parola: `\>`
 - Esempio: `ox\>`
fa match con tutte le parole che finiscano con ox
 - con parole complete:
 - Esempio: `\<Fox\>`

Espressioni regolari

▪ RE Recall

- un modo per riferirsi ai match più recente

▪ Sintassi:

- per "marcare" una porzione di espressione regolare che volete sia "ricordata": racchiuderla in `\(\)`
- per ripetere una porzione "marcata", si può utilizzare `\n`, con $n=1..9$

▪ Esempi:

- `'^\[a-z]\1'`
fa match con le linee che iniziano con due lettere minuscole identiche

▪ Domande

- Con cosa fa match `'^.*\[a-z]*\1.*\1'` ?

Espressioni regolari estese

- Il segno **?** fa match con 0 o 1 ripetizioni della espressione regolare precedente
 - Esempio: “**cort?o**” fa match con coro e corto
- Il segno **+** fa match con 1 o più ripetizioni della espressione regolare precedente, ma non 0
 - Esempio: “**[0-9]+**” fa match numeri non vuoti
- I segni **{ }** indicano il numero di ripetizioni che dalla espressione regolare precedente
 - Esempio: “**[0-9] { 5 }**” fa il match con tutti i numeri a 5 cifre
- I segni **()** servono a raggruppare espressioni regolari
 - Esempio: “**(re)***” fa match con “”, re, rere, rerere, etc.
- Il segno **|** indica un’alternativa (or)
 - Esempio: “**(bio|psico)logia**” fa match con biologia e psicologia

Intermezzo: leggere la documentazione

- **Nella documentazione e nei libri si trovano varie sintassi per la documentazione. Ad esempio, nel libro:**

command *[opt-arg] mandatory-arg {rep-arg}**

- *opt-arg* è opzionale
- *mandatory-arg* è obbligatorio
- *rep-arg* può essere ripetuto n volte, con $n \geq 0$
- le opzioni (con -) possono raggruppate, e spesso non sono racchiuse fra parentesi quadre
- corsivo per argomenti da sostituire

- **Esempio:**

uniq *-c -number [inputfile [outputfile]]*

Esempi di documentazione

- **Nella documentazione e nei libri si trovano varie sintassi per la documentazione. Ad esempio, nelle man pages:**
 - ... argomento ripetibile
 - [] argomento opzionale

NAME

mv - move (rename) files

SYNOPSIS

mv [OPTION]... SOURCE DEST

mv [OPTION]... SOURCE... DIRECTORY

mv [OPTION]... --target-directory=DIRECTORY SOURCE...

DESCRIPTION

Rename SOURCE to DEST, or move SOURCE(s) to DIRECTORY.

Comando grep

- **Il comando `grep` permette di cercare pattern in un insieme di file**
 - se non vi sono file specificati, la ricerca è nello standard input
 - il pattern è espresso come espressione regolare
 - tutte le linee che contengono il pattern vengono stampate su stdout
 - in caso di file multipli, le linee sono precedute dal pathname del file che le contiene (a meno di opzione `-h`)
 - opzione `-v`
 - fa in modo che l'output contenga tutte le linee che non contengono il pattern
 - opzione `-R`
 - analizza ricorsivamente le subdirectory
 - opzione `-c`
 - conta le occorrenze, invece di stamparle
 - opzione `-i`
 - ignora distinzioni tra caratteri maiuscoli e minuscoli

Comando grep

- **Esistono quattro versioni:**
 - **grep** [*options*] *pattern* {*file*}*
 - *pattern* è una espressione regolare
 - **fgrep** [*options*] *pattern* [*file(s)*]
 - *pattern* è una stringa fissa
 - versione più veloce
 - **egrep** [*options*] *pattern* [*file(s)*]
 - *pattern* è una espressione regolare estesa
 - **zgrep** [*options*] *pattern* [*file(s)*]
 - è anche in grado di cercare in file compressi (compressed or gzipped)

Comando grep - Esempi

```
if echo "$VAR" | grep -q txt ; then
    echo "$VAR contains the substring \"txt\""
fi
```

```
grep '[Ff]irst' *.txt
```

```
file1.txt:This is the first line of file1.txt.
```

```
file2.txt:This is the First line of file2.txt.
```


Esempi: cosa fanno queste ricerche?

- `grep -c 'ing$' /usr/dict/words`
- `grep -c '^un.*$' /usr/dict/words`
- `grep -ic '^[aeiou]' /usr/dict/words`
- `grep -ic '\(.\) \1\1' /usr/dict/words`
- `grep -c '^\(..\)\.*\1$' /usr/dict/words`
- `grep -ic '^\(.\) \(.\) \2\1$' /usr/dict/words`

Esercizio 2

- **Scrivere uno script che prenda in input da linea di comando:**
 - il pathname di una directory
 - il pathname di un file contenente testo
 - una parola chiave (opzionale)
 - un'estensione (opzionale)
 - un'opzione per indicare se la ricerca è case sensitive oppure no
- **Lo script**
 - deve individuare tutti i file contenuti nella directory (ricorsivamente), che abbiano l'estensione specificata (o tutti se assente) e che non contengano la parola chiave specificata (o tutti se assente)
 - Lo script deve aggiungere (all'inizio, prima del testo esistente) ad ognuno di questi file il testo contenuto nel file specificato.
- **Questo script può essere utile per aggiungere un commento iniziale di copyright ad un insieme di sorgenti**

AWK

■ Descrizione

- L'utility AWK effettua la scansione di uno o più file (o dello standard input) ed esegue un'azione su tutte le linee che rispettano una certa condizione
- Il nome AWK deriva da Aho, Weinberger, Kernighan
- Prende in prestito strutture di controllo e parte della sintassi dal C

■ Commento

- E' un utility molto potente, talmente potente che esistono interi libri di programmazione per awk (!)

AWK

- **Synopsis:**

```
awk [-Fc] { -f progame | program } {var=value}* {filename}*
```

- **Spiegazione:**

- Un programma awk può essere fornito su linea di comando, oppure può essere memorizzato nel file *progame*
- Il valore iniziale di un insieme di variabili può essere specificato tramite associazioni *var=valore*
- Ogni linea viene suddivisa in un certo numero di campi, separati da un delimitatore (default tab, blank). *-F* permette di specificare il delimitatore

- **Nota:**

- Se il programma AWK è fornito sulla linea di comando, deve essere racchiuso da single quote ' '
- E' molto più comune utilizzare programmi memorizzati su file

AWK

- **Un programma AWK è costituito da una o più linee (regole) con questa forma:**

[condizione] [{ azione }]

- **Condizione può essere:**

- un espressione che coinvolge operatori logici, operatori relazionali, oppure espressioni regolari
- il token speciale **BEGIN** oppure **END**

- **Azione è data da:**

- un insieme di statement C-like

- **Elenco di statement C-like:**
 - `if (conditional) statement [else statement]`
 - `while (conditional) statement`
 - `for (expression; conditional; expression) statement`
 - `break`
 - `continue`
 - `variable=expression`
 - `print list-of-expressions`
 - `printf format list-of-expressions`
 - `statement ; statement`
 - `{ statement }`

- **Come viene eseguito il programma AWK?**
 - Il testo viene analizzato linea per linea
 - Ad ogni linea, *condizione* viene valutata
 - Se *condizione* è vera, *azione* viene applicata
 - Se *condizione* è assente, *azione* viene applicata in ogni caso
 - In caso di assenza di *azione*, viene stampata la linea
 - Se sono presenti più regole, ognuna di essa viene valutata su ogni riga e l'azione associata viene eventualmente eseguita

File: folder

LS0-ES.0,0000123456,trotter,Guido,Trotter

LS0-ES.0,0000112233,gardengl,Ludovico,Gardenghi

LS0-ES.0,0000121212,cenacchi,Federica,Cenacchi

LS0-ES.0,0000122323,corrader,Roberto,Corradetti

LS0-ES.0,0000111111,nbagnasc,Nicola,Bagnasco

LS0-ES.0,0000111113,mbagnasc,Matteo,Bagnasco

LS0 ES.0,0000111888,pincopal,Pinco,Pallino

LS0-ES.0,1674002222,confalon,Roberto,Confalonieri

LS0-ES.0,1674000520;montreso,Alberto,Montresor

AWK: Script di esempio

▪ Accedere ai campi di una linea

- Il primo campo di una linea e' identificato da \$1, il secondo da \$2, etc; \$0 si riferisce all'intera linea. La variabile predefinita NF contiene il numero di campi della linea

```
% awk -F", " '{ print NF, $0 }' folder
```

```
5 LSO-ES.0,0000123456,trotter,Guido,Trotter
```

```
[...]
```

```
4 LSO-ES.0,1674000520;montreso,Alberto,Montresor
```

```
% awk -F", " '{ print NF, $2 }' folder
```

```
5 0000123456
```

```
[...]
```

```
4 1674000520;montreso
```

AWK: Script di esempio

▪ BEGIN e END

- Esistono due condizioni speciali, che vengono attivate prima della prima linea (BEGIN) e dopo l'ultima linea END

```
% cat script1.awk
```

```
BEGIN { print "Start of file:", FILENAME }
```

```
{ print $2 " is " $NF }
```

```
END { print "End of file" }
```

```
% awk -F"," -f script1.awk folder
```

```
Start of file: folder
```

```
0000123456 is Trotter
```

```
[...]
```

```
1674000520;montreso is Montresor
```

```
End of file
```

AWK: Script di esempio

▪ Condizioni

- Le condizioni possono essere espresse utilizzando gli operatori C usuali. La variabile predefinita NR, utilizzata nel prossimo esempio, contiene il numero della linea corrente

```
% cat script2.awk
```

```
NR > 1 && NF == 5 { print NR, $0 }
```

```
% awk -F"," -f script2.awk folder
```

```
2 LSO-ES.0,0000112233,gardengl,Ludovico,Gardenghi
```

```
3 LSO-ES.0,0000121212,cenacchi,Federica,Cenacchi
```

```
[...]
```

```
8 LSO-ES.0,1674002222,confalon,Roberto,Confalonieri
```

AWK: Script di esempio

▪ Variabili

- Awk supporta variabili definite dall'utente.
- Non c'e' bisogno di dichiarare variabili.
- Inizialmente, ogni variabile ha valore null o 0

```
% cat script3.awk
```

```
BEGIN { print "Scanning file" }  
NF == 5 {  
    printf "line %d: %s\n", NR, $0;  
    lineCount++;  
    wordCount += NF;  
}  
END { printf "Lines: %d, Fields: %d", lineCount,  
wordCount }
```

AWK: Script di esempio

```
% awk -F", " -f script3.awk folder
```

Scanning file

line 1: LSO-ES.0,0000123456,trotter,Guido,Trotter

line 2: LSO-ES.0,0000112233,gardengl,Ludovico,Gardenghi

[...]

line 8: LSO-ES.0,1674002222,confalon,Roberto,Confalonieri

Lines: 8, Fields: 40

AWK: Script di esempio

▪ Strutture di controllo

- awk supporta le strutture di controllo standard del C

```
% cat script4.awk
```

```
{ for (i = NF; i >= 1; i--)  
    printf "%s ", $i;  
    printf "\n" }
```

```
% awk -F"," -f script4.awk folder
```

```
Gardenghi Ludovico gardengl 0000112233 LS0-ES.0
```

```
[...]
```

```
Montesor Alberto 1674000520;montreso LS0-ES.0
```

AWK: Script di esempio

- **Espressioni regolari estese**

- La condizione per il line matching può essere espressa da una espressione regolare estesa racchiusa fra “/”

```
% awk -F", " '/^LS0-ES\.0/ { print $0 }' folder
```

```
LS0-ES.0, 0000123456, trotter, Guido, Trotter
```

```
[...]
```

```
LS0-ES.0, 0000111113, mbagnasc, Matteo, Bagnasco
```

```
LS0-ES.0, 1674002222, confalon, Roberto, Confalonieri
```

```
LS0-ES.0, 1674000520;montreso, Alberto, Montresor
```

AWK: Script di esempio

- **Condition range:**

- Una condizione può essere data da due espressioni separate da una virgola. In questo caso, awk esegue l'azione su tutte le linee comprese fra la prima che fa matching con la prima condizione fino all'ultima che soddisfa la seconda condizione

```
% awk -F", " '/trotter/, /corrader/ { print $0 }' folder
```

```
LS0-ES.0,0000123456,trotter,Guido,Trotter
```

```
LS0-ES.0,0000112233,gardengl,Ludovico,Gardenghi
```

```
LS0-ES.0,0000121212,cenacchi,Federica,Cenacchi
```

```
LS0-ES.0,0000122323,corrader,Roberto,Corradett
```


AWK: Script di esempio

▪ Funzioni predefinite

- awk supporta un certo numero di funzioni predefinite, come `substr()`, `log()`, `int()`, `length()`
- Esempio: utilizziamo `length()` per verificare che una stringa contenga solo numeri di matricola di 10 cifre

```
% cat script4.awk
```

```
/^LS0-ES\.0/
```

```
{
```

```
    if (length($2) == 10)
```

```
        printf("%s", $0);
```

```
}
```

SED

▪ **SED – Stream EDitor**

- L'utility sed scandisce uno o più file ed esegue un'azione di editing su tutte le linee che soddisfano una particolare condizione

▪ **Synopsis**

- `sed [-e script] [-f scriptfile] { filename }*`
- `-e` lo script viene dato come argomento della command line
- `-f` lo script viene dato come file
- `filename(s)` è l'insieme di file su cui si agisce

SED: Esempi

- **Sostituzione: *s/reg-exp/string/***
 - `sed -e 's/^/ /' file > file.indent`
 - Inserisce due spazi all'inizio di ogni linea

% cat file

This is the first line

The is the last line

% cat file.indent

This is the first line

This is the last line

SED: Esempi

- **Sostituzione: *s/reg-exp/string/***
 - `sed -e 's/^ *//' file > file.noindent`
 - Rimuove tutti gli spazi iniziali

```
% cat file > file.noindent
```

```
This is the first line
```

```
    The is the last line
```

```
% cat file.noindent
```

```
This is the first line
```

```
This is the last line
```

SED: Esempi

- **Cancellazione: */reg-exp/d***
 - `sed -e '/^$/d' file > file.del`
 - Rimuove tutte le linee vuote

```
% cat file > file.del
```

```
This is the first line
```

```
The is the last line
```

```
% cat file.del
```

```
This is the first line
```

```
This is the last line
```

Funzioni

- **Come un vero linguaggio di programmazione, Bash è dotato di funzioni.**
 - Una funzione è un blocco di codice che implementa una funzionalità ripetitiva, e permette di organizzare al meglio il codice.

- **Dichiarazione:**

```
function function-name {  
    command...
```

```
}
```

- **Invocazione:**

```
function-name
```

```
function-name () {  
    command...  
}
```

Funzioni

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

```
funky ()
```

```
{
```

```
    echo "This is a funky function."
```

```
    echo "Now exiting funky function."
```

```
} # Function declaration must precede call.
```

```
# Now, call the function.
```

```
funky
```

```
exit 0
```

Funzioni

```
# f1
# Would give an error, as "f1" not yet defined.
# However...
f1 () {
    echo "Calling function \"f2\" from within \"f1\"."
    f2
}
f2 () {
    echo "Function \"f2\"."
}
f1 # f2" is not actually called until this point
# although it is referenced before its definition.
# This is permissible.
```


Passaggio di parametri e valori di ritorno

■ **Le funzioni possono prendere in input parametri**

- La funzione si riferisce ai parametri in modo posizionale, con \$1, \$2, etc (distinti da argomenti di linea di comando)
- La dichiarazione è invariata
- L'invocazione assume la forma:

`function_name {arg}*`

■ **Le funzioni possono restituire valori di ritorno (exit status)**

- Se non specificato, il valore di ritorno è uguale all'exit status dell'ultimo comando eseguito
- E' possibile specificare il valore di ritorno tramite il built-in return x
- E' possibile fare riferimento al valore di ritorno tramite \$?

Esempio: Passaggio di parametri e valori di ritorno

```
#!/bin/bash
usage () {
    if [ -n "$1" ] ; then
        echo "Usage: " \
            "$1 [options]
            arguments"
    fi
}
```

```
# Prints the usage
usage $(basename $0)
```

```
#!/bin/bash
max () {
    if [ $1 -gt $2 ] ; then
        return $1
    else
        return $2
    fi
}
```

```
max 12 14
echo $?
```

Valori di ritorno

- **Il valore di ritorno è un exit status**
 - Può assumere valori positivi fra 0 e 255
 - Altrimenti, può assumere valori negativi
- **Per ritornare stringhe generali, si utilizza la variabile REPLY**

```
countlines() {  
    if [ -r /etc/passwd ] ; then  
        REPLY=$(echo $(wc -l < /etc/passwd))  
    fi  
}  
if countlines ; then  
    echo "There are $REPLY lines in /etc/passwd"  
fi
```

Spostamento file nel cestino (Gardenghi)

```
delete_list() {
  while [ ! -z "$1" ]; do
    case "$1" in
      "-L" | "-P" | "-R")
        syntaxerror
        ;;
      *)
        ABSPATH="$(cd $(dirname "$1") ; pwd)"
        PATHINTRASHCAN="${TRASHCANDIR}${ABSPATH}"
        if [ ! -d "$PATHINTRASHCAN" ]; then
          mkdir -p "$PATHINTRASHCAN"
        fi
        mv -f "$1" "$PATHINTRASHCAN/"
        shift
        ;;
    esac
  done
}
```