Interfaccia Utente a caratteri (testuale) Interprete dei comandi Shell scripting

NOTA BENE:

Questa guida contiene solo una introduzione minimale all'utilizzo dell'interprete di comandi bash, e serve per fornire agli studenti del corso di Architettura degli elaboratori le informazioni per utilizzare una semplice interfaccia a riga di comando con cui compilare programmi in linguaggio ANSI C mediante il compilatore gcc, eseguire e debuggare tali programmi, capire le interazioni tra i programmi e l'utente svolte sfruttando le funzionalità realizzate dal sistema operativo.

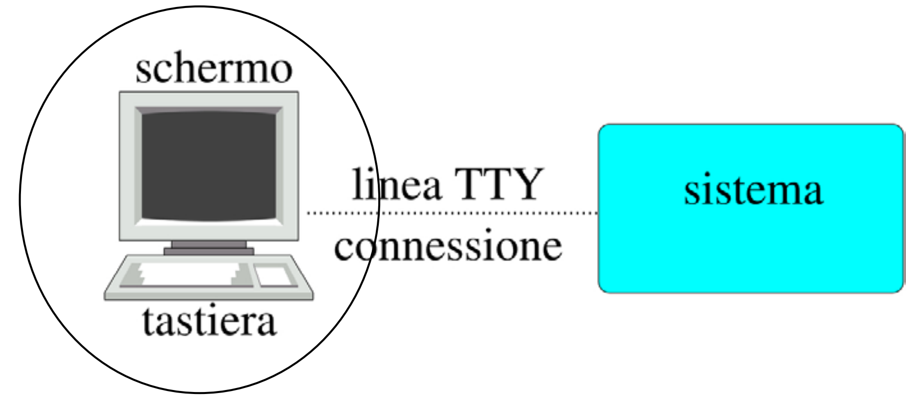
Per tale motivo, alcuni concetti sono stati volutamente semplificati, allo scopo di rendere più semplice la comprensione, anche a discapito della realtà.

Vittorio Ghini

Servizi del Sistema Operativo

- **Gestione di Risorse**: allocazione, contabilizzazione, protezione e sicurezza (possessori di informazione devono essere garantiti da accessi indesiderati ai propri dati; processi concorrenti non devono interferire fra loro).
- Comunicazioni (intra e inter-computer)
- Rilevamento di errori che possono verificarsi nella CPU e nella memoria, nei dispositivi di I/O o durante l'esecuzione di programmi utente
- Gestione del file system — capacità dei programmi e dell'utente di leggere, scrivere e cancellare file e muoversi nella struttura delle directory
- Operazioni di I/O — il SO fornisce ai programmi utente i mezzi per effettuare l'I/O su file o periferica
- Esecuzione di programmi — capacità di caricare un programma in memoria ed eseguirlo, eventualmente rilevando e gestendo, situazioni di errore
- **Programmi di sistema**
- **Chiamate di sistema**
- **Interfaccia utente**: **a linea di comando** (*Command Line Interface*, CLI) o grafica (*Graphic User Interface*, GUI).

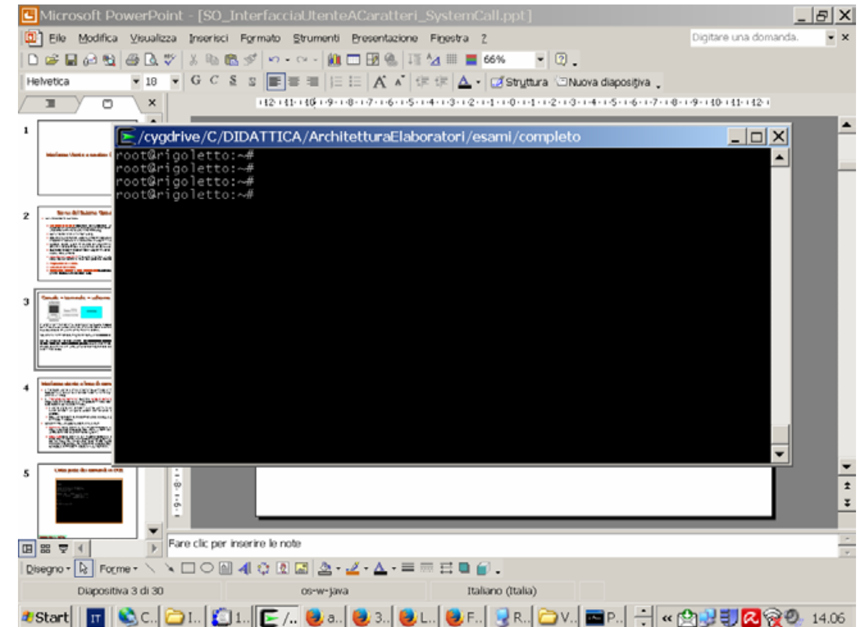
Console = terminale = schermo + tastiera



In origine i terminali (schermo+tastiera) erano dispositivi **separati** dal computer vero e proprio (detto mainframe) e comunicavano con questo mediante una linea seriale. L'output verso l'utente era di tipo solo testuale, l'input era fornito mediante tastiera.

Attualmente i terminali sono integrati nei computer (**schermo e tastiera del PC**).

Con l'avvento delle interfacce grafiche, **lo schermo del terminale viene emulato in una "finestra" dell'ambiente grafico**. Si parla di terminale virtuale. Quando la finestra del terminale è in primo piano i caratteri digitati dalla tastiera vengono passati al terminale stesso.



Interfaccia utente a linea di comando (CLI)

- L'interfaccia utente a linea di comando permette di impartire ordini al SO sotto forma di sequenze di caratteri alfa-numerici digitati sulla tastiera (o presi da un file).
- L' interprete dei comandi (o anche Shell di comandi) è un programma eseguibile che si occupa di un singolo terminale. Attende i caratteri digitati sulla tastiera per quel terminale.
  - L'utente digita dei caratteri, quando preme il tasto di invio (return) questi caratteri vengono passati all'interprete dei comandi che li gestisce.
  - Ricevuti i caratteri, la shell li interpreta ed esegue gli ordini ricevuti, poi si rimette in attesa di nuovi ordini.
- Gli ordini ricevuti possono essere di due tipi:
  - Comandi sono ordini la cui implementazione è contenuta nel file eseguibile della shell stessa (istruzione built–in del SO). Non li troviamo perciò come file separati nel file system.
  - Eseguibili sono ordini la cui implementazione è contenuta fuori della shell, cioè in altri file memorizzati nel file system. L'ordine è il nome del file da eseguire. L'interprete cerca il file specificato, lo carica in memoria e lo fa eseguire. Al termine dell'esecuzione la shell riprende il controllo e si rimette in attesa di nuovi comandi.

L’interprete dei comandi in alcuni s.o.

- Unix, Linux:
  - Diverse Shell di comandi, tra le più utilizzate **bash** (bourne again shell).
- Windows:
  - Shell DOS
  - Emulazione di interfaccia testuale Linux: bash realizzata dalla piattaforma cygwin
- Mac:
  - Diverse Shell di comandi, compresa **bash**

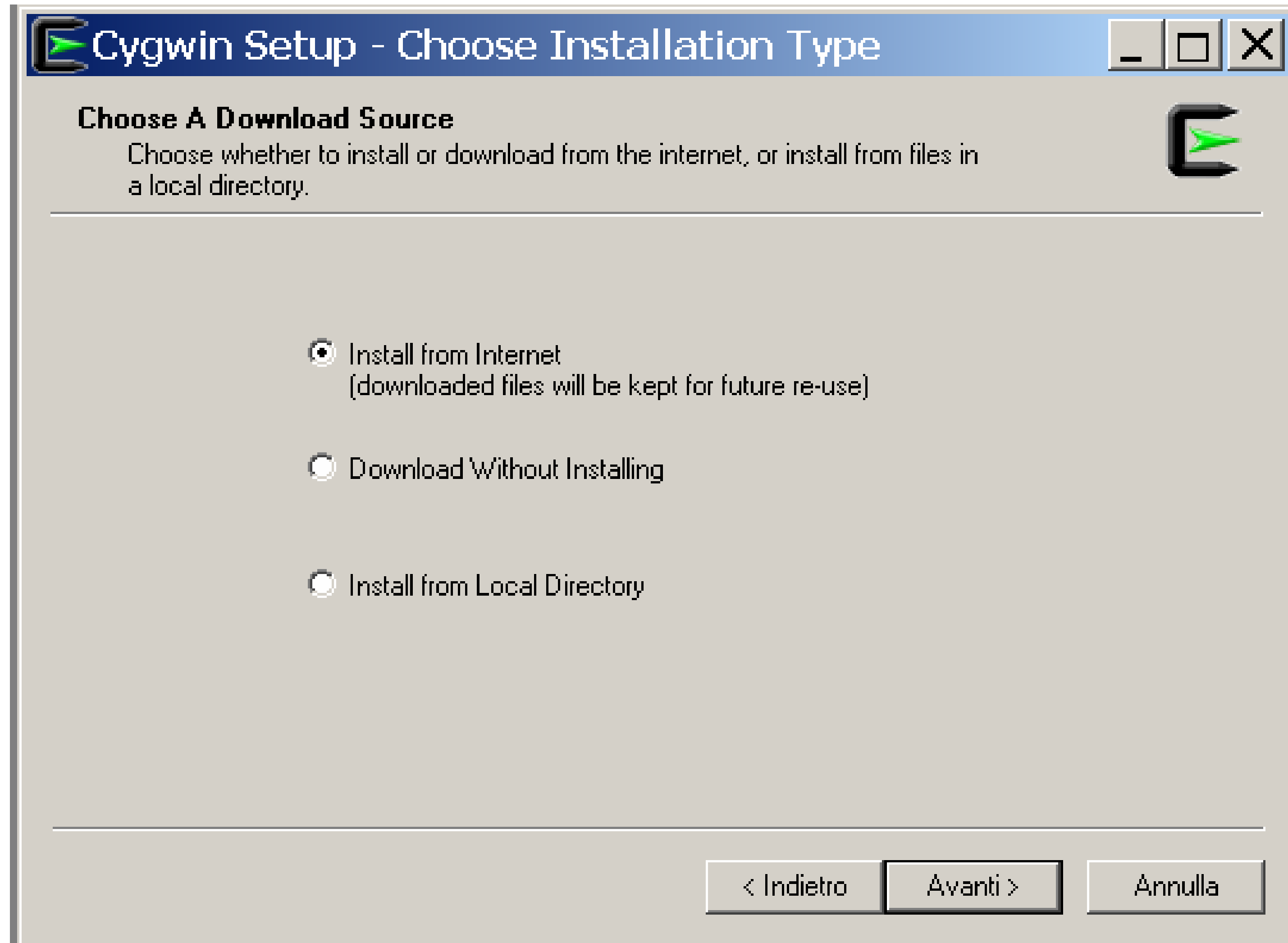
- Nel corso di architettura utilizzeremo compilatori con sola interfaccia testuale, in particolare il GNU C Compiler (gcc).
- Gli ordini al compilatore, così come l’ordine di eseguire un programma, verranno impartiti utilizzando una shell bash.
  - In Linux e su Mac esiste già un terminale con shel bash.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

Solo per Windows: installare l'emulatore di terminale Linux – cygwin (1)

- Sul sito web www.cygwin.com
- Consultare http://cygwin.com/install.html
- Scaricare ed eseguire setup-x86.exe (per sistemi a 32 bit) oppure setup-x86_64.exe (per sistemi a 64 bit).

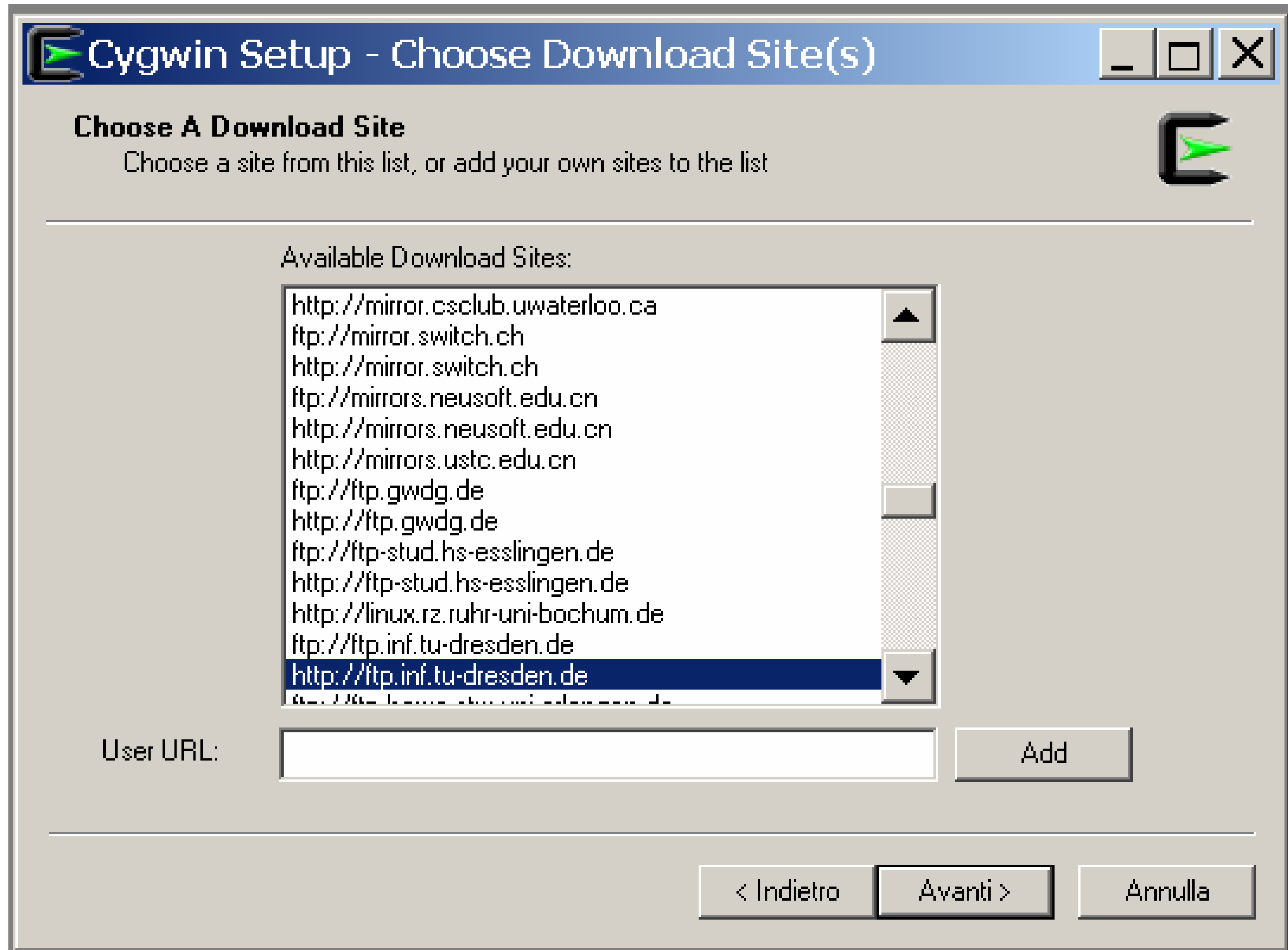
installare cygwin (2)

Selezionare l’installazione da internet



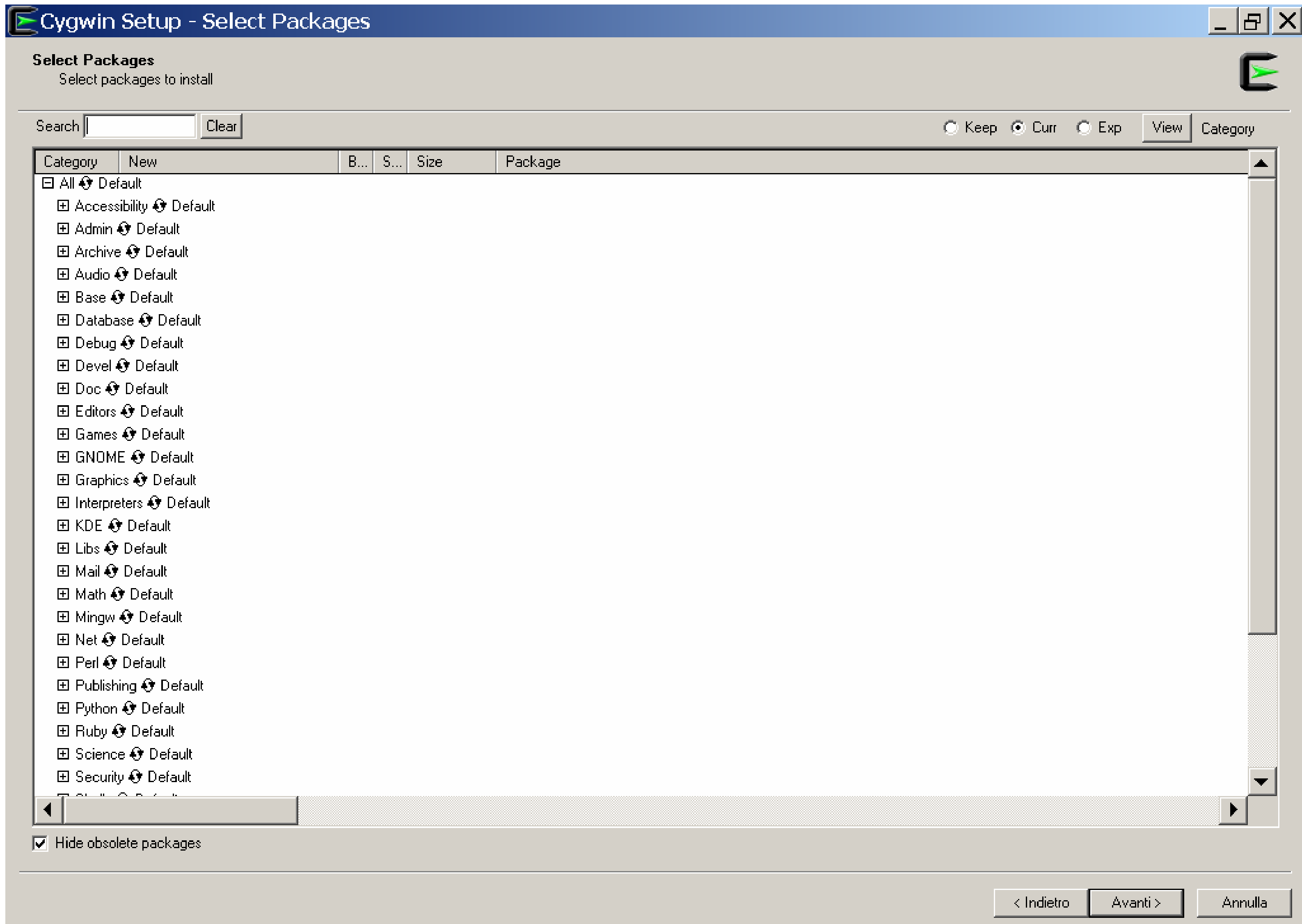
installare cygwin (3)

Selezionare un mirror da cui scaricare i files (.de (germania) veloci)



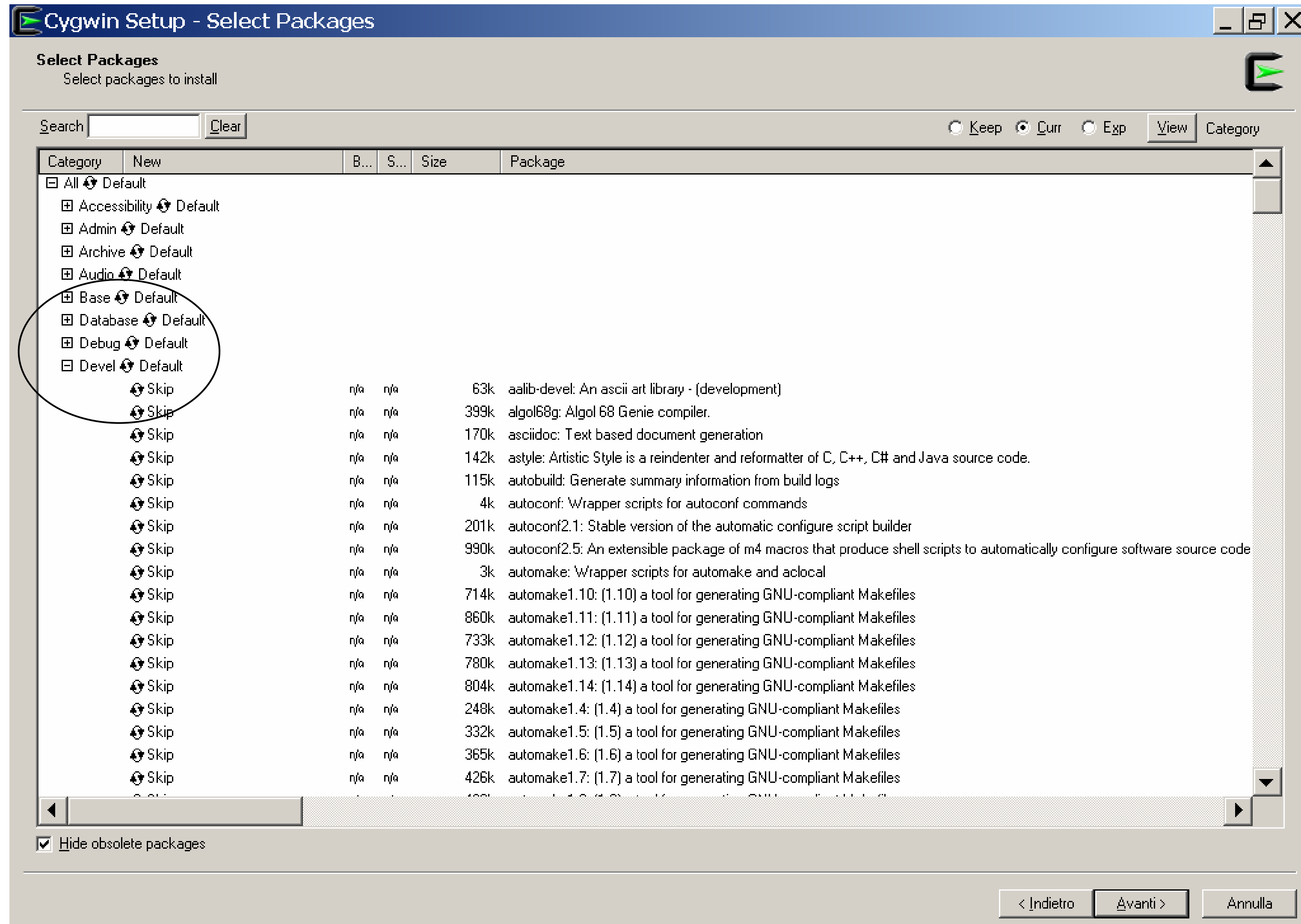
installare cygwin (4)

Selezionare le categorie di pacchetti, e per ciascuna categoria selezionare i pacchetti



installare cygwin (5)

Esempio, cliccare sulla categoria **devel**, vengono visualizzati dei pacchetti, selezionare quelli di interesse, ad es automake



installare cygwin (6)

Nella categoria **devel** selezionare anche i seguenti pacchetti:

autoconf, autoconf2.5 (ultima versione), automake, automake1.9 (ultima versione), bashdb, binutils, bison, byacc, cgdb, gcc, gdb, indent, make, patch, patchutils,

Nella categoria **editor** selezionare anche i seguenti pacchetti:

ed, nano, vim, vim-common, sed

Nella categoria **net** selezionare anche i seguenti pacchetti:

ping, openssh, openssl

Nella categoria **shell** selezionare anche i seguenti pacchetti:

bash, bash-completion,

Nella categoria **text** selezionare anche i seguenti pacchetti:

less

Nella categoria **utils** selezionare anche i seguenti pacchetti:

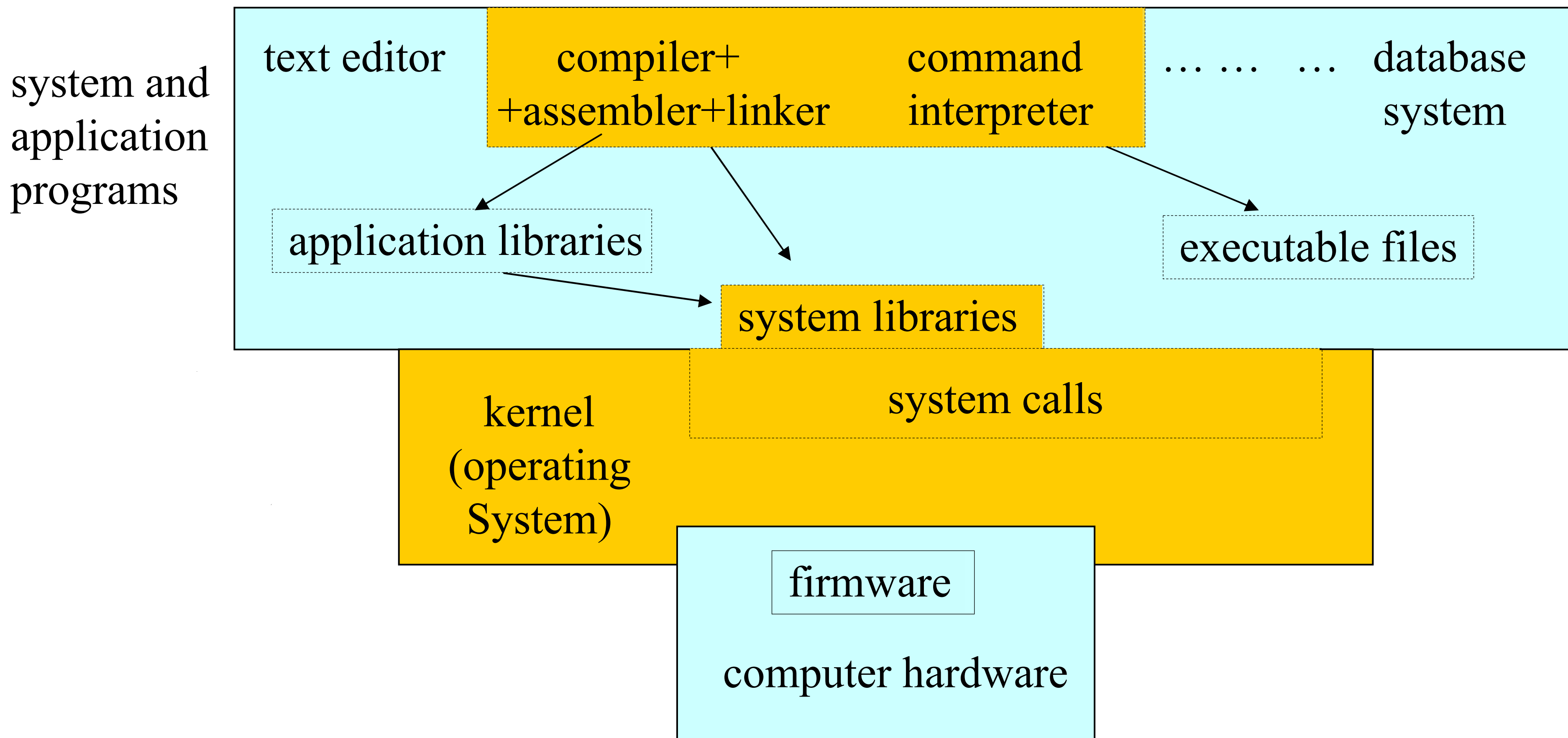
diffutils, file, time, which,

Interfaccia utente GUI

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

Librerie e Chiamate di sistema (1)

Ricordiamo l’organizzazione del computer, in particolare la relazione tra le system calls e le librerie di sistema.



Librerie e Chiamate di sistema (2)

- Le chiamate di sistema forniscono ai processi i servizi offerti dal SO:

  Controllo dei processi, Gestione dei file e dei permessi, Gestione dei dispositivi di I/O, Comunicazioni.

- Il modo in cui sono realizzate cambia al variare della CPU e del sistema operativo.
- Il modo di utilizzarle (chiamarle) cambia al variare della CPU e del sistema operativo.
- Sono utilizzate (invocate) direttamente utilizzando linguaggi di basso livello (assembly).
- Possono essere chiamate (invocate) indirettamente utilizzando linguaggi di alto livello (C o C++)
- Infatti, normalmente i programmi applicativi non invocano direttamente le system call, bensì invocano funzioni contenute in librerie messe a disposizione dal sistema operativo ed utilizzate dai compilatori per generare gli eseguibili. L'insieme di tali funzioni di libreria rappresentano le API (Application Programming Interface) cioè l'interfaccia che il sistema operativo offre ai programmi di alto livello.
- Le librerie messe a disposizione dal sistema operativo sono dette **librerie di sistema**.

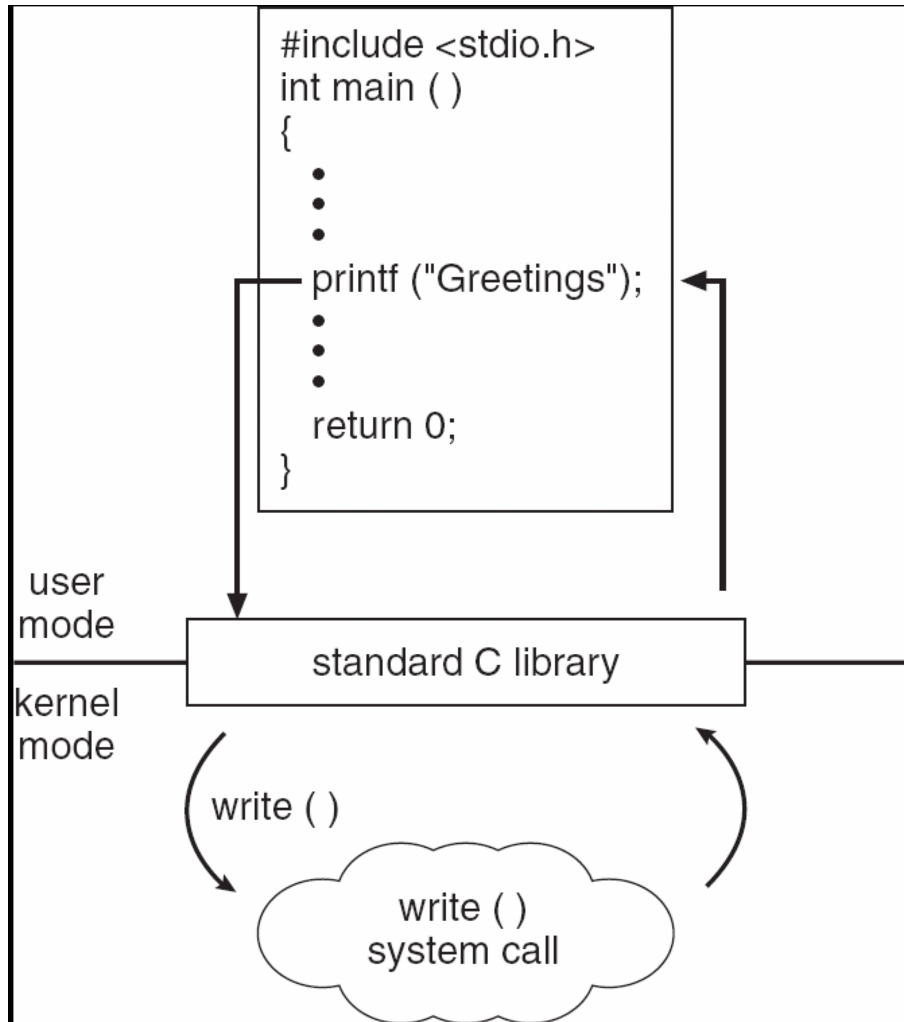
Librerie e Chiamate di sistema (3)

- Per ciascun linguaggio importante, il s.o. fornisce API di sistema da usare.
  - **Cambiando il sistema operativo oppure cambiando il processore (CPU) su cui il sistema operativo si appoggia, i nomi delle funzioni delle API rimangono gli stessi ma cambia il modo di invocare le system call, quindi cambia l'implementazione delle funzioni delle API per adattarle alle system call e all'hardware.**
  - Perché è importante programmare usando linguaggi di alto livello (es C)

    Un programma sorgente in linguaggio C che usa le API non deve essere modificato se cambio il sistema operativo o l'hardware (e questi mantengono le stesse API), poiché sul nuovo s.o. cambia l'implementazione delle API e quindi cambia l'eseguibile che viene generato su diversi s.o ed hardware.
- Alcune API molto diffuse sono:
  - Win32 API per Windows
  - **POSIX API per sistemi POSIX–based (tutte le versioni di UNIX, Linux, Mac OS X)**
  - Java API per la Java Virtual Machine (JVM).
- Possono esistere anche librerie messe a disposizione non dal sistema operativo bensì realizzate, ad esempio, da un utente.
- Solitamente tali librerie applicative sono implementate utilizzando le librerie di sstema.

Esempio con la libreria standard C

- Per Linux, la libreria standard del linguaggio C (il *run–time support system*) fornisce una parte dell'API



- Programma C che invoca la funzione di libreria per la stampa *printf()*
- La libreria C implementa la funzione e invoca la system call *write()* **nel modo richiesto dallo specifico s.o. e dalla specifica CPU.**
- La libreria riceve il valore restituito dalla chiamata al sistema e lo passa al pro-gramma utente

Nozioni per l’uso del Terminale: File system (1)

Il filesystem è la organizzazione del disco rigido che permette di contenere i file e le loro informazioni.

Lo spazio di ciascun disco rigido è suddiviso in una o più parti dette partizioni.
Le partizioni contengono

- dei contenitori di dati detti files
- dei contenitori di files, detti directories (folders o cartelle in ambienti grafici)..

In realtà ciascuna directory può a sua volta contenere dei files e anche delle altre directories formando una struttura gerarchica

In Windows le partizioni del disco sono viste come logicamente separate e ciascuna e’ indicata da una lettera (C; B: Z: …).

Se un file si chiama pippo.c ed è contenuto in una directory che si chiama vittorio che a sua volta è contenuta in una directory che si chiama home che a sua volta è contenuta nella partizione chiamata C:, allora è possibile individuare univocamente il file pippo.c indicando il **percorso** mediante il quale, partendo dalla partizione C: si arriva al file pippo.c

C:\home\vittorio\pippo.c <- percorso per raggiungere pippo.c

Notare il carattere separatore **** (si chiama backslash) che indica dove inizia il nome di una directory o di un file.

Nozioni per l’uso del Terminale: File system (2)

In Linux/Mac invece le partizioni sono viste come collegate tra loro.
Esiste una partizione principale il cui nome è **/** (slash).
Questa partizione, così come le altre partizioni, può contenere files e directories. Le directoryes possono contenere files e directories.

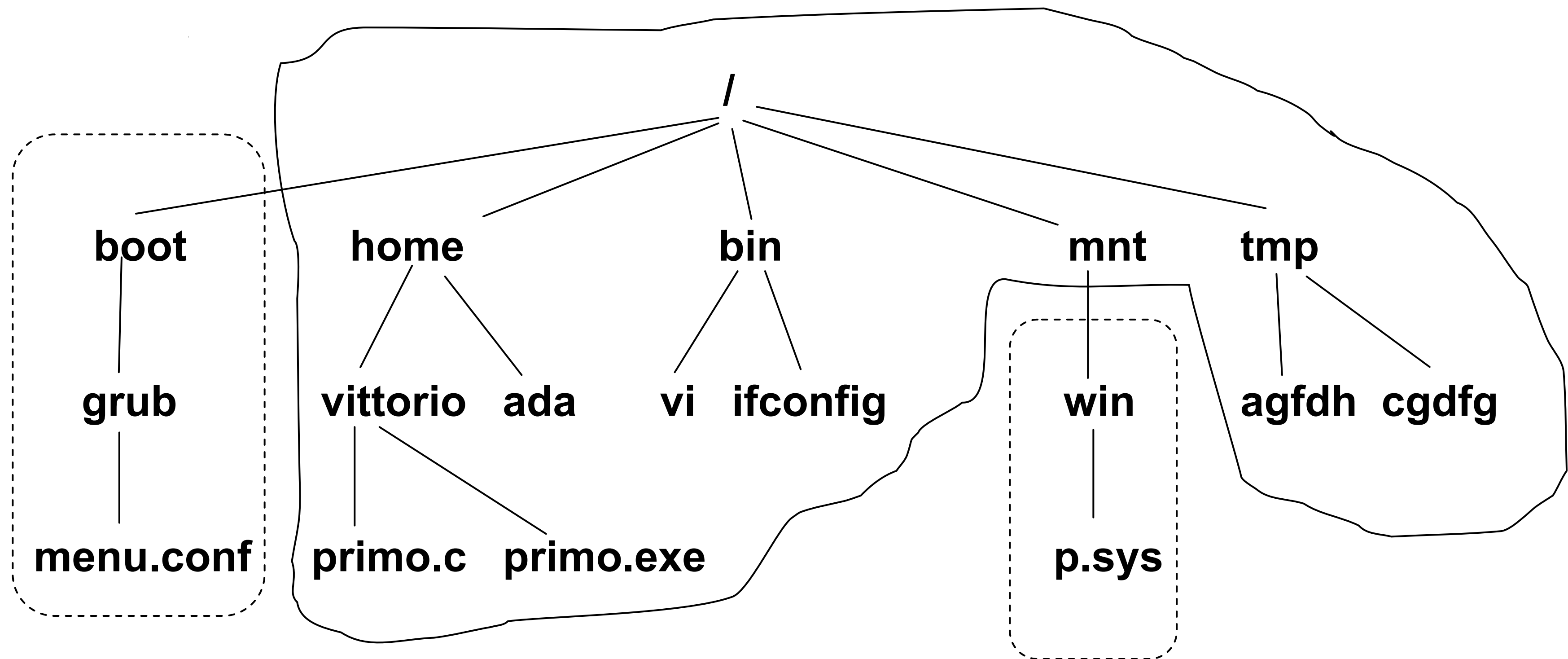
Se un file si chiama primo.c ed è contenuto in una directory che si chiama vittorio che a sua volta è contenuta in una directory che si chiama home che a sua volta è contenuta nella partizione principale, allora è possibile individuare univocamente il file pippo.c indicando il **percorso** mediante il quale, partendo dalla partizione / si arriva al file primo.c
/**home/vittorio/primo.c** <- percorso per raggiungere primo.c

Notare il carattere separatore **/** (slash) che indica dove inizia il nome di una directory o di un file. Quando il separatore / è all’inizio del percorso invece indica l’inizio della partizione principale, inizio che viene detto **root.**.

Le partizioni diverse da quella principale, si innestano (si collegano) logicamente in una qualche directory della partizione principale..
Ad esembio, una partizione chiamata boot può innestarsi direttamente nell’origine della partizione principale. Se questa partizione contiene una directory grub e quella directory contiene un file menu.conf, allora il percorso per identificare quel file menu.conf sarà:
/boot/grub/menu.conf <- percorso per raggiungere il file.conf

Notare che non si capisce se un nome indica una directory o una partizione.

Nozioni per l'uso del Terminale: File system (3)



Esempio di strutturazione in directories e files delle partizioni di un filesystem Linux:

Nell'esempio, alla partizione principale **/** sono collegate (tecnicamente si dice **montate**) altre due partizioni (circondate dalle linee tratteggiate) di nome **boot** e **win**..

Notate che mentre la partizione boot è montata direttamente come fosse una directory contenuta nella directory di inizio (detta root) della partizione principale (/), la partizione win è montata in una sottodirectory della root.

Nozioni per l'uso del Terminale: File system (4)

Nei sistemi **Linux/Mac** ciascun utente di un computer ha a disposizione dello spazio su disco per contenere i suoi files.

Per ciascun utente esiste, di solito, una directory in cui sono contenuti i files (intesi come directories e files) di quell'utente.
Quella directory viene detta **home dell'utente**.

> Per esempio, nei pc dei laboratori del corso di laurea in Informatica di Bologna, la home dell'utente "rossi" si trova in /**home/students/rossi**

Per accedere ad un computer ciascun utente deve utilizzare farsi riconoscere mediante un nome utente (**account**) e autenticarsi mediante una **password**.
L'operazione iniziale con cui l'utente si autentica per accedere ad un pc si dice login.

L'autenticazione degli utenti permette che il sistema operativo protegga i files di un utente impedendo l'accesso da parte di altri utenti o di persone esterne.
Ciascun utente può vedere quali sono i permessi di accesso ai propri files ed eventualmente modificarli.

Nozioni per l'uso del Terminale: File system (5)

Nel momento in cui si accede al terminale a riga di comando di un computer, il terminale **stabilisce la posizione logica attuale dell'utente all'interno del filesystem**, collocandolo inizialmente nella propria home directory.

Durante il lavoro l'utente può spostare la propria **posizione logica** in una diversa directory del filesystem.

NOTA BENE: spostarsi logicamente in una diversa directory VUOL DIRE VISITARE quella diversa directory e NON VUOL DIRE TRASFERIRE I PROPRI FILES IN QUELLA DIRECTORY

La directory in cui l'utente si trova logicamente in questo momento viene detta **directory corrente** e si dice che l'utente "**si trova nella**" directory corrente.

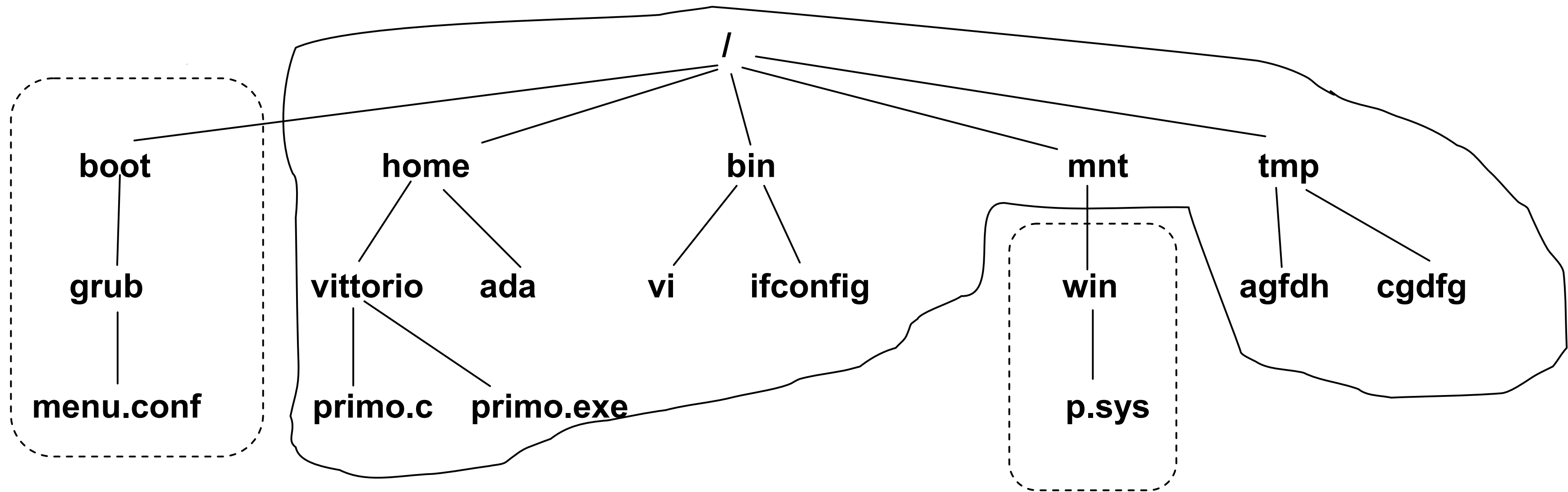
L'utente, per sapere in quale directory si trova logicamente in questo momento può eseguire il comando **pwd**, digitandolo da tastiera per farlo eseguire alla shell. Il comando pwd visualizza sullo schermo il percorso completo da / fino alla directory in cui l'utente si trova in quel momento.

```
 #  pwd
/home/students/rossi/merda
```

Per spostarsi logicamente in una diversa directory, l'utente usa il comando **cd**

```
# cd   /var/log
```

Nozioni per l’uso del Terminale: File system (6)



Supponiamo di trovarci logicamente nella directory /home/vittorio
Per spostarmi logicamente nella directory /home posso usare cd in tre modi diversi

cd /home <- specifico percorso assoluto

cd **../** <- specifico percorso relativo cioè partendo dalla directory corrente
Si noti che **il simbolo .. indica la directory superiore**

Per spostarmi dalla directory /home/vittorio alla directory /mnt/win

cd /mnt/win <- specifico percorso assoluto

cd **../../**mnt/win <- specifico percorso relativo

Per spostarmi dalla directory /boot alla directory /boot/grub

cd /boot/grub <- specifico il percorso assoluto

cd **./**grub <- specifico percorso relativo (**il simbolo . è la directory corrente**)

cd grub <- specifico percorso relativo (più semplice)

Nozioni per uso del Terminale: comando echo

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

Il comando

echo pippo pippa pippi

visualizza

pippo pippa pippi

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

Esempio: echo “pippo **;** pippa pippi”

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

pappa
ppero

Impossibile se non si usano i doppi apici

Soluzione: digitare (notando i doppi apici)

echo “pappa<INVIO>
ppero” <INVIO>

Nozioni per uso del Terminale: Variabili (1)

La shell dei comandi permette di usare delle **Variabili** che sono dei simboli dotati di un **nome** (che identifica la variabile) e di un **valore** (**che può essere cambiato**).
Il nome è una sequenza di caratteri anche numerici, ad es: PATH, PS1, USER PPID.
Attenzione, maiuscole e minuscole sono considerate diverse. Ad es P**a**TH !=P**A**TH
Il valore è anch'esso una sequenza di caratteri compresi caratteri numerici e simboli di punteggiatura, ad es: /usr:/usr/bin:/usr/sbin 17 i686-pc-cygwin

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

Le variabili di una shell possono essere stabilite e modificate sia dal sistema operativo sia dall'utente che usa quella shell.
Alcune variabili (dette d'ambiente) vengono impostate subito dal sistema operativo non appena viene iniziata l'esecuzione della shell (ad es la variabile PATH).
Altre variabili possono essere create ex-novo dall'utente in un qualunque momento dell'esecuzione della shell.

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

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

Nozioni per uso del Terminale: Variabili (2)

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

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

così la shell capisce che il nome della variabile finisce dove comincia lo spazio e sostituisce il valore al nome.
Viene visualizzato   MERDA X

Nozioni per uso del Terminale: Variabili (3)

Esiste una **variabile d'ambiente** particolare e importantissima, detta **PATH**
Viene impostata dal sistema operativo già all'inizio dell'esecuzione della shell.
L'utente può cambiare il valore di questa variabile.

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

Esempio di valore di PATH /bin**:**/sbin**:**/usr/bin**:**/usr/local/bin**:**/home/vittorio
Questa PATH definisce i percorsi che portano alle directory seguenti

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

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

Usare il Terminale a linea di comando (bash)

Per **aprire il terminale a riga di comando** Linux-like in ambiente grafico:
In Windows cercare e cliccare l'icona di "cygwin bash shell" o "cygwin terminal".
In Linux e Mac cliccare sul menù "terminal" o "terminal emulator" o "console".

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

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

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

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

Per essere eseguito, un file binario o uno script deve avere i **permessi di esecuzione**. Un utente può impostare il permesso di esecuzione di un proprio file usando il comando chmod come segue: **chmod u+x esempio_script.sh**

Come eseguire da Terminale a linea di comando

Come specificare il nome del comando o dell'eseguibile?

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

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

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

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

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

./primo.exe

Nozioni per uso del Terminale: Variabili (4)

Ogni shell supporta due tipi di variabili

Variabili locali

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

Utilizzate per computazioni locali all’interno di uno script

Variabili di ambiente

“Trasmesse” dalla shell alle subshell.

Viene creata una copia della variabile per la subshell

Se la subshell modifica la sua copia della variabile,

la variabile originale nella shell non cambia.

Solitamente utilizzate per la comunicazione fra parent e child shell

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

Per visualizzare l’elenco delle variabili, utilizzare il comando **env** (ENVironment)

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

nomevariabile=ValoreVariabile

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

export nomevariabile

Nozioni per uso del Terminale: Variabili (5)

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

var_caller.sh

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

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

echo "calling subshell"

./var_called.sh

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

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

var_called.sh

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

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

echo "modifico variabili "

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

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

echo "termina called"
```

Nozioni per uso del Terminale: Variabili (6)

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

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

Parametri a riga di comando passati al programma (1)

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

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

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

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

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

Il primo pezzo **chmod** è il nome dell’eseguibile (o anche **argomento di indice zero**).
Ciascuno degli altri pezzi è un argomento o parametro.
Nell’esempio, il pezzo u+x è l’**argomento di indice 1.**
Nell’esempio, il pezzo /home/vittorio/primo.exe è l’**argomento di indice 2.**
Nell’esempio, si dice che il numero degli argomenti passati è **2**.

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

Parametri a riga di comando passati al programma (2) Separatore di comandi ; e Delimitatore di argomenti “

Problema: Comandi Multipli su una stessa riga di comando

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

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

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

echo pippo **;** cd /tmp

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

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

echo **“**pippo ; cd /tmp “

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

Metacaratteri sostituiti dalla shell: *   ?

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

Con cosa sono sostituiti?

- ***** può essere sostituito da una qualunque sequenza di caratteri, anche vuota.
- **?** può essere sostituito da un singolo carattere oppure da nessuno.

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

Nessuna sostituzione

ls /home/vittorio	visualizza i nomi di tutti i file della directory
ls /home/vittorio/primo.c	visualizza il nome del solo file primo.c

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

ls /home/vittorio/*.exe	visualizza il nome di quei file della directory vittorio il cui nome termina per .exe (cioè primo.exe)
ls /home/vittorio/primo*	visualizza i nomi di quei file della directory vittorio il cui nome inizia per primo, cioè primo.c primo.exe

Sostituzione di ? con una singolo carattere oppure nessun carattere,

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

Comandi della bash ed eseguibili utili

Comandi

Comandi veri e propri

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

Istruzioni di controllo di flusso

for do done while if then else fi

Espressione condizionale su file

[condizione di un file **]**

Valutazione di espessione matematica applicata a variabili d'ambiente

((istruzione con espressione **))**

Eseguibili binari forniti dal sistema operativo

editor interattivi

vi nano (pico)

utilità

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

Comandi della bash (in ordine di importanza)

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

Funzione di Autocompletamento della bash. Tasto TAB (tabulazione)

Mentre sto digitando dei comandi può capitare di dover specificare il percorso assoluto o relativo di un file o di una directory.

Se comincio a scrivere questo percorso, **posso premere il tasto TAB** (tabulazione) il quale attiva la funzione di autocompletamento dei nomi di files o directory.

Tale funzione produce ogni volta uno dei quattro seguenti risultati:

1. Se non esiste nessun percorso che comincia con la parte di percorso scritta
   Non accade nulla, viene solo lanciato un beep di avvertimento.
2. Se esiste un solo percorso che comincia con la parte di percorso scritta
   Viene aggiunta alla parte scritta la parte mancante per completare il percorso.
3. Se esistono più percorsi che cominciano con la parte di percorso scritta e tutti questi percorsi hannouna parte di percorso comune oltre a quella già scritta
   Viene aggiunta alla parte scritta la parte comune a tutti i percorsi.
4. Se esistono più percorsi che cominciano con la parte di percorso scritta ma tutti questi non hanno altra parte di percorso in comune se non quella già scritta
   Viene visualizzato l'elenco dei possibili percorsi che possono essere scelti per completare il percorso già scritto.

Parametri a riga di comando passati al programma (3)

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

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

- **$#** il numero di argomenti passati allo script
- **$0** il nome del processo in esecuzione
- $1 primo argomento, $2 secondo argomento, …..
- **$*** tutti gli argomenti passati a riga di comando concatenati e separati da spazi

Esempio:

All'interno di uno script posso usarle cosi (vedere esempio_script.sh) :

```
echo "ho passato $# argomenti alla shell"
echo “tutti gli argomenti sono $*“
```

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

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

SENZA metacaratteri

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

CON metacaratteri

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

Parametri a riga di comando passati al programma (4)

File esempio_args.sh
eseguitelo chiamandolo con diversi argomenti e delimitatori
esempio_args.sh
esempio_args.sh alfa beta gamma
esempio_args.sh “alfa beta gamma”
esempio_args.sh “alfa beta” gamma

```
#!/bin/bash

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

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

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

Supponiamo di avere una prima variabile varA che contiene un valore qualunque.
Supponiamo di avere una altra variabile il cui valore è proprio il nome della prima variabile.
Voglio usare il valore della prima variabile sfruttando solo il nome della seconda variabile il cui valore è proprio il nome della prima variabile. Si dice che la seconda variabile è un riferimento indiretto alla prima variabile.

Accedere al valore di una prima variabile il cui nome è il valore di una seconda variabile è possibile nella bash a partire dalla versione 2.
Si sfrutta un operatore ${!

Esempio:

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

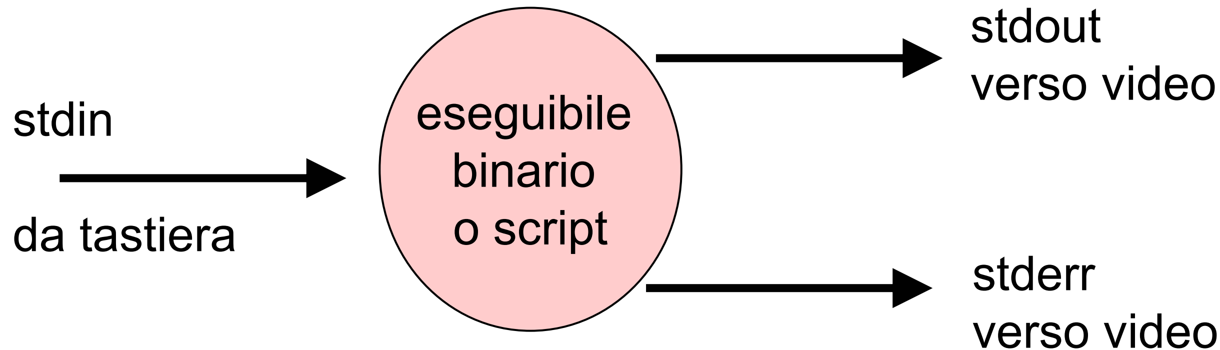
Riferimenti Indiretti a Variabili (2)
operatore **${!**
solo in bash versione 2

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

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

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

Stream di I/O predefiniti dei processi (1)



Quando un programma entra in esecuzione l'ambiente del sistema operativo si incarica di aprire 3 flussi di dati standard, che sono:

- **STANDARD INPUT** (**stdin**) serve per l'input normale (per default da tastiera). Viene identificato da una costante valore numerico 0.
- **STANDARD OUTPUT** (**stdout**) serve per l'output normale (per default su schermo video). Viene identificato da una costante valore numerico 1.
- **STANDARD ERROR** (**stderr**) serve per l'output che serve a comunicare messaggi di errore all'utente (per default anche questo su schermo video). Viene identificato da una costante valore numerico 2.

Stream di I/O predefiniti dei processi (2)

Come fare output in un programma C – la funzione printf();

esempio: vedere il file primo.c

Come fare output in uno script bash – il comando echo

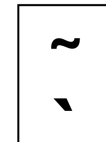
esempio già visto: echo “ciao bella”

Come catturare l’output di un programma chiamato da uno script

(NB: per far eseguire primo.exe ho usato l’apice semplice che non è l’apostrofo)

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

L’apice giusto da usare è ` quello che in alto tende a sinistra e in basso tende a destra. Viene chiamato backticks o backquotes. Nelle tastiere americane si trova nel primo tasto in alto a sinistra, accoppiato e sotto alla tilde.



Non è l’apostrofo italiano ‘ (single quote) **, il tasto del backquotes nelle tastiere italiane non esiste.**
Conviene settare la propria tastiera per comportarsi come una tastiera americana.

Ridirezionamenti di Stream di I/O (1)

Ridirezionamenti:

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

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

program **<** file_input

il programma vedrà il contenuto del file file_input come se venisse dalla tastiera.

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

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

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

Ridirezionamenti di Stream di I/O (2)

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

program **>** file_output

Nel file file_output troveremo i caratteri che il programma voleva mandare su video.
Il precedente contenuto del file verrà perso ed **alla fine dell'esecuzione del programma nel file troveremo solo l'output generato dal programma stesso**.
Si dice che il file di output è stato sovrascritto.

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

program **>>** file_output

Ridirezionamenti di Stream di I/O (3)

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

pippo
pappa

Supponiamo che il file file_output contenga queste tre righe:

uno
due
tre

Se eseguiamo il programma aggiungendolo l'output in coda al file file_output, così:

program **>>** file_output

alla fine dell'esecuzione il contenuto del file file_output sarà:

uno
due
tre
pippo
pappa

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

program **>** file_output

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

pippo
pappa

Ridirezionamenti di Stream di I/O (4)

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

```
program <  file_input > file_output
```

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

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

```
program  &>   file_error_and_output
```

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

```
program   2>  file_error    >  file_output
```

Ridirezionamenti di Stream di I/O (5)

E' possibile **eseguire due (o più) programmi contemporaneamente**, facendo si che **lo standard output del primo programma sia ridiretto nello standard input del secondo programma**, in modo tale che quello che viene mandato in output dal primo programma viene ricevuto dal secondo programma come se provenisse dalla tastiera.

Si utilizza a tal scopo l'operatore pipe **|** a concatenare i due programmi scritti sulla stessa riga di comando.

program1 **|** program2

Supponiamo che un file pippo.txt contenga le seguenti righe di testo.

ghini è un cretino
ghini è un fesso
ghini è scarso come ciclista

Usiamo il comando **cat *pippo.txt*** che manda in output le righe di testo contenute nel file pippo.txt.
Usiamo il programma **grep *un*** che riceve dallo stdin delle righe di testo e manda in output solo quelle righe che contengono la parola ***un***.
Usiamo il programma **grep *fesso*** che riceve dallo stdin delle righe di testo e manda in output solo quelle righe che contengono la parola ***fesso***.

Eseguendo i comandi concatenati dalla pipe nel modo seguente

cat pippo.txt | grep un | grep fesso

Ottengo come output del programma *grep fesso* la sola seguente linea di testo:

ghini è un fesso **(ogni riferimento a persone esistenti è puramente casuale)**

Ridirezionamenti di Stream di I/O (6)

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

program1 **;** program2 **;** program3

Con il separatore ; io faccio eseguire i diversi programmi uno dopo l’altro, cioè prima che parta il secondo programma deve finire il primo e così via.
Inoltre l’output di un programma non viene ridirezionato nell’input del programma successivo.

program1 **|** program2 **|** program3

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

Valore restituito da un programma al chiamante

Differenza tra valore restituito e output

Ogni programma o comando restituisce un valore numerico per indicare se c'è stato un errore durante l'esecuzione oppure se tutto è andato bene. Di solito un risultato 0 indica tutto bene mentre un risultato diverso da zero indica errore.

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

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

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

vedi esempio primo.c

Come restituire il risultato in uno script bash – il comando exit

esempio: **exit 9** **fa terminare lo script e restitusce 9 come risultato**

Come catturare il risultato di un programma chiamato da uno script

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

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

Sequenze di comandi condizionali e non

Sequenze non condizionali (vedi una slide precedente)

Il metacarattere "**;**" viene utilizzato per eseguire due o più comandi in sequenza

date ; pwd ; ls

Sequenze condizionali

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

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

Esempi

Eseguire il secondo comando in caso di successo del primo

$ gcc prog.c –o prog && prog

Eseguire il secondo comando in caso di fallimento del primo

$ gcc prog.c || echo Compilazione fallita

read - Lettura da standard input (tastiera o file)

Uno script può leggere dallo standard input delle sequenze di caratteri usando un comando chiamato **read**.

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

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

0 se viene letto qualcosa, **>0** se si arriva a fine file

```
while (( 1 ))
do
        read RIGA
        if (( "$?" != 0 ))
        then
                echo “eof reached "
                break
        else
                echo "read \"${RIGA}\" "
        fi
done
```

Processi in background e in foreground (1)

- Processi in foreground
  - Processi che “controllano” il terminale dal quale sono stati lanciati, nel senso che hanno il loro standard input collegato al terminale e impegnano il terminale non permettendo l’esecuzione di altri programmi collegati a quel terminale, fino alla loro terminazione
  - In ogni istante, un solo processo è in foreground
- Processi in background
  - Vengono eseguiti senza che abbiano il controllo del terminale a cui sono “attaccati”, nel senso che non hanno lo standard input collegato al terminale e permettono che vengano eseguiti nel frattempo altri programmi collegati a quel terminale.
- Job control
  - Permette di portare i processi da background a foreground e viceversa

Processi in background e in foreground (2)

- **`&`**: lancia un processo direttamente in background
  - Esempio:
    `$ prova &`
- **`^Z`**: sospende un processo in foreground
- **`^C`**: termina un processo in foreground
- **`fg`**: porta in foreground un processo sospeso
- **`bg`**: riprende l'esecuzione in background di un processo sospeso
- **`jobs`**: produce una lista dei processi in background
- **`kill`**: elimina il processo specificato con il proprio identificatore pid
  - Esempio:
    `$ kill 6152`

- Nota: **`jobs`** si applica ai soli processi attualmente in esecuzione nella shell, mentre la lista completa dei processi in esecuzione nel sistema può essere ottenuta con i comandi **`top`** o **`ps –ax`**

Esempio di script bash (1)

```
#!/bin/bash
echo "ho passato $# argomenti alla shell"
echo "l'identificatore del processo corrente e' $$"
echo "il nome del processo in esecuzione e' $0"
echo "gli argomenti passati a riga di comando sono $*"
NUM=0
for name in  $* ;  do
    echo "argomento ${name}" ;
    if [ ${name} != "picchio" ]
    then
        # controllo se il file di nome ${name} esiste gia' ed e' modificabile
        if [ -w ${name} ]
        then
            # se il file esiste lo elimino
            rm ${name}
        else
            # se invece il file non eeiste allora lo creo
            touch ${name}
            echo ciclismo > ${name}
            echo "creato il file ${name}"
            ((NUM=${NUM}+1))
        fi
    fi
done
echo "creati $NUM file"
```

Esempio di script bash (2)

continua il file esempio_script.sh della slide precedente

```
# come leggere input da tastiera

echo -e "scrivi una frase \c"
read param
echo "la frase scritta e': $param"

# eseguiamo un programma e vediamo il suo risultato

./primo.exe

echo "il processo chiamato ha restituito come valore di uscita $? "
echo "il processo chiamato ha restituito come valore di uscita $? “
# domanda: perché la seconda volta echo dà un output diverso?



# eseguiamo ancora il programma e vediamo il suo output

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

Esempio di programma ANSI C

Questo che segue è il file **primo.c**

Deve essere compilato e linkato usando il comando:

```
gcc –ansi -Wall -o primo.exe primo.c
```

Deve essere eseguito, stando nella directory dove si trova primo.exe, digitando

```
./primo.exe
```

```
#include <stdlib.h>
#include <stdio.h>
#include <unistd.h>

int main(void) {
    printf("vaffanculo!! \n");        /* stampa a video la gentile stringa di testo */
    return (11);                      /* restituisce al chiamante il valore 11 */
}
```

Esercizio per casa con gli script per bash

Spostarsi nella directory /usr/bin ,
creare una sottodirectory PIPPO,
entrarci dentro,
creare un nuovo file cacchio.sh contenente il comando
echo ghini è un cretino.

rendere eseguibile il file cacchio.sh
lanciarlo.
Alla fine dell'esecuzione eliminare il file cacchio.sh,
tornare alla directory superiore
ed eliminare la directory PIPPO

…..