Laboratorio di Sistemi Operativi

Alessandro Gerotto

Universitá degli studi di Udine

Insegnante *Scagnetto Ivan*

Anno accademico 2021-22

Shell Linux

La parte del sistema operativo UNIX dedita alla gestione dell'interazione con l'utente è la **shell**, ovvero, un'**interfaccia a carattere** dove l'utente impartisce i comandi al sistema digitando ad in apposito prompt.

Il sistema stampa sullo schermo del terminale eventuali messaggi all'utente in seguito all'esecuzione dei comandi, facendo poi riapparire il prompt, in modo da continuare l'interazione.

Tipi di shell e funzionamento

- **sh**: Bourne shell
- **bash**: Bourne again shell
- **csh**: C shell
- **tcsh**: Teach C shell
- **ksh**: Korn shell

Quando viene **invocata** una shell (automaticamente al login o esplicitamente):

1. Viene **letto un file speciale** nella home, contenente informazioni per l'inizializzazione;
2. Viene **visualizzato un prompt**, in **attesa** che l'utente invii un comando;
3. Se l'utente invia un comando, la shell lo **esegue e ritorna al punto 2**;

Pathname assoluto e relativo

- **Assoluto**: rispetto a *root/* (a.e. */home/gero/progetto/a*);
- **Relativo**: rispetto alla directory corrente (a.e. *progetto/a* trovandosi in */home/gero*).

Comandi utili:

- Present working directory: `> pwd`
- Change directory: `> cd /dir`
- Spostarsi nella dir madre: `> cd ..`

Ricerca di file

Attraversa ricorsivamente le dir in `<path>` applicando le regole specificate in `<expression>` a tutti i file e sottodirectory trovati:

`> find <path> <expression>` dove `<expression>` può essere un' opzione, una condizione o un'azione.

Alcuni esempi:

`> find <path> -name "*.txt"`	Stampa i file in <path> che terminano in ".txt".
`> find . -name '*.c' -print`	Cerca ricorsivamente partendo dalla dir corrente i file con estensione "c" e li stampa.

`> find . -name '*.bak' -ls -exec rm {} \;`	Cerca ricorsivamente partendo dalla dir corrente tutti i file con estensione bak, li stampa con relativi attributi (-ls) e li cancella (-exec rm {}. \; Il carattere \ serve per fare il "quote" del ;)
`> find /etc -type d -print`	Cerca ricorsivamente a partire dalla dir /etc tutte e solo le sottodirectory, stampandole a video

Comandi per manipolare file e directory

- Creazione directory: `> mkdir NOME`
- Rimozione directory: `> rmdir NOME`
- Copia file *f1* in *f2*: `> cp f1 f2`
- Sposta/rinomina file *f1* in *f2*: `> mv f1 f2`
- Torna byte e # linea dove f1 e f2 differiscono: `> cmp f1 f2`
- Cambiamenti da fare a f1 per renderlo =f2: `> diff f1 f2`
- Listing dei file: `> ls [OPTION]...[FILE]...`

A.e. eseguendo: `> ls -l /bin` si ottiene il seguente output:

```
...
lrwxrwxrwx    1 root     root            4 Dec  5  2000 awk -> gawk
-rwxr-xr-x    1 root     root         5780 Jul 13  2000 basename
-rwxr-xr-x    1 root     root       512540 Aug 22  2000 bash
...
```

Da sinistra a destra abbiamo:

1. **tipo di file** (**-** file normale, **d** directory, **l** link, **b** block device, **c** character device);
2. **permessi** (root, owner, group, world);
3. **numero di hard link** al file;
4. nome del **proprietario** del file;
5. nome dell'insieme di utenti che **possono accedere** al file come gruppo;
6. **grandezza** del file in byte;
7. data di **ultima modifica**;
8. **nome** del file.

Permessi e comando chmod

Linux è un sistema **multiutente**. Per ogni file ci sono 4 categorie di utenti: **root** (amministratore)**, owner, group, world** (tutti gli altri). L'amministratore del sistema (root) ha tutti i permessi (lettura, scrittura, esecuzione) su tutti i file. Per le altre categorie di utenti l'accesso ai file è regolato dai permessi:

```
> ls -l /etc/passwd
-rw-r--r--    1 root     root      981 Sep 20 16:32 /etc/passwd
```

Il blocco di caratteri `rw-r--r--` rappresenta i **permessi di accesso** al file:

- I primi 3 (`rw-`) sono riferiti all'**owner**;
- Il secondo blocco di 3 caratteri (`r--`) è riferito al **group**;
- l'ultimo blocco (`r--`) è riferito alla categoria **world**.

La prima posizione di ogni blocco rappresenta il **permesso di lettura** (r), la seconda il **permesso di scrittura** (w) e la terza il permesso di **esecuzione** (x) (Nota: per "attraversare" una directory, bisogna avere il permesso di esecuzione su di essa).

Un **trattino** (-) in una qualsiasi posizione indica l'**assenza del permesso** corrispondente.

L'owner di un file può **cambiarne i permessi** tramite il comando chmod:

- `> chmod 744 [FILE]` imposta i permessi del file [FILE] a rwxr--r--

  Infatti: rwxr--r-- 111 100 100 = 7 4 4 (leggendo ogni gruppo in ottale)
- `> chmod u=rwx,go=r f1` (produce lo stesso effetto del comando precedente) dove:
  - `u` rappresenta l'**owner**,
  - `g` il **gruppo**,
  - `o` il **world**.

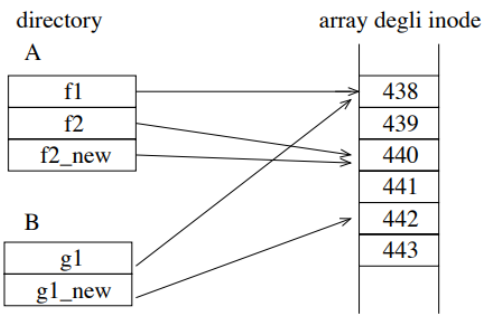
Inoltre: + aggiunge i permessi che lo seguono, - toglie i permessi che lo seguono, = imposta esattamente i permessi che lo seguono. Quindi l'effetto di chmod g+r f1 `e in generale diverso da chmod g=r f1.

Visualizzare il contenuto di un file

- Mostra **tutto** il contenuto di un file: `> cat [FILE]`
- Mostra il contenuto di un file **a blocchi**: `> more [FILE]`
- Mostra le **ultime righe** di un file: `> tail [FILE]`
- Mostra le **prime righe** di un file: `> head [FILE]`

Inodes

In UNIX, ad ogni file corrisponde un **numero di inode**, che è l'**indice in un array memorizzato su disco**. Ogni elemento contiene le informazioni relative al file (data di creazione, proprietario, dove si trova il contenuto del file su disco, …). Le **directory** sono tabelle che associano nomi di file a numeri di inode. Ogni entry di una directory è un link.



Link hard e simbolici

I link possono essere:

- **link simbolici**: file contenenti all'interno un collegamento ad un altro file);
- **link hard**: è una copia di un file che contiene al suo interno il contenuto del file originario).

Creazione di:

- **Link simbolico**: `> ln -s g1 g1_new` (`g1_new` è un text file che contiene il pathname di g1.)
- **Link hard**: `> ln f2 f2_new`

Metacaratteri della shell Unix

Un metacarattere è un carattere che rappresenta un **insieme di altri caratteri** e viene processato in modo speciale. Esempio:

`> ls *.java`	Stampa la lista dei file che terminano con .java.
`> ls prova?`	Stampa i file il cui nome è "prova?" dove "?" sta per un carattere sconosciuto.
`> ls /dev/tty[234]`	Stampa tutti i file che iniziano con "tty" e terminano con 2, 3 o 4.
`> mkdir /home/gero/Desktop{a,b,c}`	Crea le directory "a", "b", "c".

Il quoting

Il meccanismo del quoting è utilizzato per **inibire** l'effetto dei **metacaratteri**. I metacaratteri a cui è applicato il quoting perdono il loro significato speciale e la shell li tratta come caratteri ordinari. Ci sono tre modi per fare quoting:

- **Escape \:** inibisce l'effetto speciale del metacarattere che lo segue;
- **Singoli apici**: inibisce l'effetto dei metacaratteri contenuti tra singoli apici;
- **Doppi apici:** inibisce l'effetto dei metacaratteri per l'abbreviazione del pathname presenti in una stringa racchiusa tra doppi apici.

Redirezione del I/O

Di default i comandi Unix prendono l'input da tastiera (**standard input**) e mandano l'output ed eventuali messaggi di errore su video (**standard output, error**). L'input/output in Unix può essere **rediretto** da/verso file, utilizzando opportuni metacaratteri:

- `>` redirezione dell'**output**;
- `>>` redirezione dell'**output (append)**;
- `2>` redirezione dei **messaggi di errore**.

- `<` redirezione dell'**input**;
- `<<` redirezione dell'**input dalla linea di comando** ("here document");

  Il testo fornito in input al comando wc viene digitato alla console dall'utente fintanto che non si incontra una linea contenente il delimitatore delim che indica la fine dell'input, ovvero, la fine del documento temporaneo. Tuttavia il file **non esiste fisicamente** nel filesystem, ma viene creato "al volo" tramite redirezione dello standard input. Un esempio classico è il seguente:

  ```
  > wc <<delim
  > queste linee formano il contenuto
  > del testo
  > delim
  ```

  output: `2 7 44`

Pipe

Il metacarattere "|" (**pipe**) serve per comporre *n* comandi "**in cascata**" in modo che l'output di ciascuno sia fornito in input al successivo. L'output dell'ultimo comando è l'output della pipeline.

Bash: History list

L'history list è un tool che **memorizza** in essa **gli ultimi 500 comandi** inseriti dall'utente. Viene memorizzata nel file `.bash_history` nell'home directory dell'utente al momento del logout e riletta al momento del login.

`> history` → visualizza la lista dei comandi. Esempio:

`> history | tail -5`	Stampa gli ultimi 5 comandi usati.
`> ![NUMERO_EVENTO]`	Esegue il comando che corrisponde al numero dell'evento.
`> !!`	Esegue l'ultimo comando digitato;
`> ![QUALCOSA]`	Esegue l'ultimo comando che contiene [QUALCOSA].
`> !ls:s/al/i`	Sostituisci (:s) la stringa "al" con la stringa "i".

Alias

È un comando che permette di **definire altri comandi**. All'uscita dalla shell gli alias sono **automaticamente rimossi**.

Esempio:

`> alias [NOME]='[COMANDO]'` → digitando `[NOME]` esegue `[COMANDO]`;

Per **rimuovere** uno o più alias:

`> unalias [NOME_ALIAS1] [NOME_ALIAS2]` → rimuove i due alias;

Processi

Ogni processo del sistema ha un **PID** (Process Identity Number). Ogni processo può generare nuovi processi (figli). La radice della gerarchia di processi è il processo **`init`** con PID=1. `init` è il primo processo che parte al boot di sistema.

- `> ps` → fornisce i processi presenti nel sistema associati alla shell corrente;
- `> ps -af` → tutti i processi del sistema associati ad una shell (`-a`), full listing (`-f`);
- `> ps -el` → tutti i processi del sistema non associati ad una shell (`-e`), long listing (`-l`);
- `> tty` → terminale corrente

Comandi filtro

I filtri sono una particolare **classe di comandi** che possiedono i seguenti requisiti:

- prendono l'**input dallo standard input** device;
- **effettuano delle operazioni** sull'input ricevuto;
- **inviano il risultato** delle operazioni allo standard output device.

grep

Per **ricercare** una determinata stringa di caratteri in un file si utilizza il comando grep.

`> grep <stringa> <file>`

dove stringa è la parola o la frase da ricercare e file è deve essere eseguita la ricerca.

Per impostare un modello di ricerca formato da più **parole separate**, occorre racchiudere la stringa tra **virgolette singole o doppie.**

`> ls -l *.ps | grep mag`	Prima **produce un elenco dei file** che terminano per ".ps". Poi **ricerca la stringa "mag"**. Stampa quindi tutti i file creati a maggio.
`> grep -v e *`	Per ricercare le righe di un file che **non** contengono una stringa si usa l'**opzione -v**. Ricerca le righe dei file che non contengono "e".
`> grep '^b' lista`	Cerca le righe del file lista che **iniziano con "b"** (^ → inizio riga).
`> grep 'b$' lista`	Cerca le righe del file lista che **terminano con "b"** ($ → fine riga).
`> grep '^b$' lista`	Visualizza tutte le righe del file lista che **contengono solo il carattere "b"**.

`> grep 'de.' lista`	Estrae dal file lista tutti i **gruppi di tre caratteri** in cui **"de" siano i primi due** (a.e. "dei", "del", "scodella", "code", …).

fgrep ed egrep

Grep ricerca per **parola-chiave**. Per filtrare **più di una parola** si usano le espressioni regolari mediante egrep.

`> egrep "Lorem|mauris" test.txt` → ricerca le occorrenze di "Lorem" e "marius" all'interno di test.txt.

sort

Sort **ordina** le linee alfabeticamente o secondo il criterio specificato, producendo il risultato sullo standard output.

```
> sort [opzioni] [--] [file1 [file2 …] ]
```

tr

Permette di eseguire operazioni come la **conversione** di lettere minuscole in maiuscole, **cancellazione** della punteggiatura ecc.

`> tr a-z A-Z`	Converte le **minuscole** in **maiuscole**.
`> tr -c A-Za-z0-9 ' '`	**Sostituisce** i caratteri non alfanumerici con spazi (-c: complemento).
`> tr -d str`	**Cancella** i caratteri contenuti nella stringa `str`.

cut and paste

Il comando **cut** serve a **stampare** parti di **stringhe selezionate** da un FILE o dallo standard input.

```
> cut [OPTION] [FILE]
```

Con l'opzione `-d[CHAR]` si può specificare qual è il **carattere che delimita i vari campi** (`-f`).
Esempio sul file /etc/passwd:

```
> head -4 /etc/passwd
      root:x:0:0:root:/root:/bin/bash
      daemon:x:1:1:daemon:/usr/sbin:/usr/sbin/nologin
      bin:x:2:2:bin:/bin:/usr/sbin/nologin
      sys:x:3:3:sys:/dev:/usr/sbin/nologin
```

```
> head -4 /etc/passwd | cut -d: -f1 → stampa il primo campo
      root
      daemon
      bin
      sys
> head -4 /etc/passwd | cut -d: -f6 → stampa il sesto campo
      /root
      /usr/sbin
      /bin
      /dev
```

Stream editor: sed

sed (editor di flusso) è un comando che consente il **filtraggio** e la **manipolazione di testi**.

`> sed [OPZIONI]... {file regole filtraggio} [file di input]..`

Esempi:

`> echo 1:2:3:4:5 | sed s/:/-/g` → output: `1-2-3-4-5`

`> sed -n '10,50p' testo.txt` → stampa le righe dalla 10 alla 50

`> sed -z 's/\n/,/g' testo.txt` → rimpiazza tutti gli "a capo" con delle ','

Sostituzione testo con sed

Il formato dell'azione di sostituzione in sed è il seguente:

`> s/expr/new/flags` dove:

- **expr** è l'espressione da cercare,
- **new** è la stringa da sostituire al posto di expr,
- **flags** è uno degli elementi seguenti:
    - **num**: un numero da 1 a 9 che specifica quale occorrenza di expr deve essere sostituita (di default `e la prima),
    - **g**: ogni occorrenza di expr viene sostituita,
    - **p**: la linea corrente viene stampata sullo standard output nel caso vi sia stata una sostituzione)
    - **w** file: la linea corrente viene accodata nel file file nel caso vi sia stata una sostituzione.

Shell script

Gli shell script sono **programmi** interpretati dalla shell, scritti in un linguaggio i cui costrutti atomici sono i **comandi** Unix. I comandi possono essere **combinati** in sequenza o mediante i **costrutti usuali** di un linguaggio di programmazione. La sintassi varia da shell a shell.

Uno shell script va scritto in un file utilizzando per esempio il comando **cat o un editor** (vi, emacs, etc). Per poter eseguire lo script, il file deve essere reso **eseguibile**. Lo script viene eseguito invocando il nome del file. Esempio:

```
> cat >prova                    # il file prova viene editato
ls /usr/bin | wc -w            # scrittura del programma
Ctrl-d                         # fine dell'editing
> chmod 700 prova              # dirsize viene reso eseguibile
> ./prova                      # viene invocato il comando prova
459                            # risultato dell'esecuzione
```

Variabili

Le variabili della shell sono **stringhe di caratteri** a cui è **associato** un certo **spazio in memoria**. Le variabili della shell possono essere utilizzate sia sulla linea di comando che negli script. **Non c'è dichiarazione esplicita** delle variabili.

Assegnamento di una variabile (eventualmente nuova): variabile=valore (Importante: non lasciare spazi a sinistra ed a destra dell'operatore =).

Per **accedere** al valore di una variabile si utilizza il $:

```
> x=variabile
> echo il valore di x: $x
```

Output: il valore di x: variabile

Variabili d'ambiente

Le variabili definite come sopra sono **locali alla shell** o **allo script** in cui sono definite. Per rendere **globale** una variabile (e renderla una ='ambiente) si usa il comando **export**:

```
> export x
```

Esiste un insieme di **variabili di ambiente speciali** definite al momento del login:

- PS1 prompt della shell;
- PS2 secondo prompt della shell; utilizzato a.e. in caso di ridirezione dell'input;
- PWD pathname assoluto della directory corrente;
- UID ID dello user corrente;
- PATH lista di pathname di directory in cui la shell cerca i comandi;
- HOME pathname assoluto della home directory

Parametri

Le variabili `$1, $2, …, $9` sono variabili **speciali associate** al primo, secondo, …, nono parametro passato sulla linea di comando quando viene invocato uno script. Se uno script ha più di 9 parametri, si utilizza il comando shift per fare lo shift a sinistra dei parametri e poter accedere ai parametri oltre il nono.

Esempio script di nome `file` che contiene:

```
expr $1 + $2
```

Nel momento in cui si digita sulla shell:

```
> ./file 5 8
```

Da come output: 13

Variabili di stato automatiche

Sono variabili speciali che servono per **gestire lo stato** e sono aggiornate automaticamente dalla shell. L'utente può **accedervi solo in lettura**. Al termine dell'esecuzione di ogni comando unix, viene restituito un valore di uscita, exit status, uguale a:

- 0, se l'esecuzione è terminata con successo;
- diverso da 0, altrimenti (codice di errore).

La variabile speciale `$?` contiene il valore di uscita dell'ultimo comando eseguito.

Altre variabili di stato sono:

- `$?` exit status dell'ultimo comando eseguito dalla shell;
- `$$` PID della shell corrente;
- `$!` il PID dell'ultimo comando eseguito in background;
- `$-` le opzioni della shell corrente;
- `$#` numero dei parametri forniti allo script sulla linea di comando;
- `$*, $@` lista di tutti i parametri passati allo script sulla linea di comando.

Controllo di flusso if-then-else

Il comando condizionale:

```
if [CONDITION]
then
      [TRUE]
else
      [FALSE]
fi
```

Esegue il comando `[CONDITION]` e utilizza il suo **exit status** per decidere se eseguire i comandi `[TRUE]` (exit status 0) od i comandi `[FALSE]` (exit status diverso da zero).

Condizioni e comando test

Se la condizione che si vuole specificare non è esprimibile tramite l'exit status di un "normale" comando, si può utilizzare l'apposito comando **test**:

```
test expression
```

che restituisce un **exit status** pari a **0 se expression è vera**, **pari a 1 altrimenti**. Si possono costruire vari tipi di espressioni:

- Espressioni che controllano **se un file possiede certi attributi:**
  - -e STRING f restituisce vero se f esiste;
  - -f STRING f restituisce vero se f esiste ed è un file ordinario;
  - -d STRING f restituisce vero se f esiste ed è una directory;
  - -r STRING f restituisce vero se f esiste ed è leggibile dall'utente;
  - -w STRING f restituisce vero se f esiste ed è scrivibile dall'utente;
  - -x STRING f restituisce vero se f esiste ed è eseguibile dall'utente;
- Espressioni su **stringhe**:
  - -z str restituisce vero se str è di lunghezza zero;
  - -n str restituisce vero se str non è di lunghezza zero;
  - str1 = str2 restituisce vero se str1 è uguale a str2;
  - str1 != str2 restituisce vero se str1 è diversa da str2;
- Espressioni su **valori numerici**:
  - num1 -eq num2 restituisce vero se num1 è uguale a num2;
  - num1 -ne num2 restituisce vero se num1 non è uguale a num2;
  - num1 -lt num2 restituisce vero se num1 è minore di num2;
  - num1 -gt num2 restituisce vero se num1 è maggiore di num2;
  - num1 -le num2 restituisce vero se num1 è minore o uguale a num2;
  - num1 -ge num2 restituisce vero se num1 è maggiore o uguale a num2.
- **Espressioni composte**:
  - exp1 -a exp2 restituisce vero se sono vere sia exp1 che exp2;
  - exp1 -o exp2 restituisce vero se è vera exp1 o exp2;
  - ! exp restituisce vero se non è vera exp;
  - (exp) le parentesi possono essere usate per cambiare l'ordine di valutazione degli operatori (è necessario farne il quoting).

Cicli while

Vengono **eseguiti i comandi** `[COMMANDS]` **finché la condizione** `[CONDITION]` **è vera**. Sintassi:

```
while [CONDITION] do
      [COMMANDS]
done
```

Esempio:

```
while test -e $1 do
       sleep 2
done
echo file $1 does not exist
exit 0
```

Lo script precedente esegue un ciclo che dura finché il file fornito come argomento non viene cancellato. Il comando che viene eseguito come corpo del while è una pausa di 2 secondi.

Cicli uniti

Vengono **eseguiti i comandi** `[COMMANDS]` **finché la condizione** `[CONDITION]` **è falsa**. Sintassi:

```
until [CONDITION] do
[COMMANDS]
done
```

Esempio di uno script che legge continuamente dallo standard input e visualizza quanto letto sullo standard output, finché l'utente non inserisce la stringa end:

```
until false do
      read firstword restofline
      if test $firstword = end
      then
            exit 0
      else
            echo $firstword $restofline
      fi
done
```

Cicli for

Vengono eseguiti i comandi `[COMMANDS]` **per ogni elemento contenuto in wordlist** (l'elemento corrente è memorizzato nella variabile var).

Sintassi:

```
for var in wordlist do
      [COMMANDS]
done
```

Sintassi alternativa solo per shell bash (serve specificarlo includendo `#!/bin/bash` in testa allo script):

```
for (( X=0; X<var; X++)) do
      [COMMANDS]
Done
```

Case selection

L'effetto risultante è che vengono eseguiti i comandi `[COMMAND1]`, `[COMMAND2]`,... a seconda del fatto che string sia uguale a `[EXPRESSION1]`, `[EXPRESSION2]`,... I comandi `[DEFAULT_COMMANDS]` vengono eseguiti soltanto se il valore di string non coincide con nessuno fra `[EXPRESSION1]`, `[EXPRESSION2]`,... I valori `[EXPRESSION1]`, `[EXPRESSION2]`,... possono essere specificati usando le solite regole per l'espansione del percorso (caratteri jolly). Sintassi:

```
case string in
[EXPRESSION1])
      [COMMAND1]
      ;;
[EXPRESSION2])
      [COMMAND2]
      ;;
...
*)
      [DEFAULT_COMMANDS]
      ;;
esac
```

Command substitution

Il meccanismo di command substitution permette di **sostituire** ad un comando o pipeline **quanto stampato sullo standard output** da quest'ultimo. Esempi:

```
> date
Tue Nov 19 17:50:10 2002
```

```
> vardata=‘date‘
> echo $vardata
Tue Nov 19 17:51:28 2002
```

Per operare una command substitution si devono usare gli “**apici rovesciati**” o backquote (‘), non gli apici normali (’) che si usano come meccanismo di quoting. ‘

Es.1: Si supponga che sia stata impostata una variabile di ambiente di nome utente. Si scriva una pipeline che stampi a video la stringa ok se esiste un utente del sistema (contenuto in /etc/passwd) con nome di login uguale al valore della variabile utente. In caso contrario, il comando o pipeline non deve stampare nulla.

```
cat /etc/passwd | cut -d: -f1 | grep "$utente" /etc/passwd >/dev/null && echo ok
```

Es.2: Si scriva una pipeline che stampi a video l’elenco (senza ripetizioni e ordinato lessicograficamente al contrario) dei nomi di login degli utenti di sistema.

```
cat /etc/passwd | cut -d: -f1 | sort -r | uniq
```

Il linguaggio C

Il C è un **linguaggio imperativo** legato a Unix (le chiamate di sistema sono definite come funzioni C infatti qualsiasi linguaggio passa per il C per interfacciarsi con il sistema operativo), adatto all'implementazione di compilatori e sistemi operativi. È stato progettato da D. Ritchie per il PDP-11 (all'inizio degli anni '70). Il C è un linguaggio:

- **Tipato** e **compilato**;
- Ad **alto livello**, ma non "troppo" in quanto fornisce le primitive per manipolare numeri, caratteri ed indirizzi, ma non oggetti composti come liste, stringhe, vettori ecc.
- **Piccolo** perché non fornisce direttamente delle primitive di input/output. Per effettuare queste operazioni si deve ricorrere alla Libreria Standard.

Compilazione

In ambiente Linux e macOS esistono due compilatori C open source e liberamente disponibili: **GCC** (GNU C Compiler) e **Clang**. Per compilare un programma, dopo averlo salvato in un file, ad esempio `programma.c`, si invoca il compilatore:

```
> gcc programma.c -o program
```

Con l'opzione `-o` si può rinominare il file eseguibile.

Ogni file C, prima della compilazione, viene **preprocessato**. Il preprocessore trasforma il codice interpretando delle **direttive** dove le direttive sono righe di codice che cominciano con un cancelletto # (a.e. #include, #define…), seguito dal nome della direttiva vera e propria. Il compilatore vede solo il risultato del preprocessing.

Espressioni booleane

La prima differenza importante rispetto al C++ e al Java è che in C **non esiste il tipo bool**. Infatti il costrutto if e i cicli for e while, si aspettano delle espressioni di tipo **intero** nelle proprie condizioni di controllo:

- Il **valore zero** viene interpretato come **falso**;
- Qualsiasi **valore diverso da zero** viene interpretato come **vero**.

Standard input

argv (argument vector) e **argc** (argument count, ovvero il numero di stringhe puntate da argv) sono il modo in cui gli argomenti della riga di comando vengono passati a main() in C e C++ dallo **standard input**.

Notare che argc sarà 1 più il numero di argomenti, poiché praticamente tutte le implementazioni **anteporranno** il nome del programma all'array.

Possono anche essere omessi del tutto, producendo int main(), se non si intende elaborare argomenti della riga di comando.Esempio:

```c
#include <stdio.h>

void main (int argc, char **argv) {
  printf("Input 1: %s\n", argv[1]);
  printf("Input 2: %s\n", argv[2]);
  printf("Input 4: %s\n", argv[4]);
}
```

Sul terminale:

```
> gcc prova.c -o prova && ./prova casa albero nave fiore
Input 1: casa
Input 2: albero
Input 3: fiore
```

Programmazione di sistema

Le **system call** si comportano come delle funzioni. Ogni system call ha un **prototipo**. Convenzionalmente le system call restituiscono un valore negativo (tipicamente -1) per indicare che è avvenuto un **errore**.

Esistono vari tipi di system call:

- controllo di processi ()
- gestione di file
- comunicazione tra processi
- segnali.

fork()

La chiamata **crea una copia** del processo chiamante. Il **processo figlio** prosegue l'esecuzione in modo totalmente **indipendente e isolato**. Il valore restituito da fork() serve a distinguere tra processo genitore e processo figlio: Al genitore viene restituito il PID del figlio; Al figlio viene restituito 0.

```c
int main() {
   printf("Inizio del programma\n");
   pid_t pid=fork();
   if(pid == 0)
       printf("Sono il processo figlio (PID: %d).\n", (int)getpid());
   else if(pid > 0)
       printf("Sono il genitore del processo con PID %d.\n",(int)getpid());
   return 0;
}
```

Output:

```
Inizio del programma
Sono il genitore del processo con PID 37874.
Inizio del programma
Sono il processo figlio (PID: 37875).
```

Il programma inizialmente stampa la prima stringa, poi, dopo aver eseguito la fork, crea un nuovo processo figlio e termina l'esecuzione del padre stampandone il suo PID. Successivamente il programma ricomincia da capo la sua esecuzione per il processo figlio.

exec() e execl()

La chiamata **elimina** il programma originale **sovrascrivendolo** con quello passato come parametro. Quindi le istruzioni che seguono una chiamata a execl() verranno eseguite soltanto in caso si verifichi un errore durante l'esecuzione di quest'ultima ed il controllo ritorni al chiamante.

```
int execl(const char *path, const char *arg0, ...);
```

Dove:

- L'argomento **path** è l'eseguibile che si vuole lanciare;
- Gli **argomenti successivi** sono gli argomenti da riga di comando che si vogliono passare al programma, terminati da un puntatore nullo.

Allocazione dinamica della memoria

Per scrivere programmi che trattino array (quindi stringhe), bisogna a priori dichiarare la dimensione **staticamente.**

- La funzione **malloc()** alloca una zona di memoria **contigua** della dimensione richiesta e specificata come argomento e **restituisce un puntatore** (void *) all'inizio di tale zona;
- La funzione **free()** serve a **liberare la memoria** allocata dinamicamente con malloc().

Esempio di uso:

```
int main() {

   int n = 0;
   printf("Quanti numeri verranno inseriti? ");
   scanf("%d", &n);

   int *elementi = malloc(n*sizeof(int));

   printf("Inserire i numeri: ");
```

```
    for(int i = 0; i < n; ++i)
        scanf("%d", elementi + i);

    printf("La somma dei numeri inseriti e': %d\n", somma(elementi, n));
    free(elementi);
    return 0;
}
```

Tipi di dato

I tipi **base** del linguaggio C sono:

- char
- short int, int, long int unsigned int,
- float
- double

Stringhe

Seguendo la sua impostazione a basso livello, il C **non** fornisce un tipo di dato primitivo per rappresentare stringhe di caratteri. Sono **sequenze di caratteri** di cui l'ultimo, sempre presente in modo implicito, è '**\0**'. Esempio:

```
char text = {'c', 'i', 'a', 'o', '\0'}
```

Funzioni su stringhe

La libreria standard contiene molte funzioni utili a manipolare stringhe, fornite dall'header **<string.h>**:

- int **strlen**(char *s): restituisce la lunghezza di una stringa;
- int **strncmp**(char *s1, char *s2, unsigned len): se s1 = s2: restituisce 0, -1 o 1 altrimenti;
- char ***strncpy**(char *dest, char source, unsigned len): copia al max len char da source a dest;
- char ***strncat**(char *dest, char *source, unsigned len): concatena al max len char da source in fondo a dest;

Puntatori

Un puntatore è una **variabile** che **contiene l'indirizzo di memoria** di un'altra variabile. L'uso dei puntatori gira attorno a due **operatori unari:**

- **Deferenziazione (*)**: restituisce il **contenuto** dell'oggetto puntato;
- **Address of (&)**: restituisce l'**indirizzo** della variabile;

```
#include <stdio.h>
int main() {
```

```
    int x = 0;

    printf("Inserisci un intero: ");
    scanf("%d", &x); // salva nell'indirizzo dov e' contenuta 'x' il valore
immesso da
                        tastiera

    printf("Il doppio di e' %d", x*2);

    return 0;
}
```

Passaggio dei parametri a funzioni

In C tutti gli argomenti delle funzioni sono **passati per valore**, ovvero le funzioni lavorano su **copie dei parametri** e **non modificano** i parametri passati dal chiamante. Il passaggio per riferimento, presente in altri linguaggi, non esiste in C. Si può **simulare per mezzo dei puntatori**, ovvero passando l'indirizzo in memoria dell'argomento:

- Quando si **passa una variabile**, il suo valore viene copiato, per cui le **modifiche** fatte all'interno della procedura **non** hanno effetto sulle variabili del programma principale;
- Quando si **passa un indirizzo**, la funzione **è in grado di modificare** il contenuto della zona di memoria che parte da quell'indirizzo;

```
#include <stdio.h>

void fake_swap(int x, int y) {
    int z = x;
    x = y;
    y = z;
}
void swap(int *x, int *y) {
    int z = *x;
    *x = *y;
    *y = z;
}
int main() {
    int x = 42, y = 0;

    fake_swap(x, y);  // Stampa x = 42, y = 0
    printf("x = %d, y = %d\n", x, y);

    swap(&x, &y);     // Stampa x = 0, y = 42
    printf("x = %d, y = %d\n", x, y);

    return 0;
}
```

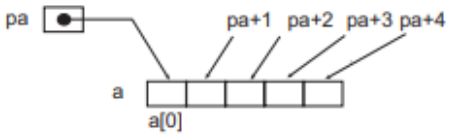
Puntatori e array

I puntatori sono lo strumento principale per la **manipolazione** degli array in C.

```
int main() {
   int a[5] = {0, 1, 2, 3, 4}; // inizializza array
   int *pa = &a[0]; // crea un puntatore ad un valore intero che punta
all'indirizzo di
                        memoria del primo elemento
   printf("%d\n", *pa); // stampa zero
}
```

Incrementando il puntatore è possibile ottenere puntatori agli **elementi successivi**:

```
printf("%d\n", *(pa + 1)); // stampa 1, equivale a a[1]
```



Array di puntatori

Essendo i puntatori delle variabili, possono essere a loro volta memorizzati in array. Ad esempio il seguente codice dichiara un array di 42 elementi, ognuno dei quali è un char *:

```
char *line[42] = { };
```

Passaggio di array a funzioni

I puntatori sono l'**unico modo** per passare array alle funzioni: per scrivere una funzione che operi su un array di input è necessario passare:

- Un **puntatore al primo elemento**;
- Un numero intero che indichi il **numero di elementi** dell'array.

FILE

Quando si lavora con i file, è necessario dichiarare un **puntatore** di tipo FILE. Questa dichiarazione è necessaria per la comunicazione tra il file e il programma. L'apertura di un file viene eseguita utilizzando la funzione **fopen("nomefile", "mode")** definita nel file di intestazione <stdio.h.> Per chiudere un file si usa invece **fclose(nomefile)**.

Esempio di programma che percepisce degli interi da un file formattato come segue, li inserisce in un array e successivamente li stampa:

```c
#include <stdio.h>
#define SIZE 150

/* file.txt contiene:
                  56,89
                  120,12
                  -1,555
                  21,67
*/

int main() {

   FILE *textfile;                        // Creazione di un puntatore ad un
file
   textfile=fopen("file.txt", "r");       // Apertura file
   int numberArray[SIZE];
   char ch;

   if (textfile == NULL) {                // File non apribile
       printf("File can't be opened\n");
       return 2;
   }

   for (int i=0; i<12; i++){
       fscanf(textfile, "%d,'\n'", &numberArray[i]); // rileva i campi
delimitati da ',' o
                                                     '\n' e li salva in
un arraay
       printf("%d,", numberArray[i]);                // stampa
"56,89,120,12,-1,555,21,67”
   }

   fclose(textfile);  // Chiude il file
   return 0;
}
```

Strutture

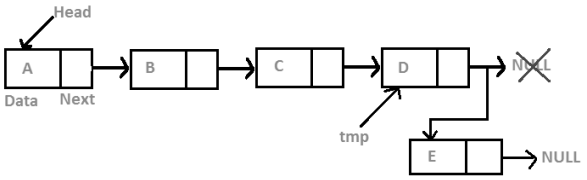
Le strutture sono un tipo **aggregato**, che raggruppa variabili di **tipo diverso** in un’unica entità. La dichiarazione di una struttura definisce un **tipo di dato** e viene fatta:

```c
struct point {
   float x;
   float y;
};
```

Liste concatenate

Una **lista concatenata** viene descritta come una **catena di nodi**, ognuno dei quali **punta al successivo** nella catena. I nodi della lista andranno aggiunti e rimossi in modo non prevedibile, quindi vanno **allocati dinamicamente**. Ogni **nodo** e' composto da:

- Un **campo data**: contiene il dato effettivo;
- Un **puntatore next**: contiene l'indirizzo al prossimo nodo della lista;




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

struct list {
   int data;
   struct list *next;
};

int main() {
   struct list *head = NULL;            // Crea un puntatore al primo elemento
   head = malloc(sizeof(struct list)); // Alloca memoria per il nodo

   head->data = 45;                     // Il primo nodo contiene il valore
45...
   head->next = NULL;                   // ... e punta a NULL

   printf("%d", head->data);
   return 0;
}
```

Nota che l'operatore **->** si può utilizzare soltanto con i puntatori ad una struttura.

Per **aggiungere dei nodi** bisogna:

- **Allocare memoria:** `head->next=malloc(sizeof(struct list));`
- **Inserire data**: `head->next->data = 2;`

Esempio di programma che crea ed aggiunge 3 nodi, inizializzandoli a 1, 2 e 3:

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

typedef struct list {
  int data;
  struct list *next;
}Node;

/* Print all the elements in the linked list */
void print(Node *head) {
   Node *current_node = head;
   while (current_node != NULL) {
       printf("%d ", current_node->data);
       current_node = current_node->next;
   }
   printf("\n");
}

int main() {
   struct list *head;

   head = malloc(sizeof(struct list));              // alloca per il nuovo nodo
   head->data = 1;                                  // primo nodo 1
   print(head);                                     // stampa 1

   head->next = malloc(sizeof(struct list));        // alloca per il nuovo nodo
   head->next->data = 2;                            // secondo nodo 2
   print(head);                                     // stampa 1 2

   head->next->next = malloc(sizeof(struct list));  // alloca per il nuovo nodo
   head->next->next->data = 3;                      // terzo nodo 3
   print(head);                                     // stampa 1 2 3

   head->next->next->next = NULL;                   // ultimo nodo punta NULL

   return 0;
}
```

Funzioni per liste concatenate:

```
//Trovare la lunghezza di una lista::
int length(struct node *head) {
   int len = 0;
   for(struct node *n = head; n; n = n->next)
       ++len;
   return len;
}

//Ricerca di un elemento in una lista:
struct node *find(struct node *head, int data) {
   for(struct node *n = head; n; n = n->next) {
       if(n->data == data)
           return n;
   }
   return NULL;
}

//Concatenazione di due liste:
struct node *last(struct node *head) {
   for(struct node *n = head; n; n = n->next) {
       if(n->next == NULL)
           return n;
   }
   return NULL;
}
struct node *append(struct node *head1, struct node *head2) {
   struct node *last1 = last(head1);
   last1->next = head2;
   return head1;
}
```

Alberi binari

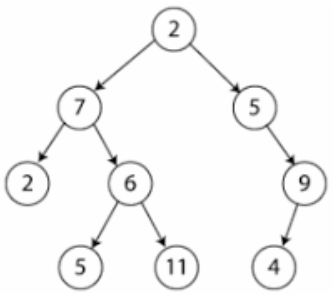
Un albero consiste di un **insieme di nodi** e un **insieme di archi orientati** che connettono coppie di nodi, con le seguenti proprietà:

- Un nodo dell'albero è designato come **nodo radice**;
- Ogni **nodo** n, a parte la radice, ha **esattamente un arco entrante**;
- Esiste un **cammino unico** dalla radice ad ogni nodo;
- L'albero **è connesso**.

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

struct tree {
   int data;
   struct tree *right;
   struct tree *left;
};

int main() {
   struct tree *head = NULL;            // Crea un puntatore al primo
elemento
   head = malloc(sizeof(struct tree));  // Alloca memoria per il nodo

   head->data = 2;                      // Il primo nodo contiene il valore
2...
   head->right = NULL;                  // ... non ha figli destri
   head->left = NULL;                   // ... non ha figli sinistri

   printf("%d", head->data);
   return 0;
}
```

Ogni **nodo** e' composto da:

- Un **campo data**: contiene il dato effettivo;
- Un **puntatore a destra**;
- Un **puntatore a sinistra**;



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

struct node {
   int data;
   struct node *left;
   struct node *right;
};

/* newNode() allocates a new node with the given data and NULL left and right
pointers. */
struct node *newNode(int data) {
   struct node *node = malloc(sizeof(struct node));  // Allocate memory for
new node
   node->data = data;  // Assign data to this node
   node->left = NULL;  // Initialize left and...
   node->right = NULL; // ...right children as NULL
   return (node);
}

int main() {
   /*create root*/
   struct node* root = newNode(1);
/*
        1
       / \
      N   N
*/

   root->left = newNode(2);
   root->right = newNode(3);
/* 2 and 3 become left and right children of 1
          1
         / \
        2   3
       / \ / \
      N   N N   N
*/

   root->left->left = newNode(4);
/* 4 becomes left child of 2
          1
         / \
        2   3
       / \ / \
      4   N N   N
     / \
    N   N
*/
```

```
    getchar();
    return 0;
}
```