

# La Struttura del Calcolatore e i Linguaggi di Programmazione

---

## Sommario

- Definizione di Informatica
- Distinzione tra Algoritmo e Programma
- Il Compilatore
- Il Sistema Operativo
- Il Processore ed il Codice Macchina

## Cosa NON è l'Informatica

---

NON è soltanto la **scienza** e la **tecnologia** dei calcolatori. Il calcolatore è **solo uno strumento**.

Computer Science is no more about computers than astronomy is about telescopes

E. W. Dijkstra

## Cosa È l'Informatica

---

È la scienza che si occupa della **rappresentazione**, dell'**organizzazione** e del **trattamento automatico** dell'**informazione** per la risoluzione di **problemi**.

L'informatica permette di risolvere problemi **velocemente** e **automaticamente**

1. I calcolatori eseguono le operazioni più **velocemente** degli esseri umani, ma sanno eseguire solo operazioni descritte in maniera **precisa e formale**
2. I calcolatori **non sono in grado** di **progettare** procedure per risolvere problemi, a differenza degli esseri umani

## L'Informatica e il Calcolatore

Il calcolatore è una macchina che **memorizza, elabora** e **distribuisce** l'informazione. Il calcolatore esegue le **istruzioni** che gli vengono impartite al fine di risolvere i problemi.

## L'Informatica e il Problem Solving

Viene fornito un problema per il quale è necessaria una soluzione:

- **Algoritmo:** si definisce una **procedura** che permette di risolvere il problema
- **Programma:** si **trasforma** la procedura in una **sequenza di operazioni** che possono essere eseguite dal calcolatore
- **Istanza:** si desidera **risolvere** il problema su uno specifico insieme di dati
- **Esecuzione:** si rappresentano i dati nel calcolatore e si fa eseguire ad esso la **sequenza di operazioni** che permette di risolvere il problema

# Definire cos'è l'Informatica

Per definire cosa è l'Informatica è necessario definire le seguenti nozioni:

- **Informazione:** notizia, dato o elemento che consente di avere conoscenza più o meno esatta di fatti, situazioni, modi di essere
- **Rappresentazione:** è una funzione che associa ad ogni elemento una sequenza unica di simboli
- **Elaborazione:** è una trasformazione  $y = f(x)$  costituita da una o più azioni elaborate (o passi di elaborazione), dove
  - $x$  è l'insieme di dati iniziali (o di ingresso)
  - $y$  è l'insieme dei dati finali (o di uscita)
  - $f$  è una regola che fa corrispondere  $y$  ad  $x$
- **Algoritmo:** è una sequenza finita di azioni elaborate che portano alla realizzazione di un compito. Esso deve essere:
  - Comprensibile
  - Corretto
  - Efficiente

## Algoritmo vs Programma

Un **programma** è la traduzione di un algoritmo in un linguaggio comprensibile dal calcolatore (**linguaggio di programmazione**).

Il linguaggio naturale è complesso ed ambiguo, mentre il calcolatore "parla" linguaggi **precisi, non ambigui** ed **estremamente semplici** in termini di:

- **Sintassi** (regole grammaticali)
- **Semantica** (significato)

## Programmazione

---

La **programmazione** è la stesura di una sequenza di istruzioni da far eseguire al calcolatore per risolvere un problema.

## Dal Sorgente all'Eseguibile

Un **compilatore** è un programma informatico che traduce una serie di istruzioni scritte in un determinato linguaggio di programmazione (**codice sorgente**) in istruzioni di un altro linguaggio (**codice oggetto**)

I compilatori attuali dividono l'operazione di compilazione in due stadi principali, il **front end** e il **back end**:

- **Front End:** il compilatore traduce il sorgente in un linguaggio intermedio (*di solito interno al compilatore*)
- **Back End:** avviene la generazione del codice oggetto

## Analisi Lessicale

Attraverso un analizzatore lessicale (**scanner** o **lexer**), il compilatore divide il codice sorgente in tanti pezzetti chiamati **token**.

I **token** sono gli elementi minimi (*non ulteriormente divisibili*) di un linguaggio, ad esempio parole chiave (**for**, **while**), nomi di variabili, operatori.

## Analisi Sintattica

L'analisi sintattica prende in ingresso la sequenza di token generata nella fase precedente ed esegue il controllo sintattico.

## Analisi Semantica

L'analisi semantica si occupa di controllare il significato delle istruzioni presenti nel codice in ingresso.

## Codice Intermedio

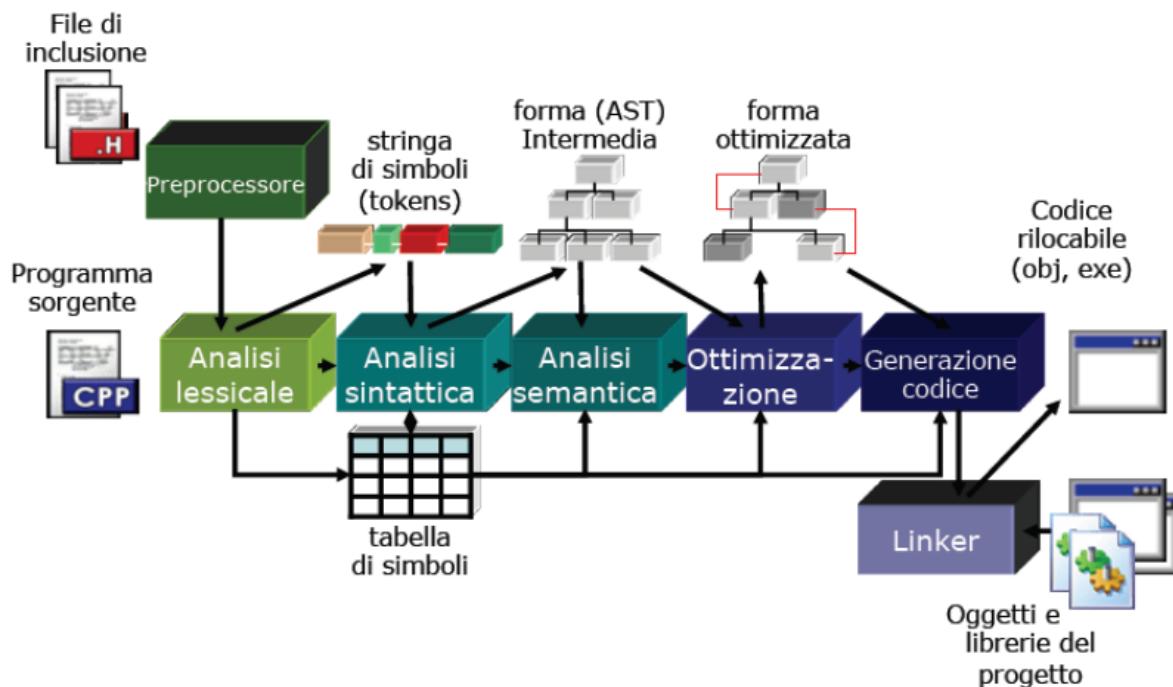
Dall'albero di sintassi viene generato il codice intermedio.

## Ottimizzazione

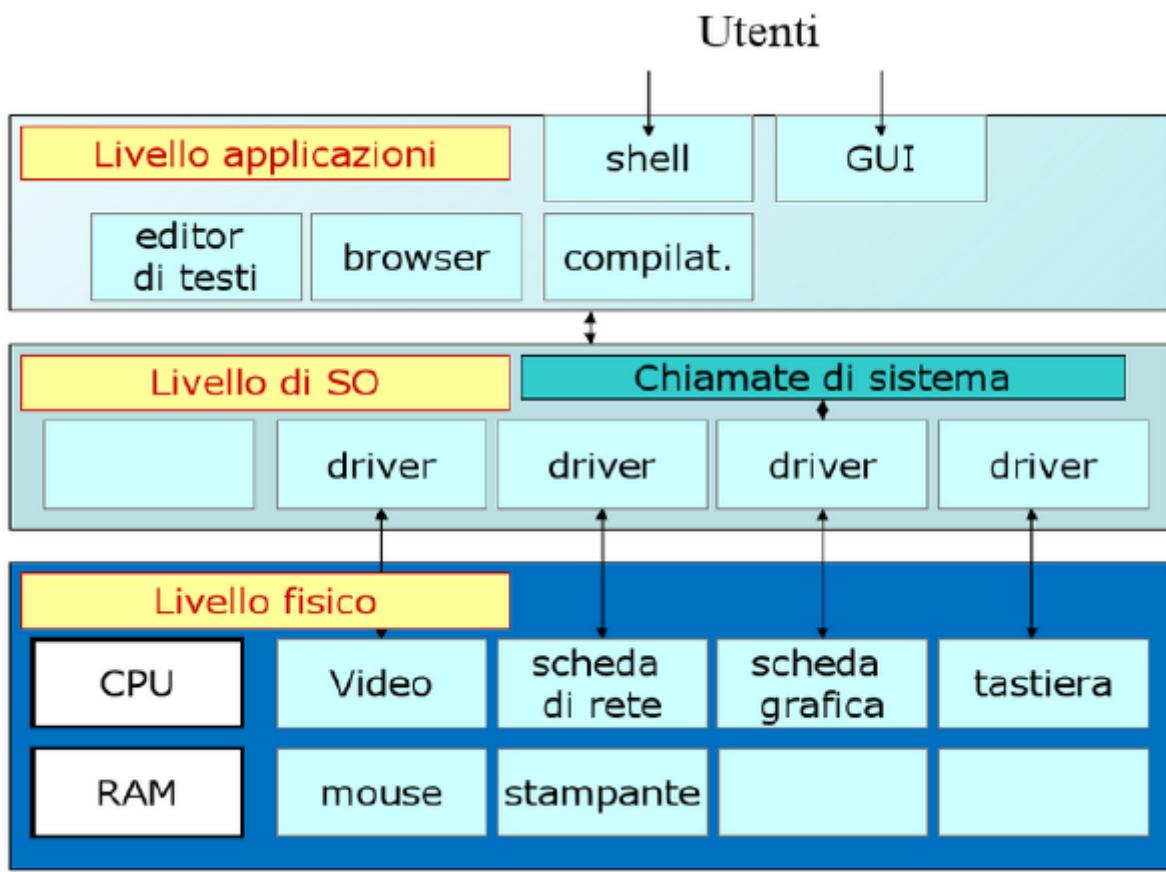
Il compilatore ottimizza il codice intermedio.

## Codice Macchina

In questa fase viene generato il codice nella forma del linguaggio target. Spesso il linguaggio target è un linguaggio macchina.



## Dal Programma al Sistema Operativo



Un **sistema di calcolo** è costituito da *quattro livelli principali*:

- **Livello utente**
- **Livello Applicazione**
- **Livello SO**
- **Livello Fisico**

Un sistema operativo rappresenta un *insieme di programmi (software)*, che gestisce gli elementi fisici di un calcolatore (**hardware**).

## Il Computer come Esecutore

"**Esecutore**": qualsiasi entità **E**(umana e non) in grado di:

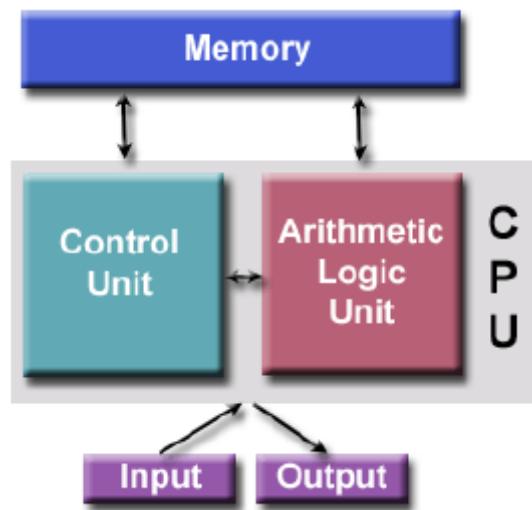
- Riconoscere un insieme finito **S** di istruzioni (**linguaggio**) scritte con l'uso di simboli di un **alfabeto (C)**
- Interpretare ogni istruzione associando a essa una ben definita, univoca e finita azione di un insieme finito di azioni (**A**)

# Rappresentare l'Informazione

## Sommario

- Il bit
- Il Sistema Binario
- Rappresentazione di Numeri Interi
- Rappresentazione di Numeri Razionali
- Il Codice ASCII

## Il Modello di Von Neumann



Gli attuali computer *general-purpose* sono di tipo **stored program** (o *a programma memorizzato*), e rispecchiano il modello generale della macchina di **Von Neumann**

John Von Neumann è stato un **matematico, fisico e letterato** ungherese naturalizzato statunitense

Dal 1948, anno in cui *il primo programma girava su un computer all'Università di Manchester*, la tecnologia ha conosciuto un'incessante evoluzione:

- Valvole
- Transistor e Circuiti Integrati (IC)
- Very Large Scale Integrated (VLSI) circuits

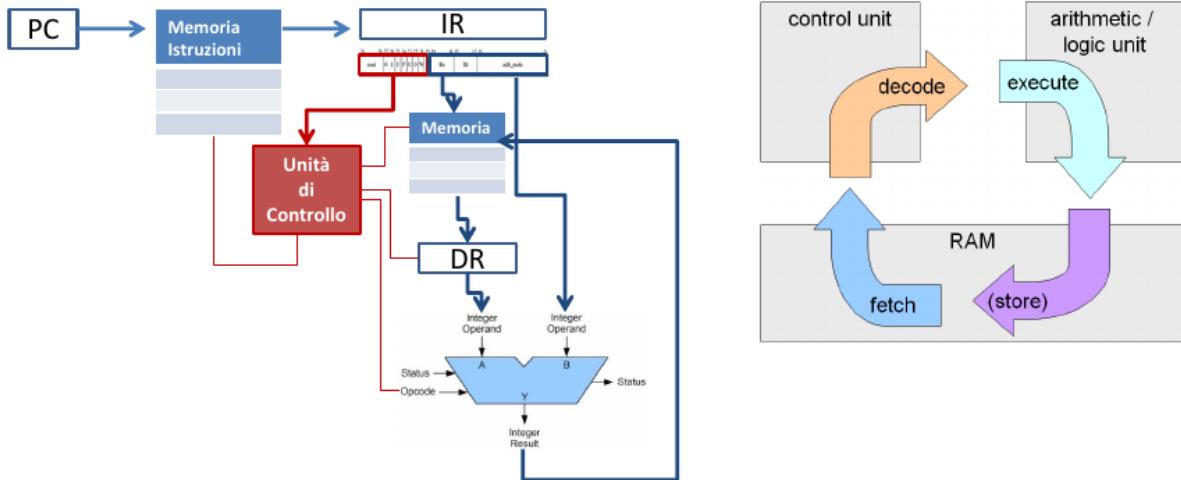
## Il Primo Calcolatore

Le **Schede Perforate** (1890) vengono introdotte da Herman Hollerit per *automatizzare la tabulazione dei dati di un censimento*.

L'**Electronic Numerical Integrator and Calculator (ENIAC)** (1946), è considerato *il primo calcolatore a valvole general-purpose programmabile* progettato da **Mauchly & Eckert** (Università della Pennsylvania). Era programmato tramite inserimento di cavi e azionamento di interruttori.

# Il Processo di Esecuzione delle Istruzioni

Un'istruzione ha un ciclo di vita che consta di **quattro fasi principali**



## II Transistor

I **computer** sono una complessa sintesi di apparecchiature che operano a velocità molto elevate.

Un moderno microprocessore è costituito da diversi milioni di transistor, ciascuno dei quali può cambiare il suo stato centinaia di milioni di volte al secondo.

II Bit

La tensione in uscita di un transistor può assumere due stati: tensione alta (1) e tensione bassa (0).

La cifra binaria è detta **bit**, e, in un numero binario, i bit estremi prendono nomi particolari:

- **MSB (Most Significant Bit)**, ovvero il bit all'**estremo sinistro** del numero binario
  - **LSB (Least Significant Bit)**, ovvero il bit all'**estremo destro** del numero binario

A binary number **1011100100** is shown. The first bit, **1**, is labeled **MSB** (*Most Significant Bit*) with a red arrow pointing to it. The last bit, **0**, is labeled **LSB** (*Least Significant Bit*) with a red arrow pointing to it.

# I Sistemi Numerali Posizionali

Nei sistemi di numerazione posizionale il numero è scritto specificando le cifre in ordine ed il suo valore dipende dalla **posizione relativa** delle cifre

- **Esempio:** sistema decimale (Base 10)

Cifre : 0 1 2 3 4 5 6 7 8 9

$$5641 = 5 \cdot 10^3 + 6 \cdot 10^2 + 4 \cdot 10^1 + 1 \cdot 10^0$$

↑↑↑↑

Posizione: 3 2 1 0

I sistemi di numerazione posizionale sono composti da:

- La **Base** del sistema di numerazione
  - Le **Cifre** del sistema di numerazione

## Sistemi a Base B

La base definisce il numero di cifre diverse nel sistema di numerazione.

La cifra di minor valore è **sempre lo 0**, mentre le altre sono, nell'ordine, **1, 2...B - 1**. Se **B > 10** occorre introdurre **B - 10** simboli in aggiunta alle cifre decimali

Un **numero intero N** si rappresenta come:

$$N = c_n B^n + c_{n-1} B^{n-1} + \dots + c_2 B^2 + c_1 B^1 + c_0 B^0$$

dove **Cn** è la **cifra più significativa** e **C0** è la **cifra meno significativa**

Un numero frazionario N' si rappresenta come

$$N' = c_1 B^{-1} + c_2 B^{-2} + \dots + c_n B^{-n}$$

Con  $n$  cifre in base **B** si rappresentano tutti gli interi positivi da **0 a  $B^n - 1$** , ovvero  **$B^n$**  numeri distinti.

## Il Sistema Binario (**B = 2**)

La base 2 è la più piccola per un sistema di numerazione ed è composta dalle sole cifre **0 e 1**

- **Esempi:**

$$(101101)_2 = 1 \cdot 2^5 + 0 \cdot 2^4 + 1 \cdot 2^3 + 1 \cdot 2^2 + 0 \cdot 2^1 + 1 \cdot 2^0 =$$

$32 + 0 + 8 + 4 + 0 + 1 = (45)_{10}$

← Forma polinomia

$$(0,0101)_2 = 0 \cdot 2^{-1} + 1 \cdot 2^{-2} + 0 \cdot 2^{-3} + 1 \cdot 2^{-4} =$$
$$0 + 0,25 + 0 + 0,0625 = (0,3125)_{10}$$
$$(11,101)_2 = 1 \cdot 2^1 + 1 \cdot 2^0 + 1 \cdot 2^{-1} + 0 \cdot 2^{-2} + 1 \cdot 2^{-3} =$$
$$2 + 1 + 0,5 + 0 + 0,125 = (3,625)_{10}$$

## Dal Bit al Byte

Un **byte** è un *insieme di 8 bit* (numero binario a 8 cifre). Con un byte si rappresentano i numeri interi interi fra **0 e  $2^8 - 1$** , ovvero l'intervallo **[0, 255]**.

00000000

00000001

00000010

00000011

.....

$2^8 = 256$  valori distinti

11111110

11111111

Il byte è l'elemento base con cui si rappresentano i dati nei calcolatori e se usano sempre dimensioni multiple: **2 byte** (16 bit), **4 byte** (32 bit), **8 byte** (64 bit)...

## Dal Byte al KiloByte

Potenze di 2

$$2^4 = 16$$

$$2^8 = 256$$

$$2^{16} = 65536$$

$$2^{10} = 1024 \quad (\text{K=Kilo})$$

$$2^{20} = 1048576 \quad (\text{M=Mega})$$

$$2^{30} = 1073741824 \quad (\text{G=Giga})$$

Qual è la relazione tra **KB** (*Kilobyte*), **MB** (*Megabyte*), **GB** (*Gigabyte*)...?

$$1 \text{ KB} = 2^{10} \text{ byte} = 1024 \text{ byte}$$

$$1 \text{ MB} = 2^{20} \text{ byte} = 1048576 \text{ byte}$$

$$1 \text{ GB} = 2^{30} \text{ byte} = 1073741824 \text{ byte}$$

$$1 \text{ TB} = 2^{40} \text{ byte} = 1099511627776 \text{ byte (Terabyte)}$$

## Da Decimale a Binario (Numeri Interi)

Si divide ripetutamente il numero intero decimale per 2 fino ad ottenere quoziente nullo:

- Le **cifre del numero binario** sono i *resti delle divisioni*
- La **cifra più significativa** è l'*ultimo resto*

**Esempio:** convertire in binario 43

$$\begin{array}{r} 43 : 2 = 21 + 1 \\ 21 : 2 = 10 + 1 \\ 10 : 2 = 5 + 0 \\ 5 : 2 = 2 + 1 \\ 2 : 2 = 1 + 0 \\ 1 : 2 = 0 + 1 \end{array}$$

bit più significativo

A destra della parola "resti" c'è un'arrow pointing to the first column of remainders (1, 1, 0, 1, 0, 1). A sinistra della parola "bit più significativo" c'è un'arrow pointing to the last remainder (1).

$$(43)_{10} = (101011)_2$$

## Da Decimale a Binario (Numeri Razionali)

Si moltiplica ripetutamente il numero frazionario decimale per 2 *fino ad ottenere una parte decimale nulla*, o, dato che la condizione potrebbe non verificarsi mai, *per un numero prefissato di volte*:

- Le **cifre del numero binario** sono le *parti intere dei prodotti successivi*
- La **cifra più significativa** è il *risultato della prima moltiplicazione*

**Esempio:** convertire in binario 0,21875 e 0,45

$$0,21875 \times 2 = 0,4375$$

$$0,4375 \times 2 = 0,875$$

$$0,875 \times 2 = 1,75$$

$$0,75 \times 2 = 1,5$$

$$0,5 \times 2 = 1,0$$



$$(0,21875)_{10} = (0,00111)_2$$

$$0,45 \times 2 = 0,9$$

$$0,90 \times 2 = 1,8$$

$$0,80 \times 2 = 1,6$$

$$0,60 \times 2 = 1,2$$

$$0,20 \times 2 = 0,4 \text{ etc.}$$



$$(0,45)_{10} \approx (0,01110)_2$$

## Da Binario a Decimale

Oltre all'espansione esplicita in **potenze di 2** (*forma polinomia*):

$$101011 = 1 * 2^5 + 0 * 2^4 + 1 * 2^3 + 0 * 2^2 + 1 * 2^1 + 1 * 2^0 = 43$$

...si può anche operare nel modo seguente:

- Si raddoppia il bit più significativo e si aggiunge al secondo bit
- Si raddoppia la somma e si aggiunge al terzo bit
- Si continua così fino al bit meno significativo

**Esempio:** convertire 101011 in decimale

bit più significativo

$$\begin{array}{r}
 \boxed{1} \times 2 = 2 \quad + 0 \\
 2 \times 2 = 4 \quad + 1 \\
 5 \times 2 = 10 \quad + 0 \\
 10 \times 2 = 20 \quad + 1 \\
 21 \times 2 = 42 \quad + 1 = 43
 \end{array}$$

$$(101011)_2 = (43)_{10}$$

## Sistema Esadecimale

La base 16 è molto usata in campo informatico

- **Cifre:** 0 1 2 3 4 5 6 7 8 9 A B C D E F

**Esempio:**

$$\begin{aligned}
 (3A2F)_{16} &= 3 \times 16^3 + 10 \times 16^2 + 2 \times 16^1 + 15 \times 16^0 = \\
 &3 \times 4096 + 10 \times 256 + 2 \times 16 + 15 = (14895)_{10}
 \end{aligned}$$

## Dai Bit all'Hex

Un numero binario di **4n** bit corrisponde ad un numero esadecimale di **n** cifre

- **Esempio:** 32 bit corrispondono ad 8 cifre decimali

1101	1001	0001	1011	0100	0011	0111	1111
D	9	1	B	4	3	7	F

$(D91B437F)_{16}$

- **Esempio:** 16 bit corrispondono a 4 cifre esadecimali

0000	0000	1111	1111
0	0	F	F

$(00FF)_{16}$

## Da Esadecimale a Binario

La conversione da esadecimale a binario *si ottiene espandendo ciascuna cifra con i 4 bit corrispondenti.*

- **Esempio:** convertire in binario il numero esadecimale 0x0c8f (notazione usata in molti linguaggi di programmazione)

0	c	8	f
0000	1100	1000	1111

## Limiti della Rappresentazione

Nelle macchine numeriche un numero deve essere rappresentato in un particolare dispositivo elettronico interno chiamato **registro**, che è paragonabile ad una cella di memoria.

La caratteristica fondamentale del registro è la sua **dimensione** (*numero di bit*) stabilita in sede di progetto. Ciò significa che in un elaboratore potremo rappresentare solo una **quantità limitata** di numeri

- **Esempio:** Se il nostro registro è "lungo" 5 bit, allora potremmo rappresentare solo i numeri compresi tra **0** (00000) e **31** (11111)

## Rappresentazione Numeri con Segno

---

Per rappresentare numeri con segno occorre utilizzare un bit per definire il segno del numero, e si possono utilizzare tre tecniche di codifica:

- Modulo e Segno
- Complemento a 2
- Complemento a 1

## Modulo e Segno

Il bit più significativo rappresenta il segno: 0 per i numeri positivi e 1 per i numeri negativi

Esiste uno zero positivo (0000) e uno zero negativo (1000)

Se si utilizzano n bit si rappresentano i numeri nell'intervallo  **$+2^{n-1} - 1$**

- **Esempio:** con 4 bit si rappresentano i numeri compresi fra -7 ( $-(2^3 - 1)$ ) e +7 ( $2^3 - 1$ )

0000	+0	1000	-0
0001	+1	1001	-1
0010	+2	1010	-2
0011	+3	1011	-3
0100	+4	1100	-4
0101	+5	1101	-5
0110	+6	1110	-6
0111	+7	1111	-7
positivi		negativi	

## Complemento a 1

Considerando numeri binari di  $n$  bit, si definisce **complemento a uno** di un numero  $A$  la quantità:

$$2^n - 1 - A$$

Viene anche detto, semplicemente, **complemento**.

**Regola pratica:**

- Il complemento a uno di un numero binario  $A$  si ottiene cambiando il valore di tutti i suoi bit (**complementandoli**, appunto)

**Esempio:**

$$A = 1011 \rightarrow \overline{A} = 0100$$

## Complemento a 2

Il **complemento a 2** di un numero binario  $A$  a  $n$  cifre è:

$$2^n - (A)_2 = 1\underset{n}{\underbrace{0\dots0}} - (A)_2$$

Il complemento a 2 di un numero si calcola sommando 1 al suo complemento a 1

$$A = 1011 \rightarrow \overline{A} = 0100 \rightarrow \overline{\overline{A}} = \overline{A} + 1 = 0101$$

**Oppure**, si può, partendo da destra, lasciare invariate tutte le cifre fino al primo 1 (**compreso**) e invertendo il valore delle rimanenti:

$$A = 1\overset{\leftarrow}{0}1|1; \quad A = 010|1$$

## Interi in Complemento a 2

I **numeri positivi** sono rappresentati come in *Modulo e Segno*

I **numeri negativi** sono rappresentati in *complemento a 2* => la cifra più significativa ha sempre valore 1

Lo **zero** stavolta non è doppio, ma è **rappresentato solo come zero "positivo"** (*sequenza di soli zeri*)

Il campo dei numeri rappresentabili diventa  $[-2^{n-1}, 2^{n-1} - 1]$

Esempio:

0000	+0	1000	-8
0001	+1	1001	-7
0010	+2	1010	-6
0011	+3	1011	-5
0100	+4	1100	-4
0101	+5	1101	-3
0110	+6	1110	-2
0111	+7	1111	-1

Nota: 0111 +7  
1000 -8

## Operazioni Binarie

### Addizione Binaria

Le regole per l'addizione di due bit sono:

- $0 + 0 = 0$
- $0 + 1 = 1$
- $1 + 0 = 1$
- $1 + 1 = 0$  con riporto di 1

N.B.: in binario l'operazione  $1 + 1$  restituisce 1 0 come risultato

- Esempio:

1 11 1	riporti	91+ 90 — 181
01011011+	01011010	
10110101		

### Addizione Binaria in Complemento a 2

È utile perchè l'operazione di somma algebrica può essere realizzata non curandosi del bit di segno, questo perchè:

- In Complemento a 1
  - Zero ha due rappresentazioni: 00000000 e 11111111
  - La somma bitwise funziona quasi sempre

$$\begin{array}{r} 00110 \\ 10101 \\ \hline 11011 \end{array} \quad \begin{array}{l} (6) \\ (-10) \\ \hline (-4) \end{array}$$

$$\begin{array}{r} 11001 \\ 11010 \\ \hline 10011 \end{array} \quad \begin{array}{l} (-6) \\ (-5) \\ \hline (-12) \end{array}$$

- In Complemento a 2
  - Zero ha una sola rappresentazione
  - La somma bitwise funziona sempre

## Somma e Sottrazione in Complemento a 2

- La **somma** si effettua direttamente, senza badare ai segni degli operandi, come se fossero due normali numeri binari.
- La **sottrazione** si effettua sommando al minuendo il complemento a 2 del sottraendo

$$A - B \rightarrow A + (-B) \quad \text{ovvero: } A + \overline{\underline{B}}$$

Esempio:

$$\begin{array}{r} 01010 + \\ 10100 = \\ \hline 11110 \end{array} \quad \begin{array}{r} 10 + \\ -12 = \\ \hline -2 \end{array}$$

## L'Overflow

L'**overflow** si ha quando *il risultato di un'operazione non è rappresentabile correttamente con gli n bit a disposizione.*

- **Esempio:** 5 bit  $\rightarrow [-16, 15]$

$$\begin{array}{r} 14 + 01110 + \\ 10 \quad 01010 \\ \hline 24 \quad \boxed{11000} \end{array} \quad \begin{array}{r} -8 + 11000 + \\ -10 \quad 10110 \\ \hline -18 \quad \boxed{101110} \end{array}$$

→ -8 → +14

Per evitare l'overflow occorre aumentare il numero di bit utilizzati per rappresentare gli operandi

Per l'overflow possiamo individuare **due casi:**

1. Operandi con **segno discorde**

- Non si può mai verificare overflow

2. Operandi con **segno concorde**

- C'è overflow quando il risultato ha segno discorde da quello dei due operandi

In ogni caso, si trascura sempre il **carry** (*riporto*) oltre il **MSB**:

$$\begin{array}{r} 0101 + 1110 + \\ 0100 = 1101 = \\ \hline 1001 \quad \cancel{1101} = \end{array}$$

**overflow!**      **carry, risultato OK**

## Fixed Point

- Si usa un numero fisso di bit per la parte intera e la parte frazionaria (non si rappresenta la virgola)

- **Esempio:** 4 + 4 bit in binario puro

$$15.9375 = \textcolor{red}{11111111}$$

$$0.0625 = \textcolor{red}{00000001}$$

↑  
virgola sottintesa

**Vantaggi:**

- **Gli operandi sono allineati**, per cui le operazioni aritmetiche risultano facili ed immediate
- **La precisione assoluta è fissa**

**Savntaggi:**

- L'intervallo di valori rappresentati è **assai modesto**
- La precisione dei numeri frazionari rappresentati è **moltò scarsa**

**Utilizzo tipico:**

- **DSP (Digital Signal Processor)**
- Sistemi digitali per applicazioni specifiche (**special-purpose**)
- Numeri interi nei calcolatori

## Numeri Interi

- In tutte le macchine numeriche i **numeri interi** vengono rappresentati in questo codice, questo a causa dell'**estrema semplicità** che presentano le operazioni aritmetiche in **complemento a 2**
- Il numero di bit dipende dalla macchina, si tratta generalmente di **16 bit (interi corti)** o **32 bit (interi lunghi)**
- La rappresentazione è nota col nome di **fixed-point** e il punto frazionario è *supposto all'estrema destra della sequenza di bit* (ovvero **parte frazionaria nulla**)

## Numeri Reali

Le rappresentazioni considerate finora hanno il pregio di rappresentare esattamente i numeri (almeno quelli interi) ma richiedono un numero di bit esorbitante quando il numero da rappresentare ha un valore elevato

- La rappresentazione dei numeri frazionari che deriva dai codici precedenti, ovvero il **fixed-point**, a causa delle forti approssimazioni che impone, è **usata raramente**
- Generalmente si usa un apposito codice noto come **floating-point**, che consente di rappresentare in un numero limitato di bit grandezze di qualsiasi valore anche se condizionate da **approssimazioni più o meno elevate**

## Floating Point

È basata sul formato esponenziale (notazione scientifica)

$$N = \textcolor{red}{\text{mantissa}} \cdot \textcolor{blue}{\text{base}}^{\text{esponente}}$$

**Vantaggi:**

- **Grande intervallo** di valori rappresentabili

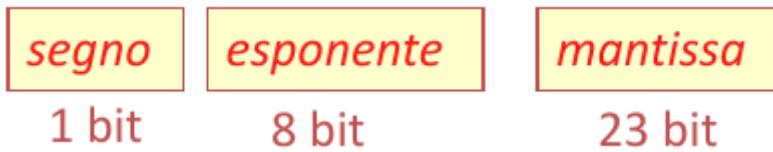
- Errore relativo fisso

Svantaggi:

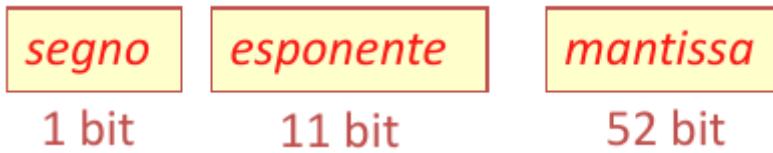
- Operandi non allineati, per cui le operazioni aritmetiche risultano molto complesse
- Errore assoluto variabile e dipendente dal valore del numero

Questa rappresentazione è standard (**IEEE-754**) ed è usata da tutti i calcolatori elettronici per rappresentare i numeri frazionari:

- Singola Precisione: circa 7 cifre decimali



- Doppia Precisione: circa 17 cifre decimali

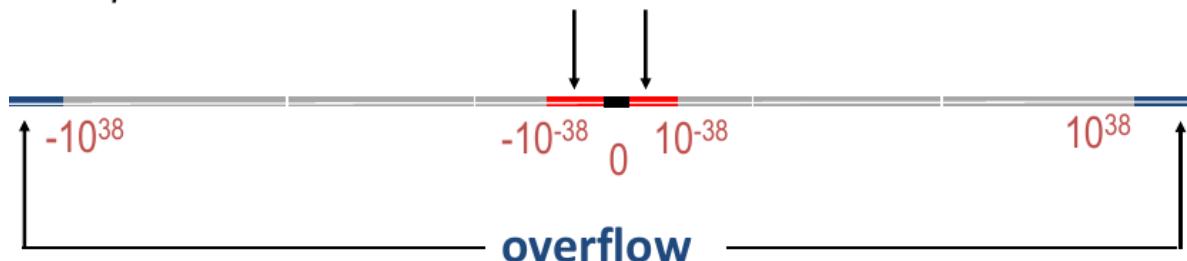


A causa della precisione variabile è possibile avere **errori di rappresentazione**:

- Overflow per numeri troppo grandi
- Underflow per numeri troppo piccoli

Esempio: IEEE P754

**underflow**



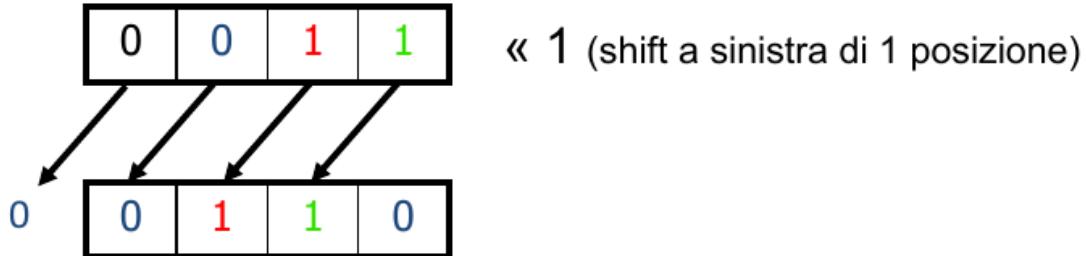
## L'Operazione di Shift

- Equivale ad una **moltiplicazione o divisione** per la base
- Consiste nel "far scorrere" i bit (a sinistra o a destra) inserendo opportuni valori nei posti lasciati liberi
- In decimale equivale a **moltiplicare** (*shift a sinistra*) o **dividere** (*shift a destra*) **per 10**
- In binario equivale a **moltiplicare** (*shift a sinistra*) o **dividere** (*shift a destra*) **per 2**

## Shift a Sinistra

- Si inserisce come **LSB** un bit a zero

- Equivale ad una **moltiplicazione per due**  
 $0011 \ll 1 = 0110 \quad (3 \times 2 = 6)$
- $0011 \ll 2 = 1100 \quad (3 \times 2^2 = 12)$
- $0011 \ll 3 = 11000 \quad (3 \times 2^3 = 24)$

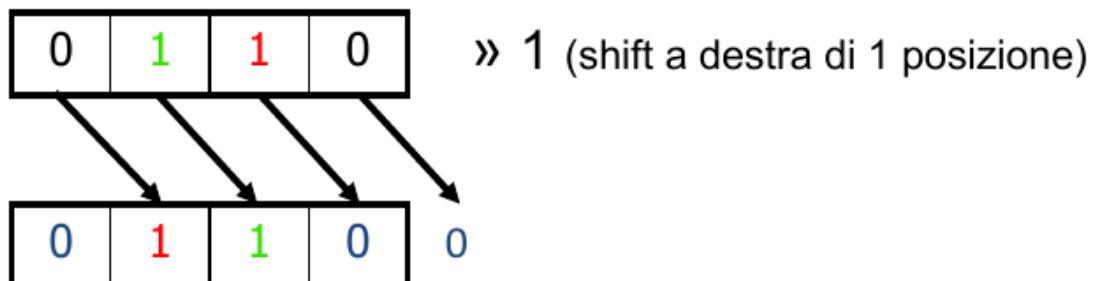


## Shift a Destra

- Si inserisce come **MSB** un bit a zero
- Equivale ad una **divisione per due**

$$0110 \gg 1 = 0011 \quad (6 : 2 = 3)$$

$$0110 \gg 2 = 0001 \quad (6 : 4 = 1) \text{ troncamento!}$$



## L'Aritmetica degli Elaboratori

L'aritmetica interna degli elaboratori differisce notevolmente dall'aritmetica classica.

Sebbene le stesse operazioni possano essere realizzate secondo modalità diverse su elaboratori diversi, si riscontrano alcune caratteristiche comuni:

- Rappresentazione binaria dei numeri
- Rango finito dei numeri rappresentabili
- Precisione finita dei numeri
- Operazioni espresse in termini di operazioni più semplici

## Rango Finito dei Numeri Rappresentabili

- Qualunque sia la codifica utilizzata, esistono sempre il **più grande** ed il **più piccolo numero rappresentabile**
- I **limiti inferiore e superiore** del range di rappresentazione dipendono sia dal **tipo di codifica** che dal **numero di bit utilizzati**

- Se il **risultato** di un'operazione **non appartiene al rango dei numeri rappresentabili** allora si è verificato un **overflow**
- Se il **risultato** di un'operazione **è più piccolo del più piccolo numero rappresentabile** allora si è verificato un **underflow**

## Precisione Finita dei Numeri

La precisione della rappresentazione di un numero frazionario è una misura di quanto essa corrisponda al numero che deve essere rappresentato.

Negli elaboratori, i numeri frazionari sono rappresentati in floating-point, utilizzando un numero finito di bit. Questo comporta che, quando un numero reale non ammette rappresentazione finita, dovrà essere codificato in maniera approssimata.

Negli elaboratori si rappresentano soltanto numeri razionali (fino ad una data precisione)

## Operazioni Espresse in Termini di Operazioni più Semplici

**La maggior parte degli elaboratori non possiede circuiti in grado di eseguire direttamente tutte le operazioni:**

- la **sottrazione** si realizza per mezzo di una **complementazione e di un'addizione**
- La **moltiplicazione** si realizza per mezzo di una **successione di addizioni e di shift**
- La **divisione** si realizza per mezzo di una **successione di shift e sottrazioni**

Quindi, mentre le **operazioni semplici** sono eseguite direttamente da appositi **circuiti hardware**, le **operazioni complesse** sono realizzate mediante esecuzione successiva di operazioni semplici.

Tutto questo è controllato dal **firmware**, ovvero un *programma opportunamente realizzato e memorizzato permanentemente*.

## Codifica dei Caratteri Alfabetici

---

Oltre ai numeri, molte applicazioni informatiche elaborano caratteri (**simboli**).

Negli elaboratori elettronici si utilizzano i numeri, i quali sono utilizzati anche per codificare caratteri e simboli, pertanto, *per poter scambiare dati in modo corretto, va definito uno standard di codifica*.

- **Esempio di codifica:**

A	—————→	01000001
3	—————→	00110011
\$	—————→	00100100

Per effettuare uno scambio di dati è necessario conoscere il tipo di codifica utilizzato.

**La codifica deve prevedere:**

- Le lettere dell'alfabeto
- Le cifre numeriche
- I simboli
- La punteggiatura
- I caratteri speciali per determinate lingue

Lo standard di codifica più diffuso è il codice **ASCII**, che sta per *American Standard Code for Information Interchange*.

## Codifica ASCII

Definisce una tabella di corrispondenza fra ciascun carattere e un codice a **7 bit** (*128 caratteri*).

I **caratteri**, in genere, sono rappresentati con **1 byte** (*8 bit*). I caratteri che hanno il **MSB a 1** (*codice compreso tra 128 e 255*) rappresentano un'**estensione della codifica**.

La tabella comprende sia **caratteri di controllo** (*codici da 0 a 31*) che **caratteri stampabili**.

I caratteri alfanumerici hanno codici ordinati secondo l'ordine alfanumerico:

0 48	A 65	a 97
1 49	B 66	b 98
.....	.....	.....
8 56	Y 89	y 121
9 57	Z 90	z 122
cifre	maiuscole	minuscole

## Caratteri di Controllo ASCII

I **caratteri di controllo** hanno **funzioni speciali** e si ottengono o con **tasti specifici** o con una **sequenza Ctrl + Carattere**

Ctrl	Dec	Hex	Code	Nota
^@	0	0	NULL	carattere nullo
^A	1	1	SOH	partenza blocco
.....	...	...	.....	.....
^G	7	7	BEL	beep
^H	8	8	BS	backspace
^I	9	9	HT	tabulazione orizzontale
^J	10	A	LF	line feed (cambio linea)
^K	11	B	VT	tabulazione verticale
^L	12	C	FF	form feed (alim. carta)
^M	13	D	CR	carriage return (a capo)
.....	...	...	.....	.....
^Z	26	1A	EOF	fine file
^[_	27	1B	ESC	escape
.....	...	...	.....	.....
^_	31	1F	US	separatore di unità

## Caratteri ASCII Stampabili

**Nota:** il valore numerico di una cifra può essere calcolato come differenza del suo codice ASCII rispetto al codice ASCII della cifra 0 (es.  $'5' - '0' = 53 - 48 = 5$ )

## Tabella ASCII Estesa

Sono i codici oltre il 127 e **non sono compresi nello standard originario**

# Linguaggio C: Elementi Fondamentali

---

## Sommario

- Il Linguaggio C
- Gli Identificatori
- I Tipi Primitivi
- Gli Operatori

## Il Linguaggio C

---

È stato sviluppato tra il **1969** ed il **1973** presso gli **AT&T Bell Laboratories** da **Ken Thompson, B. Kernighan e Dennis Ritchie**:

- Per uso interno
- Legato allo sviluppo del sistema operativo **Unix**

Nel **1978** venne pubblicata la **prima specifica ufficiale del linguaggio**, "*The C Programming Language*".

Il C è un linguaggio:

- **Imperativo**: un programma è inteso come un insieme di istruzioni, ciascuna delle quali può essere pensata come un ordine che viene impartito alla macchina
- **Strutturato**: non sono presenti salti incondizionati del tipo `go to`
  - **Teorema di Böhm-Jacopini**: *qualsiasi algoritmo può essere implementato utilizzando le sole strutture di controllo di sequenza, selezione e ciclo*
- **Tipizzato**: ogni oggetto ha un tipo
- **Elementare**: poche keyword
- **Case Sensitive**: maiuscolo e minuscolo sono diversi negli identificatori
- **Portabile**
- **Standard ANSI**

## Diffusione Attuale

I linguaggi attualmente più diffusi sono:

- C
- C++, evoluzione del C
- Java, la cui sintassi è tratta da C++
- C#, estremamente simile a Java e C++

Il linguaggio C è tra i linguaggi più diffusi e la sua sintassi è ripresa da tutti gli altri linguaggi principali.

## La Struttura di un Programma C

Un programma può essere concepito come l'unione di due parti:

- **Dichiarativa**: descrive i dati che devono essere manipolati
- **Esecutiva**: descrive le azioni che devono essere eseguite

# Il Pre-Processore

La compilazione C passa attraverso un passo preliminare che precede la vera e propria traduzione in linguaggio macchina.

Il programma che realizza questa fase è detto pre-processore, la cui funzione principale è l'espansione delle direttive che iniziano con `#`. Le direttive principali sono:

```
#include  
#define
```

- **Esempio:**

```
#include <stdio.h>  
#include <math.h>  
  
int main(){  
    // Programma vuoto  
    return 0;  
}
```

## #include

Sintassi:

- `#include <file>` per includere un **file di sistema**
- `#include "file"` per includere un **file definito dal programmatore**

Esempio: `#include "mydef.h"`

- Significa che `<file>` viene **espanso ed incluso** per intero nel file sorgente

## #define

```
#define <costante> <valore>:
```

- `<costante>`: **identificatore della costante simbolica**, convenzionalmente indicato *tutto in maiuscolo*
- `<valore>`: **valore da assegnare alla costante**

Utilizzo:

- Definizione di **costanti simboliche**
- Maggiore **leggibilità**
- Maggiore **flessibilità**

### Esempio:

```
#define PI 3.1415  
...  
double raggio = 4.7432;  
double Area = PI * r * r;  
...
```

# Funzioni di I/O

In C le **operazioni di I/O** non sono gestite vere e proprie istruzioni, bensì mediante opportune **funzioni** (concetto che verrà introdotto in futuro).

L'utilizzo di queste istruzioni richiede l'inserimento di una direttiva `#include <stdio.h>` all'inizio del file sorgente, la quale non fa altro che "includere" il file `stdio.h`, che contiene alcune dichiarazioni, tra le quali:

```
printf(<formato>, <arg1>...<argn>);      // Stampa a video
scanf(<formato>, <arg1>...<argn>);        // Lettura da tastiera
```

## printf

Sintassi:

```
printf(<formato>, <arg1>...<argn>);
```

dove:

- `<formato>` è una **sequenza di caratteri** che *determina il formato di stampa di ognuno dei vari argomenti*
  - Nella forma `%<carattere>`:
    - `%d` intero
    - `%u` unsigned
    - `%s` stringa
    - `%c` carattere
    - `%x` esadecimale
    - `%o` ottale
    - `%f` float
    - `%g` double
  - Esempio:

```
...
int a = 3;
double b = 2.5;
...
printf("Primo valore: %d", a);
printf("Secondo valore: %f", b);
...
```

- `<arg>` sono le **quantità (espressioni) che si vogliono stampare**, associate alle direttive di formato nello stesso ordine.

## scanf

Sintassi:

```
scanf(<formato>, <arg1>...<argn>);
```

dove:

- `<formato>` è una **sequenza di caratteri** che *determina il formato di stampa di ognuno dei vari argomenti*

- Nella forma %<carattere>:
  - %d intero
  - %u unsigned
  - %s stringa
  - %c carattere
  - %x esadecimale
  - %o ottale
  - %f float
  - %g double

- Esempio:

```
...
int a;
double b;
...
scanf("Primo valore: %d", &a);
scanf("Secondo valore: %f", &b);
...
```

- <arg> sono le **variabili in cui si vogliono memorizzare i valori**, associate alle direttive di formato nello stesso ordine.

## Sequenze di Escape

La tabella ASCII comprende un certo numero di valori ai quali **non corrisponde un carattere stampabile**:

- \a segnale acustico (avviso)
- \b BACKSPACE
- \f modulo continuo
- \n nuova riga
- \r ritorno a capo
- \t tabulazione orizzontale
- \v tabulazione verticale
- \' virgoletta singola
- \" virgolette doppie

Per rappresentare questi caratteri si utilizza una **sequenza di escape**, dove il carattere **backslash** (detto *carattere di escape*) assume il ruolo di segnalatore.

## Esempio di uso delle Funzioni di I/O

```
// Inclusione del file header che contiene printf e scanf
#include <stdio.h>

int main(){
    // Dichiarazione variabili per memorizzare i dati
    int b, h, prodotto, area;

    // Richiesta di inserimento dei valori per base e altezza
    printf("Inserisci la base del triangolo: ");
    scanf("%d", &b);
    printf("Inserisci l'altezza del triangolo: ");
```

```

scanf("%d", &h);

// Operazioni matematiche per il calcolo del risultato
prodotto = b * h;
area = prodotto / 2;

// Stampa del risultato
printf("L'area del triangolo è: %d", area);

return 0;
}

```

## Elementi del Linguaggio C

Da un punto di vista lessicale, un programma è una **sequenza di termini**, detti **tokens**. Il compilatore deve riconoscere i termini del linguaggio per le successive fasi di analisi.

Tipi di termini:

- **Identifieri**
- **Parole chiave**
- **Costanti**
- **Espressioni**
- **Operatori**
- **Simboli o segni speciali**

Sono invece ignorati gli **spazi bianchi**, **tabulatori**, **newlines** e **commenti**.

### Gli Identifieri

**Identificatore** è un termine usato dal programmatore per indicare **funzioni**, **variabili**, **oggetti**, **costanti** ecc.

Ogni identificatore è formato da una **sequenza di caratteri** di tipo **lettere** o **cifre** o **underscore**:

- Il **primo carattere** deve essere una **lettera** o **underscore**
- Caratteri **maiuscoli** e **minuscoli** sono **diversi**
- 31 caratteri
- Identifieri **non validi**:
  - `un amico` contiene uno **spazio**
  - `un'amica` contiene un **apostrofo**
  - `piano*forte` contiene un **simbolo**
  - `for` è una **parola chiave** del C

Gli identifieri devono **ricordare mnemonicamente** gli oggetti a cui si riferiscono.

Le **parole chiave** sono *termini che hanno significati particolari per il compilatore C* e possono essere adoperati dal programmatore solo come previsto dal linguaggio. Alcuni esempi di parole chiave sono:

- `main`: indica che il testo che segue tra parentesi graffe rappresenta il codice sorgente del programma
- `const`: definisce il nome che segue come dato costante
- `float`: definisce il nome seguente come variabile a virgola mobile (singola precisione)
- `if`, `then`, `else`: definisce costrutti di controllo del linguaggio

- ecc.

## I Dati Numerici, le Variabili e le Costanti

Sono quelli più usati in ambito scientifico nei moderni sistemi di elaborazione. Tutti gli altri tipi di dato sono trasformati in dati numerici:

- In **matematica** una variabile è **un carattere alfabetico che rappresenta un numero arbitrario**, sconosciuto, o non completamente specificato.
- In **informatica** una variabile è **una porzione di memoria destinata a contenere dei dati**, che potranno essere **letti o modificati** durante l'esecuzione di un programma.

In informatica una **costante** è una porzione di memoria destinata a contenere dei dati, che potranno essere **solo letti** e non modificati durante l'esecuzione di un programma

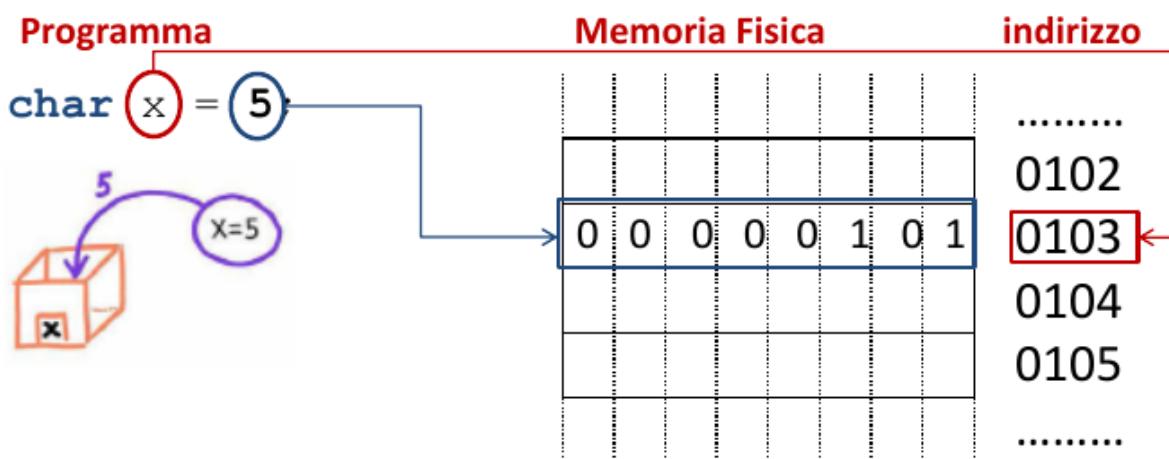
### • Il Contenuto di una Variabile

Ogni variabile, in ogni istante di tempo, possiede un certo valore:

- Le variabili appena definite hanno valore ignoto, ovvero sono non inizializzate
- In momenti diversi il valore può cambiare

In C tutti i dati **devono essere dichiarati prima di essere utilizzati**. La dichiarazione di un dato richiede:

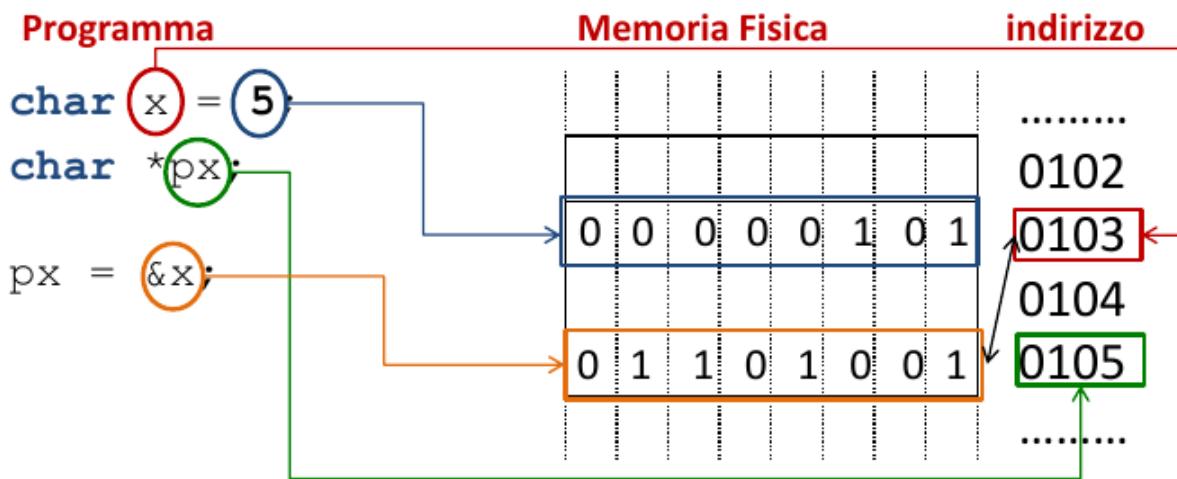
- L'**allocazione di uno spazio in memoria** atto a contenere il dato
- L'**assegnazione di un nome** a tale spazio di memoria



Ad ogni istanza di **x** nel programma viene associato il contenuto della cella di memoria, ovvero **5**. È possibile conoscere l'**indirizzo della cella di memoria** associata a **x**, scrivendo **&x**

## Il Puntatore

L'indirizzo della cella in cui è memorizzata la variabile **x** è esso stesso un **dato numerico** e può **essere** a sua volta **memorizzato** in una variabile. Una variabile che contiene l'indirizzo di una cella di memoria prende il nome di **puntatore alla variabile x**.



Ad ogni istanza di **x** nel programma viene associato il contenuto della cella di memoria, ovvero **5**. È possibile conoscere l'**indirizzo della cella di memoria** associata ad **x** scrivendo **&x**.

## Dichiarazione di Identificatori

In C dichiarare un dato significa specificarne:

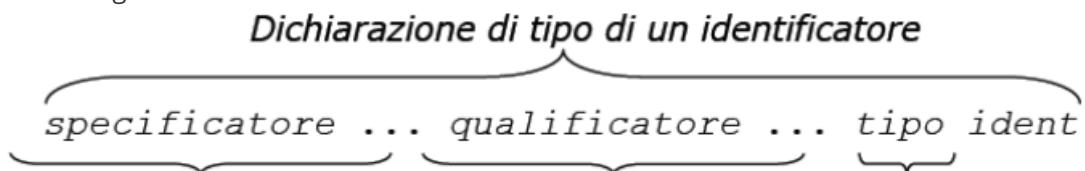
- **Nome**: definisce un identificativo unico per la variabile
- **Tipo**: è l'insieme dei valori che può assumere e l'insieme delle operazioni che possono applicarsi a tali valori
- **Modalità di accesso**: è il valore corrente della variabile
- **Tipi di base (primitivi)**

Sono quelli forniti direttamente dal C:

- `char`: **caratteri ASCII o interi** nell'intervallo [-128, 127]
- `int`: **interi in complemento a 2** nell'intervallo [-2.147.483.648, 2.147.483.647]
- `float`: **reali in floating point a singola precisione**, 7 cifre significative
- `double`: **reali in floating point a doppia precisione**, 15 cifre significative

La dimensione di questi tipi dipende dall'architettura, tuttavia la dimensione di un **char** è **sempre di 8 bit** (1 byte).

La dichiarazione di tipo informa il compilatore sul tipo assegnato ad un identificatore e ha come forma generale:



*Fornisce altre caratteristiche dell'identificatore*    *Indica come deve essere allocato il contenuto*    *tipo standard*

In cui:

- **Qualificatori**: short, long, signed, unsigned
- **Specificatori**: const, extern, static, volatile ecc.

La definizione di tipo è una dichiarazione che comporta l'allocazione di un'area di memoria per l'identificatore ma non l'inizializzazione del suo contenuto, che rimane indefinito.

- **Dimensione in Byte di un Tipo (sizeof)**

L'operatore `sizeof()` calcola il numero di byte utilizzato dai tipi di dato di base.

- **Sintassi:** `sizeof(<tipo>)`

Restituisce il numero di byte occupato da `<tipo>`, ad esempio:

```
unsigned int size;
size = sizeof(float); // Restituisce size = 4
```

L'uso dell'operatore `sizeof()` può essere esteso al calcolo dello spazio occupato da espressioni, vettori e strutture.

L'inizializzazione assegna un valore iniziale ad un identificatore già definito

- **Operazione di assegnazione:** `ident = valore`

La **dichiarazione** di un identificatore **deve precedere il suo primo utilizzo** e può avvenire in qualsiasi parte del programma.

È tuttavia **consigliabile dichiarare** tutte le costanti e le variabili **nella parte iniziale del programma (o della funzione)**, così da rendere più facile l'individuazione della sezione delle dichiarazioni nel programma.

Dichiarazione e inizializzazione **possono essere combinate**.

- **Dichiarazione di variabile:**

```
int numero;
```

- **Dichiarazione di variabile e successiva inizializzazione:**

```
int numero;
...
numero = 54;
```

- **Dichiarazione di più variabili dello stesso tipo:**

```
int numero, secondo_numero, terzo;
```

- **Dichiarazione di variabile con contestuale inizializzazione:**

```
int numero = 54;
```

- **Dichiarazione di più variabili e inizializzazione:**

```
int numero = 54, secondo_numero, terzo = 15;
```

## Simboli Speciali

Uno o anche due caratteri consecutivi che sono usati per scopi particolari, come:

- **Punto e virgola**, che servono ad indicare la fine di un'istruzione
- **Parentesi graffe**, che indicano inizio e fine di un'istruzione composta
- **Virgola** che è usata come separatore

Ad esempio, possiamo scrivere `float n, raggio;` invece di

```
float n;  
float raggio;
```

## Visibilità di un Identificatore

Ogni variabile è utilizzabile all'interno di un preciso **ambiente di visibilità**, anche detto **scope**.

- **Variabili globali**, ovvero quelle **definite all'esterno** del `main()`
- **Varibili locali**, ovvero quelle **definite all'interno** del `main()` o, più in generale, all'interno di un blocco

**N.B.** un blocco di istruzioni sono tutte quelle istruzioni racchiuse tra parentesi graffe

```
{  
    int a = 4;  
  
    a = a + 1;  
    printf("Il valore di a è: %d\n", a);  
}
```

## Le Espressioni

Le espressioni rappresentano il valore che si ottiene applicando opportune operazioni ben definite ad uno o più operandi che possono essere costanti o variabili.

In una espressione le operazioni vengono indicate con particolare simboli detti operatori, i quali possono essere di tipo:

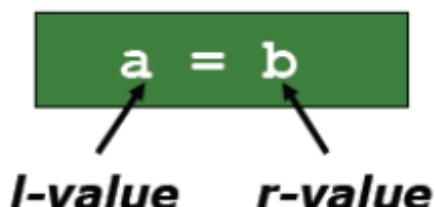
- Unario: agisce su un solo operando
- Binario: agisce su due operandi (destro e sinistro)
- Ternario: agisce su tre operandi

Inoltre, gli operatori possono essere:

- Di assegnazione
- Aritmetici
- Binari
- Relazionali
- Logici
- Incremento e Decremento
- Condizionali

## Espressioni di Assegnazione

L'operatore di assegnazione `=` copia il contenuto dell'**operatore destro (r-value)** nell'**operando sinistro (l-value)**:



- **r-value** è una qualsiasi **espressione** con valore di tipo standard

- **I-value** è una **variabile**

Il valore dell'espressione assegnazione è **r-value**

- **Esempio:** `a = b + 3`

È possibile combinare l'istruzione di assegnazione con gli operatori aritmetici.

- **Sintassi:** `<variabile> <operatore> = <espressione>;`
- **Operatori:** `+= -= *= /= %=`

**Esempi:**

```
x += 5;      // Equivalente a x = x + 5
y -= x;      // Equivalente a y = y - x
```

## Operazioni di Incremento e Decremento

Per le assegnazioni composte più comuni sono previsti degli **operandi esplici**:

```
++      // Equivalente a += 1
--      // Equivalente a -= 1
```

**Esempi:**

```
x++;      // Equivale a x = x + 1
valore--; // Equivale a valore = valore - 1
```

Questi operatori possono essere utilizzati sia in notazione **prefissa** che in notazione **postfissa**:

- **Prefissa:** la variabile viene **modificata prima di essere utilizzata** nell'espressione
- **Postfissa:** la variabile viene **modificata dopo averla utilizzata** nell'espressione

**Esempio:** assumendo `x = 4`

```
y = x++      // Si otterrà x = 5 e y = 4
y = ++x      // Si otterrà x = 5 e y = 4
```

## Espressioni Aritmetiche

Gli operatori aritmetici eseguono le principali operazioni matematiche.

Se l'operazione è **tra due numeri interi**, il **risultato** dell'operazione è **ancora** un numero **intero** (*troncamento*), ad esempio `27/4` dà come risultato `6` anziché `6.75`.

Per ottenere il resto di una divisione si usa l'operatore `%`. Ad esempio `24%4` dà come risultato `0`.

Operazione	Simbolo algebrico	Simbolo in C	Espressione algebrica	Espressione in C
<b>Addizione</b>	<code>+</code>	<code>+</code>	<code>a+b</code>	<code>a+b</code>
<b>Sottrazione</b>	<code>-</code>	<code>-</code>	<code>a-b</code>	<code>a-b</code>
<b>Moltiplicazione</b>	<code>*</code>	<code>*</code>	<code>ab</code>	<code>a*b</code>
<b>Divisione</b>	<code>:</code>	<code>/</code>	<code>a:b</code>	<code>a/b</code>
<b>Modulo</b>	<code>mod</code>	<code>%</code>	<code>a mod b</code>	<code>a%b</code>

## Regole di Precedenza

Qual è la gerarchia in `5 + 3 * 2`? Il risultato è `16` o `11`?

1. Svolgere le parentesi
2. Moltiplicazione, divisione e modulo
3. Addizione e sottrazione

Più operazioni dello stesso tipo vanno risolte **da sinistra a destra**, ad esempio:

$$5 + 4 - 3 * 4 / 2$$

$$5 + 4 - 12 / 2$$

$$5 + 4 - 6$$

$$9 - 6 = 3$$

## Tipo di un'Espressione

Se le costanti e le variabili sono tutte dello stesso tipo allora anche il valore dell'espressione sarà dello stesso tipo.

Se tutte le grandezze presenti nell'espressione sono di tipo numerico, anche se diversi, **sarà il compilatore ad effettuare tutte le opportune conversioni di tipo**.

- **Esempio:**

```
// Di che tipo è (n + f) * d?  
int n = 3;           // n è convertito a float  
float f = 2.5;       // La somma n + f è convertita a double  
double d = 4.8;      // (n + f) * d è di tipo double
```

**Regola:** non adoperare espressioni in cui sono presenti variabili di tipo diverso

## Operazioni Relazionali

Gli **operatori relazionari** sono:

- Strettamente maggiore: `>`
- Maggiore o uguale: `>=`
- Strettamente minore: `<`
- Minore o uguale: `<=`
- Uguale: `=`
- Diverso: `!=`

Eseguono il confronto fra i valori dei due operandi (di qualsiasi tipo standard) e restituiscono un valore booleano (vero/falso):

<code>a &gt; b</code>	restituisce <b>true</b> se <b>a</b> è <b>maggior</b> e di <b>b</b>
<code>a &gt;= b</code>	restituisce <b>true</b> se <b>a</b> è <b>maggior o uguale</b> a <b>b</b>
<code>a &lt; b</code>	restituisce <b>true</b> se <b>a</b> è <b>minore</b> di <b>b</b>
<code>a &lt;= b</code>	restituisce <b>true</b> se <b>a</b> è <b>minore o uguale</b> a <b>b</b>
<code>a == b</code>	restituisce <b>true</b> se <b>a</b> è <b>uguale</b> a <b>b</b>
<code>a != b</code>	restituisce <b>true</b> se <b>a</b> è <b>diverso da</b> <b>b</b>



# Linguaggio C: Strutture di Controllo

## Sommario

- Algebra di Boole
- Strutture condizionali
- Strutture iterative
- Uso dei Cicli nella programmazione
- Esempi di Algoritmi e Programmi

## Logica Booleana

Nel **1847** George Boole introdusse un nuovo tipo di **logica formale**, basata esclusivamente su enunciati di cui fosse possibile verificare in modo inequivocabile la verità o falsità.

**George Boole** è stato un **matematico e logico** britannico, ed è considerato il **fondatore della logica matematica**. La sua opera influenzò anche i settori della filosofia e diede vita alla **scuola degli algebristi della logica**.

## Variabili e Operatori Booleani

Sono variabili in grado di assumere solo due valori: Vero e Falso.

Sono descritti tramite una tavola di verità

- Per **N** operandi la tabella ha  $2^N$  righe che elencano **tutte le possibili combinazioni** di valori delle variabili indipendenti ed il valore assunto dalla variabile dipendente.

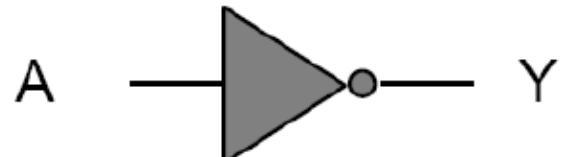
### Operatore NOT

È un **operatore logico di negazione** di una proposizione.

Data una proposizione logica **A** la negazione logica determina una seconda proposizione detta "**non A**", che risulta **vera quando A è falsa e viceversa**.

Soltanamente è *indicato con un trattino posto al di sopra della variabile logica*:

A	$\overline{A}$
falso	vero
vero	falso



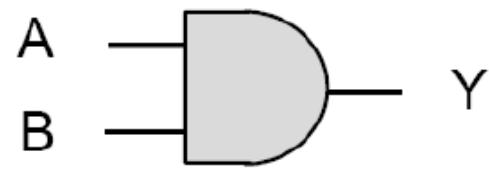
### Operatore AND

È un **operatore logico di congiunzione logica** tra due proposizioni.

Date due proposizioni **A** e **B**, la congiunzione logica determina una terza proposizione **C** che si manifesta **vera solo quando entrambe le proposizioni sono vere**.

L'operatore AND è detto anche congiunzione logica (o **moltiplicazione logica**) e solitamente si indica con il simbolo **X**:

A	B	$A \times B$
falso	falso	falso
falso	vero	falso
vero	falso	falso
vero	vero	vero



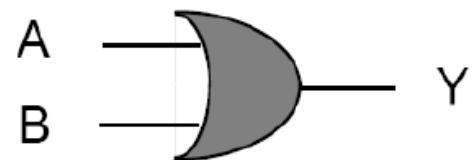
## Operatore OR

È un **operatore logico di disgiunzione logica** tra due proposizioni logiche.

Date due proposizioni **A** e **B**, la disgiunzione logica determina una terza proposizione logica **C** che si manifesta **vera quando almeno una delle due proposizioni è vera**.

L'operatore OR è *soltanente rappresentato con il simbolo +:*

A	B	$A + B$
falso	falso	falso
falso	vero	verò
vero	falso	verò
vero	vero	verò

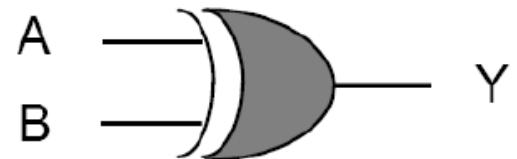


## Operatore XOR

È un **operatore logico di disgiunzione esclusiva** tra due proposizioni logiche.

Date due proposizioni logiche **A** e **B**, la disgiunzione esclusiva tra le due proposizioni è **vera** **soltanto nel caso in cui sia vera solo una delle due proposizioni**.

A	B	$A \Delta B$
falso	falso	falso
falso	vero	verò
vero	falso	verò
vero	vero	falso



## Operatori di Manipolazione dei Bit

Il C possiede una serie di operatori che possono agire direttamente sui bit delle variabili e costanti di tipo intero o carattere, dichiarate nel programma.

Si tratta di 4 operatori derivati direttamente dalle operazioni booleane di base e di altri 2 che eseguono l'operazione di shift (destra e sinistra) di un certo numero di bit:

- I primi si prestano a "mascherare" o "commutare" i bit
- I secondi possono essere utili nelle operazioni di moltiplicazione e divisione per 2

Tranne uno, sono tutti operatori binari, ovvero che agiscono su due espressioni

B	A	0	1
0		0	0
1		0	1

A AND B

B	A	0	1
0		0	1
1		1	1

A OR B

A	0	1
	1	0

NOT A

B	A	0	1
0		0	1
1		1	0

A XOR B

Operazione	Operatore	tipo
AND bit a bit	&	Binario
OR bit a bit		Binario
XOR bit a bit	^	Binario
NOT bit a bit (complemento a 1)	~	Unario
Shift a sinistra	<<	Binario
Shift a destra	>>	Binario

**unsigned char z, x = 3, y = 13;**

x    

0	0	0	0	0	0	1	1
---	---	---	---	---	---	---	---

y    

0	0	0	0	1	1	0	1
---	---	---	---	---	---	---	---

**z = x & y**  
AND

x	0	0	0	0	0	0	1	1
y	0	0	0	0	1	1	0	1
z	0	0	0	0	0	0	0	1

**z = ~ x**  
NOT

x	0	0	0	0	0	0	1	1
z	1	1	1	1	1	1	0	0

**$z = x | y$**

OR

0	0	0	0	0	0	1	1
0	0	0	0	1	1	0	1
0	0	0	0	1	1	1	1

**$z = x ^ y$**

XOR

0	0	0	0	0	0	1	1
0	0	0	0	1	1	0	1
0	0	0	0	1	1	1	0

Gli shift sono invece equivalenti alla moltiplicazione/divisione per 2.

- Sintassi:

```
<operando> >> <n.posizioni>      // Fai scorrere <operando> a destra di
<n.posizioni> bit
<operando> << <n.posizioni>      // Fai scorrere <operando> a sinistra di
<n.posizioni> bit
```

Sia `<operando>` che `<n.posizioni>` devono essere valori interi.

I due operatori si comportano diversamente a seconda del tipo numerico:

- Dati unsigned:** equivale allo **shift logico**
  - `<<` inserisce degli **0** nelle **posizioni meno significative**
  - `>>` inserisce degli **0** nelle **posizioni più significative**
- Dati signed:** equivale allo **shift aritmetico**
  - `<<` aggiunge degli **0** nelle **posizioni meno significative**, e **mantiene inalterato il bit più significativo** (bit di segno)
  - `>>` **inserisce un valore uguale al bit più significativo** (bit di segno) **mantenendo pertanto inalterato il segno**

Esempio:

```
unsigned char x = 15;    // x = 00001111
x = x << 2;            // x = 00111100 -> (15 x 2^2) = 60
x = x >> 2;            // x = 00000011 -> (15 / 2^2) = 3

char x = -15;           // x = 11110001
x = x << 2;            // x = 11000100 -> (-15 x 2^2) = -60
x = x >> 2;            // x = 11111100 -> (-15 / 2^2) = -4
```

Anche per questi operatori è **consentita la scrittura abbreviata** `<<=` o `>>=`

# Codifica dei Caratteri Alfabetici

Oltre ai numeri, molte applicazioni informatiche elaborano **caratteri** (*simboli*).

Gli elaboratori elettronici trattano i numeri, pertanto i caratteri e i simboli sono codificati per mezzo dei numeri. C'è quindi bisogno di uno **standard di codifica**:

A → 01000001

3 → 00110011

\$ → 00100100

Il tipo di codifica utilizzato deve essere ovviamente noto durante lo scambio dei dati.

La codifica deve prevedere le **lettere dell'alfabeto**, le **cifre numeriche**, i **simboli**, la **punteggiatura**, i **caratteri speciali** per certe lingue ecc.

Lo standard più diffuso è il codice **ASCII**

- *American Standard Code for Information Interchange*

## Codifica ASCII

Definisce una tabella di corrispondenza fra ciascun carattere e un codice a **7 bit** (128 caratteri).

I caratteri sono, in genere, **rappresentati con 1 byte** (8 bit). I caratteri che hanno il **MSB posto a 1** (codici da 128 a 255) rappresentano un'**estensione della codifica**.

La tabella comprende sia **caratteri di controllo** (codici da 0 a 31) sia **caratteri stampabili**

I caratteri alfanumerici hanno codici ordinati secondo l'ordine alfanumerico

0 48	A 65	a 97
1 49	B 66	b 98
.....	.....	.....
8 56	Y 89	y 121
9 57	Z 90	z 122
cifre	maiuscole	minuscole

## Caratteri di Controllo ASCII

I caratteri di controllo hanno **funzioni speciali** e si ottengono o con **tasti specifici** o con sequenze **Ctrl + Carattere**

<b>Ctrl</b>	<b>Dec</b>	<b>Hex</b>	<b>Code</b>	<b>Nota</b>
^@	0	0	NULL	carattere nullo
^A	1	1	SOH	partenza blocco
.....	...	...	.....	.....
^G	7	7	BEL	beep
^H	8	8	BS	backspace
^I	9	9	HT	tabulazione orizzontale
^J	10	A	LF	line feed (cambio linea)
^K	11	B	VT	tabulazione verticale
^L	12	C	FF	form feed (alim. carta)
^M	13	D	CR	carriage return (a capo)
.....	...	...	.....	.....
^Z	26	1A	EOF	fine file
^[_	27	1B	ESC	escape
.....	...	...	.....	.....
^_	31	1F	US	separatore di unità

## Caratteri ASCII Stampabili

Dec	Hx	Chr	Dec	Hx	Chr	Dec	Hx	Chr	Dec	Hx	Chr	Dec	Hx	Chr
32	20	SPACE	48	30	0	64	40	@	80	50	P	96	60	`
33	21	!	49	31	1	65	41	A	81	51	Q	97	61	a
34	22	"	50	32	2	66	42	B	82	52	R	98	62	b
35	23	#	51	33	3	67	43	C	83	53	S	99	63	c
36	24	\$	52	34	4	68	44	D	84	54	T	100	64	d
37	25	%	53	35	5	69	45	E	85	55	U	101	65	e
38	26	&	54	36	6	70	46	F	86	56	V	102	66	f
39	27	'	55	37	7	71	47	G	87	57	W	103	67	g
40	28	(	56	38	8	72	48	H	88	58	X	104	68	h
41	29	)	57	39	9	73	49	I	89	59	Y	105	69	i
42	2A	*	58	3A	:	74	4A	J	90	5A	Z	106	6A	j
43	2B	+	59	3B	;	75	4B	K	91	5B	[	107	6B	k
44	2C	,	60	3C	<	76	4C	L	92	5C	\	108	6C	l
45	2D	-	61	3D	=	77	4D	M	93	5D	]	109	6D	m
46	2E	.	62	3E	>	78	4E	N	94	5E	^	110	6E	n
47	2F	/	63	3F	?	79	4F	O	95	5F	_	111	6F	o
														DEL

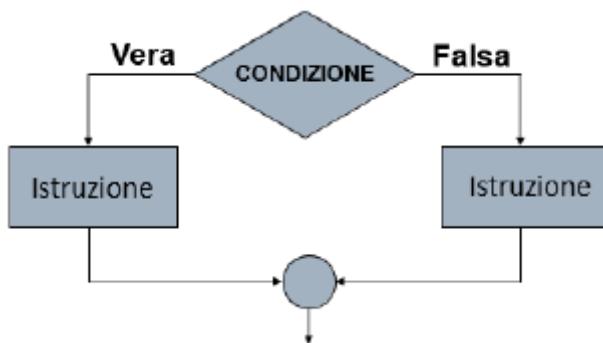
**Nota:** il valore numerico di una cifra può essere calcolato come differenza del suo codice ASCII rispetto al codice ASCII della cifra 0 (es. '5'-'0' = 53-48 = 5)

## Tabella ASCII Estesa

I codici oltre il 127 **non sono compresi nello standard originario**

128	ç	144	é	160	à	176	ø	193	±	209	∏	225	ø	241	±
129	ü	145	æ	161	í	177	¤	194	τ	210	τ	226	Γ	242	≥
130	é	146	Æ	162	ó	178	¤	195		211	Ł	227	π	243	≤
131	à	147	ö	163	ú	179		196	-	212	Ł	228	Σ	244	ƒ
132	ä	148	ö	164	ñ	180		197	+/-	213	γ	229	σ	245	∫
133	à	149	ò	165	Ñ	181		198	∫	214	γ	230	μ	246	+
134	å	150	ø	166	°	182		199		215	+	231	τ	247	≈
135	ç	151	ú	167	°	183	π	200	≤	216	≠	232	Φ	248	°
136	è	152	—	168	÷	184	π	201	≥	217	↓	233	Ω	249	.
137	ë	153	Ö	169	—	185		202	≤	218	γ	234	Ω	250	.
138	è	154	Ü	170	¬	186		203	τ	219	█	235	δ	251	¬
139	í	156	€	171	½	187	π	204	∫	220	■	236	∞	252	—
140	í	157	¥	172	¼	188	π	205	=	221	█	237	◊	253	≈
141	í	158	—	173	¡	189	π	206	≠	222	█	238	ε	254	█
142	À	159	f	174	«	190	¡	207	±	223	█	239	Λ	255	
143	Å	192	L	175	»	191	π	208	≤	224	α	240	=		

## Le Strutture Condizionali

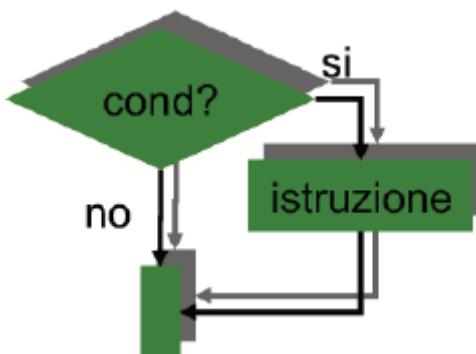


Normalmente il **flusso di esecuzione** di un programma procede in modo **sequenziale**.

Tuttavia, ci sono casi in cui l'algoritmo richiede di eseguire in alternativa un blocco di istruzioni piuttosto che un altro, in funzione del verificarsi di qualche condizione.

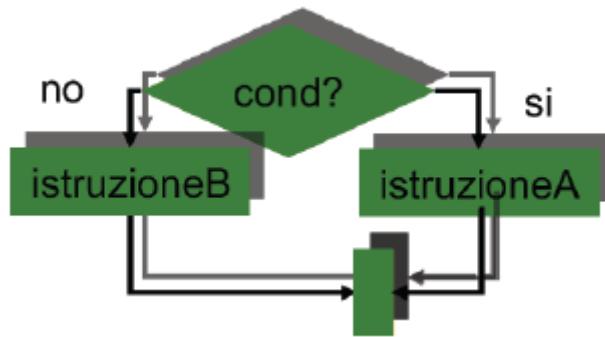
Il linguaggio C dispone di particolari istruzioni, dette **strutture di controllo**, che consentono di eseguire un blocco di istruzioni in modo condizionato, mantenendo la strutturazione e la leggibilità del programma

### Strutture Condizionali: if



L'istruzione di controllo **if (selezione)** indica al calcolatore di **eseguire una tra due istruzioni al verificarsi di una certa condizione**

- **Condizione** è un'**espressione logica**
- Se la condizione è **TRUE** allora *il programma esegue l'istruzione*, altrimenti *passa direttamente all'istruzione successiva*



Nel caso di due scelte alternative, all'istruzione `if` si aggiunge l'istruzione `else`:

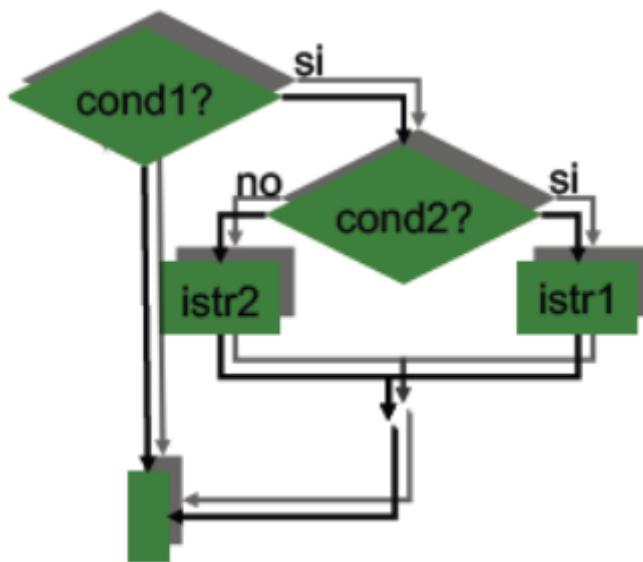
```

if(condizione){
    istruzione_1;
} else{
    istruzione_2;
}

```

- `Condizione` è un'**espressione logica**
- Se la condizione è `TRUE` allora il programma esegue `istruzione_1`
- Se la condizione è `FALSE` allora il programma esegue `istruzione_2`

## Strutture Condizionali: if/else



Se l'istruzione controllata da un `if` consiste a sua volta in un altro `if` (**istruzioni if nidificate**) allora **ogni eventuale else si riferisce sempre all'istruzione if immediatamente superiore** (in assenza di parentesi graffe). **Ad esempio:**

```

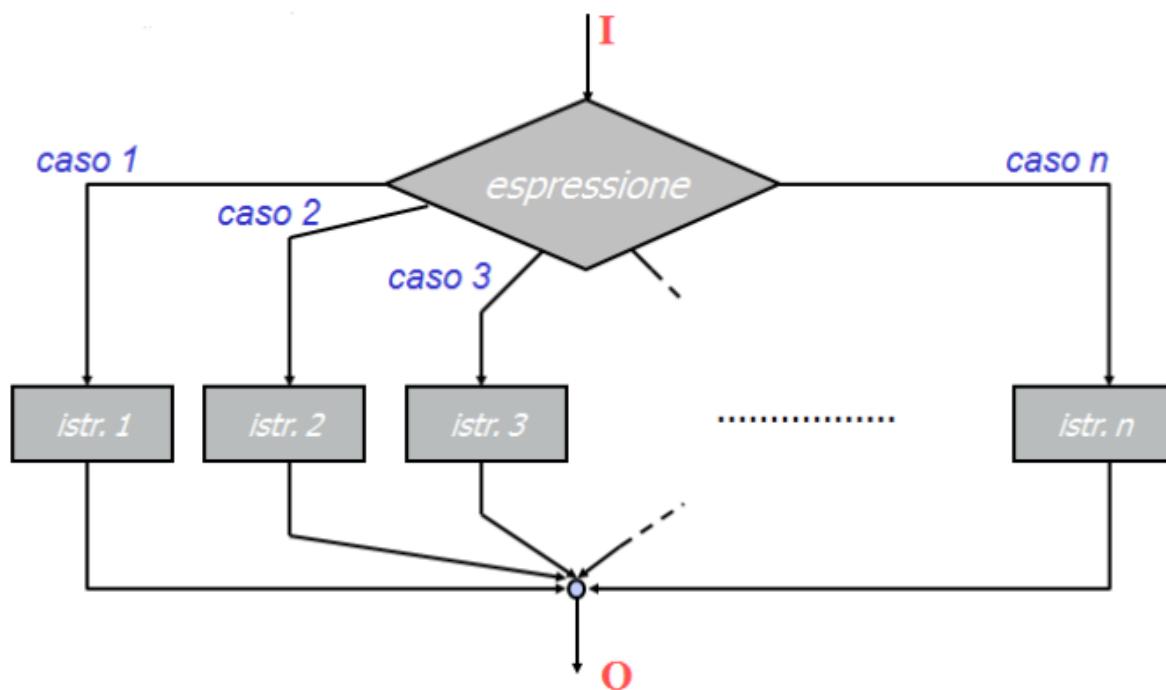
if(condizione_1)
    if(condizione_2) istruzione_1:
    else istruzione_2:

```

**Suggerimento:** usare sempre le parentesi graffe, anche in caso di istruzioni semplici:

- Consente di **controllare meglio il risultato di annidamenti** di `if`
- **Riduce gli errori** in caso di aggiunte di altre istruzioni come nel caso dell'`if` (o dell'`else`)

## Strutture Condizionali: switch



Quando la condizione da imporre non è intrinsecamente binaria ma si deve distinguere tra più casi ed eseguire le istruzioni appropriate per ogni singolo caso, l'`if` annidato può non essere conveniente per complessità e leggibilità

Per questi casi c'è l'istruzione `switch`

- **Sintassi:**

```
switch(espressione){  
    case costante_1:  
        istruzione_1;  
        break;  
    case costante_2:  
        istruzione_2;  
        break;  
    ...  
    ...  
    default:  
        istruzione_default;  
}
```

Dove:

- `espressione`: espressione a valore numerico di tipo `int` o `char`, ma non `float` o `double`
- `costante_1`, `costante_2`, ecc: sono costanti dello stesso tipo dell'espressione
- `istruzione_1`, `istruzione_2`, ecc: sono sequenze di istruzioni (senza graffe)

**Significato:**

- In base al valore di `espressione`, esegui le istruzioni del `case` corrispondente. Nel caso in cui nessun `case` venga intercettato allora esegui le istruzioni corrispondenti al caso `default` (se esiste)

I vari `case` devono rappresentare delle **condizioni mutualmente esclusive**.

I vari `case` vengono **eseguiti in sequenza**: per evitarlo è **necessario usare l'istruzione `break` alla fine di ogni blocco di istruzioni** per interrompere il flusso.

Se il blocco di `default` non è presente e nessun case viene soddisfatto allora **l'esecuzione procede con la prima istruzione che segue la `switch`**.

# Altre Strutture di Controllo

## Sommario

- Funzioni matematiche
- Casting esplicito

## Funzioni Matematiche

`math.h` è l'header file della libreria standard del C che **contiene definizioni di macro, costanti e dichiarazioni di funzioni e tipi** usati per le operazioni matematiche.

FUNZIONE MATEMATICA		FUNZIONE C
<code>\x</code>	radice quadrata di $x$ (restituisce un valore $\geq 0$ )	<code>double sqrt (double x);</code>
<code>sen x</code>	seno di $x$ (con $x$ reale)	<code>double sin (double x);</code>
<code>cos x</code>	coseno di $x$ (con $x$ reale)	<code>double cos (double x);</code>
<code>tg x</code>	tangente di $x$ (con $x$ reale)	<code>double tan (double x);</code>
<code>arcsen x</code>	arcoseno di $x$ (con $x$ nell'intervallo $[-1 ;+1]$ )	<code>double asin (double x);</code>
<code>arccos x</code>	arcoseno di $x$ (con $x$ nell'intervallo $[-1 ;+1]$ )	<code>double acos (double x);</code>
<code>arctg x</code>	arcotangente di $x$ (con $x$ reale)	<code>double atan (double x);</code>
<code>e<sup>x</sup></code>	$e$ elevato ad $x$ (con $x$ reale)	<code>double exp (double x);</code>
<code>10<sup>x</sup></code>	$10$ elevato a $x$ (con $x$ reale)	<code>double pow10 (double x);</code>
<code>ln x</code>	logaritmo in base $e$ di $x$ (con $x$ reale $\geq 0$ )	<code>double log (double x);</code>
<code>log x</code>	logaritmo di base $10$ di $x$ ( $x$ reale positivo)	<code>double log10 (double x);</code>
<code> x </code>	valore assoluto di $x$	<code>double fabs (double x);</code>
	calcolo ipotenusa di un triangolo rettangolo i di cateti $x$ ed $y$ (con $x,y$ numeri reali)	<code>double hypot (double x, double y);</code>
<code>arctg x/y</code>	arcotangente di $y/x$ . Il valore restituito è compreso fra $-\pi$ e $+\pi$ estremi compresi	<code>double atan2 (double y, double x);</code>
<code>y<sup>x</sup></code>	$x$ elevato ad $y$ (con $x,y$ numeri reali)	<code>double pow (double x, double y);</code>
	calcola mantissa ed esponente di $x$	<code>double frexp (double x, int* esp);</code>

Tutte le funzioni in C **restituiscono dei valori double** invece che float, in modo da **garantire la massima precisione**.

## Casting di Tipo Implicito

Il C esercita un controllo sui tipi e **restituisce** un messaggio di **errore quando si tenta di eseguire operazioni fra operandi di tipo non ammesso**.

Durante l'esecuzione di un programma ci sono situazioni in cui viene cambiato il tipo di dato (**casting implicito**):

- Quando un **valore di un tipo viene assegnato ad una variabile di un altro tipo compatibile**
- Quando un'**operazione contiene due elementi di tipo diverso** (ad esempio la somma di un intero con un reale)

Questo può provocare degli **errori difficilmente individuabili**.

I quattro operatori matematici si applicano a qualsiasi tipo standard, ma **i tipi dei due operandi devono essere uguali**. Nel caso in cui i due tipi siano diversi, **il compilatore esegue una conversione di tipo implicita su uno dei due operandi**, seguendo la **regola di promuovere il tipo più semplice (meno ampio in termini di bytes) a quello più complesso (più ampio in termini di**

bytes).

Il compilatore considera una **gerarchia in ordine crescente di complessità**:

- `char > unsigned char > short > unisgned short > long > unsigned long > float > double > long double`

**Esempio:** se eseguiamo la divisione `3.4/2` il secondo operando sarà trasformato in `2.0` e il risultato sarà, correttamente, `1.7`.

Nelle assegnazioni, il tipo dell'operando di destra viene sempre convertito implicitamente nel tipo dell'operando di sinistra.

- Se il tipo dell'espressione ha una precisione minore del tipo della variabile allora *non c'è alcun problema*

```
variabile_double = espressione_float
```

- Se, invece, il tipo dell'espressione ha una precisione maggiore del tipo della variabile

```
variable_float = espressione_double
```

*possono nascere due tipi di problemi:*

1. Il risultato dell'`espressione_double` contiene un numero di cifre che il tipo `float` non gestisce, comportano una **grave perdita di precisione**
2. Il risultato dell'`espressione_double` supera il massimo valore rappresentabile come `float`. Il programma potrebbe quindi avere un **comportamento imprevedibile**

**Esempio:**

```
int c;
double d;
c = d;      // Warning in fase di compilazione
```

## Casting di Tipo Esplicito

L'**operatore di casting** (o **conversione esplicita**) serve per **forzare una conversione di tipo**. Ha due operandi e in C equivale a:

```
(tipo) variabile
      tipo (variabile)
```

e consiste nell'**indicare il nuovo tipo fra parentesi davanti al nome della variabile da trasformare**.

**Tutti i tipi standard consentono il casting.** Se la variabile da trasformare è l'operando di una certa operazione allora il tipo risultante deve essere fra quelli ammissibili (altrimenti viene generato un errore di compilazione).

- **Esempio:**

Esempio\_1:

```
float r;  
r = (float)3 / 4;  
// Questo è corretto e assegna ad r il valore 0.75
```

Esempio\_2:

```
float(n) % 3;  
// Questo è errato perché l'operatore % ammette solo operandi interi
```

# Strutture di Controllo: Approfondimento

## Sommario

- Strutture di Controllo Iterative
- Istruzione `while`
- Istruzione `for`
- Istruzione `do-while`

## Strutture Iterative

Si dice **ciclo (loop)** una sequenza di istruzioni che deve **essere ripetuta più volte consecutivamente**.

Si consideri ad esempio il calcolo del fattoriale di un numero  $n > 0$ :

con il caso particolare:

Sembrerebbe che basti una semplice assegnazione, ma se non si conosce a priori il valore di  $n$  è impossibile scrivere l'istruzione che esegue il calcolo del fattoriale, poiché la formula contiene tanti fattori quanti ne indica  $n$ .

Proviamo a riscrivere la formula del fattoriale come:

Osservando la formula possiamo:

- **Attribuire** ad una variabile `fatt` il valore **1**
- **Moltiplicare** `fatt` per **2** ed **attribuire** il risultato **ancora a fatt**
- Poi **moltiplicare** `fatt` per **3** e così via fino a **n**

L'algoritmo di risoluzione può quindi essere formalizzato mediante un **processo iterativo**:

- Assegna il valore **1** a `fatt`;
- Se **n** vale **0**, termina
- Altrimenti, **per k che va da 1 a n con passo unitario**, moltiplica `fatt` per `k` e attribuisci il risultato a `fatt`

Il prodotto `fatt x k` viene eseguito  $n$  volte com'è necessario.

Nel linguaggio C i cicli iterativi sono realizzati da **tre costrutti**:

- `while`: realizza il costrutto **while-do**
- `do-while`: realizza il costrutto **repeat-until**
- `for`: realizza il **ciclo a contatore**

## Istruzione `while`

### Sintassi

```
while(condizione){           // Finchè condizione è vera
    istruzione;             // esegue istruzione, che può essere semplice o composta
}
```

## Osservazioni

Il costrutto `while` realizza il costrutto `while-do` della programmazione strutturata:

- `condizione` deve essere di **tipo logico** ed è **ricalcolata ad ogni iterazione**:
  - Se `condizione` risulta **falsa già alla prima iterazione** allora `istruzione` **non viene eseguita** neppure una volta
  - Se `condizione` non diventa **mai falsa** si finisce in un **loop infinito**
- `istruzione` è una normale istruzione e quindi **può contenere qualsiasi tipo di istruzione**, anche altri `while`, dando origine ai `while annidati`

## `break` e `continue`

Le strutture di controllo iterative sono corredate dalle istruzioni `break` e `continue`, che consentono, rispettivamente, di **anticipare l'uscita dal ciclo** o **saltare un'iterazione**:

- `break`: **provoca l'immediata terminazione del ciclo e l'uscita dall'ambito di visibilità ad esso connesso**. A seguito di ciò il controllo di flusso viene quindi **rediretto all'istruzione successiva esterna al ciclo**.  
Solitamente l'esecuzione di `break` è subordinata alla valutazione di un'espressione booleana che determina l'uscita dal ciclo **in caso di circostanze "straordinarie"** rispetto quelle per cui è prevista la naturale terminazione del ciclo
- `continue`: **provoca l'interruzione dell'iterazione corrente**. Il controllo di flusso **rimane confinato all'interno del ciclo ma viene reindirizzato all'iterazione successiva** in conseguenza di una circostanza inattesa che invalida o rende superflua l'esecuzione di tale iterazione.  
Anche l'esecuzione di `continue` è spesso subordinata alla valutazione di un'espressione booleana che determina il ritorno immediato alla valutazione dell'espressione

## Istruzione `for`

In altri linguaggi il costrutto `for` permette di eseguire un'istruzione per un numero prefissato di volte (ciclo a contatore).

Nel linguaggio C è più generale, al punto da dover essere assimilata ad una particolare riscrittura del costrutto `while`

## Sintassi

```
for(inizializzazione; condizione; aggiornamento){  
    istruzione;  
}
```

## Osservazioni

- `condizione` è un **espressione logica**
- `inizializzazione` e `aggiornamento` sono **espressioni di tipo qualsiasi**

L'istruzione `for` opera secondo il seguente **algoritmo**:

- Viene calcolata `inizializzazione`
- Finchè `condizione` è vera:
  - Viene eseguita `istruzione`
  - Viene calcolato `aggiornamento`

Di fatto il costrutto `for` è del tutto equivalente al seguente frammento di programma:

```
inizializzazione;
while(condizione){
    istruzione;
    aggiornamento;
}
```

Poichè non vi sono restrizioni sull'istruzione da eseguire nel corpo del ciclo, questa **può contenere a sua volta istruzioni `for`** ( `for` annidati) o **altri costrutti di controllo**.

## Istruzione `do-while`

### Sintassi

```
do{
    istruzione;
} while(condizione);
```

Ripeti istruzione (che può essere semplice o composta) finchè condizione è vera.

### Osservazioni

L'istruzione `do-while` realizza il costrutto `repeat-until` della programmazione strutturata

- `condizione` deve essere di **tipo logico** ed è **calcolata ad ogni iterazione**
- `istruzione`, pertanto, è **sempre eseguita almeno una volta** (anche se `condizione` è subito falsa)
- se `condizione` non diventa mai falsa si finisce in un **loop infinito**

Come per gli altri costrutti, `istruzione` è una **normale istruzione composta** e può contenere qualsiasi tipo di istruzione o costrutto, dando origine a **strutture annidate**.

# Linguaggio C: Strutture Dati

---

## Sommario

- Strutture astratte e concrete
- Vettori
- Operazioni sui vettori

## Tipi di Dati Strutturati

---

I tipi considerati finora hanno la caratteristica comune di **non essere strutturati**, ovvero **ogni elemento è una singola entità**.

Se il programma deve trattare collezioni di dati, anche se sono dello stesso tipo, ad ognuno deve essere associato un identificatore. Supponendo di dover gestire le paghe di una ditta di 3000 dipendenti sarebbe necessario definire 3000 variabili diverse.

Nella rappresentazione delle informazioni non è sufficiente considerare l'insieme dei valori, ma anche la loro **struttura**, cioè **le relazioni logiche che legano tra loro i valori stessi**:

- *un insieme di dati e la struttura che li lega* costituiscono una **struttura informatica**
  - *la parte più piccola e indivisibile* della struttura informativa si dice **elemento**

## Strutture Informative

Di conseguenza è opportuno esaminare i principali tipi di aggregati dal punto di vista della struttura logica, cioè le cosiddette **strutture astratte di dati**, e i sistemi per la loro rappresentazione nella memoria di un calcolatore, cioè **le possibili strutture concrete adatte a contenere le strutture astratte**.

In altri termini, la formulazione di un problema è espressa tenendo conto anche del tipo di rappresentazione dei dati in memoria che si pensa di adottare.

Con il termine **strutture informative**, si comprendono:

- le **strutture astratte**, proprie del problema e dipendenti unicamente da questo. Esse **studiano ed organizzano le relazioni logiche che intercorrono tra i dati**. Vengono usate per descrivere le proprietà dell'insieme dei dati indipendentemente da come questi saranno memorizzati (insieme di leggi che definiscono le relazioni esistenti fra i dati di un insieme finito)
- le **strutture concrete** analizzano il processo dal punto di vista hardware, cioè **come i dati vengono allocati nella memoria del PC**. Sono le strutture interne che vengono usate per rappresentare in memoria le strutture astratte

Definire una struttura stratta significa stabilirne il tipo, precisando:

- l'**aspetto statico**, ovvero:
  - quali sono gli **elementi di base** che la caratterizzano
  - quali sono le **relazioni possibili** tra gli elementi e come si accede ad essi
- l'**aspetto dinamico**, ovvero:
  - le **operazioni sui dati**, cioè se e in quale modo è possibile operare sui dati mediante inserimenti, variazioni e cancellazioni

Un'altra classificazione prevede la distinzione tra:

- **struttura statica**: quando il **numero di elementi** che lo compongono rimane **costante nel tempo**. Questo significa che, una volta costruita la struttura, essa viene usata solo per operazioni di ricerca o per modificare qualche valore
- **struttura dinamica**: quando il **numero degli elementi può subire variazioni nel tempo**

L'**accesso** specifica il modo in cui si "raggiunge" l'informazione all'interno della struttura (attraverso operazioni di lettura/scrittura) e può essere:

- **sequenziale**: quando, per raggiungere un elemento, è **necessario consultare tutti gli elementi che lo precedono**
- **diretto**: quando è **possibile raggiungere direttamente l'informazione desiderata**, o mediante la sua posizione all'interno della struttura o mediante altre informazioni

## Vettori

Il vettore è una **collezione di variabili tutte dello stesso tipo** (detto **tipo base**) di **lunghezza prefissata**.

Questa collezione di variabili è individuata da un unico nome, il nome del vettore appunto.

- Ogni elemento del vettore è detto **componente** ed è *individuato dal nome del vettore seguito da un indice posto tra parentesi quadre*
  - L'indice può essere **solo di tipo intero o enumerato** e determina la posizione dell'elemento nel vettore

**Vettori scalari**: contengono un solo valore

**Variabili vettoriali**: contengono **più valori dello stesso tipo**, detti **elementi**:

Se il tipo è di base (`char`, `int`, `long`, `double` ecc.) gli elementi del vettore sono variabili scalari di quel tipo. Tutti gli elementi **condividono lo stesso nome** e sono contraddistinti da un **indice indicato tra parentesi quadre**

```
int a[4];
a[1], a[2], a[3], a[4]
```

### • Definizione

```
tipo nome[n_elementi];
int vet[10];
```

Definisce un vettore di 10 elementi di tipo `int`.

- `n_elementi` deve essere una **costante intera positiva** nota al momento della compilazione (non si può chiedere all'utente mentre il programma è in esecuzione)

L'indice **deve essere un valore intero e di tipo intero**.

Il primo elemento ha indice **0** (`vet[0]`), mentre l'ultimo elemento ha indice **n-1** (nell'esempio è `vet[9]`).

Si accede ai singoli elementi indicando il **nome del vettore seguito dall'indice** dell'elemento tra parentesi quadre (costante o variabile): `vet[7]`, `vet[i]`, `vet[k+1]`.

L'indice **può essere il risultato di un'espressione**, più o meno complessa, purchè con risultato intero.

- Esempio:

```
double c, a[30];           // Definizione di a come vettore di double
int i, j, k;
x = a[2 * i + j - k];    // Espressione aritmetica per l'indice
...
char x, a[30];            // Definizione di a come vettore di caratteri
int i, j, k;
x = a[i + j - k];        // Espressione aritmetica per l'indice
```

Poichè ciascun elemento del vettore è del tipo indicato nella definizione, può essere utilizzato in tutti i contesti in cui si può usare una variabile di quel tipo

- Esempio:

```
int vet[10];
scanf("%d", &vet[4]);    // Si può fare perchè vet[4] è un intero
x = vet[4] * 5;
```

Gli elementi del vettore sono allocati in **locazioni di memoria contigue e successive**.

Il **nome di un vettore** è usato dal compilatore come **sinonimo dell'indirizzo di memoria del primo elemento** del vettore (*non è una variabile*).

La presenza di un indice suggerisce che è **possibile scandire tutti i valori di un vettore mediante un ciclo for**

```
#define N 10                      // Definizione di N = 10
int main(){
    int vet[N];                  // Definizione del vettore di 10 interi
    for(int i = 0; i < N; i++){
        scanf("%d", &vet[i]);    // Inserimento di 10 interi nel vettore
da tastiera
    }

    for(int i = N - 1; i >= 0; i--){
        printf("%d", vet[i]);   // Stampa del vettore in ordine inverso
    }
}
```

Bisogna stare attenti a non "sforare", ovvero a **non accedere a porzioni di memoria che vanno oltre i limiti del vettore**. Questo perchè **si rischia di modificare porzioni di memoria adibite ad altro**, come, per esempio, ad altre variabili.

Il compilatore può, opzionalmente, prevedere un controllo automatico per gli sforamenti, ma questo riduce le prestazioni, perchè ad ogni accesso al vettore viene fatto un controllo preliminare per lo sforamento.

Un vettore va **inizializzato**, altrimenti i valori in esso contenuti sono indeterminati. La lista degli inizializzatori può essere indicata come

- `int vet[4] = {12, 5, 3, 6};`

Gli inizializzatori **devono essere noti al compile time**, quindi devono essere espressioni costanti: **numeri, #define, valori enum, indirizzi di memoria di variabili statiche.**

**Non** possono essere: **variabili, valori const, risultati di funzioni, indirizzi di memoria di variabili automatiche.**

La **dimensione può essere opzionalmente omessa**, in quanto il compilatore può calcolarla contando i valori:

- `int vet[] = {12, 5, 3, 6};`

Se la lista contiene meno valori di quelli indicati a meno di 1 allora **quelli non specificati sono inizializzati a 0**:

- `int vet[4] = {6, 2}; // Gli altri due elementi sono posti a 0`

Con

- `int vet[10] = {0};`

si pone a zero il primo elemento e poi vengono automaticamente posti a 0 tutti gli altri elementi.

Sugli elementi del vettore agiscono gli operatori previsti per il `tipo_componente`. Pertanto **è lecito scrivere**:

- `valor_fin = vett_x[m1] + vett_y[m2];`

purchè, naturalmente, `valor_fin`, il vettore `vett_x` e il vettore `vett_y` siano dello **stesso tipo**.

Il **tempo necessario** per accedere ad un elemento di un vettore è **indipendente** dal valore dell'indice, pertanto **il vettore è una struttura ad accesso casuale**.

Non ci sono operatori che agiscono sul vettore nel suo complesso, pertanto **non è lecito** l'assegnamento di un vettore ad un altro vettore:

- `vett_x = vett_y;`

anche se i due vettori sono dello stesso tipo. Per copiare un vettore in un altro **è necessario copiare un elemento alla volta**, cosa possibile tramite strutture iterative (solitamente `for`)

- **Ma perchè questa istruzione è errata?**

Quando si definisce un vettore, il compilatore riserva un'area di memoria sufficiente per contenerlo e associa l'indirizzo iniziale di quell'area al nome simbolico (**identificatore**) da noi scelto per il vettore.

Pertanto l'identificatore **non è una vera e propria variabile** ma, piuttosto, un **puntatore**: in pratica l'identificatore è **l'indirizzo di memoria del primo elemento del vettore**.

Ecco perchè l'istruzione è errata.

# Strutture Dati: Approfondimento

## Sommario

- Operazioni elementari sui vettori
  - Definizioni
  - Copia di un vettore
  - Ricerca di un elemento
  - Vettori multidimensionali

## Ricapitolando

- Tutte le celle di un vettore avranno lo **stesso nome**
- Tutte le celle di un vettore devono avere lo **stesso tipo di base**
- La **dimensione** del vettore è **fissa** e deve essere **determinata al momento della sua definizione**
  - La dimensione è **sempre un numero intero**
- Ogni cella ha **sempre un valore**
  - Impossibile avere celle "vuote"
  - Le celle non inizializzate contengono un valore ignoto
- Ciascuna cella è identificata dal proprio **indice**
- Gli indici sono **sempre numeri interi**
  - In C gli indici **partono da 0**
- Ogni cella è a tutti gli effetti una **variabile** il cui tipo è **pari al tipo di base del vettore**
- Ogni cella, indipendentemente dalle altre:
  - deve essere **inizializzata**
  - può essere **letta/stampata**
  - può essere **aggiornata** da istruzioni di assegnazione
  - può essere **usata** in espressioni aritmetiche

## Definizione

```
#define N 10          // Dimensioni dei vettori
int v[N];           // Vettore di N interi
float r[N];         // Vettore di N reali
int i, j;           // Indici dei cicli

#define M 100          // Dimensioni dei vettori
int w[N];           // Vettore di N interi
int h[M];           // Vettore di M interi
int dato;           // Elemento da cercare
```

## Stampa di un Vettore

Occorre stampare un elemento per volta utilizzando un ciclo `for`, ricordando che:

- gli indici del vettore variano tra **0** e **N-1**

- gli utenti solitamente contano tra **1** e **N**
- `v[i]` è l'elemento (i+1)-esimo

**Esempio:**

```
printf("vettore di %d interi\n", N);
for(int i = 0; i < N; i++){
    printf("Elemento %d: ", i + 1);
    printf("%d\n", v[i]);
}
```

**Output:**

```
Stampa di un vettore di 10 interi
Elemento 1: 3
Elemento 2: 4
Elemento 3: 7
Elemento 4: 5
Elemento 5: 3
Elemento 6: -1
Elemento 7: -3
Elemento 8: 2
Elemento 9: 7
Elemento 10: 3
```

## Stampa in Linea

**Esempio:**

```
printf("vettore di %d interi\n", N);
for(int i = 0; i < N; i++){
    printf("Elemento %d: ", i + 1);
    printf("%d ", v[i]);
}
printf("\n");
```

**Output:**

```
Stampa di un vettore di 10 interi
3 4 7 5 3 -1 -3 2 7 3
```

## Copia di un Vettore

Si tratta di **copiare il contenuto di un vettore in un altro vettore**.

Per farlo **bisogna copiare ogni elemento dal primo al secondo** vettore utilizzando un ciclo `for`. I due vettori devono, ovviamente, avere lo **stesso tipo di base e lo stesso numero di elementi**:

```
for(int i = 0; i < N; i++){
    vet2[i] = vet1[i];
}
```

Come si può vedere, nonostante siano coinvolti due vettori, **basta un solo ciclo `for` e un solo indice** per accedere agli elementi di entrambi i vettori.

## Ricerca di un Elemento

Dato un valore numerico, verificare:

- se almeno uno degli elementi del vettore è uguale al valore numerico

- in caso affermativo, dire dove si trova
- in caso negativo, dire che non esiste

Si tratta di una classica istanza del problema di **ricerca di esistenza**

- Se l'array **non è ordinato** si usa la **ricerca lineare**
- Se l'array **è ordinato** si usa la **ricerca binaria** o *dicotomica*

## Ricerca Binaria

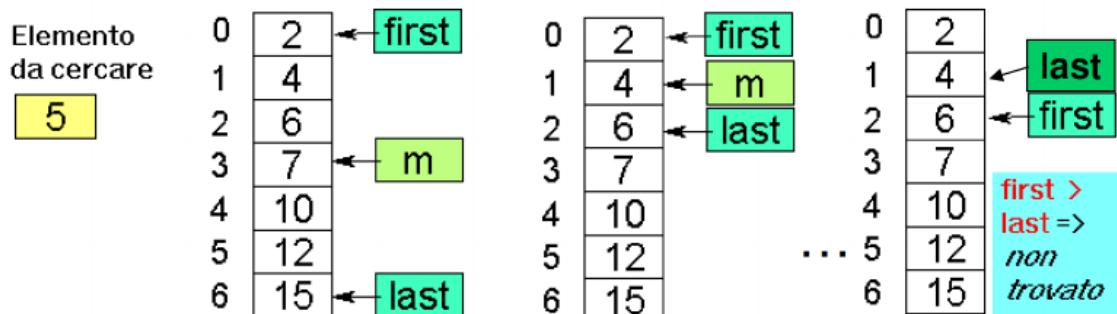
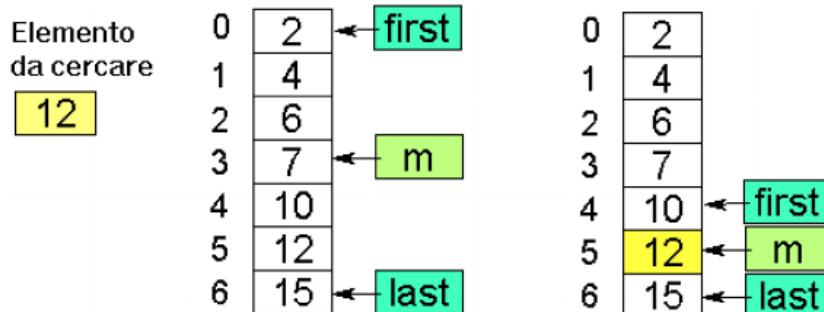
Se il vettore è ordinato allora la ricerca può essere ottimizzata

- **Definizione**

Sia `dim` la dimensione dell'array:

- se l'elemento mediano (in posizione `med`) dell'array è l'elemento da cercare allora **elemento trovato**
- Se l'elemento mediano dell'array è maggiore dell'elemento da cercare allora bisogna **cercare nella prima metà dell'array** (tra **0** e **`med-1`**)
- Se l'elemento mediano dell'array è minore dell'elemento da cercare allora bisogna **cercare nella seconda metà dell'array** (da **`med+1`** a **`N-1`**)

**Esempio:**



La tecnica di ricerca binaria, rispetto alla ricerca esaustiva, consente di **eliminare ad ogni passo metà degli elementi del vettore**:

- la prima volta che si esegue il ciclo si deve cercare fra **n** elementi
- la seconda volta **n/2** elementi
- la terza volta **n/4** elementi e così via

In generale, dopo **p** ripetizioni del ciclo, il numero di elementi che rimangono da confrontare è

Nel caso peggiore la ricerca continua finché gli elementi che rimangono da confrontare siano **<= 1**, che, matematicamente, si esprime come:

## Vettori Multidimensionali

Il concetto di vettore come collezione di elementi consecutivi può essere esteso, immaginando che gli elementi siano dei vettori a loro volta: si ottiene così un **vettore multidimensionale**, o **matrice**

La definizione di matrice ricalca pienamente quella del vettore:

- `tipo_comp nome[dim1][dim2]...;`

dove:

- `tipo_comp` può essere un **qualsiasi tipo semplice**
- `dim1, dim2`, racchiusi tra parentesi quadre, definiscono il **numero di elementi di ogni dimensione**

**Esempio:**

```
// Matrice bidimensionale di numeri interi formata da 3 righe e 5 colonne
int a[3][5];
```

<b>a</b>	<b>a[0][0]</b>	<b>a[1][0]</b>	<b>a[2][0]</b>	<b>a[3][0]</b>	<b>a[4][0]</b>	<b>a[0]</b>
	<b>a[0][1]</b>	<b>a[1][1]</b>	<b>a[2][1]</b>	<b>a[3][1]</b>	<b>a[4][1]</b>	<b>a[1]</b>
	<b>a[0][2]</b>	<b>a[1][2]</b>	<b>a[2][2]</b>	<b>a[3][2]</b>	<b>a[4][2]</b>	<b>a[2]</b>

L'**inizializzazione** di un vettore multidimensionale deve essere **effettuata per righe**:

```
int vet[3][2] = {
    {8,1},           // vet[0]
    {1,9},           // vet[1]
    {0,3}            // vet[2]
};
```

Per un vettore multidimensionale la scansione va applicata a tutte le dimensioni facendo uso dei **cicli annidati**:

```
int vet[3][5];
for(int i = 0; i < 3; i++){           // Per ogni riga
    for(int j = 0; j < 5; j++){       // Per ogni colonna
        // Elaborazione su vet[i][j]
    }
}
```

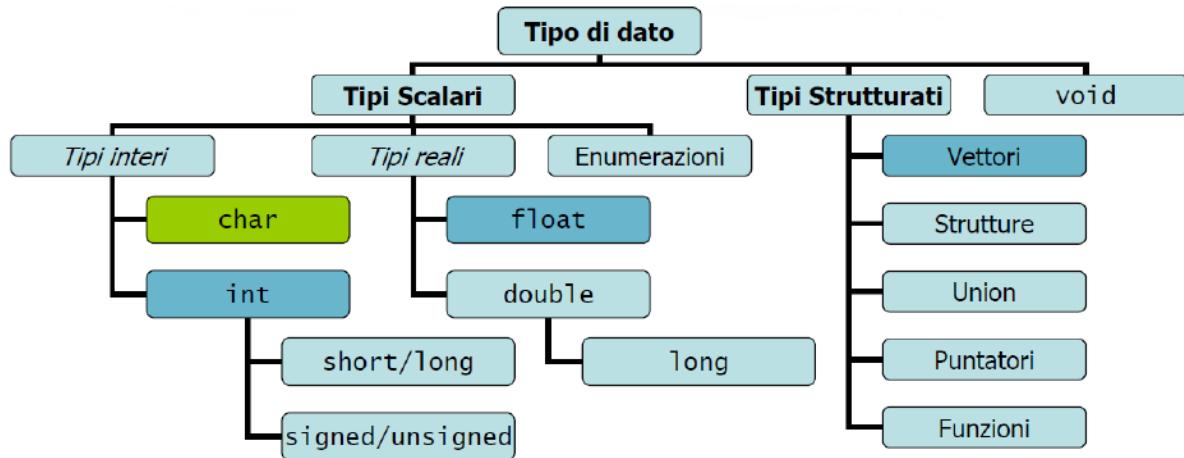
# Linguaggio C: Dati Testuali

## Sommario

- Tipi di dati testuali
- Caratteri
- Operazioni sui caratteri

## Tipi di Dati Testuali

I programmi visti finora erano in grado di elaborare esclusivamente informazioni numeriche, ma in molti casi è necessario elaborare informazioni di tipo testuale



## Rappresentazione dei Testi

Il calcolatore è **in grado di rappresentare i caratteri alfabetici, numerici e i simboli speciali** di punteggiatura.

Ad ogni diverso carattere viene assegnato, convenzionalmente, un codice numerico corrispondente.

- Il programma C lavora sempre con i codici numerici
- Le **funzioni di input/output** sono in grado di **accettare e mostrare i caratteri corrispondenti**

## Codice ASCII

La tabella ASCII è un codice convenzionale usato per la rappresentazione dei caratteri di testo attraverso i byte.

- Ad ogni byte viene fatto corrispondere un diverso carattere della tastiera (lettere, numeri, segni)
- La "vera" tabella ASCII, in realtà, **copre solo i primi 128 byte**, i byte fino al 256-esimo costituiscono la **tabella ASCII estesa**, che presenta **varie versioni a carattere nazionale**

## Caratteri e Stringhe

Il codice ASCII permette di rappresentare un singolo carattere.

Nelle applicazioni pratiche spesso serve rappresentare delle **sequenze di caratteri**, ovvero le **stringhe**

## Dualità Caratteri - Numeri

Ogni carattere è rappresentato dal suo codice ASCII, mentre ogni stringa è rappresentata dai codici ASCII dei caratteri di cui è composta

## Caratteri in C

Ogni carattere viene rappresentato dal proprio codice ASCII. Sono sufficienti **7 bit per rappresentare ciascun carattere**.

Non sono previste le **lettere accentate** né altri **simboli diacritici**: per questo c'è bisogno di **estensioni speciali e librerie specifiche**.

L'alfabeto latino, usato nella scrittura di molte lingue nel mondo, presenta una grande quantità di varianti grafiche. Queste varianti sono talmente numerose che nemmeno i 128 byte della tabella estesa bastano a rappresentarle tutte, per questo esistono diverse **estensioni della tabella ASCII**.

Per ovviare al problema è stato creato un nuovo standard internazionale chiamato **Unicode**, definito dalla **Unicode Consortium** e dalla **International Organization for Standardization (ISO 10646)**, che rappresenta i caratteri usando **2 byte**

## Caratteri di Controllo

Un **carattere di controllo** (o **carattere non visualizzabile**), è un *codice in un set di caratteri che non rappresenta in sé un simbolo scritto*. Tutti i **caratteri** nella tabella ASCII **al di sotto della posizione 32** fanno parte di questa categoria.

I caratteri di controllo nella tabella ASCII ancora d'uso comune comprendono:

- **7 (bell)**: provoca l'**emissione di un segnale sonoro** da parte del terminale ricevente
- **8 (backspace)**: utilizzato per **cancellare l'ultimo carattere visualizzato**, di solito quello immediatamente a sinistra del cursore
- **9 (horizontal tab)**: tabulatore orizzontale
- **10 (line feed)**: utilizzato per **terminare le linee di testo**
- **12 (form feed)**: per **terminare la pagina sulla stampante e avanzare al modulo successivo**
- **13 (carriage return)**: ritorno a capo, utilizzato per **terminare le linee di testo di 27 (escape)**

## Maiuscole, Minuscole e Numeri

- Lettere **maiuscole**:
  - codici ASCII compresi tra **65** e **90**, estremi inclusi
- Lettere **minuscole**:
  - codici ASCII compresi tra **97** e **122**, estremi inclusi
- Caratteri **numerici**:
  - codici ASCII compresi tra **48** e **57**, estremi inclusi

Tra il codice ASCII di una **lettera maiuscola** e quello della corrispondente **lettera minuscola** c'è una **differenza di 32 unità**

- Le **lettere maiuscole** sono tutte **consecutive ed in ordine alfabetico**
- Le **lettere minuscole** sono tutte **consecutive ed in ordine alfabetico**

- Le lettere maiuscole vengono "prima" delle minuscole
- Le **cifre numeriche** sono tutte **consecutive in ordine da 0 a 9**
- I **simboli di punteggiatura** sono sparsi

## Vettori di Caratteri: le Stringhe

Una stringa è una struttura dati capace di memorizzare **sequenze di caratteri**.

In C non esiste un tipo di dato specifico per le stringhe, pertanto si usano **vettori di caratteri**.

La **lunghezza** di una stringa è **tipicamente variabile** durante l'esecuzione del programma, quindi bisognerà gestire l'occupazione variabile dei vettori di caratteri

### Caratteristiche delle Stringhe

Memorizzate come singoli caratteri, ma il loro significato è dato dell'intera sequenza di caratteri

- Lunghezza variabile
- Mix di lettere/cifre/punteggiatura/spazi
- Solitamente non contengono caratteri di controllo

Occorre trattare l'insieme di caratteri memorizzato nel vettore come un'unica "variabile". Ogni operazione elementare sulle stringhe coinvolgerà tipicamente dei cicli che scandiscono il vettore.

Molte funzioni in libreria sono già disponibili per compiere le operazioni più frequenti e utili

### Il Tipo `char`

I caratteri in C si memorizzano in variabili di tipo `char`

- `char lettera;`

Le costanti di tipo `char` si indicano ponendo il simbolo corrispondente tra **singoli apici**

- `lettera = 'Q';`

Attenzione a non confondere i 3 tipi di apici presenti sulla tastiera:

<b>Apice singolo (apostrofo)</b>	'	<b>In C, delimita singoli caratteri</b>
<b>Apice doppio (virgolette)</b>	"	<b>In C, delimita stringhe di caratteri</b>
<b>Apice rovesciato (accento grave)</b>	`	<b>Non utilizzato in C</b>

### Dualità dei `char`

Sintatticamente, i `char` non sono altro che degli `int` di piccola dimensione:

- ogni operazione possibile su un `int` è possibile anche su un `char`
- ovviamente solo alcune di tali operazioni avranno senso sull'interpretazione testuale (ASCII) del valore numerico

## Caratteri Speciali

Per alcuni caratteri di controllo il linguaggio C definisce una particolare sequenza di escape per poterli rappresentare:

- `\n`
  - **ASCII:** LF - 10
  - **Significato:** A capo
- `\t`
  - **ASCII:** TAB - 9
  - **Significato:** Tabulazione
- `\b`
  - **ASCII:** BS - 8
  - **Significato:** Backspace
- `\a`
  - **ASCII:** BEL - 7
  - **Significato:** Emette un "bip"
- `\r`
  - **ASCII:** CR - 13
  - **Significato:** Torna alla prima colonna

Alcuni caratteri **hanno un significato particolare dentro gli apici**. Per poterli inserire come carattere esistono apposite sequenze di escape:

- `'\\\'`
  - **ASCII:** \
  - **Significato:** Immette un backslash
- `'\\'`
  - **ASCII:** '
  - **Significato:** Immette un apice singolo
- `'\\\"`
  - **ASCII:** "
  - **Significato:** Immette un apice doppio
- `'\ooo'`
  - **ASCII:** *ooo*
  - **Significato:** Immette un carattere ASCII con codice ottale *ooo*
- `'\xhh'`
  - **ASCII:** *hh*
  - **Significato:** Immette un carattere ASCII con codice esadecimale *hh*

## Input/Output di char

Esistono due insiemi di funzioni che permettono di leggere e stampare variabili di tipo `char`:

- le funzioni `printf / scanf`, usando lo specificatore di formato `%c`
- le funzioni `putchar` e `getchar`

In entrambi i casi è sufficiente includere la libreria `stdio.h`

## Osservazioni

La funzione `printf` è più comoda quando occorre stampare altri caratteri insieme a quello desiderato.

- Esempio:

```
printf("La risposta è: %c\n", ch);
printf("Codice: %c%d\n", ch, num);
```

La funzione `putchar` è più comoda quando occorre stampare semplicemente il carattere:

```
for(ch = 'a'; ch <= 'z'; ch++){
    putchar(ch);
}
```

La funzione `getchar` è generalmente più comoda in tutti i casi:

```
printf("Vuoi continuare? (s/n)");
ch = getchar();
```

## Bufferizzazione dell'Input/Output

Tutte le funzioni della libreria `stdio.h` gestiscono l'input/output **in modo bufferizzato**.

Per la maggior efficienza, **i caratteri non vengono trasferiti immediatamente** dal programma al terminale (o viceversa), **ma solo a gruppi**.

È quindi possibile che dopo una `putchar` **il carattere non compaia immediatamente** sullo schermo e, analogamente, che la `getchar` **non restituisca il carattere** finché l'utente non preme invio.

Il programma stampa l'invito ad inserire un dato

```
char ch,ch2 ;
printf("Dato: ");
ch = getchar();
ch2 = getchar();
```

L'utente immette Invio, il programma prosegue

```
char ch,ch2 ;
printf("Dato: ");
ch = getchar();
ch2 = getchar();
```

getchar blocca il programma in attesa del dato

```
char ch,ch2 ;
printf("Dato: ");
ch = getchar();
ch2 = getchar();
```

Ora `ch='a'`, il programma fa un'altra `getchar()`

```
char ch,ch2 ;
printf("Dato: ");
ch = getchar();
ch2 = getchar();
```

L'utente immette 'a', il programma non lo riceve

```
char ch,ch2 ;
printf("Dato: ");
ch = getchar();
ch2 = getchar();
```

Il programma non si blocca in attesa dell'utente, C'era già Invio! `ch2='\n'`

```
char ch,ch2 ;
printf("Dato: ");
ch = getchar();
ch2 = getchar();
```

## Soluzione

```
char ch, tmp;
printf("Dato: ");
ch = getchar();      // Leggi il dato
// Elimina eventuali caratteri successivi ed il \n che sicuramente ci sarà
do{
    tmp = getchar();
} while(tmp != '\n');

// Modo alternativo
char ch;
printf("Dato: ");
ch = getchar();
while(getchar() != '\n');
```

## Operazioni sui `char`

Le operazioni lecite sui `char` derivano direttamente dalla combinazione tra:

- le operazioni permesse sugli `int`
- la disposizione dei caratteri nella tabella ASCII
- le convenzioni lessicali della nostra lingua scritta

Una variabile di tipo `char` è allo stesso tempo:

- il **valore numerico** del codice ASCII del carattere
- il **simbolo** corrispondente al carattere ASCII

# Linguaggio C: le Stringhe

## Sommario

- Stringhe
- Operazioni sulle stringhe
- Libreria `string.h`

## Stringhe in C

Nel linguaggio C non è esplicitamente supportato alcun tipo di dato "stringa".

Le informazioni di tipo stringa vengono memorizzate ed elaborate ricorrendo a semplici **vettori di caratteri**

- **Esempio:** si realizzi un programma che acquisisca da tastiera il nome dell'utente (max 20 caratteri) e stampi a video un saluto per l'utente stesso.

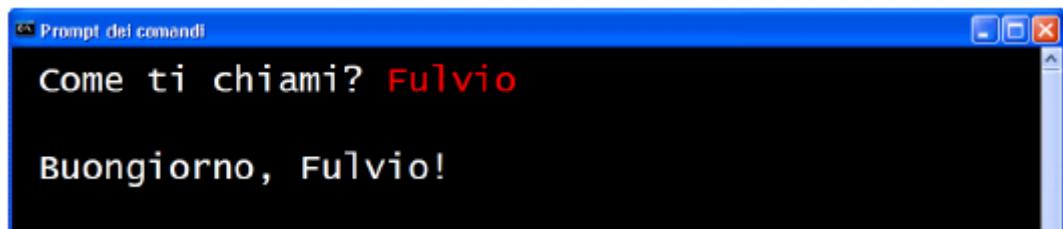
```
#define MAX 20
char nome[MAX];
int N, i;
char ch;

printf("Come ti chiami?");
N = 0;

ch = getchar();
while(ch != '\n' && N < MAX){
    nome[N] = ch;
    N++;
    ch = getchar();
}

printf("Buongiorno, ");
for(i = 0; i < N; i++){
    putchar(nome[i]);
}
printf("!\n");
```

- Che succede nel terminale:



Qualsiasi operazione sulle stringhe si può realizzare agendo opportunamente su vettori di caratteri, gestiti con **occupazione variabile**.

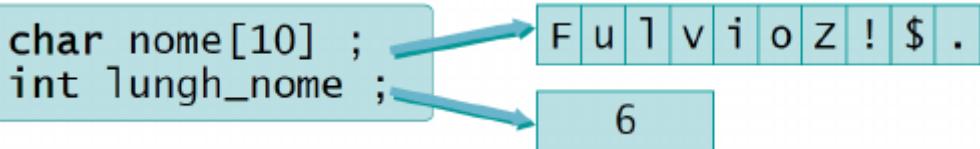
Così facendo, però, vi sono alcuni **svantaggi**:

- per ogni vettore di caratteri, occorre definire un'**opportuna variabile che ne indichi la lunghezza**

- ogni operazione, anche elementare, richiede l'uso di cicli `for` o `while`

Ci sono **due modi** per determinare la lunghezza di una stringa:

- utilizzare una **variabile intera che memorizzi il numero di caratteri validi**



- utilizzare un **carattere speciale**, con funzione di **terminatore**, dopo l'ultimo carattere valido



## Il Carattere Terminatore

Il carattere terminatore deve avere le **seguenti caratteristiche**:

- deve far parte della tabella ASCII**, ovvero deve essere rappresentabile in un `char`
- non deve mai comparire nelle stringhe utilizzate dal programma**, ovvero non deve confondersi con i caratteri "normali"

Va da sè che, ovviamente, il vettore di caratteri dovrà avere una posizione libera in più per memorizzare il terminatore stesso.

Per convenzione, in C si sceglie che **tutte le stringhe abbiano un carattere terminatore**, il quale corrisponde al carattere di codice ASCII pari a **zero**:

- `nome[6] = 0` oppure `nome[6] = '\0'`

**Esempio:**

```
#include <stdio.h>
#define MAX 10

int main(){
    char nome[MAX];

    for(int i = 0; i < MAX; i++){
        nome[i] = '0' + i;
    }

    printf("nome: %s\n", nome);
    nome[5] = '\0';

    printf("nome: %s\n", nome);
    return 0;
}
```

**Nel terminale:**

```
nome: 0123456789@
nome: 01234
```

- **Vantaggi**
  - Non è necessaria un'**ulteriore variabile intera** per ciascuna stringa
  - L'**informazione sulla lunghezza** della stringa è **interna al vettore** stesso
  - **Tutte le funzioni** della libreria standard C **rispettano questa convenzione**:
    - si aspettano che la stringa sia terminata
    - restituiscono sempre stringhe terminate
- **Svantaggi**
  - Necessario **1 byte in più**:
    - per una stringa di ***N*** caratteri serve un vettore di ***N+1*** elementi
  - Necessario **ricordare di aggiungere** sempre **il terminatore**
  - **Impossibile rappresentare stringhe contenenti** il carattere ASCII **0**

**Esempio:** si realizzi un programma in linguaggio C che acquisisca da tastiera il nome dell'utente (max 20 caratteri) e stampi a video un saluto per l'utente stesso

```
#define MAX 10

char nome[MAX];
char ch;
int i;

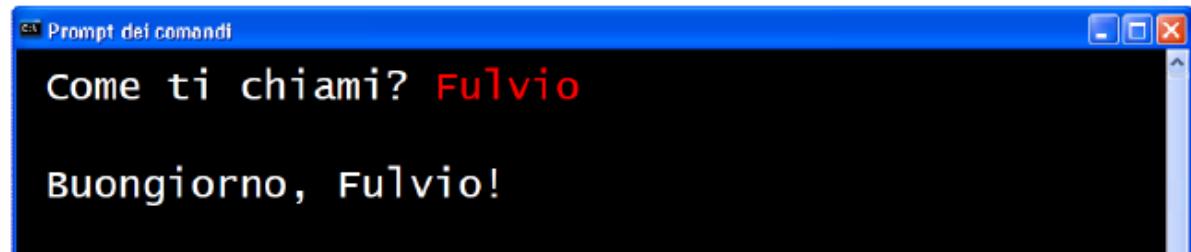
printf("Come ti chiami? ");
i = 0;

ch = getchar();
while(ch != '\n' && i < MAX){
    nome[i] = ch;
    i++;
    ch = getchar();
}

// Aggiungiamo il terminatore nullo
nome[i] = '\0';

printf("Buongiorno, ");
for(i = 0; nome[i] != '\0'; i++){
    putchar(nome[i]);
}
printf("\n");
```

Nel terminale:



# I/O di Stringhe

Diamo per scontato di utilizzare la convenzione del terminatore nullo: **si possono utilizzare sia funzioni di lettura e scrittura carattere per carattere (come negli esempi precedenti), che funzioni di lettura e scrittura di stringhe intere (scanf/printf, gets/puts).**

## Lettura di Stringhe con `scanf`

Per utilizzare la funzione `scanf` **si deve usare lo specificatore** di formato `%s` e **la variabile** da leggere deve essere il **nome di un vettore di caratteri**

- Non vanno utilizzate le parentesi quadre
- Non va utilizzata la `&`

La funzione `scanf`, utilizzata in questo modo, **legge** ciò che viene immesso da tastiera **fino al primo spazio o fine linea** (escluso)

- Non è quindi **adatta a** leggere **nomi composti** come "Pier Paolo"

**Esempio:**

```
#define MAX 20
char nome[MAX+1];
printf("Come ti chiami? ");
scanf("%s", nome);
```

## Lettura di Stringhe con `gets`

La funzione `gets` è **pensata appositamente per acquisire una stringa**.

Accetta **un parametro**, che corrisponde al **nome di un vettore di caratteri**

- Non utilizzare le parentesi quadre

**Legge tutto** ciò che viene immesso da tastiera, **compresi eventuali spazi, fino al fine linea** (escluso)

- È quindi **possibile leggere nomi composti**

**Esempio:**

```
#define MAX 20
char nome[MAX+1];
printf("Come ti chiami? ");
gets(nome);
```

## Scrittura di Stringhe con `printf`

Per utilizzare la funzione `printf` **si deve usare lo specificatore** di formato `%s` e la variabile da stampare deve essere il **nome di un vettore di caratteri**.

- Non vanno utilizzare le parentesi quadre

È **possibile combinare la stringa con altre variabili** nella stessa istruzione, ad esempio:

```
printf("Buongiorno, ");
printf("%s", nome);
printf("!\n");

// variante più compatta
printf("Buongiorno, %s!\n", nome);
```

## Scrittura di Stringhe con puts

La funzione `puts` è **pensata appositamente per stampare una stringa**.

La variabile da stampare deve essere il **nome di un vettore di caratteri**.

- Non vanno usate le parentesi quadre

Unico "svantaggio" è che questa funzione **va a capo automaticamente**, pertanto **non è possibile stampare altre informazioni sulla stessa riga**:

```
printf("Buongiorno, ");
puts(nome);
// Non c'è bisogno del \n visto che puts va a capo automaticamente
```

## Lunghezza di una Stringa

La lunghezza di una stringa si può determinare **ricercando la posizione del terminatore nullo**.

- **Esempio:**

```
#define MAX 10
char s[MAX+1];
int lun, i;

// Lettura stringa
...
for(i = 0; i != 0; i++){
    // Niente
}
lun = i;
```

Alternativamente è possibile utilizzare una libreria standard C, la `string.h`, che contiene la funzione `strlen`, la quale non fa altro che **calcolare la lunghezza della stringa passata come parametro**:

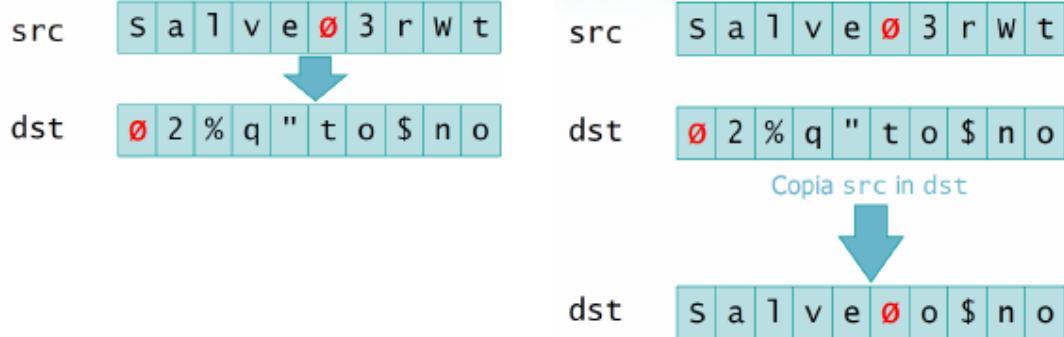
```
#include <string.h>
#define MAX 10
char s[MAX+1];
int lun;

// Lettura stringa
...
lun = strlen(s);
```

## Copia di una Stringa

L'operazione di copia prevede di **ricopiare il contenuto di una prima stringa sorgente in una seconda stringa destinazione**

```
char src[MAXS+1] ;  
char dst[MAXD+1] ;
```



Nella libreria standard C, includendo `string.h`, è disponibile la funzione `strcpy`, che **effettua la copia di stringhe**

- **Primo parametro:** stringa destinazione
- **Secondo parametro:** stringa sorgente

### Avvertenze

Nella **stringa destinazione** vi deve essere un **numero sufficiente di locazioni libere**:

- `MAXD+1 >= strlen(src) + 1`
  - dove `MAXD` è la **lunghezza della stringa destinazione** e `src` è la **stringa sorgente**

L'eventuale **contenuto della stringa destinazione viene sovrascritto**, mentre la stringa sorgente non viene modificata in alcun modo.

La `strcpy` **aggiunge automaticamente il carattere terminatore** alla fine della stringa copiata

## Concatenazione di Stringhe

L'operazione di concatenazione corrisponde a **creare una nuova stringa composta dai caratteri di una prima stringa seguiti dai caratteri della seconda stringa**



Per semplicità, in C l'operazione di concatenazione **scrive il risultato nello stesso vettore della prima stringa**, pertanto il valore precedente contenuto nella prima stringa viene perso.

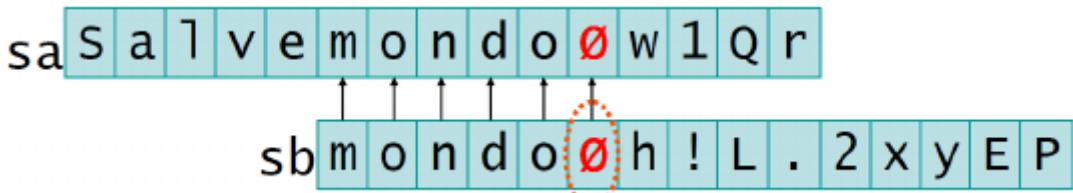
Per non perdere la prima stringa c'è bisogno di servirsi di stringhe temporanee e della funzione `strcpy`

## Algoritmo

1. Trova la fine della prima stringa



2. Copia la seconda stringa nel vettore della prima a partire dalla posizione del terminatore nullo (sovrascrivendolo)



```
#define MAX 20
char sa[MAX], sb[MAX];
int la, i;
// Lettura stringhe
...
la = strlen(sa);
for(i = 0; sb[i] != 0; i++){
    // Copia carattere per carattere
    sa[la+i] = sb[i];
}
// Aggiunta del terminatore
sa[la+i] = 0;
```

Alternativamente si può usare la **funzione** `strcat`, una funzione inclusa nell'header `string.h`, che **effetta la concatenazione automatica** di due stringhe

- **Esempio:**

```
#define MAX 20
char sa[MAX], sb[MAX];
// Lettura stringhe
...
strcat(sa, sb);
```

- **Funzionamento:**

- nella prima stringa vi deve essere un numero sufficiente di locazioni libere
  - $\text{MAX}+1 \geq \text{strlen(sa)} + \text{strlen(sb)} + 1$
- il contenuto precedente della prima stringa viene perso, mentre la seconda stringa non viene modificata
- il terminatore nullo deve essere aggiunto in coda alla prima stringa, cosa che la funzione `strcat` fa automaticamente

Per **concatenare 3 o più stringhe** occorre farlo **due a due**:

- `strcat(sa, sb)` e poi `strcat(sa, sc)`

È possibile **concatenare** anche **stringhe costanti**:

- `strcat(sa, "!")`

## Confronto di Stringhe

Il confronto di stringhe mira a determinare se:

- **le due stringhe sono uguali**, ovvero se hanno uguale lunghezza e sono composte dagli stessi caratteri nello stesso ordine
- **le due stringhe sono diverse**
- **la stringa *sa* precede la stringa *sb*** o viceversa
  - secondo l'ordine lessicografico imposto dal codice ASCII parzialmente compatibile con l'ordine alfabetico

## Confronto di Uguaglianza

Ogni carattere di ***sa*** deve essere uguale al carattere corrispondente di ***sb***, anche il terminatore nullo deve essere nella stessa posizione. I caratteri successivi al terminatore vanno ignorati

sa	S	a	l	v	e	Ø	o	4	d	1	Ø	w	1	Q	r
----	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---

sb	S	a	l	v	e	Ø	h	!	L	.	2	x	y	E	P
----	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---

```
#define MAX 20
char sa[MAX], sb[MAX];
int i, uguali;

...
// Si pone "uguale" a 1 perché si cerca la condizione sa[i] == sb[i]
uguali = 1;
// Cicla fino al terminatore di sa o sb (di fatto il ciclo termina al primo
terminatore trovato)
for(i = 0; sa[i] != 0 && sb[i] != 0; i++){
    // Verifica che i caratteri siano uguali, altrimenti poni a 0 la variabile
    "uguali"
    if(sa[i] != sb[i]){
        uguali = 0;
    }
}
// Arrivati a questo punto almeno una delle due stringhe è arrivata al
terminatore, pertanto, se per puro caso tutte le lettere analizzate fossero
risultate uguali, ora si analizza se, nella stessa posizione i, siano presenti i
terminatori in entrambe le stringhe
if(sa[i] != 0 || sb[i] != 0){
    // Se sa[i] e sb[i] sono diversi, vuol dire che una stringa è più lunga
    // dell'altra, quindi si pone "uguali" a 0
    uguali = 0;
}
```

## Confronto di Ordine

Verifichiamo se ***sa*** "è minore di" ***sb***.

Partendo con ***i = 0***:

- se ***sa[i] < sb[i]*** allora ***sa è minore***

- se `sa[i] > sb[i]` allora **sa non è minore**
- se `sa[i] == sb[i]` allora **bisogna controllare i caratteri successivi (i++)**

il terminatore nullo conta come "minore" di tutti

<code>sa</code>	S	a	1	v	e	Ø	o	4	d	1	Ø	w	1	Q	r
-----------------	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---

<code>sb</code>	s	a	1	u	t	e	Ø	!	L	.	2	x	y	E	P
-----------------	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---

```
#define MAX 20
char sa[MAX], sb[MAX];
int minore, i;

...
// Ricerca di esistenza della condizione sa[i] < sb[i]
minore = 0;
// Cicla fino al primo terminatore o finchè il valore "minore" non cambia, ovvero
// finchè non si scopre quale tra le due stringhe è "minore" dell'altra
for(i = 0; sa[i] != 0 && sb[i] != 0 && minore == 0; i++){
    if(sa[i] < sb[i]){
        // Se sa è minore di sb allora si pone "minore" a 1
        minore = 1
    }
    if(sa[i] > sb[i]){
        // Se sa NON è "minore" di sb allora si pone "minore" a -1
        minore = -1;
    }
}
// Ora dobbiamo verificare se le due stringhe hanno lunghezza diversa nel caso in
// cui tutti i caratteri fossero nelle stesse posizioni
if(minore == 0 && sa[i] == 0 && sb[i] != 0){
    minore = 1;
}
if(minore == 1){
    printf("%s è minore di %s\n", sa, sb);
}
```

Alternativamente si può usare la funzione `strcmp`, che fa parte della libreria standard C includendo l'header `string.h`, la quale **effettua il confronto di stringhe**.

- **Primo parametro:** prima stringa
- **Secondo parametro:** seconda stringa

Restituisce un valore:

- `< 0` se la **prima stringa è minore della seconda**
- `== 0` se le **stringhe sono uguali**
- `> 0` se la **prima stringa è maggiore della seconda**

Un modo per ricordarsi il significato di questi valori è immaginare che la `strcmp` esegua la sottrazione `sa - sb`:

- **negativo:** sa minore
- **positivo:** sa maggiore

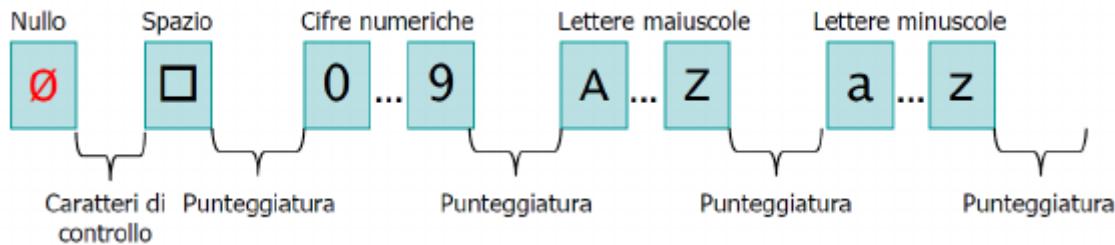
- **nullo**: uguali

```
#define MAX 20
char sa[MAX], sb[MAX];
int ris;
...
ris = strcmp(sa, sb);
if(ris < 0){
    // sa minore di sb
}
if(ris == 0){
    // sa uguale a sb
}
if(ris > 0){
    // sa maggiore di sb
}
```

## Ordinamento delle Stringhe

La funzione `strcmp` lavora **confrontando tra loro i codici ASCII dei caratteri**.

Il criterio di ordinamento è quindi **dato dalla posizione dei caratteri nella tabella ASCII**



Ogni lettera maiuscola precede ogni lettera minuscola:

- *Ciao* precede *ciao*
- *Zulu* precede *apache*

Gli spazi contano e precedono le lettere:

- *Qui Quo Qua* precede *QuiQuoQua*

I simboli di punteggiatura contano, ma non vi è una regola intuitiva. L'ordinamento che si ottiene è lievemente diverso da quello "standard" alfabetico

## Ricerca in una Stringa

È possibile concepire diversi tipi di ricerche da compiersi all'interno di una stringa:

1. verificare se un determinato carattere compare all'interno di una stringa data
2. determinare se una determinata stringa compare integralmente all'interno di un'altra stringa data, in una posizione arbitraria

## Ricerca di un Carattere

Supponendo che **s** sia una **stringa arbitraria** e che **ch** sia un **carattere qualsiasi**, *determinare se la stringa s contiene (una o più volte) il carattere ch al suo interno, in qualsiasi posizione*

```

#define MAX 20
char s[MAX], ch;
int trovato;
...
trovato = 0;
for(int i = 0; s[i] != 0 && trovato == 0; i++){
    if(s[i] == ch){
        trovato = 1;
    }
}

```

Alternativamente, includendo l'header `string.h`, abbiamo accesso alla funzione `strchr`, che **effettua la ricerca di un carattere**.

- **Primo parametro:** stringa in cui cercare
- **Secondo parametro:** carattere da cercare

Restituisce:

- `valore != NULL` se il **carattere è presente** nella stringa
- `valore == NULL` se il **carattere non è presente** nella stringa

```

#define MAX 10
char s[MAX], ch;
...
if(strchr(s, ch) != NULL){
    // Il carattere è presente nella stringa
}

```

## Ricerca di una Sotto-Stringa

Supponendo che **s** sia una **stringa arbitraria** e che **r** sia la **stringa da cercare**, determinare se la stringa **s** contiene (una o più volte) la stringa **r** al suo interno, in qualsiasi posizione

```

#define MAX 20
char s[MAX], r[MAX];
int lr, ls, pos, i, trovato, diversi;
...
ls = strlen(s);
lr = strlen(r);
trovato = 0;
for(pos = 0; pos <= ls - lr; pos++){
    // Confronta r [0...lr - 1] con s [pos...pos + lr -1]
    diversi = 0;
    for(i = 0; i < lr; i++){
        if(r[i] != s[pos + i]){
            diversi = 1;
        }
        if(diversi == 0){
            trovato = 1;
        }
    }
}
if(trovato == 1){
    // Trovato
}

```

Più convenientemente è possibile usare la funzione `strstr`, compresa nell'header `string.h` e che **effettua proprio la ricerca di una sottostringa**

- **Primo parametro:** stringa in cui cercare
- **Secondo parametro:** sotto-stringa da cercare

Restituisce:

- `valore != NULL` se la **sotto-stringa è presente**
- `valore == NULL` se la **sotto-stringa non è presente**

```
#define MAX 10
char s[MAX], r[MAX];
...
if(strstr(s, r) != NULL){
    // Sotto-stringa trovata
}
```

# Stringhe: Approfondimento

---

## Sommario

- Funzioni sulle stringhe
- Vettori di stringhe

## Funzioni di Libreria

---

Quando è possibile è sempre meglio usare le **funzioni di libreria**

- Sono **più veloci**
- Sono **maggiormente collaudate**

## Funzioni sui Caratteri

Sono definite nell'header `ctype.h`

- `isalpha`
- `isupper`
- `islower`
- `isdigit`
- `isalnum`
- `isxdigit`
- `ispunct`
- `isgraph`
- `isprint`
- `isspace`
- `iscntrl`

Analizzano un singolo carattere identificandone la tipologia.

- Lettera
  - Maiuscola
  - Minuscola
- Cifra
- Punteggiatura

### Funzioni `isalpha/isdigit`

- `isalpha`
  - **Libreria:** `ctype.h`
  - **Parametri:** un carattere `ch`
  - **Restituisce:** vero/falso
  - **Descrizione:** Restituisce vero se il carattere `ch` è una lettera maiuscola o minuscola, falso altrimenti
  - **Esempio:**

```
if(isalpha(ch)){
    // istruzioni
}
```

- `isidigit`

- **Libreria:** `ctype.h`
- **Parametri:** un carattere `ch`
- **Restituisce:** vero/falso
- **Descrizione:** Restituisce vero se il carattere `ch` è una cifra numerica, falso altrimenti
- **Esempio:**

```
if(isdigit(ch)){
    // istruzioni
}
```

## Funzioni `isupper/islower`

- `isupper`

- **Libreria:** `ctype.h`
- **Parametri:** un carattere `ch`
- **Restituisce:** vero/falso
- **Descrizione:** Restituisce vero se il carattere `ch` è una lettera maiuscola, falso altrimenti
- **Esempio:**

```
if(isupper(ch)){
    // istruzioni
}
```

- `islower`

- **Libreria:** `ctype.h`
- **Parametri:** un carattere `ch`
- **Restituisce:** vero/falso
- **Descrizione:** Restituisce vero se il carattere `ch` è una lettera minuscola, falso altrimenti
- **Esempio:**

```
if(islower(ch)){
    // istruzioni
}
```

## Funzioni `isalnum/isxdigit`

- `isalnum`

- **Libreria:** `ctype.h`
- **Parametri:** un carattere `ch`
- **Restituisce:** vero/falso

- **Descrizione:** restituisce vero se il carattere `ch` è una lettera o una cifra numerica, falso altrimenti.
- **Equivale a:** `isalpha(ch) || isdigit(ch)`
- **Esempio:**

```
if(isalnum(ch)){
    // istruzioni
}
```

- `isxdigit`
  - **Libreria:** `ctype.h`
  - **Parametri:** un carattere `ch`
  - **Restituisce:** vero/falso
  - **Descrizione:** Restituisce vero se il carattere `ch` è una cifra numerica oppure una lettera valida in base 16 (a...f, A...F), falso altrimenti
  - **Esempio:**

```
if(isxdigit(ch)){
    // istruzioni
}
```

## Funzioni `ispunct/isgraph`

- `ispunct`
  - **Libreria:** `ctype.h`
  - **Parametri:** un carattere `ch`
  - **Restituisce:** vero/falso
  - **Descrizione:** restituisce vero se il carattere `ch` è un simbolo di punteggiatura, falso altrimenti
  - **Esempio:**

```
if(ispunct(ch)){
    // istruzioni
}
```

- `isgraph`
  - **Libreria:** `ctype.h`
  - **Parametri:** un carattere `ch`
  - **Restituisce:** vero/falso
  - **Descrizione:** restituisce vero se il carattere `ch` è un qualsiasi simbolo visibile (lettera, cifra, punteggiatura), falso altrimenti
  - **Esempio:**

```
if(isgraph(ch)){
    // istruzioni
}
```

## Funzioni `isprint`/`isspace`

- `isprint`
  - **Libreria:** `ctype.h`
  - **Parametri:** un carattere `ch`
  - **Restituisce:** vero/falso
  - **Descrizione:** restituisce vero se il carattere `ch` è un qualsiasi simbolo visibile oppure uno spazio, falso altrimenti
  - **Esempio:**

```
if(isprint(ch)){  
    // istruzione  
}
```

- `isspace`
  - **Libreria:** `ctype.h`
  - **Parametri:** un carattere `ch`
  - **Restituisce:** vero/falso
  - **Descrizione:** restituisce vero se il carattere `ch` è invisibile (spazio, tab, a capo), falso altrimenti
  - **Esempio:**

```
if(isspace(ch)){  
    // istruzioni  
}
```

## Funzione `iscntrl`

- **Libreria:** `ctype.h`
- **Parametri:** un carattere `ch`
- **Restituisce:** vero/falso
- **Descrizione:** restituisce vero se `ch` è un carattere di controllo (ASCII 0...31, 127), falso altrimenti
- **Esempio:**

```
if(iscntrl(ch)){  
    // istruzioni  
}
```

## Funzioni `toupper`/`tolower`

- `toupper`
  - **Libreria:** `ctype.h`
  - **Parametri:** un carattere `ch`
  - **Restituisce:** `char` maiuscolo

- **Descrizione:** se `ch` è una lettera minuscola allora restituisce l'equivalente maiuscolo, altrimenti restituisce `ch` stesso

- **Esempio:**

```
for(int i = 0; s[i] != 0; i++){
    s[i] = toupper(s[i]);
}
```

- `tolower`

- **Libreria:** `ctype.h`

- **Parametri:** un carattere `ch`

- **Restituisce:** `char` minuscolo

- **Descrizione:** se `ch` è una lettera maiuscola allora restituisce l'equivalente minuscolo, altrimenti restituisce `ch` stesso

- **Esempio:**

```
for(int i = 0; s[i] != 0; i++){
    s[i] = tolower(s[i]);
}
```

## Funzioni `strcpy`/`strncpy`

- `strcpy`

- **Libreria:** `string.h`

- **Parametri:** stringa destinazione `dst`, stringa sorgente `src`

- **Descrizione:** Copia il contenuto di `src` all'interno di `dst` (che deve avere lunghezza sufficiente)

- **Esempio:**

```
strcpy(s1, s2);
strcpy(s, "");
strcpy(s1, "ciao");
```

- `strncpy`

- **Libreria:** `string.h`

- **Parametri:** stringa destinazione `dst`, stringa sorgente `src`, numero max caratteri `n`

- **Descrizione:** copia `n` caratteri da `src` a `dst`

- **Esempio:**

```
strncpy(s1, s2, 20);
strncpy(s1, s2, MAX);
```

## Funzioni `strcat`/`strncat`

- `strcat`

- **Libreria:** `string.h`
- **Parametri:** stringa destinazione `dst`, stringa sorgente `src`
- **Descrizione:** accoda il contenuto di `src` alla fine di `dst` (che deve avere lunghezza sufficiente)
- **Esempio:**

```
strcat(s1, s2);
strcat(s1, " ");
```

- `strncat`

- **Libreria:** `string.h`
- **Parametri:** stringa destinazione `dst`, stringa sorgente `src`, numero max caratteri `n`
- **Descrizione:** accoda `n` caratteri di `src` alla fine di `dst`
- **Esempio:**

```
strncat(s1, s2, MAX);
```

## Funzioni `strcmp`/`strncmp`

- `strcmp`

- **Libreria:** `string.h`
- **Parametri:** stringa `s1`, stringa `s2`
- **Restituisce:** risultato confronto come `int`
- **Descrizione:**
  - **risultato < 0** se `s1` precede `s2`
  - **risultato == 0** se `s1` è uguale a `s2`
  - **risultato > 0** se `s1` segue `s2`
- **Esempio:**

```
if(strcmp(s, r) == 0){
    // istruzioni
}
while(strcmp(r, fine) != 0){
    // istruzioni
}
```

- `strncmp`

- **Libreria:** `string.h`
- **Parametri:** stringa `s1`, stringa `s2`, numero max caratteri `n`
- **Restituisce:** risultato confronto come `int`
- **Descrizione:** funziona come `strcmp` ma confronta solo i primi `n` caratteri, ignorando i successivi

- o **Esempio:**

```
if(strncmp(r, "buon", 4) == 0){
    // istruzioni
}
```

## Funzioni atoi/ atof

- **atoi**

- o **Libreria:** `stdlib.h`
- o **Parametri:** una stringa `s`
- o **Restituisce:** valore estratto come `int`
- o **Descrizione:** analizza la stringa `s` ed estraе il valore intero in essa contenuto (a partire dai primi caratteri)
- o **Esempio:**

```
n = atoi(s);
n = atoi("232abc");
```

- **atof**

- o **Libreria:** `stdlib.h`
- o **Parametri:** una stringa `s`
- o **Restituisce:** valore estratto come `double` o `float`
- o **Descrizione:** analizza la stringa `s` ed estraе il valore reale in essa contenuto (a partire dai primi caratteri)
- o **Esempio:**

```
x = atof(s);
x = atof("2.32abc");
```

## Matrici di Caratteri

Nel definire una matrice è ovviamente possibile usare il tipo base `char`.

Permette di memorizzare una tabella  **$N \times M$**  di caratteri ASCII, la quale, in ogni posizione `[i][j]` **deve contenere un carattere**:

- non può essere vuota
- non può contenere più di un carattere

```
char tris[3][3] ;
```

	0	1	2
0	O	X	.
1	.	X	.
2	.	.	.

## Vettori di Stringhe

Una matrice di caratteri può anche essere vista come:

- un **vettore di caratteri**
- un **vettore di stringhe**

Si tratta di un metodo diverso di interpretare la stessa struttura dati

**char nomi[5][10];**

	0	1	2	3	4	5	6	7	8	9
0	F	u	l	v	i	o	\0	x	!	w
1	A	n	t	o	n	i	o	\0	.	Z
2	C	r	i	s	t	i	n	a	\0	u
3	E	l	e	n	a	\0	5	g	r	d
4	D	a	v	i	d	e	\0	\$	2	e

## Qualche Esempio

La **stampa del contenuto di un vettore di stringhe** si ottiene semplicemente **stampando ciascuno degli elementi**. Per farlo si possono utilizzare `puts` o `printf`

```
for(int i = 0; i < N; i++){
    puts(vet[i]);
}
```

Acquisire da tastiera un vettore di stringhe:

- un ciclo per ciascuna delle stringhe da leggere
- lunghezza nota a priori
- lunghezza determinata dalla lettura di un certo dato (es. "FINE").

```
char vet[MAX] [LUN+1];
char s[MAX];
int N, end;
...
do{
    printf("Quante stringhe? ");
    gets(s);
    N = atoi(s);
    if(N < 1 || N > MAX){
        printf("Valore errato: deve essere tra 1 e %d\n", MAX);
    }
} while(n < 1 || n > MAX);

N = 0;
end = 0;
do{
    printf("Stringa %d: ", N+1);
    gets(s);
    if(strlen(s) == 0){
        printf("Vuota. Ripeti.\n");
    } else if (strlen(s) > LUN){
        printf("Troppo lunga.\n");
    }
}
```

```
    } else if (strcmp(s, "FINE") == 0){
        end = 1;
    } else{
        strcpy(vet[N], s);
        N++;
    }
} while(end == 0);
```

## Errore Frequenti

Confondere una **stringa (vettore di caratteri)** con un **vettore di stringhe (matrice di caratteri)**

```
char s[LUN+1];
// s[i] è un singolo carattere
// s è l'intera stringa
char v[MAX][LUN+1];
// v[i][j] è il singolo carattere
// v[i] è un'intera stringa
// v è l'intera matrice
```

# Linguaggio C: i Record

## Sommario

- I tipi enumerativi
- I record
- I tipi definiti dall'utente

## I Limiti del Tipo Array

Il termine **tipo aggregato** si riferisce ai **vettori** e ai **tipi struct**:

- un **vettore** è un **raggruppamento di variabili dello stesso tipo**
- una **struct** è un **raggruppamento di variabili anche di tipo diverso**

## I Record

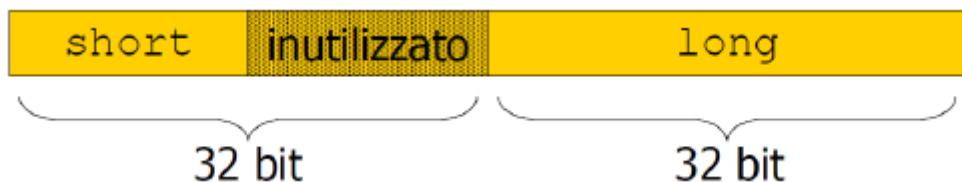
Insieme di più variabili denominate **membri** in genere di **tipo diverso** identificate da un **nome comune (tag)**.

In memoria i membri sono **allocati contiguamente** e nello **stesso ordine di dichiarazione**.

Tra un membro e il successivo **possono esserci** (dipendente dal tipo di microprocessore) **spazi intermedi di allineamento** della memoria:

- **non indirizzabili** (inutilizzabili)
- dal **contenuto indefinito**

Ad esempio, supponendo uno `short` su 16 bit e un `long` su 32 bit in una macchina con **allineamento a 32 bit**:



Non c'è mai spazio di allineamento prima del primo membro, quindi l'indirizzo di una variabile `struct` **coincide con quello del suo primo membro**.

## Dichiarazione di Struct

```
struct nomeTag{  
    tipo1 membro1;  
    tipo2 membro2;  
    ...  
};
```

La dichiarazione **non riserva memoria**, ma **crea un nuovo tipo di dato**.

Due struct completamente uguali a **meno del tag** sono considerate di **tipo diverso**.

La dichiarazione di una **struct anonima (senza tag)** è sempre considerata avente tipo diverso da ogni altra struct (con tag o anonima che sia), anche se ha gli stessi membri.

## Definizione di Variabili Struct

Lo **scope** del nome dei membri è **confinato alla sola struttura dove sono dichiarati**: le variabili del programma e i membri di altre strutture possono avere gli stessi nomi di un membro.

- **Esempio:**

```
struct punto{  
    int x;  
    int y;  
};
```

dove:

- punto è il **tag**
- x e y sono due **membri scalari** di tipo **int**

Dichiarare una variabile **struct** **riserva memoria** e ha la consueta forma:

- **tipo, var1, var2...**

dove, però, **tipo** è una **struct**.

La **definizione di variabili** può essere **contestuale alla dichiarazione del tipo** (il tag può essere omesso se non serve definire in seguito altre variabili di questo tipo).

- **Esempio:**

```
struct punto{  
    int x;  
    int y;  
} pt1, pt2, pt3; // 3 variabili
```

## Operazioni su Struct

La definizione può essere separata dalla dichiarazione del tipo (il tag non può essere omesso in questo caso):

```
struct punto{  
    int x;  
    int y;  
};  
struct punto pt4, pt5, pt6;
```

In entrambi i casi è possibile inizializzare una variabile **struct**:

- con **valori costanti (tra parentesi graffe)**
- mediante **assegnazione di un'altra variabile struct dello stesso tipo** o **chiamando una funzione che restituisca una struct dello stesso tipo**, nel caso di variabili automatiche

**Esempio:**

```
struct punto pt7 = {12, 14};  
struct punto pt8 = pt7;  
struct punto pt9 = creapunto(5, 7);
```

Per accedere ai singoli membri di una variabile di tipo `struct` si usa la forma:

- `nomeVar.nomeMembro`

dove, attenzione, `nomevar` è il **nome della variabile**, NON quello del tag.

È possibile assegnare un valore ad un membro scalare:

- `pt1.x = 4`

L'assegnazione di una variabile `struct` ad un'altra avviene mediante **copia del contenuto** (non del puntatore). L'assegnazione è possibile **solo se sono dello stesso tipo**:

- `pt1 = pt2`

**Definizione di un puntatore a struct:**

- `struct punto *p`

**Determinazione dell'indirizzo** di una variabile di tipo `struct`:

- `p = &pt4`

L'operatore `.` ha una **priorità maggiore** dell'operatore di deriferimento `*`, quindi, per indicare il membro `x` della variabile di tipo `struct` puntata da `p`, **servono le parentesi**:

- `(*p).x = 12`

Questo perchè `*p.x` equivale a `*(p.x)`, ovvero l'oggetto puntato da `x` (se fosse un puntatore).

Per non sbagliarsi, **si preferisce** scrivere `*(p).x` mediante **l'operatore freccia**:

- `p -> x`

Quindi si ha `p -> x = 12`.

- La priorità dell'operatore `->` è la più alta in assoluto, quindi `++p -> x` equivale a `++(p -> x)`, ovvero incrementa `x`, non `p`.

Per determinare l'indirizzo di un membro si usa `&pt4.y` in quanto `&` ha **priorità inferiore** a `.`

## Membri di una Struct

I membri possono essere di **tipo scalare, o aggregato** (vettoriale, altre struct). L'inizializzazione avviene come già indicato, le parentesi graffe interne possono essere tralasciate (vedi inizializzazione matrice).

```
struct rettangolo{
    struct punto bassoSinistra;
    struct punto altoDestra;
} rett = {{2,3}, {12,9}};
```

L'accesso ai membri interni richiede l'indicazione del "percorso" da seguire:

- `rett.altoDestra.x = 14`
  - `rett` è la **variabile**
  - `altoDestra` è il **membro di** `rett`
  - `x` è il **membro di** `altoDestra`

# Campi di Bit

Sono insiemi di bit che costituiscono un valore di tipo intero (**signed o unsigned**):

```
struct cartaDaGioco{  
    unsigned valore : 4;  
    unsigned seme : 2;  
    unsigned colore : 1;  
};
```

Il numero intero a destra di ciascun membro **indica**, per ciascun campo, **da quanti bit esso sia costituito**.

I singoli campi **si comportano come valori interi** e quindi possono comparire in espressioni, essere assegnati, confrontati ecc.

La **dimensione massima** di ciascun campo è la **dimensione di un int** (**caratteristica non portabile**, inoltre **dipendente dal compilatore**).

I campi vengono accorpati a costituire **gruppi di byte** delle dimensioni di un **int** (non è specificato se da sinistra a destra o viceversa).

Non si può determinare il puntatore/offset di un campo di bit (può essere in mezzo ad un byte).

Un **campo anonimo** (*senza nome della variabile*) può servire come **riempitivo (padding)** con quel numero di bit, ma **non può essere utilizzato per contenere valori**. Un campo anonimo con dimensione 0 **forza l'allineamento di memoria** del membro seguente al successivo **int**:

```
struct cartaDaGioco{  
    // Allocati nel primo int  
    unsigned valore : 4;  
    unsigned : 5;  
    unsigned seme : 2;  
    unsigned : 0;  
    // Allocato nel secondo int  
    unsigned colore : 1;  
};
```

- Esempio:

```
struct switching{  
    unsigned light : 1;  
    unsigned fridge : 1;  
    int count;  
    // 4 byte  
    unsigned stove : 4;  
    unsigned : 4;  
    unsigned radio : 1;  
    unsigned : 0;  
    unsigned flag : 1;  
} onoffpower;
```

Member Name	Storage Occupied	Total	Total
light	1 bit	1 bit	32 bits
fridge	1 bit	1 bit	
(padding up to 30 bits)	To the next int boundary	30 bits	
count	The size of an int (4 bytes)	$4 \times 8 = 32$ bits	32 bits
stove	4 bits	4 bits	32 bits
(unnamed field)	4 bits	4 bits	
radio	1 bit	1 bit	
(padding up to 23 bits)	To the next int boundary (unnamed field)	23 bits	
flag	1 bit	1 bit	32 bits
(padding up to 31 bits)	To the next int boundary	31 bits	
	16 bytes = 64 bits	$4 \times 32$ bits = 128 bits	128 bits

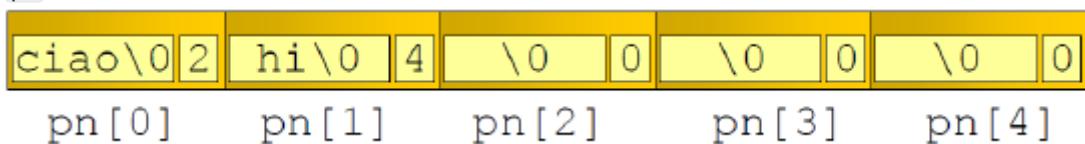
## Vettori di Struct

Ogni elemento di un vettore di `struct` è una variabile di tipo `struct`:

```
struct numParole{
    char parola[20];
    int num;
} pn[5] = {{"ciao",2}, {"hi",4}};
```

Il vettore `pn` ha 5 elementi, ciascuno è una `struct numParole`, i primi 2 elementi sono inizializzati, i successivi sono `""` e `0` (le graffe interne possono essere omesse).

- `pn`:



## Confronto di Variabili Struct

Per verificare se due variabili dello stesso tipo `struct` sono uguali (stesso contenuto), **si deve confrontare ciascun membro**:

```

if(pt1.x == pt2.x && pt1.y == pt2.y){
    // istruzioni
}

// NON si può fare invece:
if(pt1 == pt2){
    // istruzioni
}

```

## Tipo union

Permette di definire una variabile che può contenere un solo elemento, ma di tipo diverso.

La dichiarazione è simile alle `struct`, ma **solo un membro per volta tra quelli indicati nella dichiarazione può essere in uso**.

- **Esempio:**

```

union tris{
    int x;
    double y;
    char nome[10];
} var;

```

In questo esempio la variabile `var` è considerata di tipo `int` **se viene usata come `var.x`**, di tipo `double` **se usata come `var.y`** e `stringa di 10 caratteri` **se usata come `var.nome`**.

Sta al programmatore mantenere memoria del tipo attuale di `var` e usarla coerentemente.

L'utilizzo inizia con un'assegnazione:

```

alfa.y = 23.23;
// Da questo punto alfa è di tipo double e non esistono né alfa.x né alfa.nome

alfa.x = 12;
// Da questo punto alfa è di tipo int e non esistono né alfa.y né alfa.nome

strcpy(alfa.nome, "ciao");
// Da questo punto alfa è un vettore di char e non esistono né alfa.x né alfa.y

```

## Operatore `typedef`

**Dichiara il nome di un nuovo tipo di dato** (in realtà un'*abbreviazione*) **a partire da altri tipi** (scalari, aggregati ecc.).

- `typedef tipoEsistente nuovoTipo`

La **dichiarazione di tipo è identica alla definizione di una variabile**, ma è preceduta dalla clausola `typedef`.

Per comprendere correttamente che cosa produce una dichiarazione `typedef` è utile pensare al tipo che avrebbe la variabile se non ci fosse `typedef` e poi considerare che il nome della variabile è in realtà il nome del nuovo tipo

- **Esempio:**

```
typedef char string[80];
```

Se non ci fosse `typedef`, `string` sarebbe una **variabile di tipo vettore di 80 char**, mentre, grazie a `typedef`, `string` è **il tipo vettore di 80 char**. Quindi:

- o `string parola`

definisce la variabile `parola` di tipo `string`, cioè di tipo `char[80]`

- **Esempio:**

```
typedef char *strup;
```

Dichiara il tipo `strup` come **puntatore a char**, quindi:

- o `strup par`

definisce la variabile `par` di tipo `strup`, cioè `char*`

I nomi dei tag di `struct` e `union` possono essere omessi quando li si usa soltanto nella `typedef` (ovvero se la scrittura `struct TAG` non compare altrove nel programma)

```
typedef struct rett{  
    struct punto bassoSinistra;  
    struct punto altoDestra;  
} rettangolo;
```

Dichiara il tipo `rettangolo` come `struct rett`, quindi:

- `rettangolo r = {{2, 3}, {12, 9}}`

definisce la variabile `r` di tipo `struct rett`.

```
typedef struct{  
    int x;  
    int y;  
} vpunti[10];
```

Dichiara il tipo `vpunti` come **vettore di 10 elementi** di una struttura `struct` dichiarata (**anonima**), ovvero alla quale **non è assegnato un nome**.

- `vpunti vett`

Definisce la variabile `vett` di tipo `vpunti`, ovvero un **vettore di 10 struct**, utilizzabile come:

- `vett[0].x = 12`

I nomi dei nuovi tipi non devono essere nomi utilizzati da altri identificatori.

I nomi dei tag sono **scorrelati dai nomi di variabili, costanti, tipi, ecc.** (teoricamente appartengono a name space diversi), quindi **è possibile dichiarare un nome di tipo con lo stesso nome di un tag**:

```
typedef struct rett{  
    int x;  
    int y;  
} rett;
```

Si preferisce utilizzare un nome di tipo con iniziale maiuscola (`T`) o terminante con `_t` (`rett_t`) come d'uso nella libreria standard.

L'operatore `typedef` viene spesso utilizzato per "nascondere" come il compilatore realizza internamente una certa funzionalità, fornendo al programmatore un comportamento standard. Questo si traduce in una **migliore portabilità del codice**: indipendenza dalla piattaforma hardware, dal sistema operativo, dal compilatore ecc.

Ad esempio, per qualsiasi compilatore ANSI C il tipo `size_t` è sempre il tipo più appropriato (per quella combinazione hardware/OS/compilatore) per memorizzare la dimensione in byte di una variabile o la lunghezza di una stringa. Internamente il compilatore potrebbe usare un `unsigned int` o un altro `long`, ma usando `size_t` non ci si deve preoccupare di questi dettagli.

Il valore restituito **richiede un cast** per l'assegnazione ad una variabile di altro tipo:

```
int len;
len = (int)strlen(stringa);
```

È più chiaro dichiarare esternamente (anche mediante `include`) con `typedef` quei nuovi tipi che verranno utilizzati in più funzioni.

Questo **non è strettamente necessario** in quanto la compatibilità di tipo di due variabili viene determinata "smontando" la dichiarazione `typedef` nella sua struttura basata sui tipi primitivi:

```
typedef int *intptr;
intptr p;
int *q;
q = p; // Lecito perchè sono dello stesso tipo
```

Quando un tipo è dichiarato con `typedef` su strutture aggregate anonime (`struct` e `union` senza tag), le variabili di quel nuovo tipo **sono considerate dello stesso tipo** (idealmente, nell'operazione di "smontaggio", le strutture aggregate anonime ricevono lo stesso **tag fittizio**, diverso per ogni `typedef`)

```
typedef struct{
    int a;
    int b;
} Miastruct;

Miastruct a = {0, 0};
Miastruct b;
b = a; // Lecito perchè sono dello stesso tipo
```

## `typedef` e `const`

Se il tipo `T` è dichiarato con una `typedef`, la posizione della `const` **non è significativa** perché il modificatore `const` si applica all'intera `typedef`, quindi **le due definizioni seguenti sono equivalenti**:

```
const T var;
T const var;
```

Se, ad esempio, `T` è dichiarato come:

- `typedef int* T`

entrambe le definizioni sono equivalenti a:

- `int* const var`

## Attenzione

Se, invece, `T` viene definito con una `define`, allora `T` non è un vero nuovo tipo e la posizione della `const` diventa significativa.

Se, ad esempio, `T` è definito come:

- `#define int* T`

la definizione `const T var` equivale a:

- `const int* var`

mentre la definizione `T const var` equivale a:

- `int* const var`

che non è equivalente alla precedente.

## Operatore `sizeof`

Restituisce il numero di byte di cui è composto un tipo di dato o una variabile (scalare o aggregata).

È un operatore (*non una funzione*) e viene valutato in fase di compilazione, può essere usato in una `define`

- Sintassi:

```
sizeof(nomeTipo);  
sizeof nomeVariabile;
```

Le parentesi sono necessarie se si indica il nome di un tipo di dato, facoltative se si indica il nome di una variabile

- Esempio:

```
sizeof(double); // Restituisce il numero di byte richiesti da double  
sizeof pt1; // Restituisce il numero di byte richiesti da struct punto
```

Il tipo del valore restituito è `size_t`.

Il valore restituito richiede un cast per l'assegnazione ad una variabile, in quanto `size_t` potrebbe essere in realtà un `unsigned long` (genera un warning):

```
int len;  
len = (int)sizeof(double);
```

Definendo la variabile `vet` come:

```
struct x{  
    int b;  
} vet[10], s;
```

`sizeof` produce i seguenti valori:

```
n = sizeof vet; // Restituisce il numero di byte richiesti da un vettore di 10  
n = sizeof vet / sizeof(struct x); // Restituisce il numero di elementi del  
vettore vet  
n = sizeof vet / sizeof s; // Restituisce il numero di elementi del vettore vet
```

# Record: Ulteriori Informazioni

Si vuole realizzare un tipo `struct` utilizzato per informazioni su operazioni di vendita, aventi i seguenti campi:

- **codice**: numero **intero** indicante il codice di riferimento dell'articolo venduto
- **nome**: **stringa** di lunghezza inferiore a 20 caratteri
- **prezzo**: numero **reale** (`float`) corrispondente al prezzo unitario dell'articolo
- **n.pezzi**: numero **intero** di pezzi venduti

Di tale tipo si vogliono successivamente dichiarare due variabili scalari, `x` e `y`, e un vettore `v` di **dimensione 100**.

Si definiscano in C il tipo di `struct` e le variabili `x`, `y`, `v` utilizzando le varie alternative possibili.

- **Schema di dichiarazione 1**: dichiarazione del tipo `struct operazione`, definizione delle variabili utilizzando il tipo `struct operazione`

```
struct operazione{
    int codice;
    char nome[20];
    float prezzo;
    int npezzi;
};

...
struct operazione x, y, v[100];
```

- **Schema di dichiarazione 2**: dichiarazione contestuale di tipo e variabili

```
struct operazione{
    int codice;
    char nome[20];
    float prezzo;
    int npezzi;
} x, y, v[100];
```

- **Schema di dichiarazione 3**: dichiarazione contestuale di tipo e variabili, senza nome per il tipo `struct`

```
struct{
    int codice;
    char nome[20];
    float prezzo;
    int npezzi;
} x, y, v[100];
```

- **Schema di dichiarazione 4**: generazione del tipo `operazione_t` mediante `typedef`

```

typedef struct operazione{
    int codice;
    char nome[20];
    float prezzo;
    int npezzi;
} operazione_t;
operazione_t x, y, v[100];

```

- **Schema di dichiarazione 4.1:** generazione del tipo `operazione_t` mediante `typedef`

```

struct operazione{
    int codice;
    char nome[20];
    float prezzo;
    int npezzi;
};
typedef struct operazione operazione_t;
operazione_t x, y, v[100];

```

## Errori di Dichiarazione

```

struct operazione{
    int codice = 201;
    char nome[20] = "video al plasma";
    float prezzo = 634.5;
    int npezzi = 7;
} op;

```

Il **tipo di inizializzazione** proposta (nella definizione di `struct`) **non è legale**.

È invece possibile inizializzare esplicitamente la variabile `op` come riportato di seguito:

```

struct operazione{
    int codice;
    char nome[20];
    float prezzo;
    int npezzi;
} op = {201, "video al plasma". 634.5, 7};

```

```

struct{
    int x,
    int y,
    int z;
} a, b;

```

La dichiarazione multipla di campi dello stesso tipo **non è corretta**. Sono possibili **due schemi alternativi**:

1. `struct{`  
 `int x;`  
 `int y;`  
 `int z;`  
`} a, b;`

```
2. struct{
    int x, y, z;
} a, b;
```

## Definizione di Record

Si vuole realizzare un tipo `struct` adatto a contenere le informazioni anagrafiche di persone (cognome, nome, data di nascita, indirizzo di residenza). In particolare, i campi relativi a cognome e nome possono essere realizzati come **stringhe di lunghezza inferiore a 30 caratteri**, mentre la data di nascita e l'indirizzo di residenza sono **sotto-strutture** realizzate, a loro volta, da campi:

- **data**: giorno, mese, anno (**interi**)
- **indirizzo**: via (**stringa**), numero civico (**intero**), codice postale (**stringa di 5 caratteri**), città (**stringa di meno di 30 caratteri**), provincia (**due caratteri**)

Si definiscano in C i tipi `struct` adatti a rappresentare tali dati (si realizzino i 3 tipi definendoli mediante costrutto `typedef`)

```
#define MAXS 30
#define NCAP 6
#define NPROV 3

typedef struct{
    int g, m, a;
} data_t;

typedef struct{
    char via[MAXS];
    int numero;
    char cap[NCAP], citta[MAXS], prov[NPROV];
} indirizzo_t;

typedef struct{
    char cognome[MAXS], nome[MAXS];
    data_t dataDiNascita;
    indirizzo_t indirizzo;
} persona_t;
```

Si ridefinisce ora la struttura precedente utilizzando un'**unica definizione**:

```
#define MAXS 30
#define NCAP 6
#define NPROV 3

typedef struct{
    char cognome[MAXS], nome[MAXS];
    struct{
        int g, m, a;
    } dataDiNascita;
    struct{
        char via[MAXS];
        int numero;
        char cap[NCAP], citta[MAXS], prov[NPROV];
    } indirizzo;
} persona_t;
```



# [Lezione 15] Linguaggio C: i Puntatori

## Sommario

- Puntatori
- Aritmetica dei puntatori
- Assegnamento dei puntatori
- Allocazione e deallocazione memoria

## Puntatori

### Introduzione

I puntatori sono un tipo di variabile che **contiene un indirizzo di una locazione di memoria**, ovvero l'indirizzo di un'altra variabile.

```
int* pippo;
// pippo è una variabile di tipo int*, ovvero un tipo "puntatore ad intero"
int *pippo;
// *pippo è il valore puntato da pippo ed è una "espressione di tipo intero"
```

### Sintassi

Entrambi possono essere **sia letti che assegnati** (possono comparire da entrambi i lati di un assegnamento)

```
pippo;      // Valore del puntatore
*pippo;     // Valore dell'oggetto puntato
```

### Preambolo

Cosa succede normalmente:

- **il compilatore assegna** alla variabile `pippo` **una locazione di memoria** (ad esempio la locazione `0x612A22C`)
- Inoltre **riserva 4 byte di memoria** per la variabile `pippo` (la dimensione di un `int`)

### Significato

```
int* pippo;
```

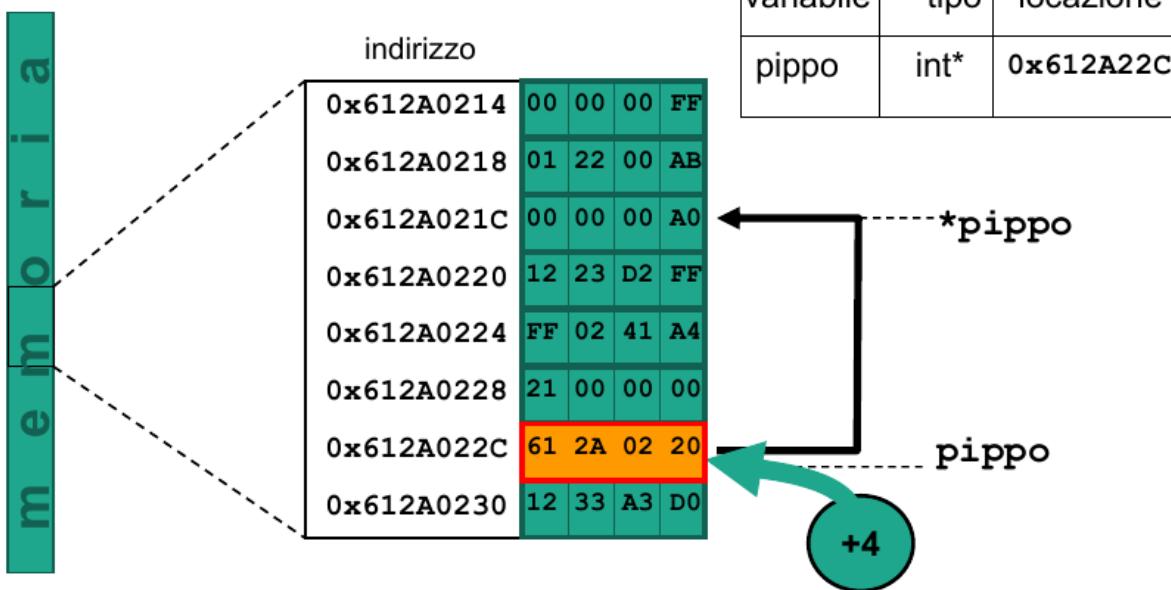


**pippo** (il **puntatore** stesso) vale... 0x612A021C

**\*pippo** (la **variabile puntata** da pippo) vale... 0x000000A0

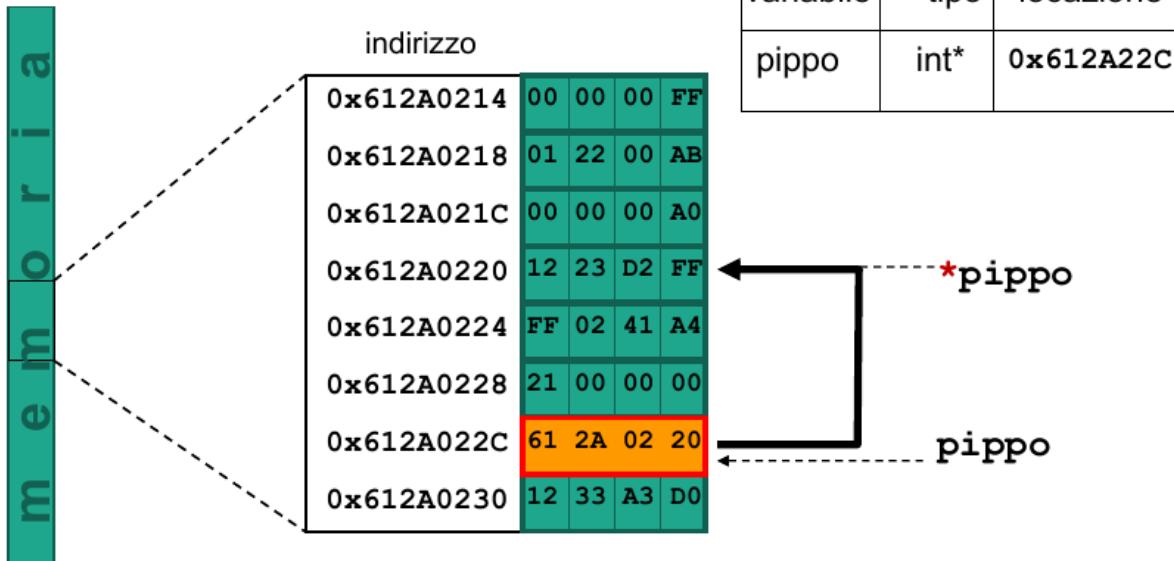
## Cambiare il Valore del Puntatore

```
pippo++;
```



- **pippo** (il **puntatore**) vale 0x612A021C
- **\*pippo** (la **variabile puntata**) vale 0x000000A0

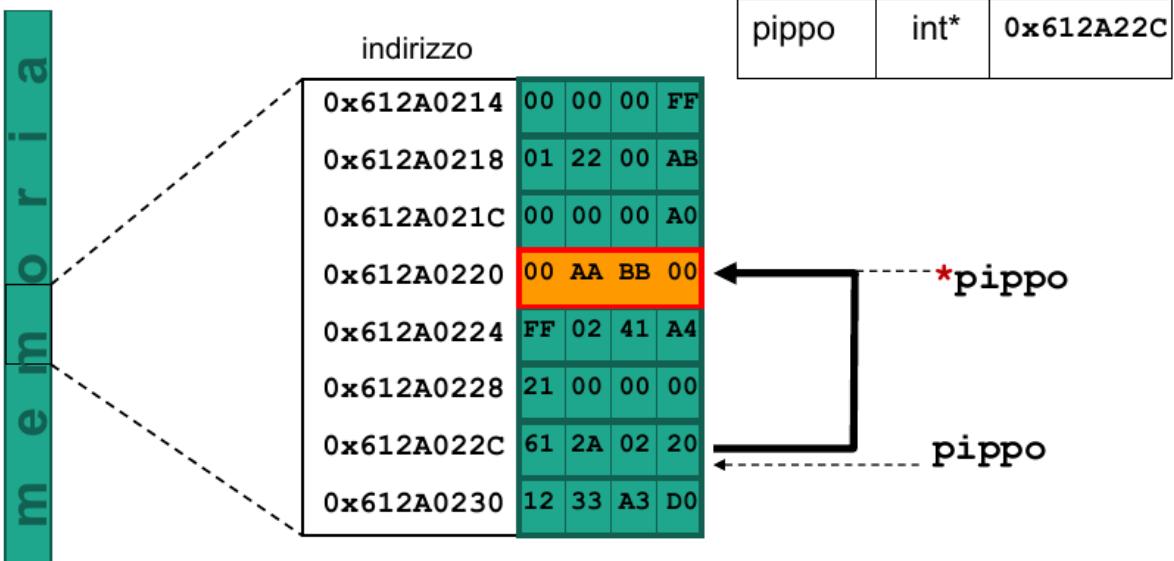
```
pippo++;
```



- `pippo` ora vale `0x612A0220`
- `*pippo` ora vale `0x1223D2FF`

## Cambiare il Valore Puntato

```
*pippo = 0x00AABB00;
```



## Considerazione sull'Efficienza

```
const int I = 10; // I è una costante intera
int i;           // i è una variabile intera
int* ip;         // ip è un puntatore ad intero
int x;
...
x = I;
// In compilazione:
// STORE    10      0xcc000000

x = i;
// In compilazione:
// READ     TEMP    0xaa000000
```

```
// STORE    TEMP    0xCC000000
x = *ip;
// In compilazione:
// READ     TEMP0   0xBB000000
// READ     TEMP1   TEMP0
// STORE    TEMP1   0xCC000000
```

Considerazioni su numero di **lettura** e **scrittura**:

```
a = 15;
// 1 accesso in SCRITTURA
// 0 accessi il LETTURA
a = b;
// 1 accesso in SCRITTURA
// 1 accesso il LETTURA
a = *p;
// 1 accesso in SCRITTURA
// 2 accessi il LETTURA
*p = 15;
// 1 accesso in SCRITTURA
// 1 accesso il LETTURA
*p = b;
// 1 accesso in SCRITTURA
// 2 accessi il LETTURA
*p = *p2;
// 1 accesso in SCRITTURA
// 3 accessi il LETTURA
```

## Aritmetica dei Puntatori

**Somma con un intero:**

<**puntatore** ad un tipo **T**> + <**intero**>

**espressione di tipo puntatore ad un tipo T (T\*)**

**Semantica:** `p + i`

- è il puntatore che punta ad una locazione `i` elementi (di tipo `T`) dopo `p`
- come indirizzo di memoria si ha `(p + i) * (dimensione di T)`

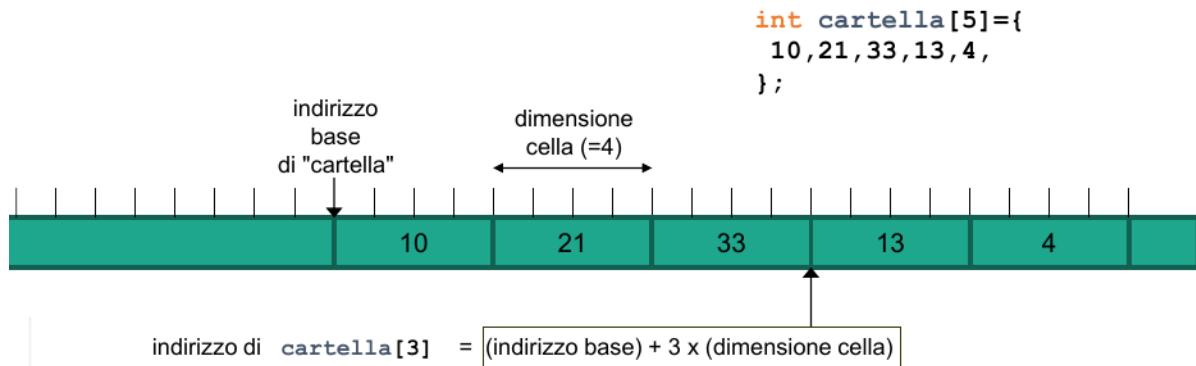
**Esempio:**

```
double *p, *q;
...
q = p + 3;
*(p + 3) = 2.0;
q++;
q--;
q += 2;
```

**Richiamo sui vettori:**

- in memoria, gli **elementi di un array** sono memorizzati in una **serie di celle contigue**

- ogni cella ha la **stessa grandezza**
- per questo gli array sono **strutture ad accesso casuale**



Quindi, se dichiariamo:

- `double *p`

l'accesso

- `p[5]`

equivale a

- `*(p + 5)`

## Puntatori e Record

```
typedef struct{  
    char nome[24];  
    char cognome[24];  
    int peso;  
} Persona;  
Persona *p;
```

Come accediamo al campo peso della `Persona` puntata da `p`?

- `*p.peso` **non va bene**, perché interpretato come `*(p.peso)`

L'unico **modo corretto** è:

- `*(p).peso` o, **equivalentemente**, `p -> peso`

## Assegnare i Puntatori

In memoria, un puntatore è un **indirizzo di memoria** (*di una variabile di cui è noto il tipo*).

Ma quale indirizzo?

- **Modo 1:** *prendere l'indirizzo di una variabile esistente*
  - il puntatore punterà a quella variabile
- **Modo 2:** *allocare della memoria libera*
  - il puntatore punterà ad una nuova variabile, memorizzata nella memoria così riservata
  - la nuova variabile è **allocata dinamicamente**

## Modo 1

Si fa uso dell'operatore &.

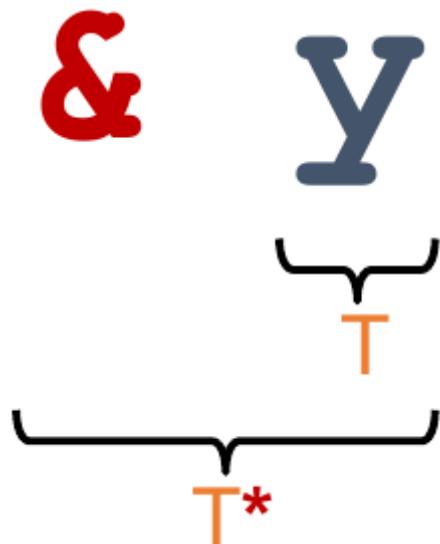
- **Esempio:**

```
double d = 9.0;
double *p;

p = &d;           // p punta all'indirizzo di memoria che contiene d
printf("%f", *p); // stampa il valore di *p, ovvero il valore puntato da
                  // p, che sarebbe d = 9.0
*p = 21.5;        // si scrive il valore 21.5 nell'area di memoria puntata
                  // da p, ovvero quella che contiene d
printf("%f", d); // si stampa d che risulterà uguale proprio a 21.5
```

## Operatore & e Tipi

Se `y` è una variabile di tipo `T`:



## Operatore & e Vettori

```
int numeri[] = {10, 20, 30, 40};
int *punt;

punt = &numeri;
// SBAGLIATO, perchè numeri non è di tipo int
punt = &(numeri[0]);
// CORRETTO, anche se si può scrivere ancora meglio come:
punt = numeri;
```

- **Esempio:** stampa di un array

```

// Metodo classico
for(int i = 0; i < 4; i++){
    printf("%d", numeri[i]);
}

// Utilizzando i puntatori
for(int i = 0; i < 4; i++){
    printf("%d", *(punt++))
}

```

# [Lezione 16]

## Utilizzo dei Puntatori

Sia `p` che `*p` sono **L-value modificabili**, ovvero sono un "qualcosa" a cui si può assegnare un valore.

L'operatore `*` ha **priorità superiore a quella degli operatori matematici**:

- `x = 6 * *p` equivale a `x = 6 * (*p)`

Per visualizzare il valore di un puntatore si può utilizzare la direttiva `%p` in una `printf`

## Tipi e Puntatori

L'informazione relativa al tipo è necessaria per permettere ai puntatori di conoscere la dimensione dell'oggetto puntato (usata nell'aritmetica dei puntatori).

Poichè oggetti di tipo diverso possono avere dimensione diversa, **l'assegnazione tra puntatori di tipo diverso è in genere errata** e il compilatore genera un warning.

```

int *p, x = 12;
long *q, y = 26;
p = &x; // Corretto
q = &y; // Corretto
q = p; // Errato, genera un warning
q = &x; // Errato, genera un warning

```

## Puntatori a `void`

Sono **puntatori generici** e **non possono essere dereferenziati** (non si può scrivere `*p`), possono essere usati solo come contenitori temporanei di valori di tipo puntatore (a qualsiasi tipo)

- `void *h`

**Non serve il cast (`void *`)** per copiare un puntatore non-`void` in un puntatore `void`:

- `h = p` è lecito supponendo `int* p`

Qualsiasi tipo di puntatore può essere confrontato con un puntatore a `void`

- `NULL` è definito come `(void *) 0`

Per dereferenziare il valore di un puntatore a `void` **è necessario assegnarlo ad un puntatore al tipo appropriato** per poter conoscere le dimensioni dell'oggetto puntato.

Può essere necessario il cast (`tipo *`) per copiare un puntatore `void` in un puntatore non-`void` (non richiesto dal compilatore C, ma richiesto dal compilatore C++).

In riferimento all'esempio precedente:

```
int *q;  
q = h;  
q = (int *)h;  
*q = 23;  
  
// Nota: sono state precedentemente eseguite le assegnazioni  
p = &x;  
h = p;
```

## Puntatori e Vettori

Il nome (*senza parentesi*) di un `vettore-di-T` è un valore costante di tipo `puntatore-a-T` e corrisponde all'**indirizzo di memoria del primo elemento del vettore**:

```
int vet[100];  
int *p;  
p = vet;  
// L'indirizzo di memoria di vet viene messo in p ed equivale a  
p = &vet[0];  
// Le parentesi hanno priorità maggiore di &
```

Attenzione:

- `vet = p` è **SBAGLIATO!**

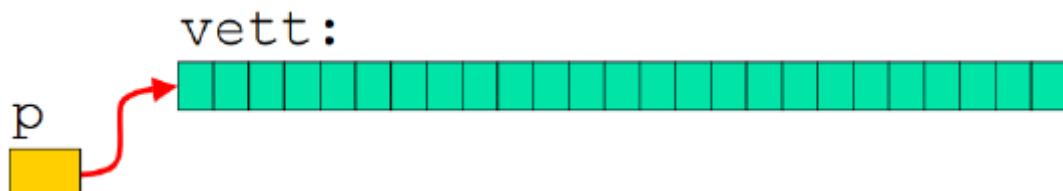
Non si può assegnare un valore a `vet` in quanto questo **NON è una variabile puntatore**, ma è un "*sinonimo*" di un **indirizzo di memoria**.

Gli indirizzi di memoria **sono valori costanti stabiliti dal compilatore**, non sono variabili e quindi **non hanno uno spazio in memoria modificabile per contenere un valore**.

Il termine "puntatore" viene comunemente (e impropriamente) usato al posto di "indirizzo di memoria" (es. "`&a` dà il puntatore ad `a`").

Una variabile di tipo `puntatore-a-T`, assegnata in modo che contenga l'indirizzo di un oggetto di tipo `vettore-di-T`, può essere utilizzata come se fosse un `vettore-di-T`:

```
int vet[25];  
int *p = vet;
```



In questo esempio, `p[3]` equivale a `vet[3]`.

Il compilatore internamente trasforma le espressioni con notazione vettoriale `[]` in espressioni con i puntatori.

Una variabile di tipo `puntatore-a-T` non "sa" se il valore a cui punta è singolo o è l'elemento di un vettore, lo sa solo il programmatore a cui **spetta il compito di usarlo in modo coerente**:

```
int x = 10, vet[10], *p, *q;  
p = &x;  
// NO, non esiste l'oggetto puntato da p + 1  
p++;  
q = p + 1;  
p = vet;  
// SI, perchè p punta ad un vettore, quindi esiste p + 1  
p++;  
q = p + 1;
```

## Priorità dell'Operatore \*

Dalla tabella delle priorità si vede che l'operatore di deriferimento `*` ha priorità quasi massima, inferiore solo alle parentesi (e a `>>` e a `.`), e **associatività da destra a sinistra**.

Quindi, considerando che gli operatori `*` e `++` hanno stessa priorità e **associatività da D a S**:

- `*p++` equivale a `*(p++)` e incrementa `p`
- `++p` equivale a `*(++p)` e incrementa `p`
- `++*p` equivale a `++(*p)` e incrementa `*p`

Inoltre:

- `(*p)++` incrementa `*p`
- `*p + 1` equivale a `(*p) + 1` e non a `*(p + 1)`

## Puntatori e Stringhe

Si noti la differenza tra le seguenti definizioni:

- `char str[100]`
  - **riserva spazio per contenere i caratteri**, è una variabile e il suo contenuto può essere modificato
- `char *s`
  - **non riserva spazio per contenere i caratteri**, quindi per essere utilizzata come stringa le si deve assegnare una stringa "vera"

### Assegnazione di una stringa variabile:

```
// Corretto  
s = str;  
scanf("%s", s);  
// Sbagliato  
s = "salve";  
scanf("%s", s);
```

Si considerino i **seguenti esempi**:

- `char str[] = "ciao"`
  - È l'**inizializzazione** di una variabile `stringa`
  - Il compilatore **riserva memoria** per `str` e vi copia i caratteri di `"ciao"`. La stringa costante `"ciao"` **non esiste in memoria**: è usata dal compilatore per inizializzare la

- stringa `str`, ma esiste in memoria la stringa variabile "ciao"
  - Pertanto scrivere `str[0] = 'm'` è corretto
- `char *s = "hello"`
  - È l'**inizializzazione** di una variabile **puntatore**
  - Il compilatore determina l'indirizzo della stringa costante "hello" (che **esiste in memoria**) e lo assegna alla variabile puntatore
  - Scrivere `s[0] = 'b'` è **ERRATO**, perché "hello" è **costante**

## Puntatore Variabile a Valore Costante

```
int const *p;
const int *p;
// Sono equivalenti
```

`p` è una variabile di tipo **puntatore-a-costante** (a un oggetto costante di tipo `int`)

```
const int x, y;
const int *p;

p = &x;      // Si, p è una variabile
p = &y;      // Si, p è una variabile
*p = 13;     // No, *p è costante
```

L'assegnazione di un valore di tipo **puntatore-a-costante** (l'indirizzo di un valore costante) ad una variabile di tipo **puntatore-a-variabile** **genera un warning** del compilatore perchè **permette di bypassare la restrizione** `const`

```
const int x = 12;
int y = 10;
int *p;           // Puntatore a variabile
const int *q;     // Puntatore a costante
p = &x;          // Warning
*q = 5;          // Non dà errore
q = &x;          // OK
*p = 5;          // Dà errore
```

- `int * const p`

`p` è una costante di tipo **puntatore-a-variabile** (a un oggetto di tipo `int`).

Le costanti possono essere **solo inizializzate**

```
int x, y;
int * const p = &x; // Inizializzazione
*p = 13;          // Si, perchè *p è una variabile
p = &y;          // No, perchè p è una costante
```

## [Lezione 17]

## Vettori di Puntatori

Gli elementi di questi vettori sono puntatori:

- `int *vet[10]`
  - Definisce un vettore di 10 puntatori a `int`
  - Le `[]` hanno priorità maggiore dell'operatore `*`

### Esempio di inizializzazione:

```
int a, b, c;
int *vet[10] = {NULL};
vet[0] = &a;
vet[1] = &b;
vet[2] = &c;

int *v[4] = {&a, &b, &c};
v[0] = a;
```

I valori da `vet[3]` a `vet[9]` sono tutti `NULL` in quanto è stato inizializzato il primo elemento e **tutti i successivi vengono posti automaticamente a 0**.

### Esempio di inizializzazione errata:

```
int a, b, c;
int *vet[10] = {&a, &b, &c};
```

È **errato** perché questi inizializzatori sono **indirizzi di variabili automatiche**.

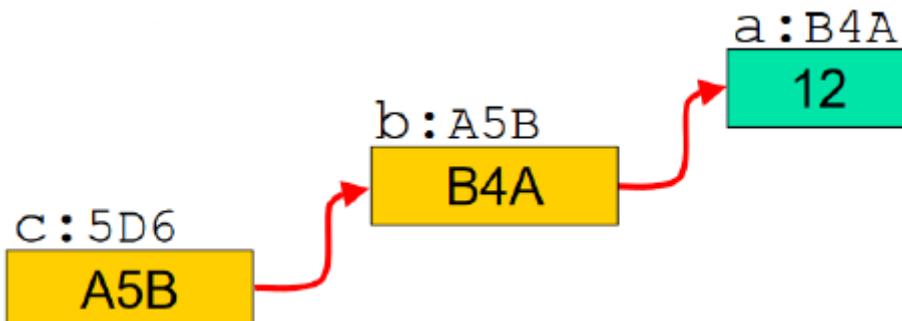
## Puntatori a Puntatori

Sono variabili che contengono l'**indirizzo di memoria di un puntatore**.

- **Esempio:**

```
int a, *b, **c;
a = 12;
b = &a;
c = &b;
```

Il puntatore `c` punta ad una variabile `(b)` che, a sua volta, punta ad un `int (a)`.



- **Esempio:**

```

int a = 10, b = 20, c = 30;
int *v[3], int **w;
v[0] = &a;
v[1] = &b;
v[2] = &c;
w = v;
w++;

```

- `v` è un **vettore di puntatori**, cioè è l'**indirizzo di memoria** (puntatore) **di un puntatore**. Quindi `v + 1` è lecito e punta al secondo puntatore del vettore `v`, ovvero `v[1]`, il quale punta a sua volta a `b`
- `w` è anch'esso un **puntatore a puntatore** e, dato che punta a `v`, incrementandolo punterà anch'esso a `v[1]`

Una matrice è un vettore di vettori e quindi, considerando che l'associatività di `[]` è da sinistra a destra, si ha:

- `int Mx[7][5]`

che definisce un **vettore di 7 elementi**, ciascuno dei quali è un **vettore di 5 int**.

Gli elementi del vettore `Mx` sono i 7 vettori identificati da `Mx[i]`, quindi `Mx[i]` è l'**indirizzo di memoria di ciascuno dei 7 vettori di 5 int**.

- `Mx` è di tipo `puntatore-a-int` come `int *p`? NO
- `Mx` è di tipo `puntatore-a-puntatore-a-int` come `int **p`? NO
- `Mx` è di tipo `puntatore-a-vettore-di-5-int` come `int(*p)[5]`, quindi `Mx + 1` punta al secondo vettore di 5 `int`.

## Puntatori e Matrici

Nella definizione di puntatore seguente:

- `int (*p)[5]`

è necessario che la dimensione delle colonne (5) sia specificata, perché definisce un **puntatore-a-vettore-di-cinque-integeri**.

Poichè `Mx` è un `puntatore-a-vettore-di-5-integeri` (e *non un puntatore ad intero*), allora:

- `Mx + 1` **punta** all'elemento successivo, ossia **al successivo vettore di 5 interi** (seconda riga della matrice)

## Ricapitolando

- `int Mx[7][5]`
  - è un `puntatore-a-vettore-di-5-integeri`
- `Mx[1][2]`
  - è un **valore scalare**
  - è di **tipo int**
  - è **modificabile**
  - `Mx[1][2] + 1` somma 1 al contenuto di `Mx[1][2]`
- `Mx[1]`
  - è l'**indirizzo di un vettore di 5 int**
  - è di tipo `puntatore-a-int`

- non è modificabile
- `Mx[1] + 1` punta a `Mx[1][1]`
- `Mx`
  - è l'indirizzo di un vettore di 7 vettori di 5 int
  - è di tipo puntatore-a-vettore-di-int
  - non è modificabile
  - `Mx + 1` è l'indirizzo di memoria `Mx[1]`

Inoltre:

```
int *p;           // puntatore-a-int
int (*q)[5];     // puntatore-a-vettore-di-5-int
p = Mx;          // NO, è errata
q = Mx;          // OK, q + 1 è l'indirizzo del secondo vettore di 5 elementi
(Mx[1])
p = &Mx[0][0];   // OK, p è l'indirizzo del primo elemento di Mx[0]
```

## Vettori di Stringhe

Essendo una stringa un **vettore di caratteri**, un vettore di stringhe è, in realtà, un **vettore di vettori di caratteri**, ossia una **matrice di char**.

- `char str[4][20] = {"uno", "due"}`

Definisce un **vettore di 4 stringhe di 20 char**. Le 4 stringhe sono identificate da `str[i]`.

**str:**

<code>str[0]</code>	u	n	o	\0	\0	\0	\0	\0	\0	\0	\0	\0	\0	\0	\0	\0	\0	\0	\0
<code>str[1]</code>	d	u	e	\0	\0	\0	\0	\0	\0	\0	\0	\0	\0	\0	\0	\0	\0	\0	\0
<code>str[2]</code>	\0	\0	\0	\0	\0	\0	\0	\0	\0	\0	\0	\0	\0	\0	\0	\0	\0	\0	\0
<code>str[3]</code>	\0	\0	\0	\0	\0	\0	\0	\0	\0	\0	\0	\0	\0	\0	\0	\0	\0	\0	\0

- **Puntatori a Puntatori: ATTENZIONE**

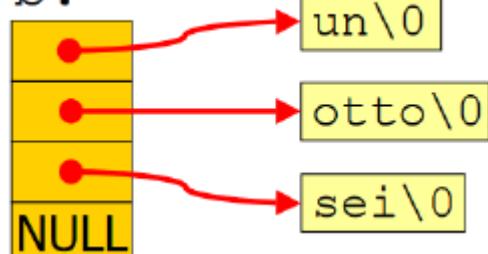
Si notino le **differenze** tra:

```
char a[4][8] = {"un", "otto", "sei"};
char *b[4] = {"un", "otto", "sei"};
```

**a:**

un\000000
otto\0000
sei\0000
\00000000

**b:**



- `a[i]` è l'**indirizzo costante** della riga `i` di `a`, tale riga è una stringa variabile di 8 char
- `b[i]` è una **variabile puntatore** con l'indirizzo della riga `i`, stringa costante di 4 caratteri: ad esempio `b[2]` contiene l'indirizzo di `sei\0`

## Vettori di Puntatori e Matrici

```
a[2] = "hello";
// ERRORE, a[2] non è un puntatore
strcpy(a[2], "hello");
// CORRETTO

b[2] = "hello";
// CORRETTO, b[2] è una variabile puntatore a cui viene assegnato l'indirizzo di memoria di una stringa costante
strcpy(b[2], "hello");
// ERRORE, b[2] punta ad una stringa costante

a[1][0] = 'm';
// CORRETTO, l'oggetto puntato da a[1] è una stringa variabile
b[1][0] = 'm';
// ERRORE, l'oggetto puntato da b[1] è una stringa costante
```

### const per Puntatori a Puntatori

- `int **x`
  - `x` è una variabile di tipo **puntatore a un puntatore variabile a un oggetto variabile** di tipo `int`
- `const int **x`
  - `x` è una variabile di tipo **puntatore a un puntatore variabile a un oggetto costante** di tipo `int`
- `int **const x`
  - `x` è una costante di tipo **puntatore a un puntatore variabile a un oggetto variabile** di tipo `int`
- `int *const *x`
  - `x` è una variabile di tipo **puntatore a un puntatore costante a un oggetto variabile** di tipo `int`
- `const int *const *x`
  - `x` è una variabile di tipo **puntatore a un puntatore costante a un oggetto costante** di tipo `int`
- `const int *const *const x`
  - `x` è una costante di tipo **puntatore a un puntatore costante a un oggetto costante** di tipo `int`

## Modo 2

Si tratta di **allocare della memoria libera**

- Il puntatore punterà ad una nuova variabile, memorizzata nella memoria così riservata
- La nuova variabile è **allocata dinamicamente**

# Organizzazione della Memoria

## Quattro aree:

- **area codice**: qui viene tenuto il **codice** che viene eseguito (linguaggio macchina)
- **area variabili globali**: qui risiedono le **variabili globali** (non l'avrei mai detto)
- **area stack**: qui risiedono le **variabili locali**
- **heap (free store)**: è il "**resto**", memoria libera. Riserva da cui si attinge lo spazio per le variabili allocate dinamicamente (`malloc`, `calloc`)

Cosa conosce il compilatore (staticamente):

- l'**indirizzo delle variabili globali**
- l'**offset delle variabili locali**
  - rispetto al **record di attivazione**
  - `vero indirizzo = posizione attuale record attivazione + offset`
- il **valore delle costanti**

```
const int A=10;
int b=10;

int main(int p) {
    int d;
    ...
}
```

ide.	tipo	locazione o valore	o offset
A	int	---	10
b	int	0xA12F345A	---
p	int	---	0x000000020
d	int	---	0x000000030

**NOTA:** mentre l'area codice e l'area per le variabili globali hanno **dimensione fissa**, l'area stack e l'heap hanno **dimensione variabile**. Per convenzione lo **stack cresce "verso il basso"** e l'heap **cresce "verso l'alto"**.

## Allocazione

- `void* malloc(unsigned int n)`

Funzione `malloc`: **m-emory alloc-ation**:

1. alloca `n` bytes di memoria
  2. restituisce l'indirizzo della memoria appena allocata
    - sotto forma di **puntatore generico** (`void *`). È un puntatore senza tipo, un **semplice indirizzo di memoria**
- **Esempio:**

```
int *p;
p = malloc(4);
// ERRORE, malloc restituisce un void*, quindi bisogna fare cast:
int *p;
p = (int*)malloc(4);
```

## Memoria Esaurita

- `void* malloc(unsigned int n)`

Se non c'è più memoria l'allocazione fallisce e `malloc` restituisce il valore speciale `NULL`:

- semanticamente `NULL` è un "puntatore che non punta a nulla"

- in memoria `NULL` è rappresentato dal valore **0**

Il valore restituito dalle `malloc` va controllato:

```
int *p;
p = (int*)malloc(4);
if(!p){ // Significa if(p == NULL)
    // Memoria Finita
}
```

## Allocazione e Record

```
typedef struct{
    // Dichiarazioni
} NuovoTipo;

NuovoTipo* p;
p = (NuovoTipo*)malloc(sizeof(NuovoTipo));
```

Il costrutto `sizeof` è estremamente utile con le `malloc` e si usa sempre, anche con i tipi base.

- **Nota:** il C non prescrive quanti bytes occupano

## Deallocazione

- `void free(void* p)`

Libera la memoria che era stata allocata all'indirizzo `p`.

- **Nota:** `p` deve essere il risultato di una `malloc`

```
int* p;
p = (int*)malloc(sizeof(int));
// uso p
free(p);
```

Se ci si dimentica di deallocare si ha un cosiddetto **memory leak**.

- **Nota:** non c'è alcuna garbage collection in C

```
int main(){
    int k;      // Vengono automaticamente allocati 4 byte per k
    k = 15;    // k viene inizializzato a 15
    ...
    // Uso k
    return 0;
    // Alla fine dell'esecuzione i 4 byte sono resi di nuovo disponibili
}
```

Con l'**allocazione dinamica**:

```

int main(){
    int* k;
    // Dichiarazione del puntatore k
    k = (int*)malloc(sizeof(int));
    // Allocazione esplicita di k
    *k = 15;
    // Inizializzazione di k
    ...      // uso k
    free(k);
    // Deallocazione esplicita di k
    return 0;
}

```

## Allocazione di Vettori

- `(void*)calloc(unsigned int n, unsigned int size)`

Funzione calloc: c-contiguous allocation

- Alloca `n` elementi **contigui** ciascuno grande `size`. In pratica alloca un'area di memoria grande `n*size`
- Per il resto funziona come `malloc`

### Esempio:

```

int* p;
p = (int*)calloc(100000, sizeof(int));
// Alloca un vettore di 100000

```

### Da ricordare:

```

int* v = (int*)calloc(10000, sizeof(int));
// Vettore allocato dinamicamente
int v[10000];
// Vettore a dimensione fissa

```

In entrambi i casi:

- si ha una vettore di 10000 interi
- si può quindi scrivere: `v[2] = v[0] + 3 * v[1]`

# [Lezione 18] Linguaggio C: le Funzioni

## Sommario

- Le Funzioni
- Chiamata di una Funzione e Ritorno da una Funzione
- Parametri Formali e Parametri Attuali
- Passaggio di parametri
- Passaggio di vettori e matrici

## Le Funzioni

### Struttura Modulare

Per **semplificare la struttura** di un programma complesso è possibile **suddividerlo in moduli**.

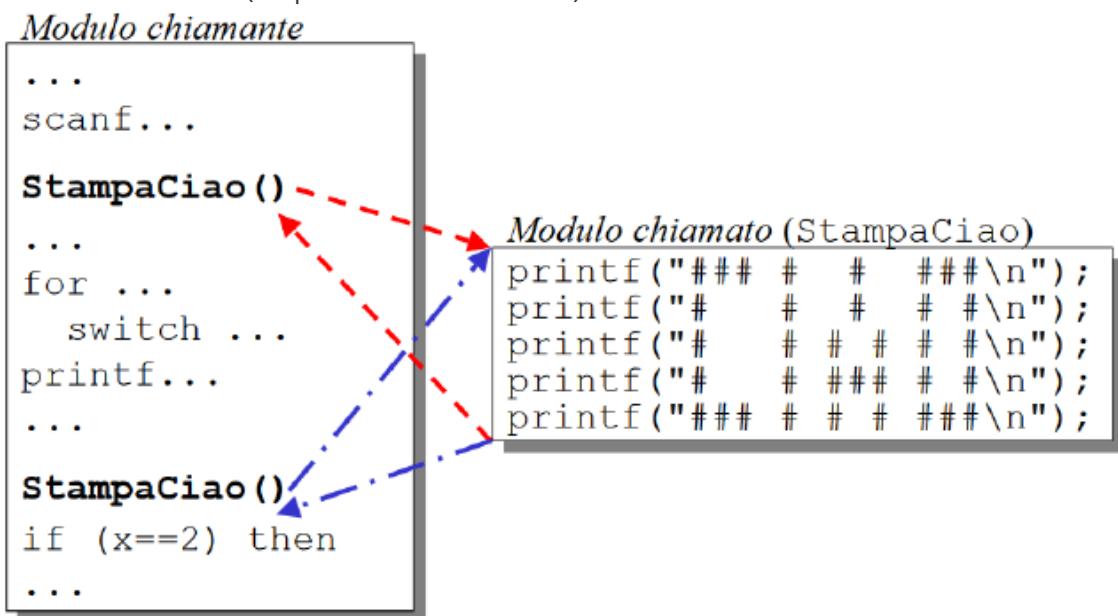
Un **modulo** è un *blocco di codice che assolve ad un compito preciso e a cui è stato dato un nome*.

Un programma consta di un modulo principale (`main`) ed eventuali altri moduli di supporto.

Quando un modulo richiama un altro modulo, il chiamante **viene sospeso** finché il chiamato non ha terminato la sua esecuzione.

In C i moduli sono chiamati **funzioni**. Ogni funzione può richiamare qualsiasi altra funzione (anche sè stessa).

- **Esempio:** le due chiamate del modulo `StampaCiao` fanno eseguire ogni volta le istruzioni che lo costituiscono (sospendendo il chiamante)



### Vantaggi

- Poichè i moduli "nascondono" al loro interno i dettagli di come una certa funzionalità venga realizzata, il programma complessivo ha un **livello di astrazione maggiore**
  - Il modulo viene visto come una "**black box**", ovvero un insieme di macro-istruzioni
- Il codice per ottenere una certa funzionalità viene scritto una volta sola e viene richiamato ogni volta che è necessario (la chiamata richiede tempo)

- Il codice complessivo è più corto
- Essendo più piccoli, i **moduli sono più semplici da realizzare e verificare**
- Il codice di un modulo correttamente funzionante può essere riutilizzato in altri programmi

## Variabili Locali

Ogni funzione è un piccolo programma a sè stante, isolato dalle altre funzioni.

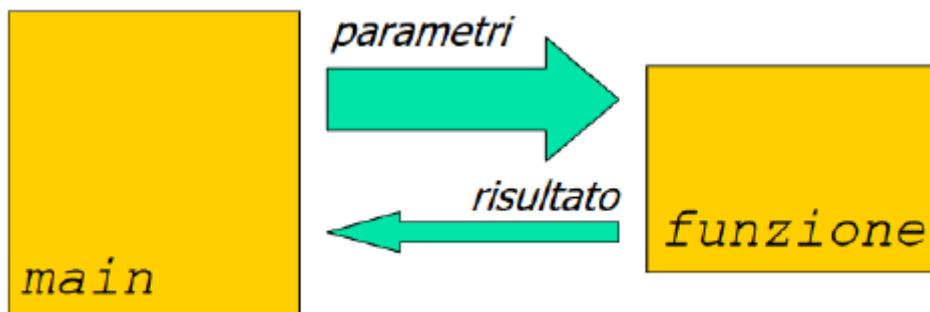
All'interno di una funzione possono essere definite delle **variabili locali** (ovvero con **scope locale**), ovvero delle funzioni che le altre funzioni non "vedono". Proprio per questo si possono usare variabili con lo stesso nome in funzioni diverse.

Le variabili locali **vengono create ogni volta che si entra nella funzione e distrutte quando si esce**. Le inizializzazioni avvengono ad ogni chiamata, senza inizializzazione il contenuto è indefinito.

## Parametri e Valore Restituito

Essendo le variabili interne locali, private, per far passare ad una funzione i dati da elaborare è necessario utilizzare variabili speciali dette **parametri**.

La funzione comunica al modulo chiamante il risultato della sua elaborazione producendo un unico valore detto valore restituito o valore di ritorno



## Definizione di Funzione

```

tipo nomeFunzione(parametri){
    // Definizione variabili locali
    // Istruzioni
    // Eventuale return
}
  
```

**tipo** indica il **tipo del valore restituito** (es. `sqrt` restituisce un `double`).

Se la funzione non restituisce valori (come la funzione `StampaCiao`) allora bisogna indicare il tipo `void`:

- `void StampaCiao(...)`

Se si omette il tipo allora viene supposto `int`.

Se la funzione non ha parametri (come `StampaCiao`) allora si indica `void` tra le parentesi:

- `void StampaCiao(void)`

# Chiamata di Funzione

Si chiama una funzione indicandone il **nome seguito da una coppia di parentesi contenenti i valori da elaborare** (separati da virgolette):

- `eleva(y, 2)`

Se la funzione non richiede parametri bisogna lasciare vuote le parentesi (che devono esserci comunque):

- `StampaCiao()`

Il valore restituito può essere assegnato ad una variabile o utilizzato in un'espressione, altrimenti viene semplicemente scartato:

```
x = eleva(y, 2);
y = 3 * eleva(2, k) - 4 * k;
eleva(3, 5);
```

## Ritorno da una Funzione

La funzione termina (si torna al modulo chiamante) quando viene eseguita l'istruzione di `return risultato`.

- Una funzione può avere più istruzioni di `return`
- `risultato` è il valore restituito dalla funzione al chiamante, è un'espressione qualsiasi
  - es. `return x * 2`

Se il tipo restituito dalla funzione è `void` allora non si deve indicare il risultato, inoltre la `return` che precede la graffa di chiusura (e solo questa) **può essere omessa**.

I valori delle variabili locali **vengono persi**.

Quando si è nel `main`, la `return` **termina il programma**. Per terminare il programma mentre si è in una funzione si possono utilizzare:

- `exit(status)`
- `exit(EXIT_SUCCESS)`
  - dichiarata in `stdlib.h`

## Esempio

```
int main(void)
{
    int x, y ;
    /* leggi un numero
     * tra 50 e 100 e
     * memorizzalo
     * in x */
    /* Leggi un numero
     * tra 1 e 10 e
     * memorizzalo
     * in y */
    printf("%d %d\n",
           x, y) ;
```

```
int main(void)
{
    int x, y ;
    x = leggi(50, 100) ;
    y = leggi(1, 10) ;
    printf("%d %d\n",
           x, y) ;
```

```
int leggi(int min,
          int max)
{
    int v ;
    do {
        scanf("%d", &v) ;
    } while( v<min || v>max) ;
    return v ;
}
```

## Tipo di una Funzione

Il tipo di una funzione è determinato dal tipo del valore restituito e dal tipo, numero e ordine dei suoi parametri.

- `int eleva(int b, int e)`

`eleva` è una funzione che ha un primo parametro `int`, un secondo parametro `int` e restituisce `int`

## Scope di una Funzione

Lo scope di una funzione (nome e tipo) si estende dal punto in cui viene definita fino alla fine del file.

La funzione può essere utilizzata **solo dalle funzioni che nello stesso file seguono la sua definizione** (vale anche per il main)

```
f1(){  
    ...  
}  
f2(){  
    ...  
}  
main(){  
    ...  
}
```

`f1` non "vede" e non può usare `f2`, al contrario `f2` vede `f1` mentre il `main` vede sia `f1` che `f2`.

il compilatore verifica che le chiamate a funzione siano coerenti con le corrispondenti definizioni (cioè abbiano lo stesso tipo).

È necessario che la chiamata a funzione **sia nello scope della funzione stessa**. Se il compilatore trova una funzione di cui non conosce il tipo (non è in scope, come, ad esempio, `f1` che chiama `f2`) allora presuppone che essa sia definita altrove e quindi:

- non fa controlli sugli argomenti
- presuppone che restituisca un `int`
- segnala il possibile problema con un **warning**

## Prototipo di una Funzione

Il prototipo di una funzione è una **dichiarazione che estende lo scope della funzione** (nome e tipo).

Il corpo della funzione (la sua definizione) può essere quindi collocato:

- **in un punto successivo a dove viene chiamata** (nell'esempio sottostante, `eleva` è definita dopo il `main` dove viene utilizzata)
- **in un altro file di codice sorgente C**
- **in una libreria** (compilata)

lo scopo primario degli header file è quello di **fornire al compilatore i prototipi delle funzioni delle librerie del C** (ad esempio, `stdio.h` contiene i prototipi di `scanf`, `printf`, `getchar` ecc.)

## • Esempio di una funzione

```
#include <stdio.h>

// Prototipo
int eleva(int b, int e);

int main(){
    int x, y;

    printf("Introduci numero: ");
    scanf("%d", &x);
    y = eleva(x, 2);
    printf("%d^%d = %d\n", x, 2, y);
    return 0;
}

int eleva(int b, int e){
    int k = 1;
    while (e-- > 0){
        k *= b;
    }
    return k;
}
```

I prototipi possono essere collocati:

- **prima del `main`** (come nell'esempio)
- **tra una funzione e l'altra**
- **insieme alle definizioni delle variabili locali** di una funzione

Il prototipo **estende lo scope della funzione** (nome e tipo) dal punto dove è indicato:

- **fino alla fine del file** se esso è *collocato esternamente alle funzioni* (prima del `main` o tra due funzioni)
- **fino alla fine della funzione** se è *interno ad una funzione* (collocato con le variabili locali di una funzione)

Il prototipo di una funzione è **simile alla definizione della funzione**, salvo che:

- **manca il corpo**
- **i nomi dei parametri possono essere omessi** (i tipi devono essere presenti)
  - nell'esempio di `eleva` si può scrivere il prototipo come `int eleva(int, int)`
- **ha un punto e virgola alla fine**

I nomi dei parametri dei prototipi:

- se non sono omessi, **possono essere diversi da quelli usati nella definizione della funzione**
- **sono scorrelati dagli altri identificatori** (nomi uguali si riferiscono comunque ad identificatori diversi)
- **sono utili per descrivere il significato dei parametri**
  - `int eleva(int base, int esponente)`

## Parametri Formali ed Attuali

- **Parametri formali:** sono le *variabili indicate tra le parentesi nella definizione della funzione*
  - `int eleva(int b, int e)`
- **Parametri attuali** (o **argomenti**): sono i *valori (variabili, costanti, espressioni) indicati tra le parentesi alla chiamata di una funzione*
  - `eleva(x, 2)`

Nella chiamata ad una funzione bisogna indicare un argomento per ciascuno dei parametri formali.

I parametri attuali e quelli formali **corrispondono in base alla posizione** (primo attuale al primo formale ecc.)

I nomi dei parametri formali sono scorrelati (e quindi tipicamente diversi) dai nomi di eventuali variabili usate come argomenti

I parametri formali hanno lo stesso nome delle variabili locali della funzione

Le funzioni possono ricevere dei parametri dal proprio chiamante

Nella funzione:

- **Parametri formali**
- Nomi "interni" dei parametri

```
int leggi(int min,
          int max)
{ ... }
```

Nel chiamante:

- **Parametri attuali**
- Valori effettivi (costanti, variabili, espressioni)

```
int main(void)
{
    ...
    y = leggi(1, 10);
    ...
}
```

```
int main(void)
{
    int x, y;

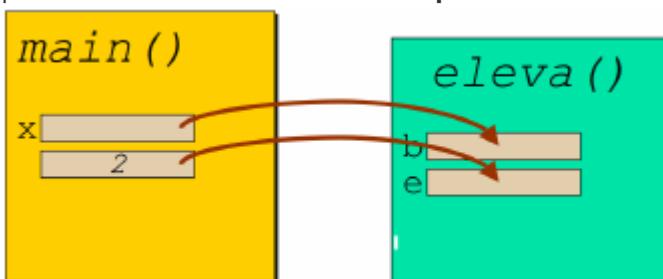
    x = leggi(50, 100);
    y = leggi(1, 10);

    printf("%d %d\n",
           x, y);
}
```

## Passaggio dei Parametri (per Valore)

I dati possono essere passati ad una funzione **esclusivamente per valore (by value)**:

- alla chiamata della funzione vengono create nuove variabili con i nomi di ciascuno dei parametri formali e in esse viene copiato il **valore del corrispondente parametro attuale**



Come per le assegnazioni, se il parametro attuale e il corrispondente formale sono di tipo diverso c'è una **conversione automatica al tipo del parametro formale** (se è di tipo meno capiente può essere generato un warning).

Poiché in memoria i parametri formali e quelli attuali sono completamente distinti e indipendenti, **cambiare il valore di un parametro formale non modifica il parametro attuale corrispondente**, neppure se questo è una semplice variabile (nell'esempio di sopra la modifica di `b` non si ripercuote su `x`)

# [Lezione 19]

## Variabili Locali Static

Le variabili locali hanno **classe di allocazione automatica**:

- vengono create ogni volta che si esegue la funzione ed eliminate ogni volta che questa termina (perdono il valore)

Le variabili locali di **classe di allocazione statica**, invece, **non vengono mai rimosse dalla memoria**, quindi non perdono il loro valore quando la funzione termina (resta quello che aveva al termine della chiamata precedente)

Le variabili statiche non richiedono la ri-allocazione della memoria ad ogni chiamata della funzione, quindi il programma può essere più veloce.

Si richiede una classe di allocazione statica e non automatica mediante lo **specificatore di classe di allocazione static**:

- `static int cont = 0`

Se non inizializzate esplicitamente, **vengono inizializzate automaticamente a 0** (che, nel caso dei **puntatori**, vengono **automaticamente convertiti** in `NULL`).

L'inizializzazione delle variabili `static` avviene idealmente solo la prima volta che si esegue la funzione (in realtà i valori vengono inizializzati dal compilatore)

Le variabili locali `static` possono essere inizializzate dal compilatore solo con espressioni costanti:

- numeri e `#define`
- indirizzi di memoria di variabili statiche

Non possono invece essere inizializzate con:

- valori `const`
- variabili e risultati di funzioni
- indirizzi di memoria di variabili automatiche

```
int conta(void){  
    static int cont = 0;  
    return cont++;  
}
```

È una semplice funzione di conteggio. Ogni volta che viene richiamata, la funzione incrementa il contatore `cont` e ne restituisce il valore.

Se `cont` non fosse statica la funzione restituirebbe sempre il valore 1, perché `cont` verrebbe re-inizializzata a 0 ad ogni chiamata.

```

char *nomeMese(int n){
    static char *nome[] = {"inesistente", "gennaio", "febbraio", ecc};
    if(n < 1 || n > 12){
        return nome[0]; // inesistente
    } else{
        return nome[n];
    }
}

```

La stringa di cui viene restituito il puntatore può essere utilizzata dal chiamante in quanto, essendo statica, non viene rimossa dalla memoria

## Passaggio dei Parametri (per Riferimento)

Il passaggio dei parametri nella modalità **per riferimento (by reference)**, prevede che la **modifica del parametro formale si ripercuota identica sul corrispondente parametro attuale** (deve essere una variabile).

In C non esiste il passaggio per riferimento, ma questo **può essere simulato passando** per valore alla funzione **il puntatore al dato** da passare (che deve essere una variabile, non può essere il risultato di un calcolo).

Nella `scanf` le variabili scalari sono precedute da `&` perchè devono essere modificate dalla funzione e quindi se ne passa l'indirizzo:

```

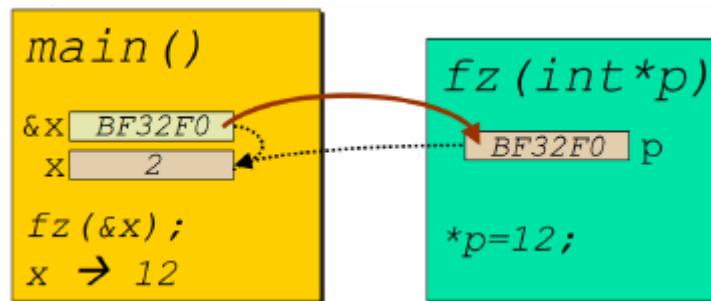
#include <stdio.h>

// Prototipi
void fz(int*);

int main(){
    int x = 2;           // Alloca x e gli assegna 2
    fz(&x);            // Chiama fz passandole l'indirizzo di x
    printf("%d\n", x);
    return EXIT_SUCCESS;
}

void fz(int *p){
    *p = 12;           // fz accede a x come *p modificandola in 12
}

```



## Passaggio di Vettori

Per passare un vettore come argomento **basta indicare il suo nome senza parentesi**:

- `x = media(vettore)`

Il parametro formale corrispondente definirà un vettore dello stesso tipo (in genere senza dimensione, perché ininfluente)

- `float media(int v[])`

La funzione deve conoscere in qualche modo la dimensione del vettore (indicarlo tra parentesi non serve a nulla):

- viene **passato come argomento**:

- `float media(int v[], int dim)`

- **noto a priori** (es. con una `#define`)

- si usano **variabili esterne**

Possono essere passati vettori con dimensioni diverse, ma dello stesso tipo `T`.

Quando si passa un `vettore-di-T`, poichè si indica il nome del vettore, in realtà **si passa l'indirizzo di memoria del suo primo elemento** (che non dà alcuna informazione né restrizione sulla lunghezza del vettore).

Il parametro formale è quindi, in realtà, un `puntatore-a-T`, la forma `v[]` viene convertita automaticamente in `*v`, si possono usare le due definizioni indifferentemente:

- `float media(int *v)`

## Passaggio di Matrici

---

Per passare una matrice come argomento **si indica il suo nome senza parentesi**:

- `x = media(matrice)`

Il parametro formale corrispondente dichiara una matrice dello stesso tipo (in genere senza la prima dimensione in quanto è ininfluente):

- `void func(int matrice[][10])`

La funzione deve conoscere in qualche modo le dimensioni della matrice e, anche in questo caso, indicarle nelle parentesi non serve a niente.

- Possono essere passate matrici con diverso numero di righe, ma devono avere lo stesso numero di colonne e gli elementi devono essere dello stesso tipo `T`
- Poichè una matrice è un `vettore-di-vettori`, quando essa viene passata ad una funzione, **viene in realtà passato l'indirizzo del primo elemento** del `vettore-di-vettori`
- Il tipo del parametro formale corrispondente è quindi un `puntatore-a-vettore` e non un `puntatore-a-puntatore`

Nel caso dell'esempio il parametro formale è un `puntatore-a-vettore-di-10-int`:

- `int (*matrice)[10]`

La forma `matrice [] [10]` viene convertita automaticamente in `(*matrice) [10]`, si possono usare le due definizioni indifferentemente:

- `void funz(int (*matrice)[10])`

È quindi un **errore** scrivere:

- `void funz(int **matrice)`

perchè, oltre all'errore di tipo, **si perde la dimensione delle colonne** e quindi non si può determinare la posizione degli elementi della matrice

# Passaggio di Matrici e Vettori Multidimensionali

In memoria l'elemento `Mx[i][j]` viene determinato con il calcolo

- `indirizzo_di_Mx + NC * i + j`

dove `NC` è il **numero delle colonne**. Come si vede **il numero delle righe non serve**.

Per passare ad una funzione una matrice con qualsiasi numero di righe e qualsiasi numero di colonne si fa ricordo all'**allocazione dinamica**.

Quanto visto per le matrici può essere esteso ai **vettori multidimensionali**. In particolare:

- nel parametro formale **si può tralasciare la dimensione del solo primo elemento**
- il parametro formale è un **puntatore ad un vettore di X vettori di Y vettori di Z vettori ... di tipo T**
- la funzione **deve conoscere i valori di tutte le dimensioni**
- possono essere passate matrici **con la sola prima dimensione diversa**

## Parametri `const`

Il modificatore `const` applicato ai parametri formali impedisce che all'interno della funzione si possa modificarne il valore

- `int funzione(const int v)`

Permette di **proteggere i parametri da una successiva incauta modifica** (per prevenire errori di programmazione)

Ad esempio, questo richiede al compilatore di segnalare se un `puntatore-a-dato-costante` viene assegnato ad un `puntatore-a-dato-variabile` (ad esempio passandolo come parametro), cosa che **bypasserebbe la restrizione**

**Puntatore variabile a dati variabili:**

```
int f(int *p){  
    *p = 12;      // OK, dato variabile  
    p++;         // OK, puntatore variabile  
}
```

Si può passare un `int*`, ma non si può passare un `const int*`

- Non c'è conversione automatica di tipo perchè dentro la funzione nulla vieterebbe di poter cambiare il valore della variabile puntata:

```
int x = 12;  
const int *y = &x;  
f(y);        // ERRORE
```

**Puntatore variabile a dati costanti:**

```
int f(const int *p){  
    *p = 12;      // NO, dato costante  
    p++;         // OK, puntatore variabile  
}
```

Si può passare un `int*` (c'è conversione di tipo automatica in quanto si passa ad un tipo più restrittivo). Si può passare un `const int*`

- **Note:** tipicamente utilizzato per passare un puntatore ad una struct o ad un vettore impedendo che possano essere modificati.

Non si può passare senza cast un `Tipo**` dove è richiesto un `const Tipo**`

### Puntatore costante a dati variabili:

```
int f(int *const p){  
    *p = 12;      // OK, dato variabile  
    p++;         // NO, puntatore costante  
}
```

Si può passare un `int*`

### Puntatore costante a dati costanti:

```
int f(const int* const p){  
    *p = 12;      // NO, dato costante  
    p++;         // NO, puntatore costante  
}
```

Si può passare un `int*`

## Variabili Esterne

Vengono **definite** (riservando memoria) **esternamente al corpo delle funzioni**:

- **in testa al file**, tipicamente dopo le direttive `#include` e `#define`
- **tra una funzione e l'altra**

Sono visibili e condivisibili da tutte le funzioni che nello stesso file seguono la definizione.

Possono essere utilizzate come metodo alternativo per comunicare dati ad una funzione e per riceverne.

A questo scopo si usino con parsimonia: **rendono poco evidente il flusso dei dati all'interno del programma**.

Le variabili esterne hanno **classe di allocazione statica**:

- **non vengono mai rimosse dalla memoria** e, salvo inizializzazione esplicita, **vengono inizializzate automaticamente a 0** (`NULL` per i puntatori)

Una variabile locale con lo stesso nome di una variabile esterna **copre la visibilità di quella esterna** (la variabile locale ha priorità superiore), rendendo impossibile accedere alla variabile esterna (non è buona pratica di programmazione)

```

#include<...>
int uno;
main()
{
    uno = 12;
}
long due;
void fun1()
{
    uno = 21; due=55;
}
int tre;
int fun2()
{
    return uno + due + tre;
}

```

Lo specificatore di classe di allocazione `extern` permette di estendere la visibilità delle variabili esterne.

La clausola `extern` viene premessa ad una definizione di variabile per trasformarla in dichiarazione (**non riserva memoria**)

- `extern int x`

Indica al compilatore che la variabile è definita (con allocazione di memoria, senza `extern`) altrove (più avanti nello stesso file o in un altro file). In seguito il **linker** ricondurrà tutte le dichiarazioni all'unica definizione

## Documentazione delle Funzioni

È utile scrivere sempre la documentazione relativa allo scopo e all'uso di una funzione come commento iniziale contenente:

- **Scopo:** a cosa serve la funzione
- **Parametri:** tipo e descrizione di ciascuno
- **Valore restituito:** tipo e descrizione

Possibilmente anche:

- **Pre-condizioni:** requisiti particolari sui parametri che devono essere soddisfatti da chi invoca la funzione (es. `param > 29`)
- **Post-condizioni:** garanzie date dalla funzione sul valore restituito o sullo stato del programma, purchè le precondizioni siano state soddisfatte (es. `risultato >= 23 && risultato <= 32`)

## Chiamata di Funzioni - Dettagli

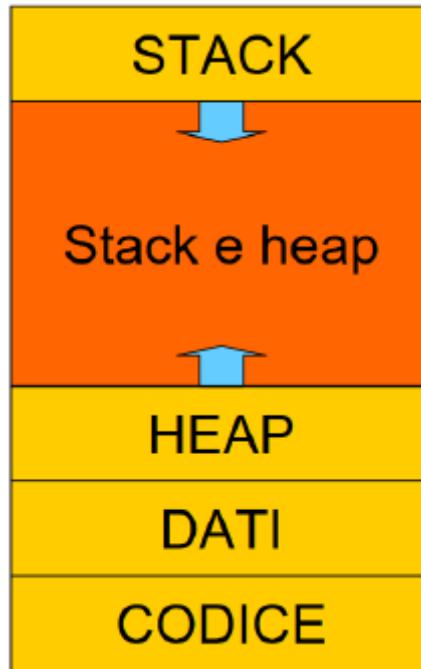
Il programma compilato è costituito da **due parti distinte**:

- **code segment:** codice eseguibile
- **data segment:** costanti e variabili note alla compilazione (statiche ed esterne)

Quando il programma viene eseguito, il sistema operativo alloca spazio di memoria per:

- il **code segment (CS)**

- il **data segment (DS)**
- **stack e heap** (condiviso)



Lo stack contiene inizialmente le variabili locali della funzione `main()`.

Lo heap inizialmente è vuoto e serve per contenere i blocchi di memoria allocati dinamicamente con funzioni `malloc()`.

Stack e heap crescono nell'area condivisa nel senso indicato dalle frecce

- Quando viene chiamata una funzione, sullo stack **vengono prima copiati i valori dei suoi argomenti e poi vi viene allocato un Activation Record** (o **stack frame**) per contenere tutte le variabili locali (e altro)
- Quando la funzione termina, gli **argomenti e l'AR vengono rimossi dallo stack** che quindi **ritorna nello stato precedente alla chiamata**

Nell'Activation Record viene anche **memorizzato l'indirizzo di ritorno della funzione**: la locazione di memoria che contiene l'istruzione del chiamante da cui continuare dopo che la funzione è terminata.

Queste **operazioni di allocazione e deallocazione** di spazio sullo stack e, in generale, il meccanismo di chiamata e ritorno da una funzione **richiedono tempo**.

In casi estremi di necessità di elevate performance si può cercare di limitare il numero delle chiamate a funzione, a costo di ricopiare lo stesso codice in più punti (eventualmente utilizzando macro con argomenti)

## Progetti Multi-File

---

È possibile suddividere le funzioni che costituiscono un eseguibile su più file (detti **translation unit**).

Ciascun file viene compilato separatamente e il **linker** li assembla per costituire un unico file eseguibile.

Uno solo dei file deve contenere la definizione della funzione principale `main`.

L'insieme dei file sorgenti viene spesso chiamato **progetto**.

In ciascun file si collocano funzioni che insieme forniscono una certa funzionalità.

Ciascuno dei file si comporta come una **libreria di funzioni**, salvo che queste vengono compilate (e non solo linkate) con il programma principale.

Un file con funzioni specifiche per fornire una determinata funzionalità può essere facilmente riutilizzato in altri programmi: basta includerlo nel progetto.

Ciascun file ha bisogno delle sole direttive `#include` e `#define` necessarie al codice di quel file.

Per usare in un file una funzione dichiarata in un altro file (non può essere `static`), si deve indicarne il prototipo (`extern opzionale`).

Spesso si raggruppano tutte le `#define`, le variabili esterne e i prototipi di tutte le funzioni (non può essere `static`) di un progetto in un unico file `.h` e i file `.c` del progetto che ne hanno bisogno lo includono (con virgolette):

- `#include "mioheader.h"`

Le variabili esterne usate in tutti i file devono essere definite solo in uno dei file, mentre gli altri devono avere la dichiarazione `extern` corrispondente.

Non si può usare `extern` con variabili esterne con specificatore di classe di allocazione `static` in un altro file (sono utilizzabili solo dalle funzioni di quel file).

## Linkage di Variabili e Funzioni

---

Il **linkage** esprime la corrispondenza di identificatori (variabili o costanti) omonimi presenti in più blocchi e/o in più translation unit diverse (linkate insieme) e/o librerie.

Un identificatore con **linkage esterno** è **visibile in più translation unit** (es. variabili e funzioni esterne non `static`).

Un identificatore con **linkage interno** è **visibile solo nella translation unit dove è definito** (es. variabili e funzioni `static`).

Un identificatore **non ha linkage** se è **locale al blocco dove è definito** (es. variabili locali, parametri di funzioni, tag, membri, ecc.)

## [Lezione 20]

---

### Sommario

- Puntatori a Funzione
- La Funzione `main`

### Puntatori a Funzione

---

#### Indirizzo di una Funzione

Si può chiamare una funzione utilizzando l'indirizzo di memoria dal quale inizia il codice eseguibile della funzione stessa.

L'indirizzo di memoria di una funzione può essere assegnato ad una variabile puntatore, memorizzato in un vettore, passato ad una funzione, restituito da una funzione.

## Definizione di Variabili

- `tipo (*nome)(parametri)`

Definisce la variabile `nome` come **puntatore ad una funzione che restituisce un valore del tipo indicato e richiede i parametri indicati**.

Le parentesi intorno al nome sono necessarie, altrimenti si ha:

- `tipo *nome(parametri)`

che costituisce il **prototipo di una funzione che restituisce un puntatore del tipo indicato e richiede i parametri indicati**.

**Qualche esempio:**

- `double (*fp)()`
  - definisce `fp` come variabile puntatore ad una funzione che restituisce un `double`, nulla è indicato per i suoi parametri, quindi non vengono controllati.
- `double (*fp)(double, double)`
  - definisce `fp` come variabile puntatore ad una funzione che ha come parametri due `double` e restituisce un `double`
- `int (*fpv[3])(long)`
  - definisce `fpv` come vettore di 3 elementi, ciascuno dei quali è un puntatore ad una funzione che ha come parametri un `long` e restituisce un `int`

Le definizioni esterne inizializzano a `NULL` le variabili puntatori a funzioni.

Il nome di una funzione equivale al suo indirizzo di memoria (come per i vettori).

Con i seguenti prototipi:

```
double pow(double, double);
double atan2(double, double);
int f1(long), f2(long);
```

si possono avere le seguenti inizializzazioni:

```
double (*fp)(double, double) = NULL;
double (*fp)(double, double) = pow;
int (*fpv[3])(long) = {f1, f2};           // +1 NULL
int (*fpv[3])(long) = {NULL};            // 3 NULL
```

## Assegnazione e Confronto

Il nome di una funzione equivale al suo indirizzo di memoria (come per i vettori).

L'operatore `&` è ininfluente se applicato al nome di una funzione.

- **Esempi** (l'operatore `&` può essere omesso):

```
fp = &atan2;
fp = atan2;
fpv[2] = &f1;
fpv[2] = f1;
```

il confronto fra puntatori a funzione è un normale confronto tra puntatori, permette ad esempio di determinare se il puntatore si riferisce ad una certa funzione o no, oppure se è `NULL`:

```
if(fp == pow){  
    printf("fp punta a pow\n");  
}
```

## Chiamata della Funzione

La funzione il cui puntatore è stato memorizzato nella variabile `fp` può essere richiamata in due modi equivalenti:

- `(*fp)(argomenti)`
- `fp(argomenti)`

Il **primo metodo** rende più evidente che si tratta di un puntatore a funzione e non del nome di una funzione, il **secondo** è più leggibile e comodo:

```
x = (*fp)(2.23, 4.76);  
x = fp(2.23, 4.76);  
  
i = (*fpv[1])(22);  
i = fpv[1](22);
```

## Passaggio a Funzione

Un puntatore a funzione può essere passato ad una funzione come argomento.

Questo permette di passare ad una funzione puntatori a funzioni diverse per ottenere comportamenti diversi dalla stessa funzione.

Il **parametro attuale** deve essere il **nome di una funzione**.

Il **parametro formale** deve essere il **prototipo delle funzioni chiamabili** (è lì che viene definito il nome del puntatore visto all'interno della funzione stessa)

- **Esempio:** si supponga di avere le due funzioni seguenti

```
int piu(int a, int b); // Somma di due numeri interi  
int meno(int a, int b); // Differenza di due numeri interi  
  
int piu(int a, int b){  
    return a + b;  
}  
int meno(int a, int b){  
    return a - b;  
}  
int calc(int x, int y, int (*funz)(int, int)){  
    return (*funz)(x, y);  
}  
  
int main(){  
    int m, n;  
    m = calc(12, 23, piu);      // Calcola la somma dei due valori  
    n = calc(12, 23, meno);    // Calcola la differenza dei due valori  
    ...
```

```
}
```

## Valore Restituito da una Funzione

Una funzione può restituire un puntatore a funzione.

Si definisce con l'istruzione `typedef` il tipo del `puntatore-a-funzione` restituito dalla funzione chiamata:

- `typedef int (*TipoFunz)(int, int)`

`TipoFunz` è un nuovo tipo: `puntatore-a-funzione-che-ha-2-argomenti-int-e-restituisce-un-int`, e lo si usa nel solito modo:

- `TipoFunz operaz(int x)`

```
typedef int (*TipoFunz)(int, int);
int piu(int a, int b){
    return a + b;
}
int meno(int a, int b){
    return a - b;
}
TipoFunz operaz(int f){
    static TipoFunz fpv[2] = {piu, meno};
    return fpv[f];
}

int main(){
    int y;
    TipoFunz op;          // Variabile puntatore
    op = operaz(1);       // Scelta operazione
    y = (*op)(12, 23);
}
```

## La Funzione main

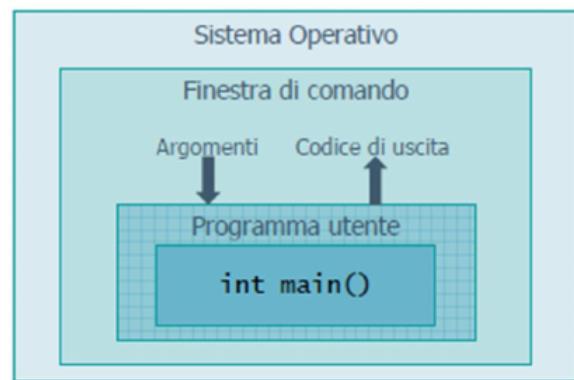
La funzione `main()`, presente in tutti i programmi C, è una funzione come le altre, ma ha, come unica particolarità, quella di **venire chiamata dal sistema operativo appena il programma viene avviato**:

- non esiste mai una chiamata a `main()`
- l'interfaccia della funzione viene definita dalle caratteristiche del sistema operativo

Ricordiamo che il linguaggio C si è evoluto con interfacce a caratteri. L'attivazione di un programma avviene digitandone il nome in una finestra di comando

La finestra di comando permette di passare al programma una serie di argomenti:

- nessuna, una o più stringhe di testo
- utilizzati dal programma come dati in ingresso



### Numero variabile di argomenti

- Anche nessuno

### Tipo di argomenti

- Numeri
- Caratteri o stringhe

Il sistema operativo non ha modo di sapere quanti parametri servono al programma, né di che tipo. Verranno, perciò, **trattati in modo standardizzato**.

Gli argomenti sulla linea di comando vengono trattati come un **vettore di stringhe**.

Il programma riceve:

- una **copia del vettore di stringhe**
- un **valore numerico** che indica **quante stringhe sono presenti**

```

C:\progr>quadrato          ➤ Numero argomenti = 0
C:\progr>quadrato 5         ➤ Numero argomenti = 1
                           ➤ Argomento 1 = "5"
C:\progr>quadrato 5 K       ➤ Numero argomenti = 2
                           ➤ Argomento 1 = "5"
                           ➤ Argomento 2 = "K"

```

- `int argc`: numero di parametri sulla linea di comando

- Incrementato di uno, in quanto **il nome del programma viene considerato come un parametro**
  - Se non vi sono argomenti vale `1`
  - Se vi sono  $k$  argomenti vale  `$k + 1$`
- `char* argv`: vettore di stringhe, ognuna delle quali è un parametro del programma
  - Vi sono `argc` diverse stringhe
    - La prima stringa (`argv[0]`) è il nome del programma
    - La seconda stringa (`argv[1]`) è, se esiste, il primo argomento
    - E così via...

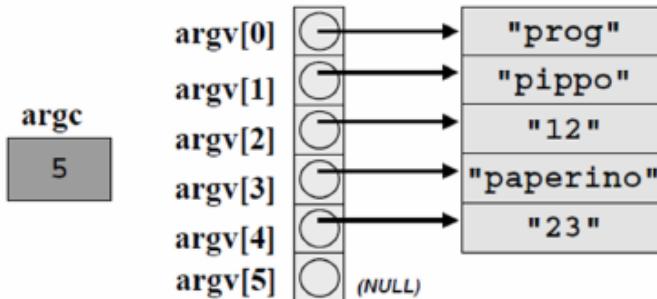
Il vettore `argv` contiene i dati sotto forma di stringa, ciò significa che, se abbiamo bisogno di dati numerici, **occorre effettuare la conversione da stringa a numero**:

```
i = atoi(argv[1]);
r = atof(argv[1]);
```

Se, invece, il parametro è una stringa, è possibile:

- utilizzare un puntatore alla stringa di interesse
- copiarlo in una variabile

C:\progr>**progr pippo 12 paperino 23**



```
#include <stdio.h>
int main(int argc, char *argv[])
{
    int i ;
    printf("argc = %d\n", argc) ;
    for(i=0; i<argc; i++)
    {
        printf("argv[%d] = \"%s\"\n",
               i, argv[i]) ;
    }
}
```

## Esempio

Scrivere un programma che, dato un carattere e una stringa da linea di comando, conti quante volte il carattere compare nella stringa

- C:\> contaOccorrenze c stringa

## Algoritmo

- Recuperare gli argomenti
- Scandire la stringa carattere per carattere, contando le occorrenze del carattere dato
- Scrivere il numero di occorrenze a video

## Codice

```
#include <stdio.h>

int main(int argc, char* argv[]){
    int cont = 0, i = 0;
    char ch = argv[1][0];
    char *s = argv[2];

    while(s[i] != '\0'){
        if(s[i++] == ch){
            cont++;
        }
    }
    printf("Totale occorrenze: %d", cont);
}
```

## Il Preprocessore

Modifica il codice C prima che venga eseguita la traduzione vera e propria.

Le direttive al preprocessore riguardano:

- **inclusione di file** (`#include`)
- **definizione di simboli** (`#define`)
- **sostituzione di simboli** (`#define`)
- **compilazione condizionale** (`#if`)
- **macroistruzioni con parametri** (`#define`)

Non essendo istruzioni C non richiedono il `;` finale.

La riga con `#include` viene sostituita dal contenuto testuale del file indicato

- `#include <stdio.h>`

Il nome del file può essere completo di percorso e ha due forme:

- `#include <file>`: il file viene cercato nelle directory del compilatore
- `#include "file"`: il file viene cercato prima nella directory dove si trova il file C e poi, se non trovato, nelle directory del compilatore

I file inclusi possono a loro volta contenere altre direttive `#include`.

La direttiva `#include` viene in genere collocata in testa al file sorgente C, prima della prima funzione. Generalmente i file inclusi non contengono codice eseguibile ma solo dichiarazioni e altre direttive

- `#define nome`

definisce il simbolo denominato `nome`

- `#define DEBUG`

I simboli vengono utilizzati dalle altre direttive del preprocessore (ad esempio, si può verificare se un simbolo è stato definito o no con `#if`).

Lo scope di nome si estende dalla riga con la definizione fino alla fine di quel file sorgente e non tiene conto dei blocchi di codice.

`nome` ha la stessa sintassi di un identificatore (nome di variabile), non può contenere spazi e viene, per convenzione, scritto in maiuscolo.

- `#undef nome`

annulla una `#define` precedente

## Inclusione Condizionale

Permette di includere o escludere parte del codice dalla compilazione ed dal preprocessing stesso

```
#if espressione_1
    istruzioni
#elif espressione_2
    istruzioni
...
#else
    istruzioni
#endif
```

Solo uno dei gruppi di istruzioni sarà elaborato dal preprocessore e poi compilato.

Le espressioni devono essere costanti intere (non possono contenere `sizeof()` o `cast`) e sono considerate vere se `!= 0`.

L'espressione `defined (nome)` produce `1` se `nome` è stato definito, altrimenti restituisce `0`.

`#ifdef nome` equivale a:

- `#if defined(nome)` e verifica che `nome` sia definito

`#ifndef nome` equivale a:

- `#if !defined(nome)` e verifica che `nome` non sia definito

Nel caso in cui un file incluso ne includa altri a sua volta, per evitare di includer più volte lo stesso file, si può usare lo schema seguente (quello che segue è il file `hdr.h`):

```
#ifndef HDR
#define HDR
    // Contenuto di <hdr.h>
#endif
```

Se venisse incluso una seconda volta, il simbolo `HDR` sarebbe già definito e il contenuto non verrebbe nuovamente incluso nella compilazione.

Per escludere dalla compilazione un grosso blocco di codice (anche con commenti):

```
#if 0
    // Codice da non eseguire
#endif
```

Per isolare istruzioni da usare solo per il debug:

```
#ifdef DEBUG
    printf("valore di x: %d\n", x);
#endif
```



# Algoritmi di Ordinamento

---

Esistono sostanzialmente due tecniche:

- **Ordinamenti Interni:** si usano quando la lista dei dati non è troppo lunga e può essere memorizzata per intero nella memoria del computer, solitamente in un vettore.
  - Corrisponde all'ordinare le carte di un mazzo disponendole su un tavolo, in modo che siano visibili ed utilizzabili contemporaneamente da chi ordina.
- **Ordinamenti Esterne:** si usano per insiemi di dati molto grandi, memorizzati in file su dischi esterni o su nastri, che non conviene caricare nella loro interezza nella memoria del computer.
  - Corrisponde ad ordinare le carte disponendole in mucchietti o pile, in modo che solo la carta in cima sia visibile ed utilizzabile

Un'altra classificazione possibile si basa sull'efficienza o sull'economia nel tempo di questi algoritmi: una buona misura si ottiene contando il **numero di confronti** tra chiavi e il **numero di movimenti** necessari per il riordino.

Gli Algoritmi di Ordinamento si dividono in due gruppi:

- **Metodi Diretti**
  - Tecniche semplici e ovvie
  - Adatti ad illustrare le caratteristiche dei maggiori principi di ordinamento
  - Facili da capire e realizzare
  - Complessità nell'ordine di  $n^2$ :
    - Rapidi per  $n$  abbastanza piccolo
    - Inutilizzabili per  $n$  grande
- **Metodi Avanzati**
  - Tecniche più complesse e meno intuitive
    - A differenza dei Metodi Diretti, che per ogni passo spostano un elemento per un'unica posizione, i Metodi Avanzati si basano sul principio di spostare, con un unico salto, gli elementi su distanze maggiori
  - Generalmente richiedono un numero di confronti nell'ordine di  $n \cdot \log(n)$

## Metodi Diretti

---

- **Ordinamento per Selezione (Selection Sort)**
  - Anche detto *ordinamento per minimi o massimi successivi*
- **Ordinamento per Inserimento (Insertion Sort)**
  - Basato sul concetto della *inserzione ordinata*
- **Ordinamento per Scambi (Bubble Sort)**

## Selection Sort

## Algoritmo

- Si considera il vettore di **n** elementi **[0, n - 1]**
- Si trova il minimo nell'intervallo e lo si porta nella posizione con indice più piccolo, restringendo poi l'intervallo a **[1, n - 1]**
- Si continua allo stesso modo finchè il vettore non è completamente ordinato

## Codice

```
int min, tmp;
for(int i = 0; i < n; i++){
    min = i;                                // Poniamo l'indice min uguale all'indice i attuale
    for(int j = i; j < n; j++){
        if(vet[j] < vet[min]){ // Se si trova un valore nel vettore minore di vet[min]
            min = j;          // allora aggiorniamo l'indice min
        }
    }
    tmp = vet[min];                         // Alla fine della ricerca del minimo
    vet[min] = vet[j];                      // globale eseguiamo l'operazione di
    vet[j] = tmp;                          // scambio tra vet[i] e vet[min]
}
```

## Insertion Sort

### Algoritmo

Si basa sull'operazione di push di un elemento **elem** in un vettore:

- Riceca della posizione **i** del vettore dove inserire **elem** ( $vet[i] > elem$ )
- Spostamento in avanti di un posto di tutti gli elementi **vet[j]** con  $j = i \dots nelem$ , dove **nelem** è il numero di elementi presenti nel vettore
- Incremento di **nelem** di un'unità
- Inserzione di **elem** nel vettore

Se applicato ad un vettore pre-esistente, questo algoritmo **riordinerà il vettore**:

- **Caso 1: inserzione ordinata di elementi**
  - Si inserisce **vet[0]**
  - Per  $i = 1 \dots n - 1$ 
    - Si cerca il primo elemento **vet[j] > vet[i]** nella sottolista **vet[0]...vet[n - 1]**
    - Il nuovo elemento va aggiunto in posizione **j**
    - Gli elementi **vet[i - 1]...vet[j]** vanno spostati a destra di una posizione
    - Il nuovo elemento va assegnato a **vet[j]**:  $vet[i] \rightarrow vet[j]$
- **Caso 2: riordinamento di un vettore preesistente**
  - Come nel caso precedente
    - Al ciclo *i-esimo* viene prelevato **vet[i]**
    - **vet[i]** va posizionato al posto giusto in **vet[0]...vet[i]**
    - Gli elementi ancora da ordinare restano nella posizione iniziale

## Codice

```
int j, tmp;
for(int i = 0; i < n; i++){
    tmp = vet[i];                                // si copia vet[i] in tmp per
    "salvarlo"
    j = 0;
    while((vet[j] < vet[i]) && (j < i)){      // si pone j = 0 ad ogni iterazione
        j è minore dell'indice i                  // Fino che vet[j] < vet[i] e l'indice
        j++;                                         // si incrementa l'indice j
    }
    for(int k = i - 1; k >= j; k--){            // si trasla tutto l'array a destra
        vet[k + 1] = vet[k];                      // di una posizione
    }
    vet[j] = tmp;                                // si posiziona in vet[j] il valore
    salvato precedentemente
}
```

## Complessità

Lo spostamento  $i$ -esimo richiede  $i + 1$  operazioni (tra inserzione e spostamenti)

- Se inserisco un elemento al posto di  $k$  avrò  $k$  confronti e  $i - k$  spostamenti, più l'inserimento:
  - **N. operazioni:**  
con  $i = 2 \dots n$
  - **N. medio di operazioni per inserimento:**
- In definitiva
  - che è dell'ordine di  $n^2$

## Bubble Sort

### Algoritmo

Ordinamento mediante scambi tra *coppie successive*

- Si esamina la prima coppia (**vet[0], vet[1]**) e, se non è ordinata, si esegue lo scambio
- Si prosegue allo stesso modo per tutte le coppie fino ad arrivare alla coppia (**vet[n - 2], vet[n - 1]**)
- Al termine del ciclo l'elemento massimo occuperà l'ultima posizione del vettore
- Il procedimento viene ripetuto per il sottoinsieme **vet[0]...vet[n - 2]** e così via fino ad arrivare al sottoinsieme **vet[0]...vet[1]**

Il riordinamento ha diversi **casi**

- **Worst Case:** lista contro-ordinata
  - $n - 1$  cicli
  - Per ogni ciclo  $n - i$  confronti, con  $i = 1 \dots n - 1$ 
    - 
    -
  - N. medio confronti:
  - Totale confronti:

- **Best Case:** lista già ordinata
  - $n - 1$  confronti nell'unico ciclo
- **Average Case:** distribuzione uniforme degli elementi
  - Metà del caso peggiore:

Ne deriva che la complessità sia dell'ordine di  $n^2$

## Codice: Versione Iterativa

```

int nswaps = 0, ncycles = 0, tmp; // Definiamo i contatori di cicli e scambi e
// la variabile temporanea
for(int i = 1; i < n; i++){
    ncycles++; // Incrementiamo ncycles ad ogni nuovo ciclo
    for(int j = 0; j < n - 1; j++){
        if(vet[j] > vet[j + 1]){ // se l'elemento in posizione j è più grande
            dell'elemento in posizione
            nswaps++; // j + 1, incrementa il contatore degli
            scambi
            tmp = vet[j]; // ed esegui lo scambio
            vet[j] = vet[j + 1];
            vet[j + 1] = tmp;
        }
    }
}
printf("Sono stati eseguiti %d scambi\n", nswaps); // Stampa degli scambi
// totali
printf("Sono stati eseguiti %d cicli\n", ncycles); // e dei cicli totali

```

## Codice: Versione Migliorata

```

int nswaps = 0, ncycles = 0, i, tmp; // Definiamo i contatori, l'indice i
// e la variabile tmp
bool scambi = TRUE; // Definiamo la variabile booleana
scambi inizializzata a TRUE
i = 1; // Inizializziamo a 1 l'indice i
while((i <= n - 1) && (scambi == TRUE)){
    scambi = FALSE; // Poniamo a FALSE la variabile
    scambi
    ncycles++; // Incrementiamo il contatore dei
    cicli
    for(int j = 0; j < n - 1; j++){
        if(vet[j] > vet[j + 1]){ // se l'elemento in posizione j è
            più grande dell'elemento
            nswaps++; // in posizione j + 1, allora
            incrementiamo il contatore
            tmp = vet[j]; // degli scambi ed eseguiamo lo swap
            tra vet[j] e
            vet[j] = vet[j + 1]; // vet[j + 1]
            vet[j + 1] = tmp;
            scambi = TRUE;
        }
    }
    i++;
}
printf("Sono stati eseguiti %d scambi\n", nswaps); // Stampa degli scambi
// totali

```

```
printf("Sono stati eseguiti %d cicli\n", ncycles); // e dei cicli totali
```

## Codice: Versione Personale

```
int tmp, swap; // Definiamo la variabile tmp e la swap
che useremo a mò di bool
do{
    swap = 0; // Inizializziamo a FALSE la variabile
    swap ad ogni ciclo
    for(int i = 0; i < n - 1; i++){
        if(vet[i] > vet[i + 1]){ // Se l'elemento in posizione j è più
            grande dell'elemento in
            tmp = vet[i]; // posizione j + 1 allora eseguiamo lo
            swap // e poniamo a TRUE la variabile swap
        }
    }
} while(swap == 1); // Il ciclo si ripete finchè viene
eseguito almeno uno swap
```

## Complessità

In termini di complessità è ovvio che il **Bubble Sort** sia l'algoritmo **più complesso** tra i tre appena elencati

## Counting Sort

Si basa sull'ipotesi che ognuno degli **n** elementi in input sia un intero nell'intervallo **[1, k]** dove **k** è un numero non troppo grande.

Si determina, per ogni elemento **x** in input, il numero di elementi minori di **x**. Questa informazione può essere usata per porre l'elemento **x** direttamente nella sua esatta posizione nell'array di output.

- **Es:** se ci sono 10 elementi minori di **x**, allora **x** va posizionato all'undicesima posizione nell'array di output

L'algoritmo deve gestire situazioni in cui alcuni elementi abbiano lo stesso valore, infatti non si vuole metterli tutti nella stessa posizione nell'array di output

## Caratteristiche

- **Veloce**, perchè fa delle ipotesi sull'input, infatti assume che l'input consista di numeri interi in un piccolo intervallo
- **Stabile**
- **Richiede 3 array**
  - **Array A:** array di dimensione **n**, contenente gli **n** numeri da ordinare
  - **Array B:** array di dimensione **n**, contenente la sequenza ordinata degli **n** numeri
  - **Array C:** array **temporaneo di lavoro** di dimensione **k**, dove **k** è il massimo numero trovato in **A**

## Codice

```
int i = 0, max = 0;
// Vettore A preesistente
int *B = NULL, *C = NULL;
for(int i = 0; i < n; i++){
    if(vet[i] > max){
        max = A[i];
    }
}
printf("Il massimo è: %d", max);

// Allocazione dinamica vettore B
B = (int*)calloc(n, sizeof(int));
if(!B){
    printf("Memoria insufficiente\n");
    return -1;
}
// Allocazione dinamica vettore C
C = (int*)calloc(max + 1, sizeof(int));
if(!C){
    free(B);
    printf("Memoria insufficiente\n");
    return -1;
}

// Riempiamo il vettore C
for(int i = 0; i < n; i++){
    +C[A[i]];
}
// Accumuliamo le posizioni in C
for(int i = 1; i <= max; i++){
    C[i] = C[i] + C[i - 1];
}
// Ordiniamo costruendo B
for(int i = n - 1; i >= 0; i--){
    B[C[A[i]] - 1] = A[i];
    C[A[i]] = C[A[i]] - 1;
}

for(int i = 0; i < n; i++){
    A[i] = B[i];
    free(B);
    free(C);
}
```

# [Lezione 23] Linguaggio C: Liste

## Sommario

- Tipi Ricorsivi
- Le Liste
- Liste Semplicemente Concatenate
- Operazioni su Liste Semplicemente Concatenate

## Tipi Ricorsivi

### Definire i Tipi Ricorsivi

```
typedef struct{
    char nome[20];
    char cognome[20];
    int peso;
    Persona padre;
} Persona;
```

Concettualmente sbagliato, ricorsione infinita.

```
typedef struct{
    char nome[20];
    char cognome[20];
    int peso;
    Persona *padre;
} Persona;
```

Concettualmente giusto, ma **non compila** perchè, al momento della dichiarazione del campo `padre` il tipo `Persona` non esiste ancora.

```
typedef struct P{
    char nome[20];
    char cognome[20];
    int peso;
    struct P *padre;
} Persona;

Persona a, b;
a.padre = &b;
```

```

typedef struct Persona{
    char nome[20];
    char cognome[20];
    int peso;
    struct Persona *padre;
};

struct Persona a, b;
a.padre = &b;

```

## Liste

Una lista è una **struttura dati ricorsiva**, ovvero una *struttura dati formata da elementi dello stesso tipo e collegati insieme*, la cui **lunghezza può variare dinamicamente**.

I suoi elementi sono **variabili dinamiche** e vengono creati e/o distrutti a tempo di esecuzione producendo una struttura dati che **cresce o diminuisce a seconda delle esigenze** del programma in esecuzione.

È possibile implementare liste anche tramite array, ma ciò può avvenire solo quando si conoscono esattamente le dimensioni della lista.

Ogni elemento di una lista è definito come una struttura **costituita da uno o più campi dati e da un campo puntatore contenente l'indirizzo dell'elemento successivo**.

```

struct elem{
    int info;
    struct elem *succ;
};

```

### • Strutture Concatenate

Una struttura è detta **concatenata** quando è **costituita**, oltre che dai suoi normali membri, **anche da uno o più membri aggiuntivi**, dichiarati come puntatori alla struttura stessa.

```

struct rec{
    int info;
    struct rec *next;
};

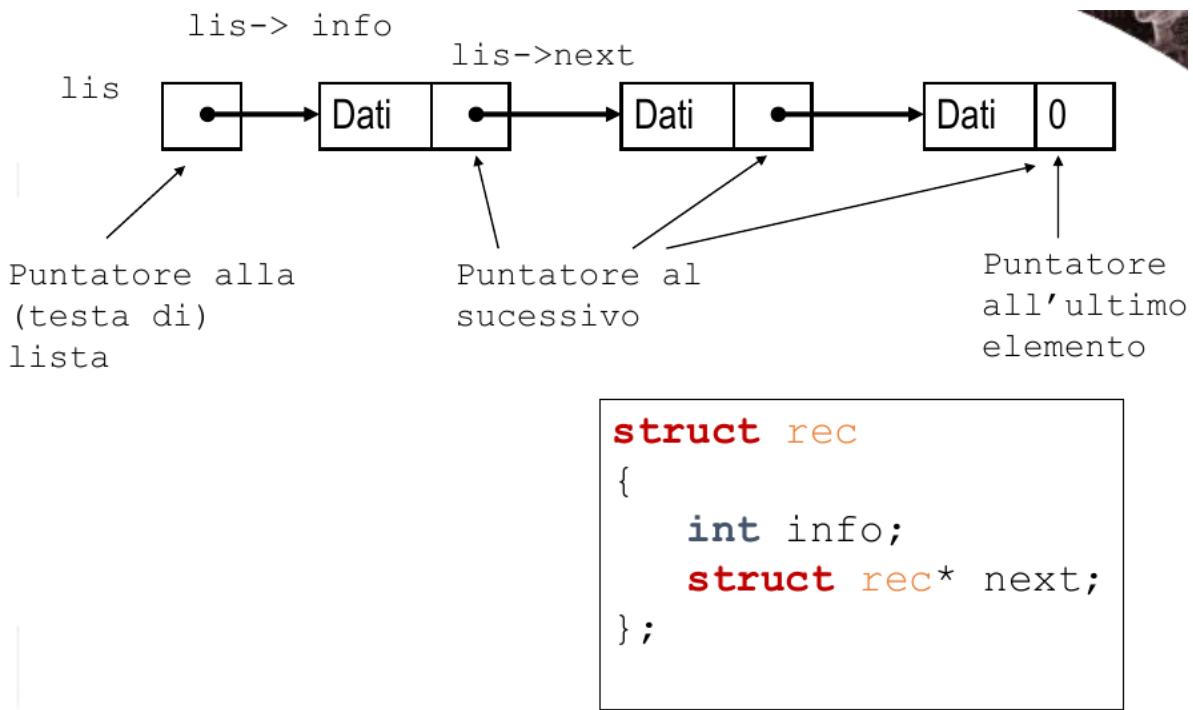
```

**La definizione** di una struttura concatenata è, di solito, **accompagnata da un certo numero di funzioni**, che hanno il compito di gestirla, cioè di **eseguire le operazioni** di inserimento, eliminazione e ricerca di oggetti.

Ogni lista è definita da una **variabile puntatore** che **punta al primo elemento della lista**.

Nel caso di assenza di elementi (lista vuota) tale variabile puntatore **assume valore NULL**.

In una lista il campo puntatore dell'ultimo elemento **assume sempre valore NULL**.



## Allocazione Dinamica di Liste

L'allocazione dinamica della memoria si presta alla gestione di liste di oggetti, **quando il loro numero non è definito a priori**.

Queste liste **possono aumentare e diminuire di dimensioni dinamicamente** in base al flusso del programma, e quindi devono essere gestite in un modo più efficiente dell'allocazione di memoria permanente sotto forma di array.

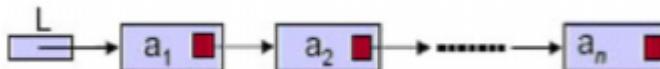
Una **lista concatenata (linked list)** è un insieme di oggetti, caratterizzati dal fatto di essere **istanze di una struttura concatenata**.

In ogni oggetto, i membri puntatori alla struttura **contengono l'indirizzo di altri oggetti della lista**, creando così un legame fra gli oggetti e rendendo la stessa lista **percorribile**, anche se gli oggetti non sono allocati consecutivamente in memoria.

Se la struttura possiede un solo membro puntatore a sé stessa allora è detta **single-linked** (o **monodirezionale**), se ne possiede due è detta **double-linked** (o **bidirezionale**).

## Tipi di Liste

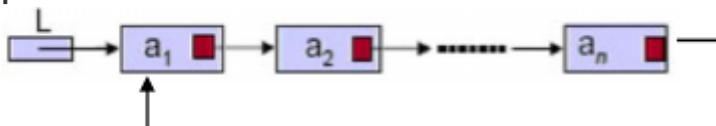
- **Lista monodirezionale**: può essere visitata in **un solo senso**. Un solo puntatore per nodo



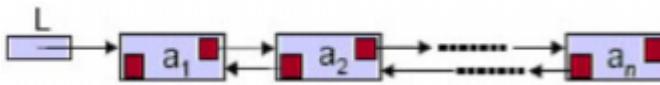
- **Lista monodirezionale con sentinella**: ha una cella in più detta **sentinella**, che è direttamente indirizzata da **L**, ma non contiene il valore di alcun elemento



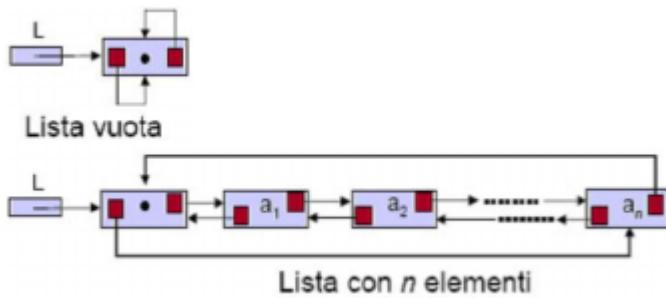
- **Lista monodirezionale circolare**: l'ultimo puntatore non ha valore **NULL**, ma **punta al primo nodo**



- **Lista bidirezionale**: può essere visitata nei **due sensi**. Due puntatori per nodo



- **Lista bidirezionale circolare:** il puntatore all'elemento successivo dell'ultimo nodo punta al primo elemento della lista e il puntatore all'elemento precedente del primo punta all'ultimo elemento della lista



## Operazioni su Liste

- Creazione della Lista (vuota e successivi inserimenti)
- Lettura di una Lista
- Stampa di una Lista
- Cancellazione di una Lista
- Inserimento in lista
- Estrazione da Lista

### Struttura del nodo:

```
typedef struct Nodo_SL{
    int dato;
    struct Nodo_SL *next;
} Nodo_SL;
```

### Struttura della lista:

```
typedef struct Lista_SL{
    Nodo_SL *next;
} Lista_SL;
```

## Creazione della Lista

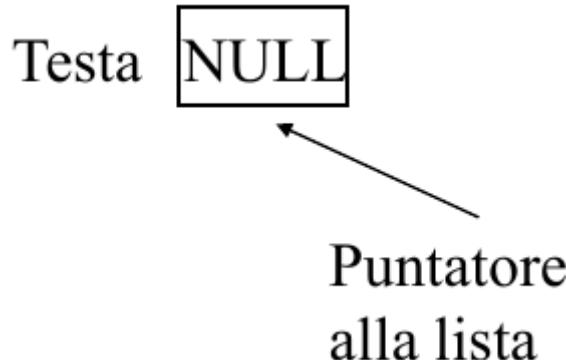
Per **creare una lista**, basta definirla, ovvero è sufficiente **creare il modo di riferirsi ad essa**.

L'unica cosa che esiste sempre della lista è la sua **testa** (o **radice**), ossia il **puntatore al suo primo elemento**.

Questa è l'unica componente allocata staticamente ed è inizializzata a `NULL`, poichè all'inizio (creazione della lista) non punta a niente in quanto non ci sono elementi.

- **Esempio:**

```
Lista_SL Testa;
Testa.next = NULL;
```

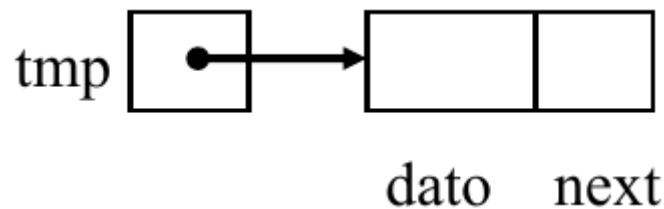


## Creazione di un Nuovo Nodo

La creazione di un nuovo nodo (in qualsiasi fase dell'esistenza di una lista) avviene **creando una nuova istanza della struttura tramite allocazione dinamica**, utilizzando, di solito, un puntatore di appoggio (`tmp`).

- **Esempio:**

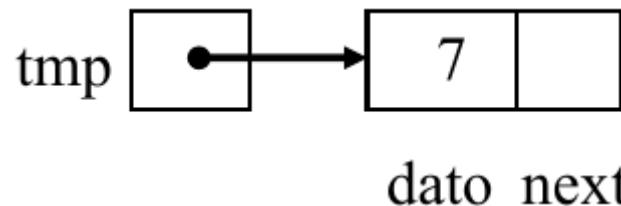
```
Nodo_SL *tmp = (Nodo_SL *)malloc(sizeof(Nodo_SL));
```



## Assegnazione di Valori ai Campi Dati

L'assegnazione di valori ai campi dati si ottiene **dereferenziando il puntatore al nodo e accedendo ai singoli dati**, ovvero utilizzando direttamente l'operatore `->`

```
Nodo_SL *CreaNodo_SL(int dato){
    Nodo_SL *tmp;
    tmp = (Nodo_SL *)malloc(sizeof(Nodo_SL));
    if(!tmp){
        return NULL;
    } else{
        tmp -> next = NULL;
        tmp -> dato = dato;
    }
    return tmp;
}
```



Le operazioni di inserimento di un elemento (e, analogamente, quelle di cancellazione) possono avvenire secondo diverse modalità (ovvero in diverse posizioni della lista), assumendo di volta in volta caratteristiche specifiche.

In ogni caso l'inserimento di un nuovo elemento nella lista prevede sempre **i seguenti passi**:

1. Creazione di un nuovo nodo (allocazione dinamica)
2. Assegnazione di valori ai campi dati
3. Collegamento del nuovo elemento alla lista esistente
  - o aggiornamento del campo puntatore del nodo
  - o aggiornamento dei puntatori della lista

Queste due ultime operazioni caratterizzeranno la **tipologia dell'inserimento**

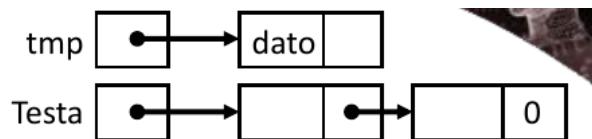
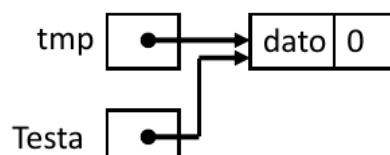
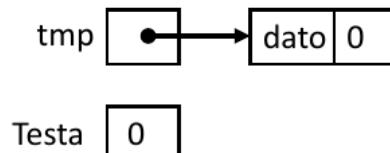
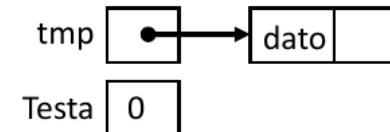
## Inserimento in Testa

Il **caso più semplice** è costituito dall'inserimento in testa, in quanto si dispone di un riferimento esplicito a questa (la testa della lista Testa).

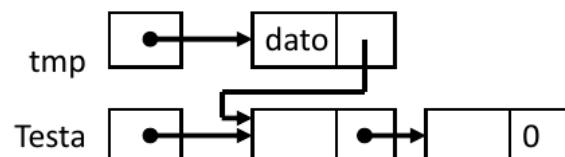
- il campo `next` del nuovo nodo punterà allo stesso valore a cui punta il campo `next` di `Testa`
- `Testa` sarà collegato al nuovo nodo:

```
tmp -> next = Testa.next;
Testa.next = tmp;
```

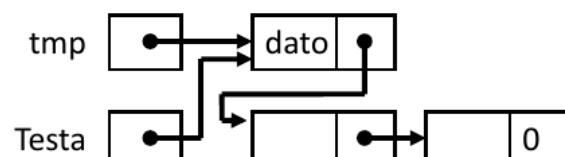
**NB:** funziona anche se la lista è vuota



**tmp->next = Testa.next;**



**Testa.next = tmp;**

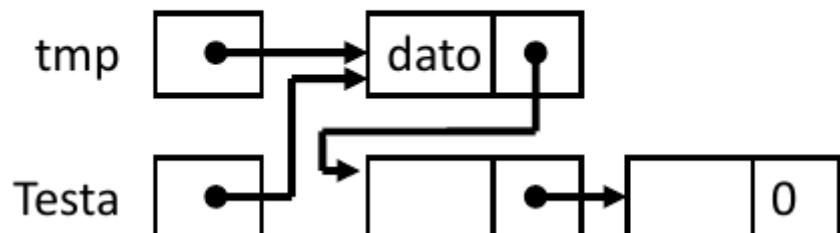
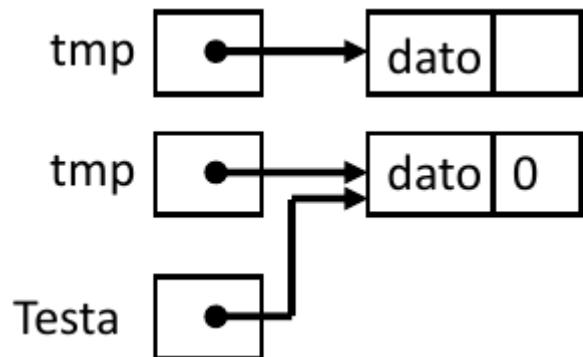


```
void InserisciInTesta_SL(Lista_SL *Testa, int dato){
    Nodo_SL *temp = NULL;
    if(!Testa -> next){
        temp = CreaNodo_SL(dato);
        if(temp){
            Testa -> next = temp;
            return;
        }
    } else{
        temp = CreaNodo_SL(dato);
        if(temp){
            temp -> next = Testa -> next;
            Testa -> next = temp;
        }
    }
}
```

```

    }
}
return;
}

```



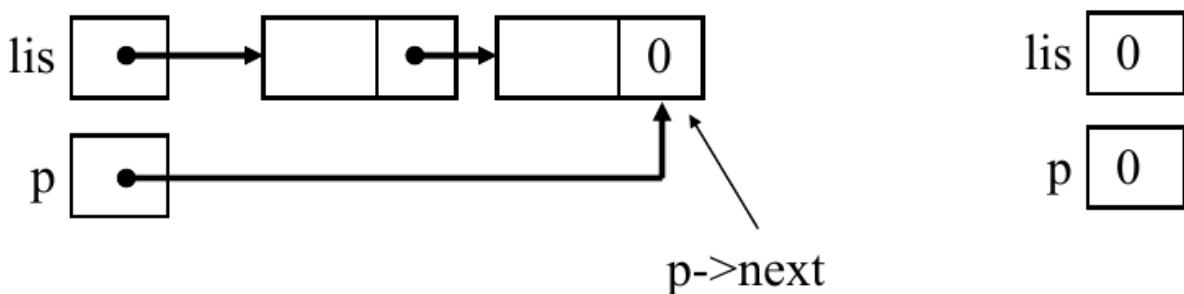
## Ricerca dell'Ultimo Elemento

Per cercare l'ultimo elemento possiamo **scorrere la lista tramite un puntatore ausiliario `p`**, inizializzato a `Testa.next`.

```

Nodo_SL *p = Testa.next;
while(p != NULL && p -> next != NULL){
    p = p -> next;
}

```



## Inserimento in Coda

L'inserimento in coda è **più complesso**, in quanto **non abbiamo un puntatore esplicito all'ultimo elemento**, ma dobbiamo prima scorrere la lista per cercarlo.

Supponiamo di averlo trovato e che sia il puntato `p`:

- il campo `next` del nuovo nodo punterà a `NULL` (in quanto è l'ultimo)

Il campo `next` dell'ex-ultimo nodo punterà al nuovo nodo

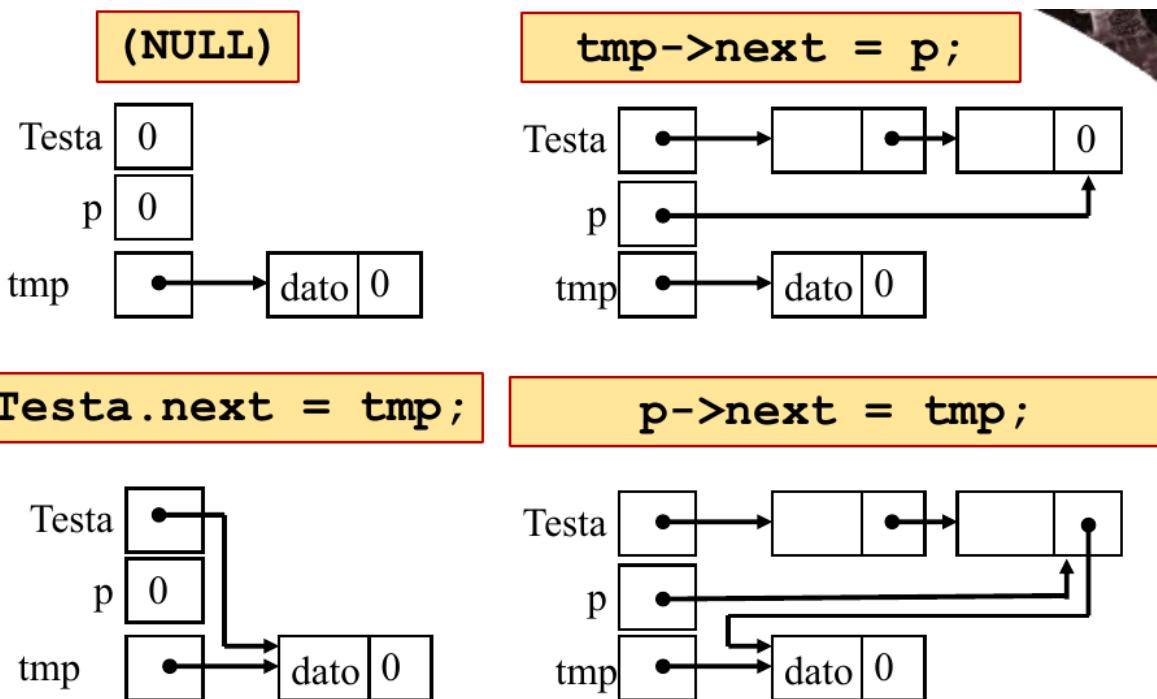
- **Esempio~:**

```

tmp -> next = NULL;
p -> next = tmp;

```

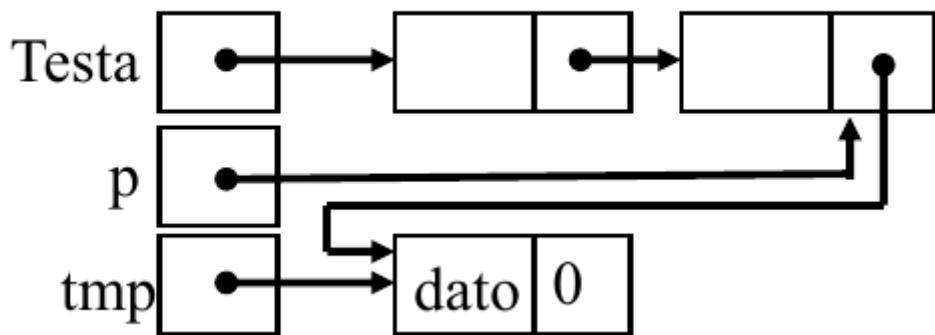
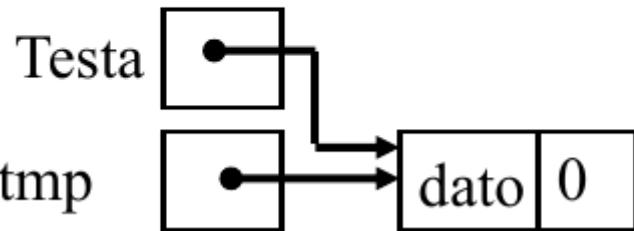
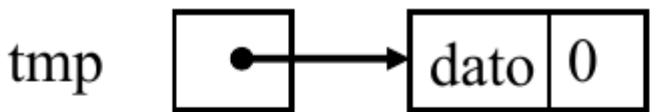
La lista vuota va gestita come un **caso particolare**



```

void InserisciInCoda_SL(Lista_SL *Testa, int dato){
    Nodo_SL *temp = NULL;
    Nodo_SL *p = Testa -> next;
    if(!Testa -> next){
        temp = CreaNodo_SL(dato);
        if(temp){
            Testa -> next = temp;
            return;
        }
    } else{
        temp = CreaNodo_SL(dato);
        while(p -> next){
            p = p -> next;
        }
        p -> next = temp;
    }
    return;
}

```



## Liberare la Lista

Liberare la lista consiste nel **deallocare sequenzialmente tutti i nodi** presenti nella lista.

```

void LiberaLista_SL(Lista_SL *Testa){
    Nodo_SL *tmp = Testa -> next;
    while(Testa -> next){
        tmp = Testa -> next;
        Testa -> next = tmp -> next;
        printf("\nElimino %d", tmp -> dato);
        free(tmp);
    }
    Testa -> next = NULL;
}

```

## Stampa degli Elementi nella Lista

Stampare la lista consiste nel mostrare a video sequenzialmente tutti i campi nei nodi della lista.

```

void StampaLista_SL(Lista_SL *Testa){
    Nodo_SL *tmp = Testa -> next;
    while(tmp && tmp -> next){
        printf("%d -> ", tmp -> dato);
        tmp = tmp -> next;
    }
    if(tmp){
        printf("%d -|", tmp -> dato);
    }
}

```

## Inserimento in una Posizione Specifica

L'inserimento in una posizione specifica **richiede** preventivamente **l'individuazione di tale posizione all'interno della lista** e dipende dalla condizione che si vuole verificare, per cui dobbiamo prima scorrere la lista per determinarla.

Nel caso di **inserimento in ordine crescente**, la lista risultante **deve rimanere in ogni momento ordinata**.

Pertanto, all'inserimento di un nuovo valore, si dovrà scorrere la lista fino alla posizione corretta per l'inserimento (ovvero, fin quando il campo `dato` dei nodi esistenti risulta minore del dato da inserire)

## Ricerca di un Elemento Qualsiasi

La condizione più sicura da utilizzare in una ricerca è riferirsi direttamente al puntatore all'elemento nella condizione di scorrimento.

In tal modo, però, si sorpassa l'elemento cercato.

Per questo nella ricerca della posizione di inserimento si usano di solito due puntatori, `p` e `q`, che **puntano rispettivamente all'elemento precedente e al successivo**.

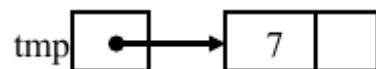
- **Esempio:**

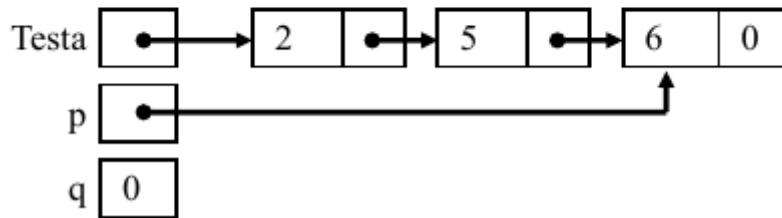
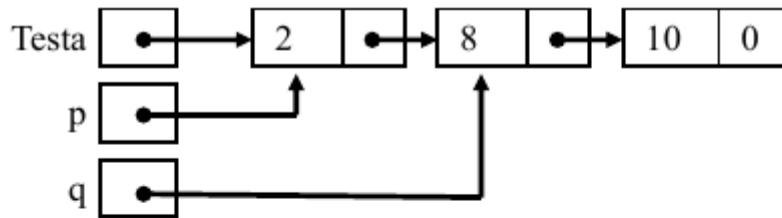
```
Nodo_SL *q = Testa.next;
Nodo_SL *p = Testa.next;
while(q != NULL && q -> dato < dato){
    p = q;
    q = q -> next;
}
```

## Ricerca di una Posizione Specifica

- **Esempio:**

```
Nodo_SL *q = Testa.next;
Nodo_SL *p = Testa.next;
while(q != NULL && q -> dato < dato){
    p = q;
    q = q -> next;
}
```





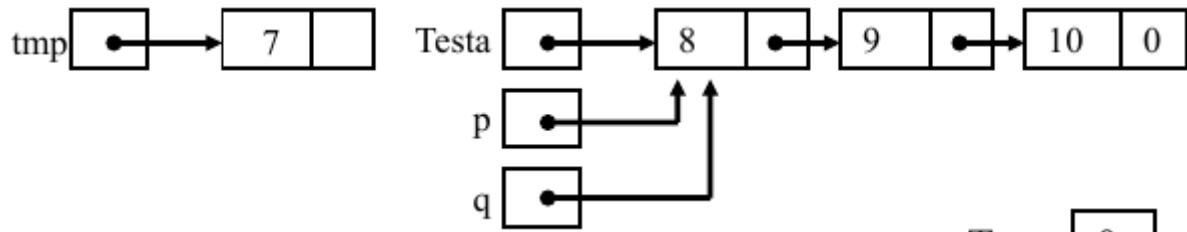
## Casi Particolari

- Esempio:

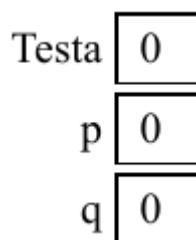
```

Nodo_SL *q = Testa.next;
Nodo_SL *p = Testa.next;
while(q != NULL && q -> dato < dato){
    p = q;
    q = q -> next;
}

```



Se `q == Testa.next` allora `p` non contiene l'elemento precedente



Quindi, nel **caso generale**:

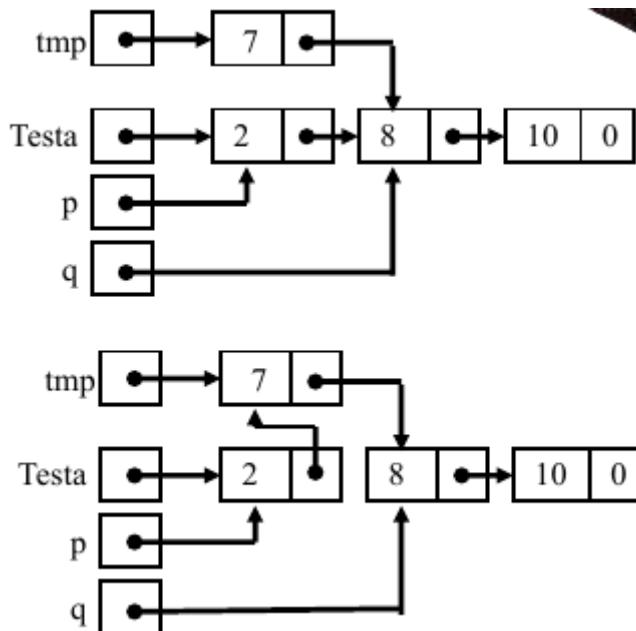
```

tmp -> next = q;
p -> next = tmp;

```

ma se `q == Testa -> next` (**inserimento in testa**):

```
Testa.next = tmp;
```



## Ricerca del Nodo Predecessore

```
Nodo_SL *CercaPredecessore_SL(Lista_SL Testa, int dato){
    Nodo_SL *q = Testa.next;
    Nodo_SL *p = Testa.next;
    while(q != NULL && (q -> dato < dato)){
        p = q;
        q = q -> next;
    }
    return p;
}
```

## Ricerca di un Dato Specifico

```
Nodo_SL *CercaElemento_SL(Lista_SL Testa, int dato){
    Nodo_SL *q = Testa.next;
    while(q != NULL && (q -> dato != dato)){
        q = q -> next;
    }
    return q;
}
```

## Inserimento in Ordine Crescente

L'inserimento di valori in ordine crescente è effettuata **inserendo i nuovi elementi subito dopo averne trovato il predecessore**.

```

void InserisciInordine_SL(Lista_SL *Testa, int dato){
    Nodo_SL *temp = NULL;
    Nodo_SL *nuovo = NULL;
    if(Testa -> next == NULL || Testa -> next -> dato > dato){
        InserisciInTesta_SL(Testa, dato);
    } else{
        nuovo = CreaNodo_SL(dato);
        temp = CercaPredecessore_SL(*Testa, dato);
        nuovo -> next = temp -> next;
        temp -> next = nuovo;
    }
    return;
}

```

## Inserimento Dopo un Elemento

L'inserimento di un valore dopo un elemento specifico è effettuata **cercando il valore e inserendo l'elemento come nodo successivo**.

```

void InserisciDopoElemento_SL(Lista_SL *Testa, int predecessore, int dato){
    Nodo_SL *temp = NULL;
    Nodo_SL *nuovo = NULL;
    if(Testa -> next == NULL){
        return;
    } else{
        temp = CercaElemento_SL(*Testa, predecessore);
        if(temp){
            nuovo = CreaNodo_SL(dato);
            nuovo -> next = temp -> next;
            temp -> next = nuovo;
        }
    }
    return;
}

```

## Ordine Decrescente

Supponiamo di voler creare una lista contenente i valori:

- $V = \{5, 9, 7, 12, 0\}, n = |V|$

in **ordine decrescente**.

```

Lista_SL Testa_tmp;
Lista_SL *tmp = NULL;
Testa_tmp.next = NULL;
for(int i = 0; i < 5; i++){
    InserisciInordine_SL(&Testa_tmp, v[i]);
}
StampaLista_SL(Testa_tmp);
tmp = Testa_tmp.next;
Testa.next = NULL;
while(tmp){
    InserisciInTesta_SL(&Testa, tmp -> dato);
    tmp = tmp -> next;
}

```

```
StampaLista_SL(Testa;)
```

## Eliminazione di un Nodo

L'eliminazione di un nodo dalla lista prevede:

- Ricerca del nodo da eliminare (se necessaria)
- Salvataggio del nodo in una variabile ausiliaria
- Scollegamento del nodo dalla lista (aggiornamento dei puntatori della lista)
- Distruzione del nodo (deallozazione della memoria)

In ogni caso, bisogna verificare inizialmente che la lista non sia già vuota!

- `if(Testa.next != NULL)`

Come per l'inserimento, il caso più semplice è costituito dall'eliminazione del nodo di testa, in quanto esiste il puntatore `Testa.next` a questo elemento.

Negli altri casi, si procede come per l'inserimento.

### Eliminazione di un Nodo di Testa

Bisogna aggiornare il puntatore alla testa `Testa.next` che dovrà puntare al nodo successivo a quello da eliminare.

```
Node_SL *tmp = Testa.next; // Salvataggio del nodo da eliminare  
Testa.next = tmp -> next; // Aggiornamento della lista  
free(tmp); // Distruzione del nodo
```

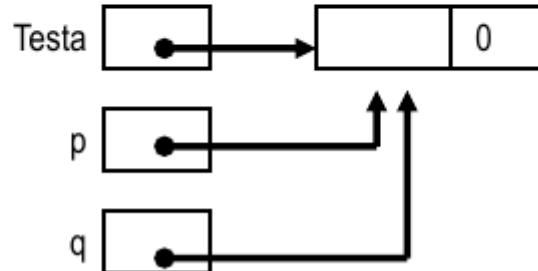
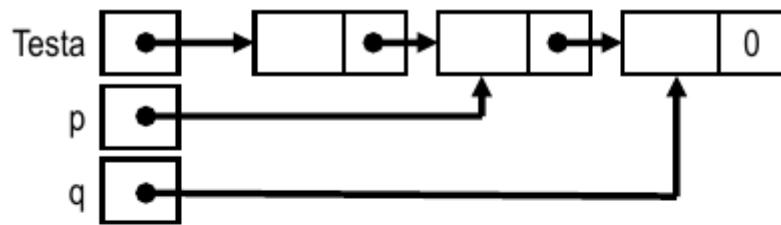
### Eliminazione del Nodo di Coda

Bisogna aggiornare il campo `next` relativo al penultimo nodo, che ora diventa l'ultimo (e quindi assume valore `NULL`).

Per questo nella ricerca della posizione di eliminazione si usano di solito due puntatori, `p` e `q`, che **puntano rispettivamente all'elemento precedente e successivo**.

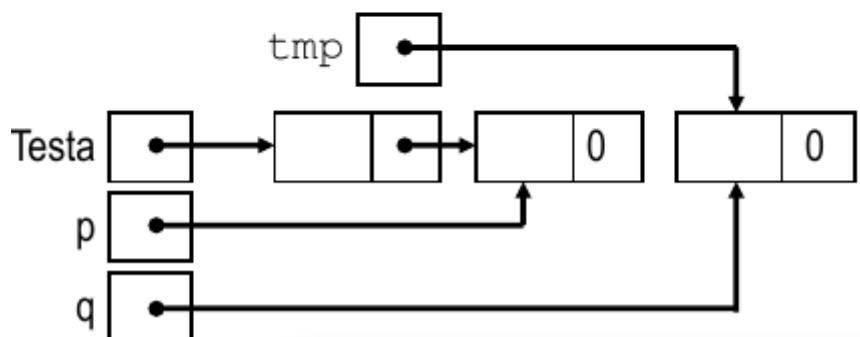
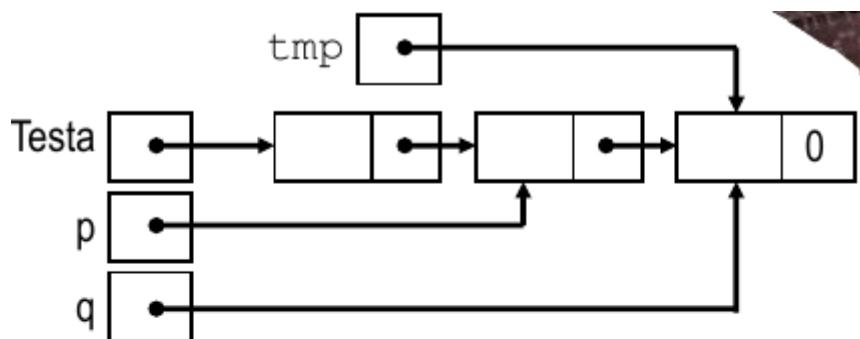
- **Esempio:**

```
Nodo_SL *q = Testa.next;  
Nodo_SL *p = Testa.next;  
while(q != NULL && q -> next != NULL){  
    p = q;  
    q = q -> next;  
}
```



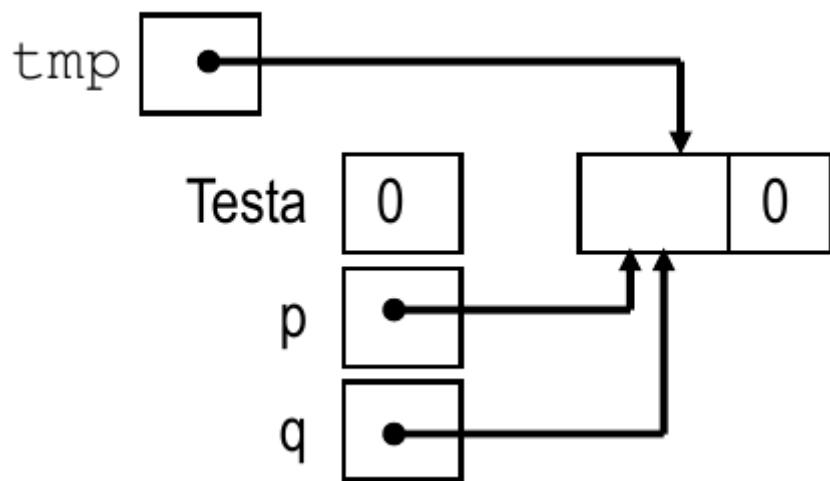
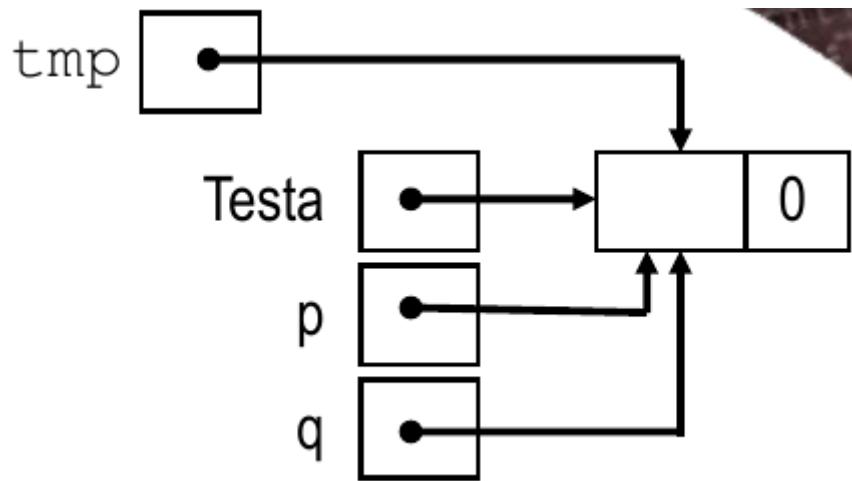
**Casi particolari:** l'elemento da eliminare è l'unico della lista

```
Nodo_SL *tmp = q;
p -> next = tmp -> next;
// Oppure
p -> next = NULL;
```



Se `q == Testa.next` allora `p` non contiene l'elemento successivo.

```
Nodo_SL *tmp = q;
Testa.next = tmp -> next;
// Oppure
Testa . next = NULL;
```



```

void CancellaInCoda_SL(Lista_SL *Testa){
    Nodo_SL *q = Testa -> next;
    Nodo_SL *p = Testa -> next;
    if(!q){
        return;
    } else{
        while(q != NULL && q -> next != NULL){
            p = q;
            q = q -> next;
        }
        p -> next = NULL;
        free(q);
    }
}

```

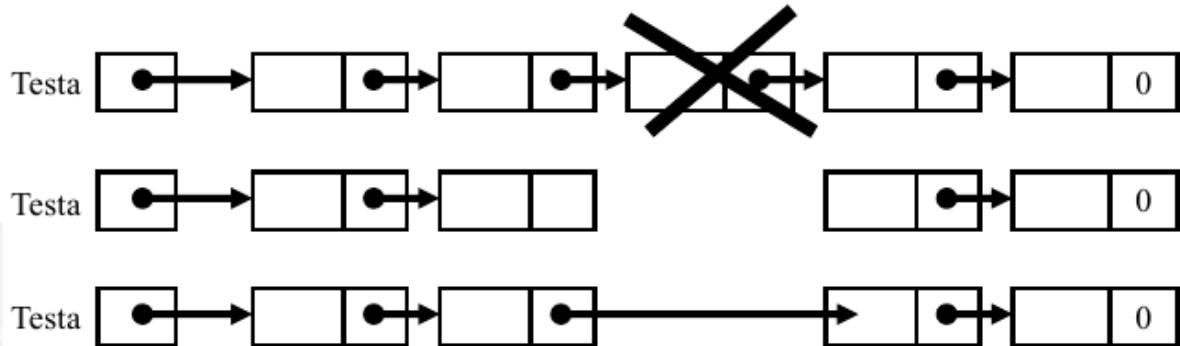
## Eliminazione di un Nodo

Individuato il nodo, **bisogna evitare che la sua rimozione spezzi la lista.**

```

// Salvataggio del nodo da eliminare
Nodo_SL *pre = CercaPredecessore_SL(Testa, dato);
Nodo_SL *nodo = pre -> next;
// Aggiornamento della lista
pre -> next = nodo -> next;
// Distruzione del nodo
free(nodo);

```



In generale è necessario aggiornare il puntatore `next` dell'elemento precedente

## [Lezione 25]

### Liste Semplicemente Concatenate

#### Pro

- Permettono di **utilizzare in modo efficiente la memoria** in fase di esecuzione. Viene allocata solo la memoria richiesta per rappresentare il contenuto attuale della lista.
- **Le operazioni di inserimento e rimozione sono efficienti**. Non è necessario riorganizzare completamente la struttura dati.

#### Contro

- È possibile attraversare la lista in **una sola direzione**: dalla testa verso la coda.
- Le operazioni che richiedono l'**attraversamento dalla coda verso la testa** sono complesse e **generalmente costose**.

### Liste Doppiaamente Concatenate

Le **liste doppiaamente concatenate** estendono la rappresentazione delle liste concatenate, **introducendo un puntatore al nodo precedente**.

A differenza delle liste semplicemente concatenate, le liste doppiaamente concatenate **possono essere attraversate facilmente in entrambe le direzioni**.

Le implementazioni di alcune funzioni sono semplificate dall'introduzione del nuovo puntatore.

Aggiungono un ulteriore overhead di memoria alla struttura dati: **è necessario memorizzare due puntatori distinti in ogni nodo**.

Non offrono vantaggi rispetto alle liste concatenate semplici relativamente all'accesso alla coda della lista

Possiamo rappresentare il tipo di dato lista come **puntatore a struttura** contenente:

- Un campo (`dato`) di tipo `int`
- Un campo puntatore (`prev`) ad una struttura dello stesso tipo
- Un campo puntatore (`next`) ad una struttura dello stesso tipo

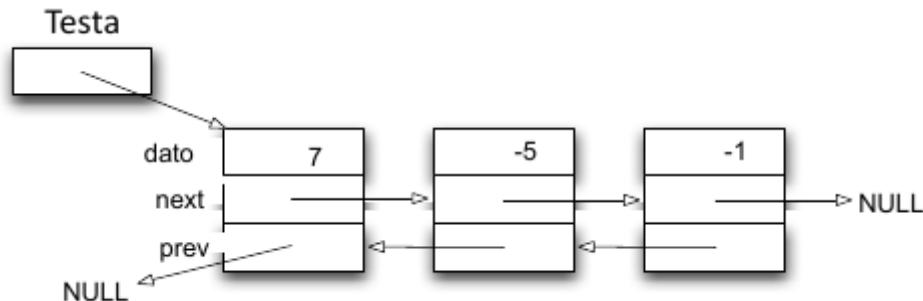
**Struttura del nodo:**

```
typedef struct Nodo_DL{
    int dato;
    struct Nodo_DL *prev;
    struct Nodo_DL *next;
} Nodo_DL;
```

### Struttura della lista:

```
typedef struct Lista_DL{
    Nodo_DL *next;
} Lista_DL;
```

Da un nodo è possibile accedere sia al nodo successivo (campo `next`) che al nodo precedente (campo `prev`) nella lista:



## Creazione della Lista

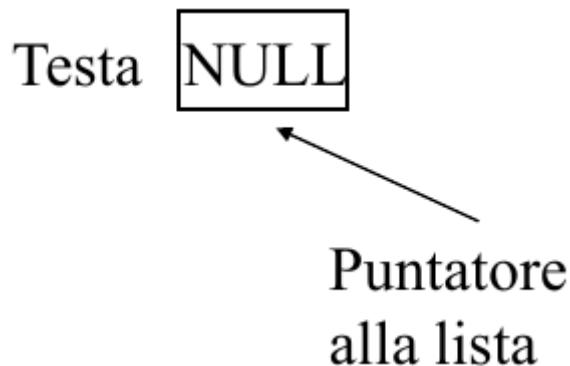
Per creare una lista basta definirla, ovvero **è sufficiente creare il modo di riferirsi ad essa**.

L'unica cosa che esiste sempre della lista è la sua **testa** (o *radice*), ossia il **puntatore al suo primo elemento**.

Questa è l'unica componente **allocata staticamente** ed è inizializzata a `NULL` poiché all'inizio (creazione della lista) non punta a niente in quanto non ci sono elementi

- **Esempio:**

```
Lista_DL Testa;
Testa.next = NULL;
```

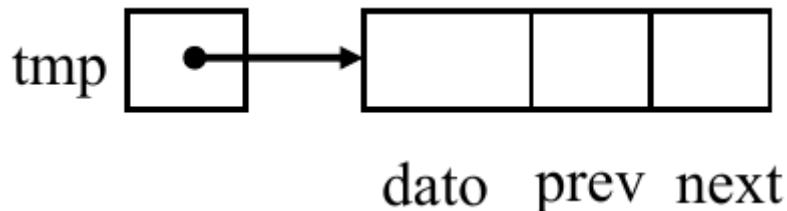


## Creazione di un Nuovo Nodo

La creazione di un nuovo nodo (in qualunque fase dell'esistenza di una lista) avviene creando una nuova istanza della struttura tramite **allocazione dinamica**, utilizzando di solito un puntatore di appoggio (`tmp`).

- **Esempio:**

```
Nodo_DL *tmp = (Nodo_DL*)malloc(sizeof(Nodo_DL));
```



L'assegnazione di valori ai campi dati si ottiene **dereferenziando il puntatore al nodo e accedendo ai singoli dati**, ovvero utilizzando direttamente l'operatore `->`

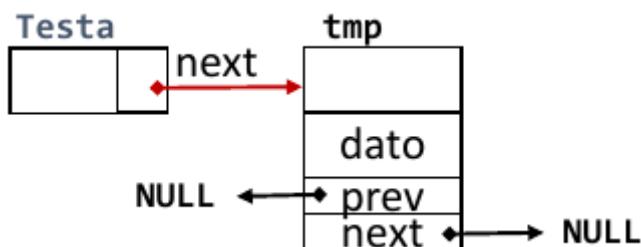
```
Nodo_DL *CreaNodo_DL(int dato){  
    Nodo_DL *tmp;  
    tmp = (Nodo_DL*)malloc(sizeof(Nodo_DL));  
    if(!tmp){  
        return NULL;  
    } else{  
        tmp -> prev = NULL;  
        tmp -> next = NULL;  
        tmp -> dato = dato;  
    }  
    return tmp;  
}
```

La funzione `CreaNodo_DL` è essenzialmente implementata come visto per le liste concatenate. Il nodo di una lista concatenata contiene il puntatore al nodo precedente oltre che quello al nodo successivo, per cui è necessario inizializzare a `NULL` entrambi i puntatori nella struttura `Nodo_DL`.

## Inserimento in Testa

Se la lista è vuota, si inserisce il nuovo nodo come unico nodo puntato da `Testa.next`.

- `Testa.next = tmp`

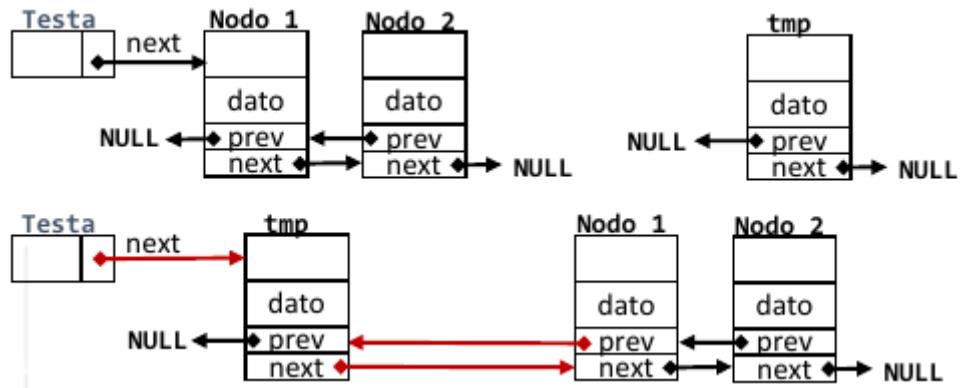


Se la lista non è vuota, si inserisce il nuovo nodo come primo nodo puntato da `Testa.next`, e si aggiornano i puntatori del nodo in testa e del nodo appena inserito:

```

tmp -> next = Testa.next;
Testa.next -> prev = tmp;
Testa.next = tmp;

```



```

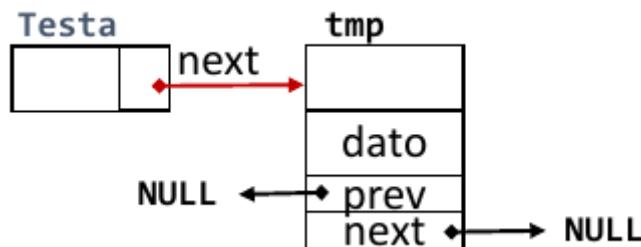
void InserimentoInTesta_DL(Lista_DL *Testa, int dato){
    Nodo_DL *tmp = NULL;
    if(!Testa -> next){
        tmp = CreaNodo_DL(dato);
        if(tmp){
            Testa -> next = tmp;
            return;
        }
    } else{
        tmp = CreaNodo_DL(dato);
        if(tmp){
            tmp -> next = Testa -> next;
            Testa -> next -> prev = tmp;
            Testa -> next = tmp;
        }
    }
    return;
}

```

## Inserimento in Coda

Se la lista è vuota, si inserisce il nuovo nodo come unico nodo puntato da `Testa.next`.

- `Testa.next = tmp`

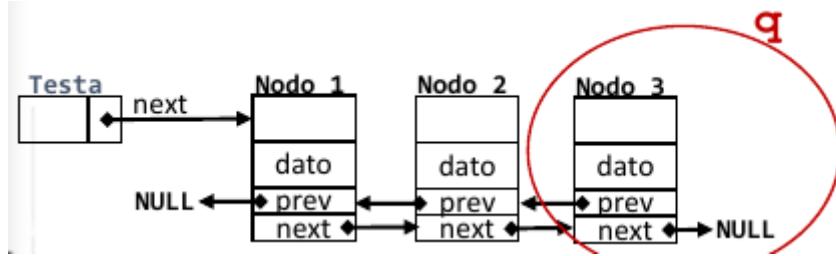


Se la lista non è vuota, troviamo il nodo di coda scorrendo la lista a partire da `Testa.next`.

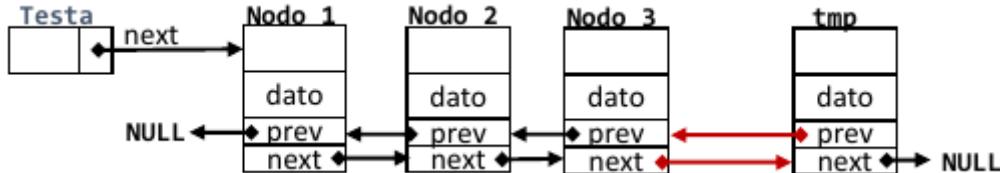
```

Nodo_DL *q = Testa.next;
while(q != NULL && q -> next != NULL){
    q = q -> next;
}

```



```
q -> next = tmp;
tmp -> prev = q;
```



```
void InserisciInCoda_DL(Lista_DL *Testa, int dato){
    Nodo_DL *tmp = NULL;
    Nodo_DL *q = Testa -> next;
    if(!Testa -> next){
        tmp = CreaNodo_DL(dato);
        if(tmp){
            Testa -> next = tmp;
            return;
        }
    } else{
        tmp = CreaNodo_DL(dato);
        while(q -> next){
            q = q -> next;
        }
        q -> next = tmp;
        tmp -> prev = q;
    }
    return;
}
```

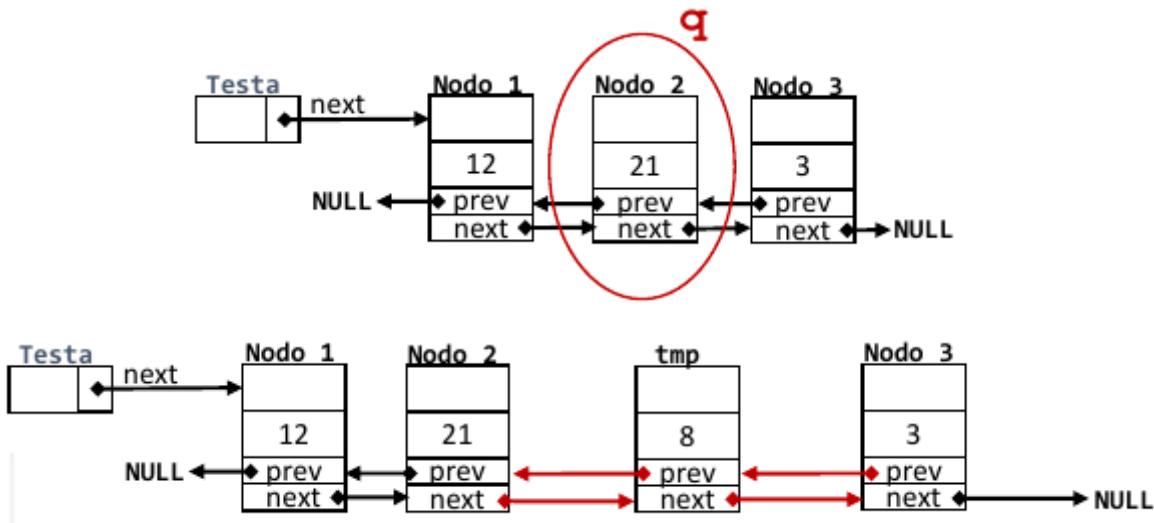
## Ricerca di un Dato Specifico

```
Nodo_DL *CercaElemento_DL(Lista_DL Testa, int dato){
    Nodo_DL *q = Testa.next;
    while(q != NULL && (q -> dato != dato)){
        q = q -> next;
    }
    return q;
}
```

## Inserimento Dopo un Elemento

Se la lista è vuota, l'elemento non esiste e il nuovo nodo non viene inserito. In alternativa, se l'elemento viene trovato, si crea un nuovo nodo e si inserisce dopo tale elemento, aggiornando i puntatori.

```
InserisciDopoElemento_DL(Testa, 21, 8);
q = CercaElemento_DL(Testa, 21);
tmp -> prev = q;
q -> next -> prev = tmp;
tmp -> next = q -> next;
q -> next = tmp;
```



L'inserimento di un valore dopo un elemento specifico è effettuata **cercando il valore e inserendo l'elemento come nodo successivo**.

```
void InserisciDopoElemento_DL(Lista_DL *Testa, int predecessore, int dato){
    Nodo_DL *tmp = NULL;
    Nodo_DL *nuovo = NULL;
    if(Testa -> next == NULL){
        return;
    } else{
        tmp = CercaElemento_DL(*Testa, predecessore);
        if(tmp){
            nuovo = CreaNodo_DL(dato);
            nuovo -> prev = tmp;
            tmp -> next -> prev = nuovo;
            tmp -> next = nuovo;
            nuovo -> next = tmp -> next;
        }
    }
    return;
}
```

## Eliminazione di un Nodo

L'eliminazione di un nodo dalla lista prevede:

- Ricerca del nodo da eliminare (se necessaria)
- Salvataggio del nodo in una variabile ausiliaria
- Scollegamento del nodo dalla lista (aggiornamento dei puntatori della lista)
- Distruzione del nodo (deallocazione della memoria)

In ogni caso, bisogna verificare inizialmente che la lista non sia già vuota:

- `if(Testa.next != NULL)`

Come per l'inserimento, il caso più semplice è costituito dall'eliminazione del nodo di testa, in quanto esiste il puntatore `Testa.next` a questo elemento.

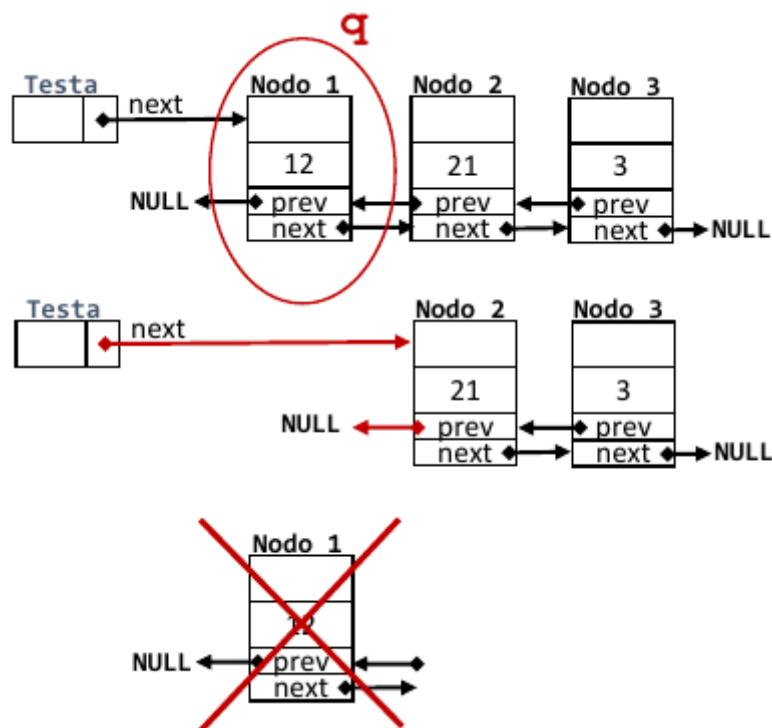
Negli altri casi si procede come per l'inserimento.

## Eliminazione del Nodo di Testa

Bisogna aggiornare il puntatore alla testa `Testa.next`, che dovrà puntare al nodo successivo a quello da eliminare.

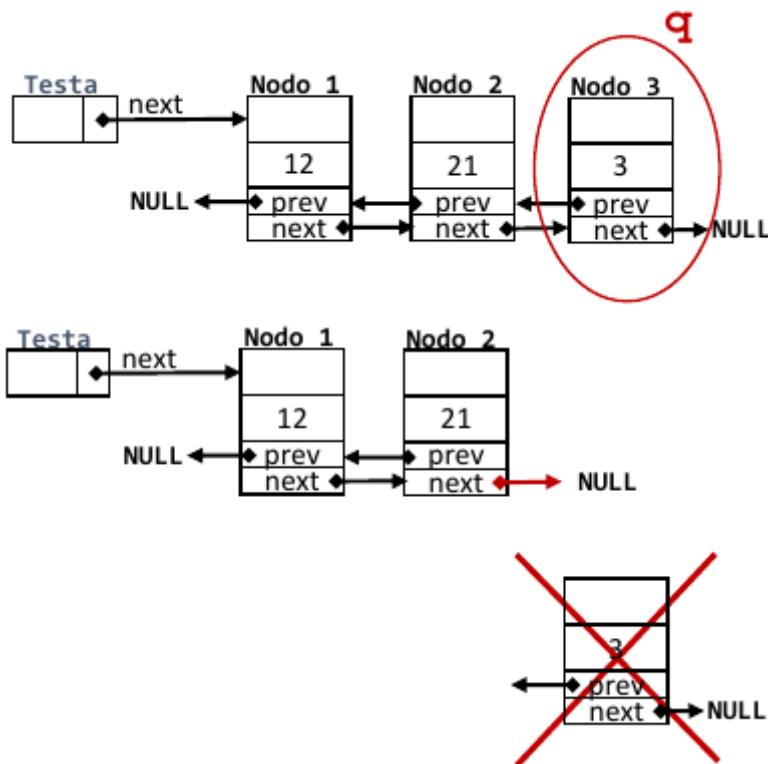
```
// Salvataggio del nodo da eliminare
Nodo_DL *tmp = Testa.next;
// Aggiornamento della lista
Testa.next = tmp -> next;
tmp -> next -> prev = NULL;
// Distruzione del nodo
free(tmp);
```

```
void CancelllaInTesta_DL(Lista_DL *Testa){
    Nodo_DL *q = Testa -> next;
    if(!q){
        return;
    } else{
        if(q -> next){
            Testa -> next = q -> next;
            Testa -> next -> prev = NULL;
        } else{
            Testa -> next = NULL;
            free(q);
        }
    }
}
```



## Eliminazione del Nodo di Coda

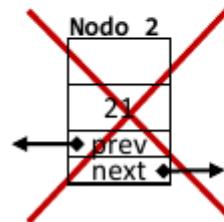
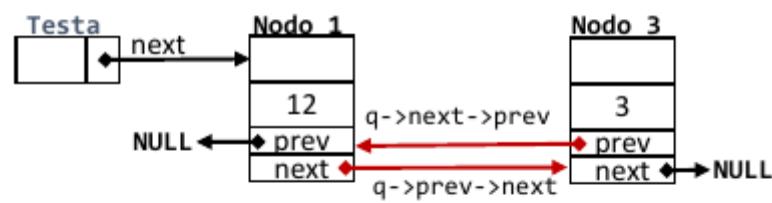
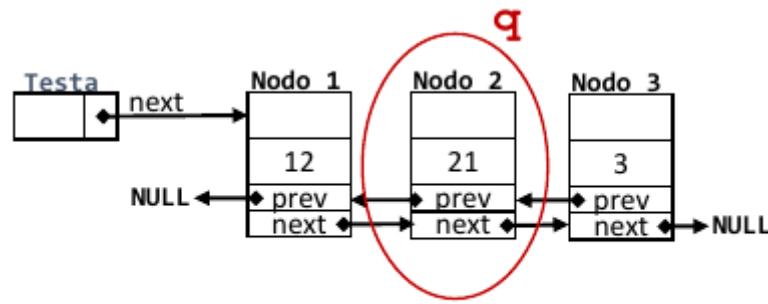
```
void CancellaInCoda_DL(Lista_DL *Testa){  
    Nodo_DL *q = Testa -> next;  
    if(!q){  
        return;  
    } else{  
        while(q != NULL && q -> next != NULL){  
            q = q -> next;  
        }  
        if(q -> prev){  
            q -> prev -> next = NULL;  
        }  
        free(q);  
    }  
}
```



## Eliminazione di un Nodo Generico

```
void CancellaElemento_DL(Lista_DL *Testa, int dato){  
    Nodo_DL *tmp = NULL;  
    if(Testa -> next == NULL){  
        return;  
    } else{  
        tmp = CercaElemento_DL(*Testa, dato);  
        if(tmp){  
            if(tmp -> next == NULL){  
                CancellaInCoda_DL(Testa);  
            } else if(tmp -> prev == NULL){  
                CancellaInTesta_DL(Testa);  
            } else{  
                tmp -> prev -> next = tmp -> next;  
                tmp -> next -> prev = tmp -> prev;  
                free(tmp);  
            }  
        }  
    }  
}
```

```
        }
    }
}
return;
}
```



# Linguaggio C: Code e Stack

## La Struttura Dati "Coda"

Una **coda** (o *queue*) è una **struttura dati astratta** le cui modalità di accesso sono di tipo **FIFO**

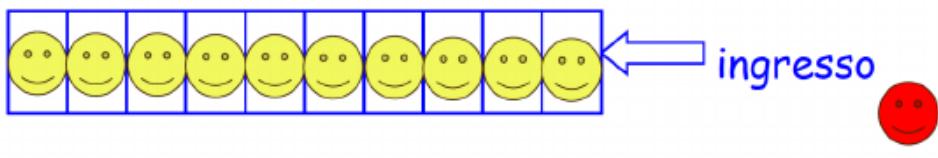
- **FIFO (First In First Out)**: i dati sono estratti rispetto all'ordine di inserimento

La politica FIFO è comunemente adottata in numerosi contesti differenti.

- Nella vita reale è utilizzata in (quasi) tutti i contesti in cui è necessario attendere per poter ottenere un servizio (coda al supermercato, allo sportello bancario ecc.)

In informatica è utilizzata per la gestione dei processi su un sistema operativo, per la gestione del flusso di dati tra periferiche ecc.

**Dequeue:** operazione  
che corrisponde alla  
estrazione di un elemento  
dalla coda



**Enqueue:** operazione  
che corrisponde  
all'inserimento di un  
elemento dalla coda



Esistono **due approcci** per implementare una struttura dati di tipo coda in C:

- **code con capacità illimitata**: struttura dati di tipo **lista concatenata**
- **code con capacità limitata**: implementazione ad-hoc tramite **array**

Una struttura dati di tipo coda può essere vista come una struttura dati di tipo lista che supporta un numero limitato di operazioni:

- **Enqueue**: inserimento in coda (o, equivalentemente, in testa)
- **Dequeue**: rimozione in testa (o in coda)

Anche in questo caso ci concentriamo unicamente su implementazioni di code che ci permettano di memorizzare interi (tipo di dato `int`).

- Vogliamo che il nostro codice sia **efficiente** in termini di utilizzo di memoria e tempo di calcolo
- Vogliamo che il nostro codice sia **modulare**: le funzioni devono essere semplici, specifiche e riutilizzabili

## Definizione di Tipo

Il nodo del tipo di dato `coda` è equivalente al nodo del tipo di dato `lista`.

Modifichiamo la struttura dati di accesso in modo da conservare sia un puntatore alla testa che un puntatore al nodo terminale.

- **Struttura del nodo:**

```

typedef struct Nodo_SL{
    int dato;
    struct Nodo_SL *next;
} Nodo_SL;

```

- Struttura della coda:

```

typedef struct Coda{
    Nodo_SL *head;
    Nodo_SL *tail;
} Coda;

```

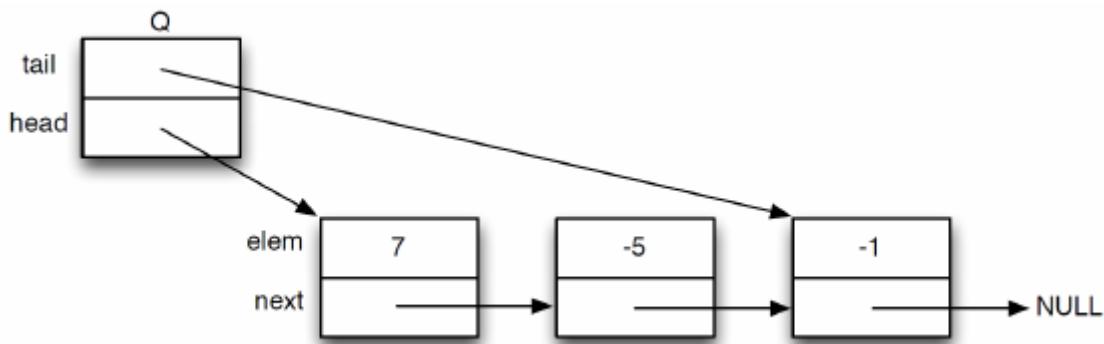
## Funzioni per la Gestione

Consideriamo i seguenti **prototipi** che definiscono le **operazioni richieste su una coda**:

- `int enqueue(Coda *Q, int dato);`
- `int dequeue(Coda *Q, int *dato);`
- `int LeggiInTesta(Coda *Q, int *dato);`

L'operazione di rimozione in testa è efficiente per tutte le rappresentazioni concatenate viste.

L'operazione di inserimento in coda è resa efficiente grazie al puntatore alla coda inserito nella struttura `Coda`



## Creazione della Coda

Per creare una coda basta definirla, ovvero **è sufficiente creare il modo di riferirsi ad essa**.

L'unica cosa che esiste sempre della coda è il suo **punto di accesso** (o *radice*).

- Questa è l'**unica componente allocata staticamente**

All'inizio i puntatori `head` e `tail` sono inizializzati a `NULL`, in quanto non ci sono elementi

- **Esempio:**

```

Coda Q;
Q.head = NULL;
Q.tail = NULL;

int CodaVuota(Coda *Q){
    return (Q -> head == NULL);
}

void CreaCoda(Coda *Q){

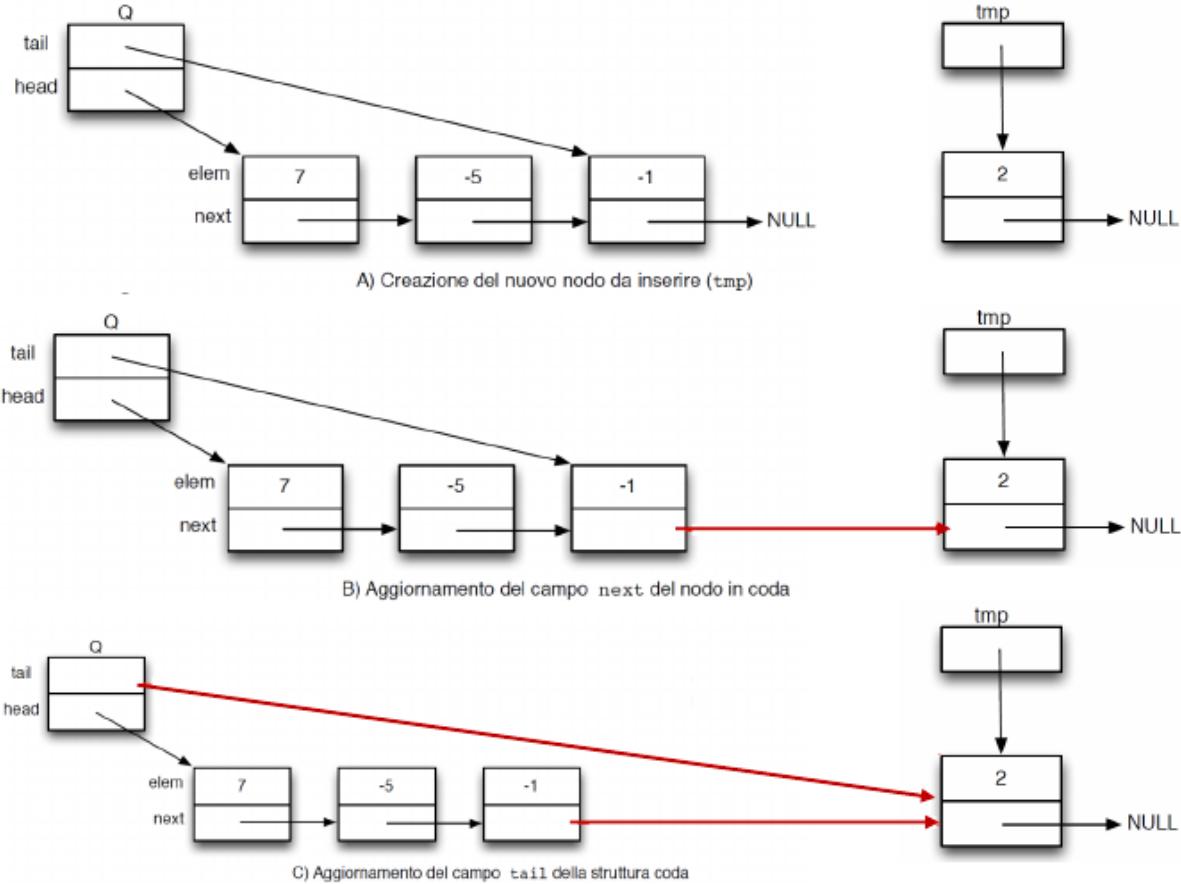
```

```

if(Q){
    Q -> head = NULL;
    Q -> tail = NULL;
}

```

## Inserimento di un Elemento nella Coda



L'operazione di inserimento di un elemento nella coda viene gestita come un inserimento in coda ad una lista:

- il nodo viene creato da `creaNodo_SL`, ovvero la **stessa funzione definita per le liste**

È necessario gestire in modo specifico **due casi distinti**:

- La coda inizialmente è vuota**: sia il puntatore alla testa che il puntatore alla coda puntano al nuovo nodo allocato
- La coda contiene almeno un elemento**: l'operazione di inserimento è gestita come inserimento in coda in una lista concatenata. Non è necessario modificare il puntatore alla coda.

```

void enqueue(Coda *Q, int dato){
    Nodo_SL *tmp;
    if(Q == NULL){
        return;
    } else if(codaVuota(Q)){
        Q -> head = CreaNodo_SL(dato);
        Q -> tail = Q -> head;
        return;
    } else{
        tmp = CreaNodo_SL(dato);
    }
}

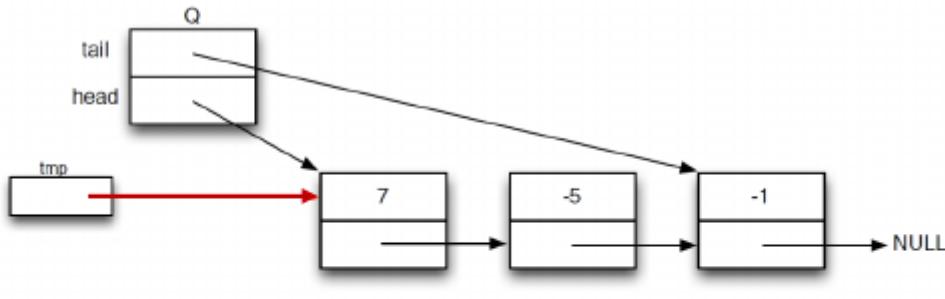
```

```

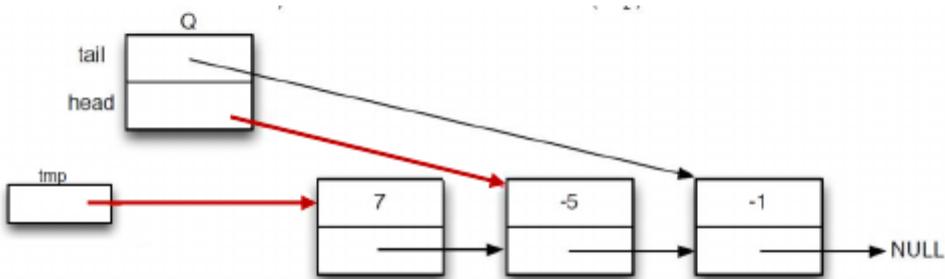
    if(tmp != NULL){
        Q -> tail -> next = tmp;
        Q -> tail = tmp;
    }
    return;
}
}

```

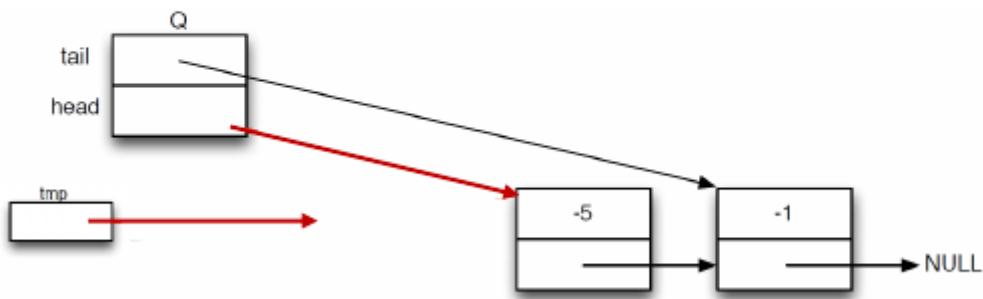
## Eliminazione di un Elemento dalla Coda



A) Puntatore al nodo da rimuovere (`tmp`)



B) Aggiornamento puntatore alla testa della coda (head)



C) Deallocazione del nodo da rimuovere

L'operazione di rimozione di un elemento dalla coda viene gestito come rimozione dalla testa di una lista.

È necessario gestire in modo specifico **tre casi distinti**:

- **coda vuota:** la funzione termina
- **la coda contiene un solo elemento:** è necessario aggiornare sia il puntatore alla testa che alla coda della lista
- **la coda contiene almeno due elementi:** l'operazione è gestita come rimozione dalla testa in una lista concatenata. Non è necessario modificare il puntatore alla coda della lista

```

void dequeue(Coda *Q, int *dato){
    Nodo_SL *tmp;
    if(Codavuota(Q) || dato == NULL){
        return;
    } else if(Q -> head == Q -> tail){
        *dato = Q -> head -> dato;
    }
}

```

```

        free(Q -> head);
        Q -> head = Q -> tail = NULL;
        return;
    } else{
        tmp = Q -> head;
        *dato = Q -> head -> dato;
        Q -> head = Q -> head -> next;
        free(tmp);
        return;
    }
}

```

## Funzioni Aggiuntive sulle Code

La seguente funzione permette di svuotare completamente una coda e di verificare se una coda è vuota.

L'implementazione della funzione di distruzione di una coda **rimuove iterativamente l'elemento in testa**.

```

void LiberaCoda(Coda *Q){
    Nodo_SL *tmp;
    if(Q != NULL){
        while(Q -> head != NULL){
            tmp = Q -> head;
            Q -> head = Q -> head -> next;
            free(tmp);
        }
        Q -> head = Q -> tail = NULL;
    }
}

```

L'operazione di accesso al primo elemento in coda è gestita come l'operazione di selezione dell'elemento in testa ad una lista.

```

void LeggiInTesta(Coda *Q, int *dato){
    if(codavuota(Q) || dato == NULL){
        return;
    } else{
        *dato = Q -> head -> dato;
        return;
    }
}

```

## La Struttura Dati "Stack"

Uno **stack** (o *pila*) è una **struttura dati astratta** le cui modalità di accesso sono di tipo **LIFO**.

- **LIFO (Last In First Out)**: i dati sono estratti in ordine inverso rispetto al loro inserimento.

Una struttura dati pila supporta essenzialmente **due sole operazioni**:

- **Push**: inserisce un oggetto in cima alla pila
- **Pop**: rimuove un oggetto in cima alla pila e ne ritorna il valore

Nonostante la sua semplicità, lo stack è utilizzato in numerosi contesti informatici.

- Ad esempio, quasi tutti i linguaggi di alto livello utilizzano uno stack di record di attivazione per gestire la chiamata a funzione

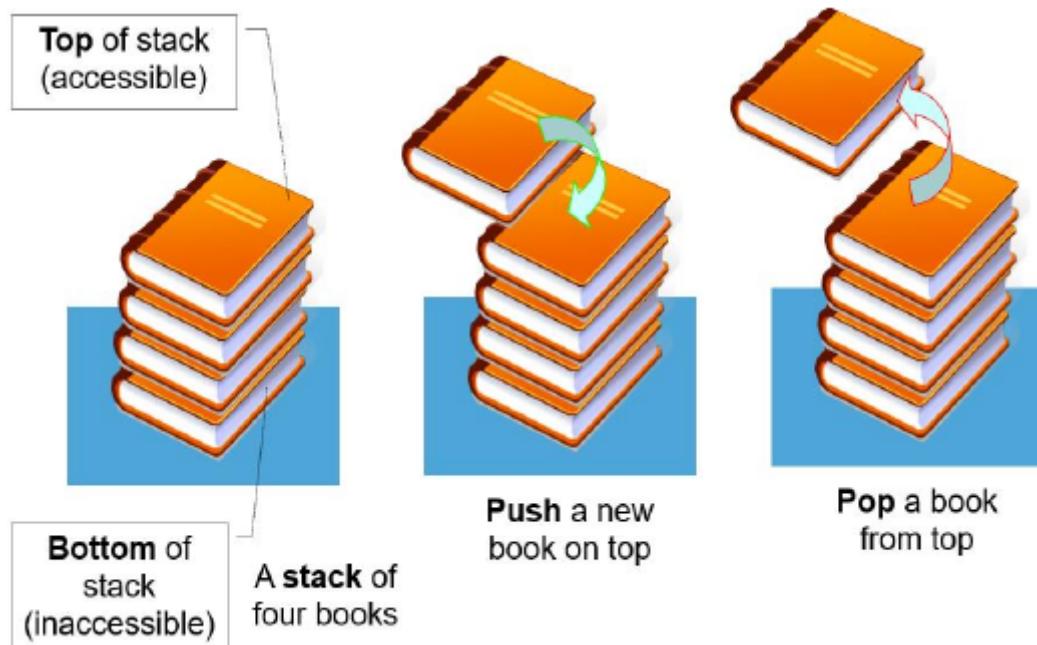
Una struttura dati di tipo stack può essere vista come una struttura dati di tipo lista che **supporta un numero limitato di operazioni**.

Le operazioni di Push e Pop sono equivalenti a quelle di inserimento e rimozione in testa (o in coda) della lista.

Esistono diversi approcci per implementare una struttura dati di tipo pila in C:

- **capacità illimitata**: implementa una struttura dati di tipo lista
- **capacità limitata**: implementata con un array

Anche in questo caso ci concentriamo unicamente su implementazioni di pile che permettano di memorizzare interi



## Definizione di Tipo

Uno stack con capienza illimitata **può essere implementato tramite una lista concatenata**.

Utilizziamo la libreria che implementa le liste concatenate per implementare una struttura dati di tipo stack.

Le operazioni di inserimento e rimozione in testa sono efficienti per tutte le rappresentazioni concatenate viste.

Le operazioni in cima allo stack possono essere implementate efficientemente come operazioni in testa ad una lista.

**Il tipo di dato `stack` è equivalente al tipo di dato `lista`**.

È necessario includere il file header in cui è definito il tipo di dato lista per poter definire il tipo di dato stack.

```
#include "liste.h"
typedef Lista_SL Stack;
```

## Funzioni di Gestione

```
// Creazione di un oggetto di tipo stack inizialmente vuoto
void Creastack(Stack *S){
    if(S){
        S -> next = NULL;
    }
}

// Stampa di tutti gli elementi nello stack
void StampaStack(Stack S){
    Nodo_SL *tmp = S.next;
    printf("\n[Stack] -> ");
    while(tmp && tmp -> next){
        printf("%d -> ", tmp -> dato);
        tmp = tmp -> next;
    }
    if(tmp){
        printf("%d -| ", tmp -> dato);
    }
}

// Inserire un elemento nello stack
void Push(Stack *S, int *dato){
    if(S){
        InserisciInTesta_SL(S, dato);
    }
}

// Estrarre un elemento dallo stack
void Pop(Stack *S, int *dato){
    if(S && S -> next){
        *dato = S -> next -> dato;
        CancellatInTesta_SL(S);
    }
}

// Leggere un elemento in cima allo stack\
void Top(Stack *S, int *dato){
    if(S && S -> next){
        *dato = S -> next -> dato;
    }
}

// Svuotare l'intero contenuto di un oggetto di tipo stack
void LiberaStack(Stack *S){
    if(S){
        LiberaLista_SL(S);
    }
}

// Testare se lo stack è vuoto
int StackVuoto(Stack *S){
    return (!S || S -> next == NULL);
}
```

## Stack con Array

```
typedef struct AStack{
    unsigned int size;
    unsigned int last;
    int *dato;
} AStack;

void CreaStack(AStack *S, unsigned int n){
    S -> size = 0;
    S -> last = 0;
    S -> dato = (int*)malloc(n*sizeof(int));
    if(S -> dato != NULL){
        S -> size = n;
    }
    return;
}

void LiberaStack(AStack *S){
    if(S != NULL){
        free(S -> dato);
        S -> dato = NULL;
        S -> last = 0;
        S -> seize = 0;
    }
}

int AStackVuoto(AStack *S){
    return (S == NULL || S -> last == 0)
}

int AStackPieno(AStack *S){
    return (S -> last == S -> size);
}

int APush(AStack *S, int dato){
    if(AStackPieno(S)){
        return 1;
    } else{
        S -> dato[S -> last++] = dato;
        return 0;
    }
}

int APop(AStack *S, int *dato){
    if(AStackVuoto(S) || dato == NULL){
        return 1;
    } else{
        *dato = S -> dato[S -> last-- - 1];
        return 0;
    }
}

int ATop(AStack *S, int *dato){
    if(AStackVuoto(S) || dato == NULL){
        return 1;
    } else{
        *dato = S -> dato[S -> last - 1];
    }
}
```

```
    return 0;
}

void StampaAStack(Astack s){
    int i = 0;
    printf("\n[Stack] -> ");
    if(AstackVuoto(&s) == 0){
        for(i = s.last - 1;i > 0; i--){
            printf("%d -> ", s.dato[i]);
        }
        printf("%d -|", s.dato[0]);
    }
}
```

# [Lezione 27] Linguaggio C: la Ricorsione

## Calcolo del Fattoriale Iterativo

```
int fact(int n){  
    int i;  
    // Inizializzazione del fattoriale  
    int F = 1;  
    for(i = 2; i <= n; i++){  
        F = F * i;  
    }  
    return F;  
}
```

La variabile `F` accumula risultati intermedi:

- se `n = 3` allora, inizialmente, `F = 1`
- al primo ciclo sarà `n = 2` e quindi `F` assume il valore `2`
- all'ultimo ciclo `i = 3` e `F = 6`

## Processo Computazionale Iterativo

Nell'esempio precedente il risultato viene sintetizzato "in avanti".

L'esecuzione di un algoritmo di calcolo che compiti "in avanti", per accumulo, è un **processo computazionale iterativo**.

La **caratteristica fondamentale** di un processo computazionale iterativo è che, ad ogni passo, è disponibile un risultato parziale:

- dopo  $k$  passi, si ha a disposizione il risultato parziale relativo al caso  $k$

Questo **non è vero nei processi computazionali ricorsivi**, in cui nulla è disponibile finché non si è giunti fino al caso elementare.

## La Ricorsione

Una **funzione matematica** è **definita ricorsivamente** quando, nella sua definizione, compare un riferimento a sè stessa.

La ricorsione consiste nella possibilità di definire una funzione mediante sè stessa ed è basata sul **principio di induzione matematica** (se una proprietà vale per  $n$  ed  $n + 1$  allora vale **per ogni  $n$** ).

Operativamente, risolvere un problema con approccio ricorsivo comporta:

- identificare un **caso base** ( $n = n0$ ) in cui la soluzione sia nota
- riuscire ad esprimere la soluzione nel caso generico  $n$  in termini dello stesso problema in uno o più casi semplici ( $n - 1, n - 2$  ecc)

**Esempio:** fattoriale di un numero naturale

```

fact(n) = n!

n! : N -> N
n! vale 1 se n = 0
n! vale n * (n - 1)! se n > 0

```

## La Ricorsione in C

In C è possibile definire funzioni ricorsive:

- il corpo di ogni funzione ricorsiva **contiene almeno una chiamata alla funzione stessa**

**Esempio:** *definizione in C della funzione ricorsiva fattoriale*

```

int fact(int n){
    if(n <= 0){
        return 1;
    } else{
        return n * fact(n - 1);
    }
}

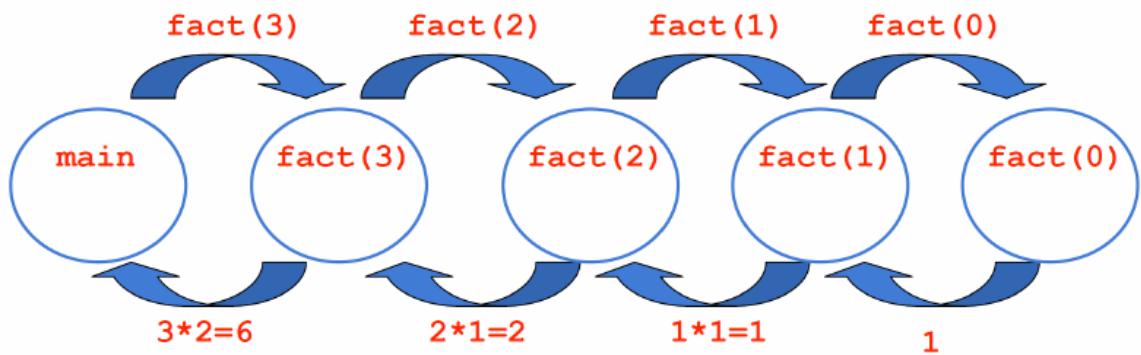
int main(){
    int fz, f6, z = 5;
    fz = fact(z - 2);
}

```

La funzione `fact` è **sia servitore che cliente** (di sè stessa).

**Algoritmo:**

- La funzione `fact` lega il parametro `n` a `z - 2 = 3`. Essendo `3` positivo, si passa al ramo `else`.  
Per calcolare il risultato della funzione, però, è necessario effettuare una nuova chiamata di funzione `fact(2)`.
- Per calcolare il risultato della funzione è necessario effettuare una nuova chiamata di funzione. `n - 1` nell'enviroment di `fact` vale `2`, quindi viene chiamata `fact(1)`.
- Il nuovo servitore lega il parametro `n` a `1`. Essendo `1` positivo si passa al ramo `else`.  
Per calcolare il risultato della funzione è necessario effettuare una nuova chiamata di funzione. `n - 1` nell'enviroment di `fact` ora vale `0`, quindi viene chiamata `fact(0)`.
- Il nuovo servitore lega il parametro `n` a `0`. La condizione `n <= 0` adesso è vera e la funzione `fact(0)` restituisce `1` come risultato e termina.
- Il controllo torna al servitore precedente, ovvero `fact(1)`, che può valutare l'espressione `n * 1` (valutando `n` nel suo enviroment dove vale `1`) ottenendo come risultato `1` e terminando.
- Il controllo torna al servitore precedente, `fact(2)`, che può valutare l'espressione `n * 1` (valutando `n` nel suo enviroment dove vale `2`) ottenendo come risultato `2` e terminando.
- Il controllo torna al servitore precedente, `fact(3)`, che può valutare l'espressione `n * 2` (valutando `n` nel suo enviroment dove vale `3`) ottenendo come risultato `6` e terminando.



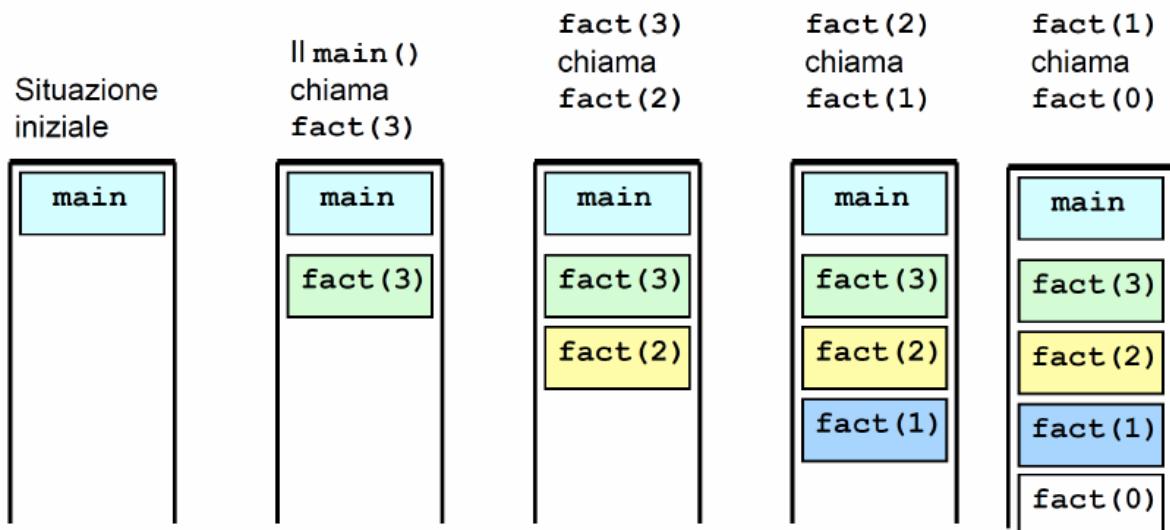
main	fact(3)	fact(2)	fact(1)	fact(0)
Cliente di fact(3)	Cliente di fact(2)	Cliente di fact(1)	Cliente di fact(0)	Servitore di fact(1)

Servitore  
del main

Di Servitore  
di fact(3)

Di Servitore  
di fact(2)

## Cosa Succede nello Stack

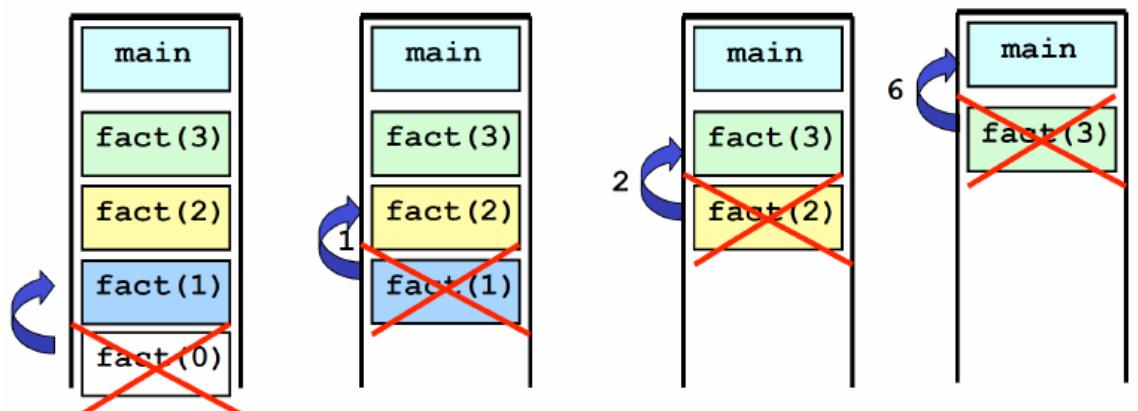


fact(0) termina restituendo il valore 1. Il controllo torna a fact(1)

fact(1) effettua la moltiplicazione e termina restituendo il valore 1. Il controllo torna a fact(2)

fact(2) effettua la moltiplicazione e termina restituendo il valore 2. Il controllo torna a fact(3)

fact(3) effettua la moltiplicazione e termina restituendo il valore 6. Il controllo torna al main.



## Somma dei Primi N Numeri Naturali

- **Problema:** calcolare la somma dei primi N numeri naturali
- **Specifica:** considera la somma  $1 + 2 + 3 \dots + (N - 1) + N$  come composta di due termini:
  - il **primo termine** non è altro che lo stesso problema in un caso più semplice: **calcolare la somma dei primi N - 1 numeri interi**
  - il **secondo termine** è il valore noto **N**
- Esiste un **caso banale**, il **caso base**, ovvero che la somma fino ad 1 vale, banalmente, 1

**Algoritmo ricorsivo:** Somma  $N \rightarrow N$

- Somma( $n$ ) vale 1 se  $n = 1$
- Somma( $n$ ) vale  $n + \text{Somma}(n - 1)$  se  $n > 0$

```
#include <stdio.h>
int SommaFinoA(int n);

int main(){
    int dato;
    printf("\nInserisci un intero positivo: ");
    scanf("%d", &dato);
    if(dato > 0){
        printf("\nRisultato: %d", SommaFinoA(dato));
    } else{
        printf("ERRORE!");
    }
}

int SommaFinoA(int n){
    if(n == 1){
        return 1;
    } else{
        return SommaFinoA((n - 1) + n);
    }
}
```

**Esempio di esecuzione** con  $N = 4$ :

```
// In avanti
N = [4 + SommaFinoA(3)]
N = [4 + [3 + SommaFinoA(2)]]
N = [4 + [3 + [2 + SommaFinoA(1)]]]
// All'indietro
N = [4 + [3 + [2 +      1      ]]]
N = [4 + [3 +          3      ]]
N = [4 +              6      ]
N =               10
```

## Funzioni: Modello Run-Time

Ogni volta che viene invocata una funzione:

- si crea una **nuova istanza** del servitore
- **viene allocata memoria** per i parametri e le variabili locali
- si effettua il **passaggio dei parametri**

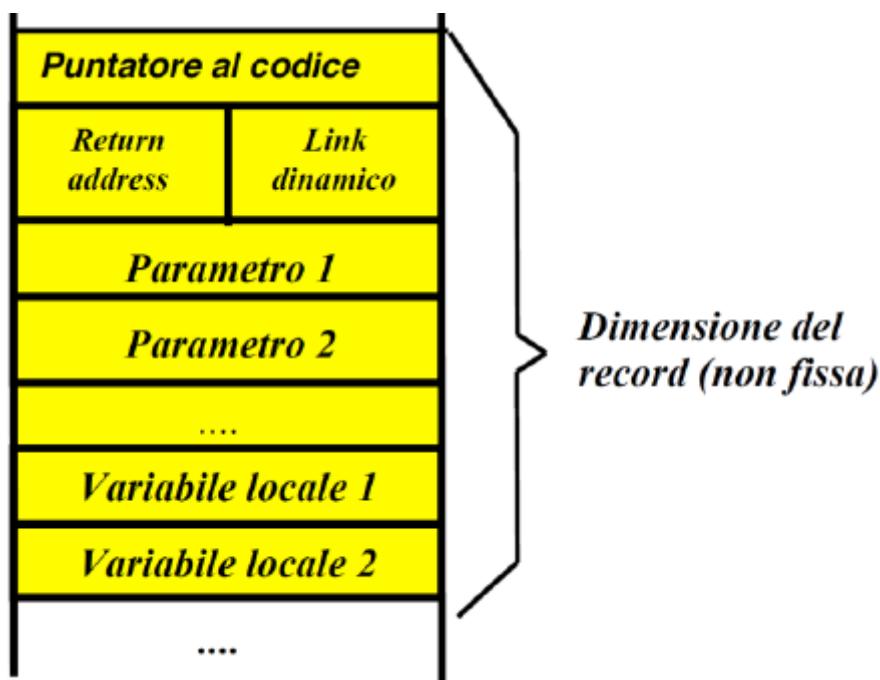
- **si trasferisce il controllo** al servitore
- **si esegue il codice** della funzione

Al momento dell'invocazione viene creata dinamicamente una **struttura dati** che **contiene i binding dei parametri e degli identificatori** definiti localmente alla funzione detta **record di attivazione**.

## Record di Attivazione

Contiene tutto ciò che serve per la chiamata della funzione alla quale è associato:

- **parametri formali**
- **variabili locali**
- **indirizzo di ritorno (Return Address RA)** che indica il punto a cui tornare (nel codice del cliente) al termine della funzione, per permettere al cliente di proseguire una volta che la funzione termina
- un **collegamento al record di attivazione** del cliente (Link Dinamico DL)
- l'**indirizzo del codice della funzione** (puntatore alla prima istruzione del corpo)



Il record di attivazione associato ad una chiamata di una funzione *f*:

- è **creato al momento** dell'invocazione di *f*
- **permane per tutto il tempo** in cui la funzione *f* è in esecuzione e **viene deallocated al termine** dell'esecuzione **della funzione** stessa

Ad ogni chiamata di funzione viene creato un nuovo record, specifico per quella chiamata di quella funzione.

La **dimensione** del record di attivazione:

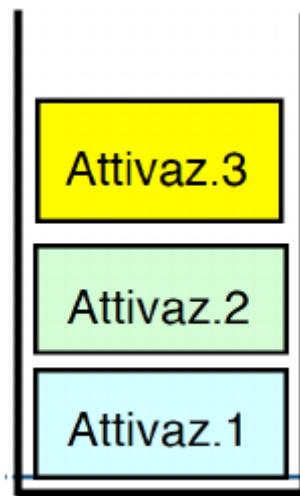
- varia da una funzione all'altra
- per una data funzione, è fissa e calcolabile a priori

Funzioni che chiamano altre funzioni danno luogo ad una **sequenza di record di attivazione**:

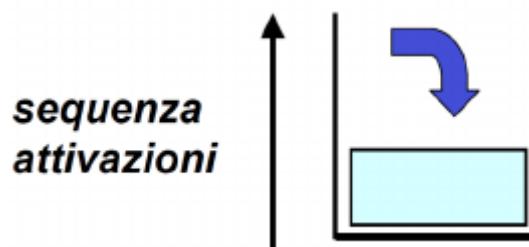
- allocati secondo l'ordine delle chiamate
- deallocated in ordine inverso

La sequenza dei link dinamici costituisce la cosiddetta **catena dinamica**, che **rappresenta la storia delle attivazioni** ("chi ha chiamato chi").

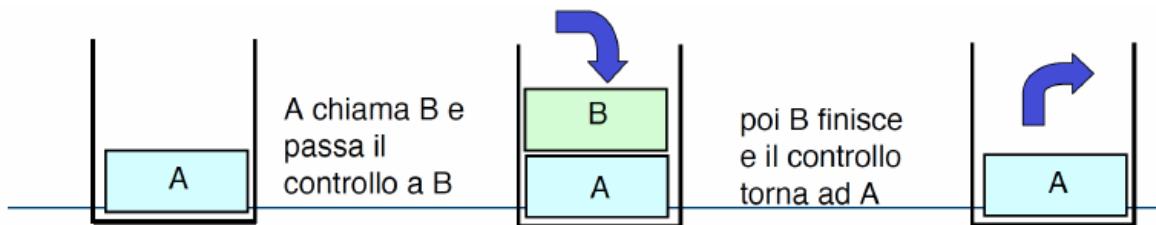
L'area di memoria in cui vengono allocati i record di attivazione viene **gestita come uno stack**.



Normalmente lo stack dei record di attivazione si disegna nel modo seguente:



Quindi, se la funzione A chiama la funzione B, lo stack evolve nel modo seguente:

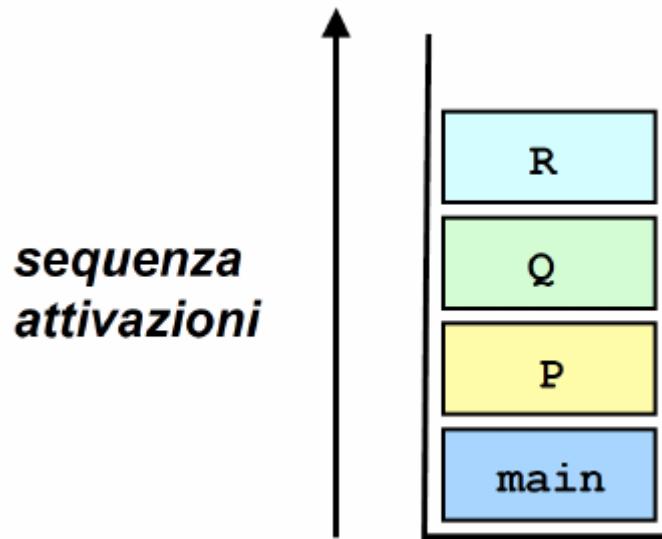


**Programma:**

```
int R(int A){  
    return A + 1;  
}  
  
int Q (int x){  
    return R(x);  
}  
int P(void){  
    int a = 10;  
    return Q(a);  
}  
  
int main(){  
    int x = P();  
}
```

- **Sequenza chiamate:**

- o S.O. -> main -> P() -> Q() -> R()



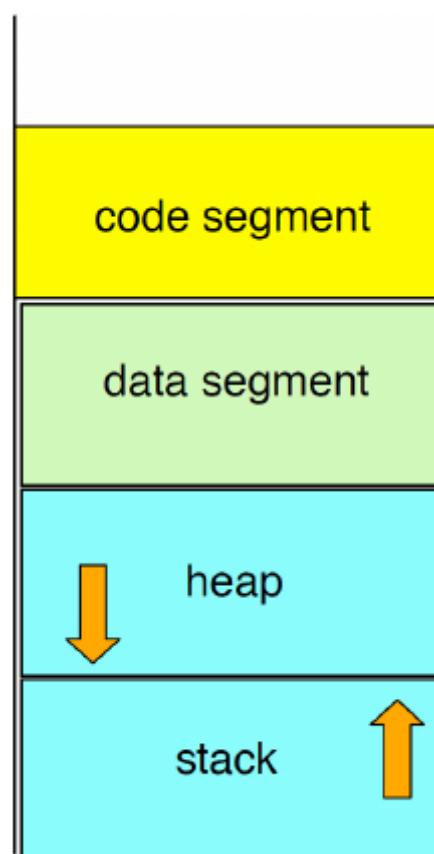
## Spazio di Indirizzamento

La memoria allocata ad ogni programma in esecuzione è suddivisa in varie parti (segmenti), secondo lo schema seguente:

- **code segment**: contiene il **codice eseguibile** del programma
- **data segment**: contiene le **variabili globali**
- **heap**: contiene le **variabili dinamiche**
- **stack**: è l'area dove vengono allocati i **record di attivazione**

Code segment e data segment sono di **dimensione fissata staticamente** (a tempo di compilazione).

La dimensione dell'area associata a stack + heap è **fissata staticamente**: man mano che lo stack cresce, diminuisce l'area a disposizione dell'heap e viceversa



È possibile imporre che una variabile locale ad una funzione abbia un tempo di vita pari al tempo di esecuzione dell'intero programma, utilizzando il qualificatore `static`:

```
void f(){
    static int cont = 0;
    ...
}
```

La variabile `static int cont`:

- è creata all'inizio del programma, inizializzata a `0` e deallocata alla fine dell'esecuzione
- la sua visibilità è limitata al corpo della funzione `f`
- il suo tempo di vita è pari al tempo di esecuzione dell'intero programma
- è **allocata nel data segment** (area dati globale)

**Esempio:**

```
#include <stdio.h>
int f(){
    static int cont = 0;
    cont++;
    return cont;
}

int main(){
    printf("%d\n", f());
    printf("%d\n", f());
}
```

La variabile `static int cont` è allocata all'inizio del programma e deallocata alla fine dell'esecuzione. Essa persiste tra un'attivazione e l'altra di `f`: la prima `printf` stampa 1, la seconda `printf` stampa 2 e così via.

## [Lezione 28] Ricorsione e Complessità

### Sommario

- Esempi di Ricorsione
- Ricorsione e Funzioni Iterative con Stack
- Cenni sulla Complessità

### Divide et Impera

Metodo di approccio ai problemi che **consiste nel dividere il problema dato in problemi più semplici**

- I risultati ottenuti risolvendo i problemi più semplici vengono combinati insieme per costruire la soluzione del problema originale
- Generalmente, quando la semplificazione del problema consiste essenzialmente nella semplificazione dei dati da elaborare (ad es. la riduzione della dimensione del vettore da elaborare), si può pensare ad una soluzione ricorsiva

### La Ricorsione

Una funzione è detta **ricorsiva se chiama sè stessa**.

Se due funzioni si chiamano l'un l'altra allora sono dette **mutualmente ricorsive**.

La funzione ricorsiva sa risolvere direttamente solo casi particolare di un problema detti **casi di base**:

- se viene invocata passandole dei dati che costituiscono uno dei casi di base, allora ne restituisce il risultato
- se, invece, viene chiamata passandole dei dati che non costituiscono uno dei casi di base, allora chiama sè stessa (passo ricorsivo) passando dei dati semplificati/ridotti

## Analisi

L'apertura delle chiamate ricorsive semplifica il problema ma non calcola ancora nulla.

Il valore restituito dalle funzioni viene utilizzato per calcolare il valore finale man mano che si chiudono le chiamate ricorsive: **ogni chiamata genera valori intermedi a partire dalla fine**.

Nella ricorsione vera e propria non c'è un mero passaggio di un risultato calcolato nella chiamata più interna a quelle più esterne, ossia le return non si limitano a passare indietro invariato un valore, ma c'è un'elaborazione intermedia

## Pro e Contro

**Pro:**

- spesso la ricorsione **permette di risolvere un problema anche molto complesso con poche linee di codice**

**Contro:**

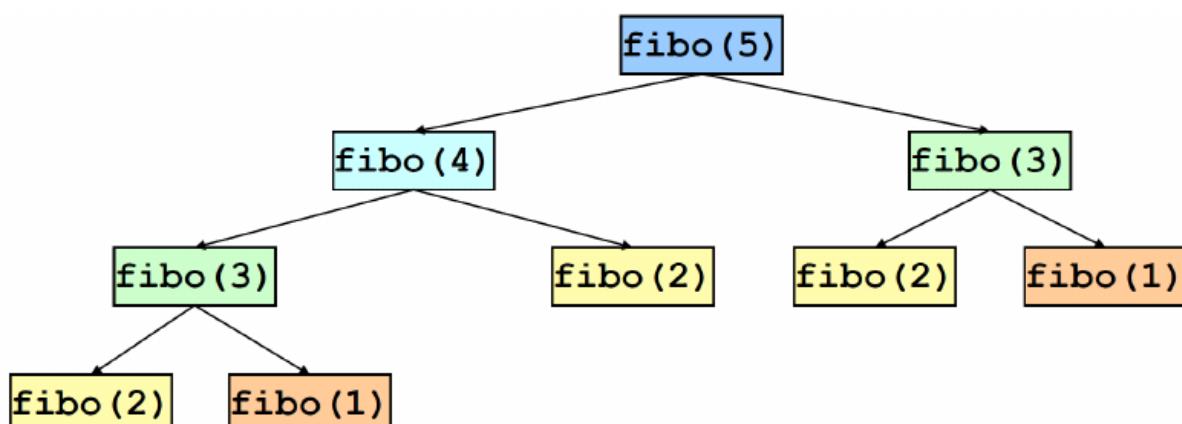
- La **ricorsione è poco efficiente** perchè richiama molte volte una funzione e questo:
  - richiede tempo per la gestione dello stack (allocare e passare i parametri, salvare l'indirizzo di ritorno e i valori di alcuni registri della CPU)
  - consuma molta memoria (alloca un nuovo stack frame ad ogni chiamata, definendo una nuova ulteriore istanza delle variabili locali non static e dei parametri ogni volta)

## Considerazioni

Si osservi che:

- `fibo(n)` chiama `fibo(n - 1)` e `fibo(n - 2)`
- `fibo(n - 1)` chiama `fibo(n - 2)` ecc.

Si hanno dei **calcoli ripetuti**, il che è **inefficiente**!



## Ricorsione con Stack Esplicito

L'uso di uno stack esplicito permette di **simulare le chiamate ricorsive ad una funzione**.

**Ricorsione:**

- Stack dei record di attivazione
- Caso base -> return valore
- Caso generico n -> chiamata ricorsiva su m < n
- Condizione di arresto = caso base

**Iterazione:**

- Stack esplicito S
- Caso base -> accumulo in una variabile F
- Caso generico n -> inserisco m < n nello stack
- Condizione di arresto = stack vuoto

## Fibonacci con Stack Esplicito

```
#include <stdio.h>
#include <stdlib.h>
// Libreria per le funzioni di gestione dello stack creata nelle lezioni
precedenti
#include "stack.h"

long fibo(long n){
    if(n <= 1){
        return 1;
    } else{
        return (fibo(n - 1) + fibo(n - 2));
    }
}

int main(int argc, char* argv[]){
    int n, F = 0;
    AStack S;
    Creastack(&S, 100);
    n = atoi(argv[1]);
    APush(&S, n);
    while(AStackVuoto(&S) == 0){
        APop(&S, &n);
        if(dato <= 1){
            F = F + 1;
        } else{
            APush(&S, n - 1);
            APush(&S, n - 2);
        }
        StampaAStack(S);
        printf("      F = %d", F);
    }
    printf("\nIl valore di Fibonacci per %d è %d\n", atoi(argv[1]), F);
}
```

Nel terminale:

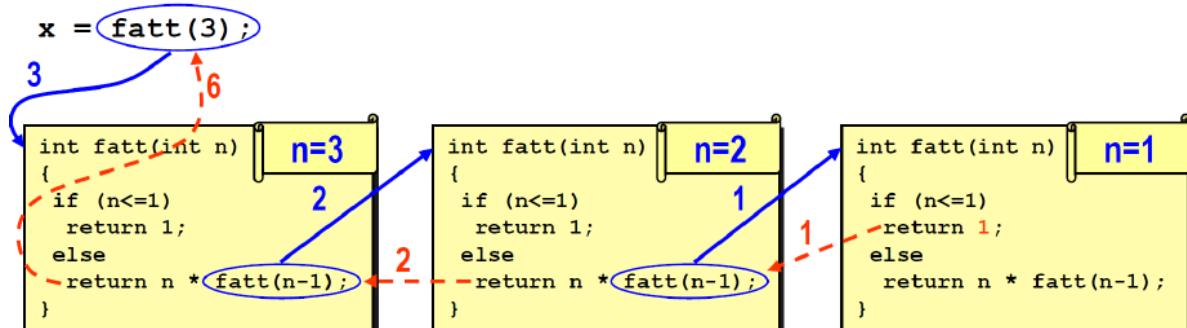
```
[Stack]-> 2 -> 3 -| F=0
[Stack]-> 0 -> 1 -> 3 -| F=0
[Stack]-> 1 -> 3 -| F=1
[Stack]-> 3 -| F=2
[Stack]-> 1 -> 2 -| F=2
[Stack]-> 2 -| F=3
[Stack]-> 0 -> 1 -| F=3
[Stack]-> 1 -| F=4
[Stack]-> F=5
```

Il valore di Fibonacci per 4 è 5

## Fattoriale: Funzione Ricorsiva

```
int fact(int n){
    if(n <= 0){
        return 1;
    } else{
        return (n * fact(n - 1));
    }
}

int main(){
    int x = fact(3);
}
```



## Fattoriale con Stack

```
#include <stdio.h>
#include <stdlib.h>
#include "stack.h"

int fact(int n){
    if(n <= 0){
        return 1;
```

```

        } else{
            return (n * fact(n - 1));
        }
    }

int main(int argc, char* argv[]){
    int n, F = 1;
    AStack S;
    CreaAStack(&S, 100);
    n = atoi(argv[1]);
    APush(&S, n);
    while(AStackVuoto(&S) == 0){
        APop(&S, &n);
        if(n <= 0){
            F = 1 * F;
        } else{
            F = n * F;
            APush(&S, n - 1);
        }
        StampaAStack(S);
        printf("      F = %d", F);
    }
    printf("\nIl fattoriale di %d è %d\n", atoi(argv[1]), F);
}

```

Nel terminale:

[Stack] ->	5 -	F=6
[Stack] ->	4 -	F=30
[Stack] ->	3 -	F=120
[Stack] ->	2 -	F=360
[Stack] ->	1 -	F=720
[Stack] ->	0 -	F=720
[Stack] ->		F=720
Il Fattoriale di 6 è 720		

## Complessità di un Algoritmo

- **$T(n)$  = tempo di elaborazione**
  - numero di operazioni elementari eseguite
- **$S(n)$  = spazio di memoria**
  - numero di celle di memoria utilizzate durante l'esecuzione
- **$n$  = dimensione (taglia) dei dati in ingresso**
  - Es. intero positivo  $x$ :  $n = 1 + \lceil \log(x) \rceil$ , cioè il numero di cifre necessarie per rappresentare  $x$  in notazione binaria
  - Es. vettore di elementi:  $n$  = numero di elementi
  - Es. grafo:  $n, m$  = numero dei vertici, numero archi

# Tempo di Elaborazione T(n)

- **Caso peggiore (spesso)**
  - $T(n) =$  tempo massimo dell'algoritmo su qualsiasi input di dimensione n
- **Caso medio (talvolta)**
  - $T(n) =$  tempo atteso su tutti gli input di dimensione  $n = \text{tempo di ogni input} * \text{la probabilità che ci sia quell'input}$  (*media pesata*).  
È necessaria un'assunzione sulla distribuzione statistica degli input (spesso distribuzione uniforme)
- **Caso migliore (fittizio, probabilità che non si verificherà mai)**
  - Ingannevole per algoritmi lenti che sono veloci su qualche input

## Caso Peggiorre

Generalmente **si cerca un limite superiore** perchè:

- **Fornisce una garanzia all'utente:** l'algoritmo non potrà impiegare più di così
- Per alcuni algoritmi **si verifica molto spesso**
  - Es. ricerca in un DB di un'informazione non presente
- **Il caso medio spesso è cattivo** quasi quanto quello peggiore
- Non sempre è evidente cosa costituisce un input medio
- **Esempio:  $T(n)$  di una Funzione Iterativa**

```
Min(A)
min = A[0]
for i = 1 to A.length
    if A[i] < min
        min = A[i]
return min
```

## Costo Numero di volte

$c_1$	1
$c_2$	n
$c_3$	$n-1$
$c_4$	$n-1$

$$T(n) = c_1 + n*c_2 + (n-1)*c_3 + (n-1)*c_4 = (c_2+c_3+c_4)*n + (c_1-c_3-c_4) = \\ = a*n + b$$

funzione lineare

# Indipendenza dalla Macchina

Qual è il tempo di calcolo di un algoritmo nel caso peggiore?

- Dipende dal computer usato:
  - **velocità relativa**: confronto sulla stessa macchina
  - **velocità assoluta**: confronto su macchine diverse

**IDEA:** Analisi Asintotica

- ignorare le costanti dipendenti dalla macchina
- studiare il tasso di crescita di  $T(n)$  con  **$n$  che tende ad inf**

**Nota** - a causa dei fattori costanti e dei termini di ordine inferiore che vengono ignorati, se

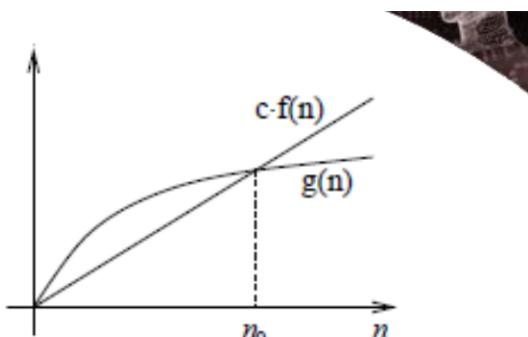
$T_1(n) > T_2(n)$  asintoticamente:

- per input piccoli algoritmo1 può richiedere meno tempo di algoritmo 2, ma...
- per input sufficientemente grandi algoritmo2 sarà eseguito più velocemente di algoritmo1

## Definizioni

Definizione1.  $T(n)$  è  $O(g(n))$  se esistono due numeri positivi  $c$  ed  $N$  tali che  $T(n) \leq c g(n)$  per qualsiasi  $n \geq N$ .

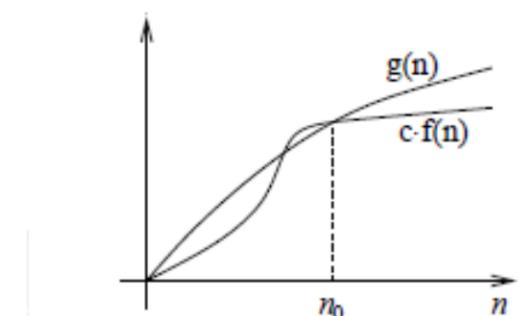
Si scrive  $T(n) = O(g(n))$



- $g(n)$  limita superiormente  $T(n)$
- $g(n)$  approssima asintoticamente  $T(n)$  dall'alto
- $g(n)$  è una buona approssimazione superiore per  $T(n)$  quando  $n$  è molto grande

Definizione2.  $T(n)$  è  $\Omega(g(n))$  se esistono due numeri positivi  $c$  ed  $N$  tali che  $T(n) \geq c g(n)$  per qualsiasi  $n \geq N$ .

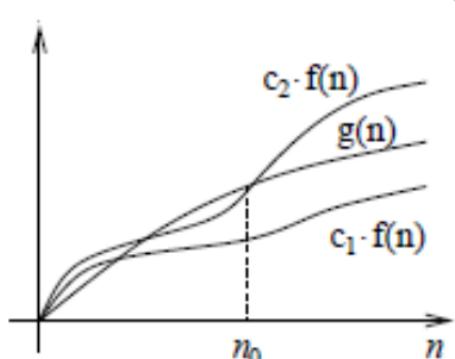
Si scrive  $T(n) = \Omega(g(n))$



- $g(n)$  limita inferiormente  $T(n)$
- $g(n)$  approssima asintoticamente  $T(n)$  dal basso
- $g(n)$  è una buona approssimazione inferiore per  $T(n)$  quando  $n$  è molto grande

Definizione3.  $T(n)$  è  $\Theta(g(n))$  se esistono numeri positivi  $c_1, c_2$ ,  $N$  tali che  $c_1 g(n) \leq T(n) \leq c_2 g(n)$  per qualsiasi  $n \geq N$

Si scrive  $T(n) = \Theta(g(n))$



- $g(n)$  limita  $T(n)$
- $g(n)$  approssima asintoticamente  $T(n)$
- $g(n)$  è una buona approssimazione per  $T(n)$  quando  $n$  è molto grande

1. Se  $T(n) = c$ , allora  $T(n) = O(1)$ ,  $T(n) = \Omega(1)$ ,  $T(n) = \Theta(1)$
2. Se  $T(n) = c \cdot f(n)$ , allora  $T(n) = O(f(n))$ ,  $T(n) = \Omega(f(n))$ ,  $T(n) = \Theta(f(n))$
3. Se  $g(n) = O(f(n))$  e  $f(n) = O(h(n))$ , allora  $g(n) = O(h(n))$  [anche per  $\Omega$  e  $\Theta$ ]
4.  $f(n) + g(n)$  ha complessità  $O(\max(f(n), g(n)))$  [anche per  $\Omega$  e  $\Theta$ ]
5. Se  $g(n) = O(f(n))$  e  $h(n) = O(q(n))$ , allora  $g(n) \cdot h(n) = O(f(n) \cdot q(n))$  [anche per  $\Omega$  e  $\Theta$ ]

## Classi di Complessità

- $T(n) = O(1)$ : complessità costante (cioè  $T(n)$  non dipende dalla dimensione  $n$  dei dati di ingresso).
- $T(n) = O(\log n)$ : complessità logaritmica.
- $T(n) = O(n)$ : complessità lineare.
- $T(n) = O(n \cdot \log n)$ : complessità pseudolineare (così detta da  $n \cdot \log n = O(n^{1+\epsilon})$  per ogni  $\epsilon > 0$ ).
- $T(n) = O(n^2)$ : complessità quadratica.
- $T(n) = O(n^3)$ : complessità cubica.
- $T(n) = O(n^k)$ ,  $k > 0$ : complessità polinomiale.
- $T(n) = O(\alpha^n)$ ,  $\alpha > 1$ : complessità esponenziale.

## Complessità Asintotica e Tempo

complessità \ dim. input		10	$10^3$	$10^6$
costante	- $O(1)$	1 $\mu$ sec	1 $\mu$ sec	1 $\mu$ sec
logaritmica	- $O(\lg n)$	3 $\mu$ sec	10 $\mu$ sec	20 $\mu$ sec
lineare	- $O(n)$	10 $\mu$ sec	1 msec	1 sec
$n \lg n$	- $O(n \lg n)$	33 $\mu$ sec	10 msec	20 sec
quadratica	- $O(n^2)$	100 $\mu$ sec	1 sec	$10^{12}$ sec
cubica	- $O(n^3)$	1 msec	$10^9$ sec	$10^{18}$ sec
esponenziale	- $O(2^n)$	10 msec	$10^{301}$ sec	$10^{301030}$ sec

11.6 gg

31709 anni !!

16.7 min

# **Complessità di un Problema**

## **Complessità di un algoritmo:**

- misura del numero di passi che si devono eseguire per risolvere il problema

## **Complessità di un problema:**

- complessità del migliore algoritmo che lo risolve