

PROGRAMMAZIONE I

**Corso del Prof. Daniel Riccio
(2020/2021)**

Indice

L'Informatica	3
• Tutte le Basi	4
 Il Codice Binario e Rappresentazione	13
• Codice Binario	14
• Rappresentazione	15
 Il Linguaggio C	23
• Il Linguaggio	24
 <hr/> Algebra di Boole	32
 <hr/> Le Strutture Condizionali	35
• Strutture di Controllo Iterative	36
 I Vettori	40
Le Matrici	42
La Rappresentazione dei Testi	43
• Stringa	43
 <hr/> I Record	53
I Puntatori	55
I Puntatori a Funzioni	73
Gli Ordinamenti	75
Le Liste	80
La Ricorsione	95
 Fine	98

Informatica

Cos'è L'Informatica?

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

Informazione Automatica

Informatica

Dato



Informazione



Conoscenza ←

Permette di risolvere problemi velocemente e automaticamente

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

Calcolatore

Il **calcolatore** è una macchina che **memorizza, elabora e distribuisce l'informazione**.



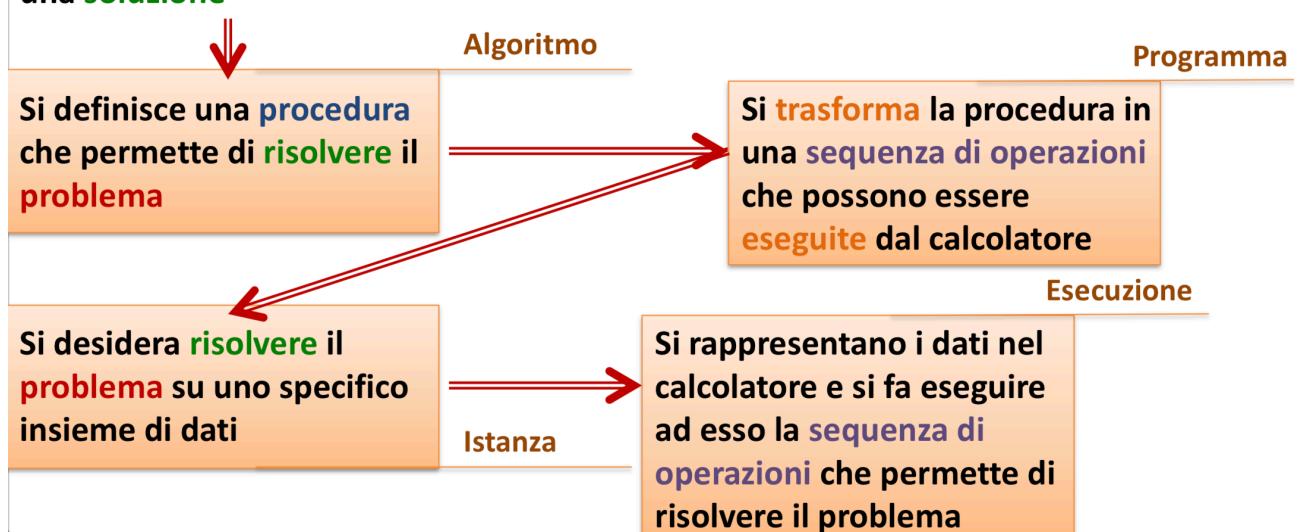
Il calcolatore esegue istruzioni, che gli vengono impartite al fine di risolvere problemi.

Per definire bene l'informatica abbiamo bisogno di acquisire 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 elaborative (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 elaborative che portano alla realizzazione di un compito. Esso deve essere
 - Comprensibile
 - Corretto
 - Efficiente

Problem Solving:

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



Programma

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

Il linguaggio naturale (italiano, inglese, ...) è complesso ed ambiguo, mentre il calcolatore “parla” linguaggi precisi, non ambigui ed estremamente semplici in termini di

- **sintassi** (regole grammaticali)
- **semantica** (significato)

La Programmazione

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

Problema: calcolare l'area di un triangolo, considerando base e altezza

Soluzione (ad alto livello)

Semiprodotto della base nell'altezza

Soluzione (elementare)

Moltiplico la base per l'altezza e divido il risultato per 2.

Soluzione (algoritmica)

Sia **b** la base del triangolo **T**

Sia **h** l'altezza del triangolo **T**

Calcolo **p** come **b×h**

Calcolo **A** come **p : 2**

Programma C

```
1 #include<stdio.h>
2
3 int main ()
4 {
5     int base, altezza, area;
6     printf ("digita base: ");
7     scanf ("%d", &base);
8     printf ("digita altezza: ");
9     scanf ("%d", &altezza);
10    area = base * altezza / 2;
11    printf ("Area: %d ", area);
12    return 0;
13 }
```

Esecuzione

```
digita base: 4
digita altezza: 2
Area: 4
```

Il Compilatore

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.

Dal Sorgente all'eseguibile...

Steps:

- **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 (pippo), 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**

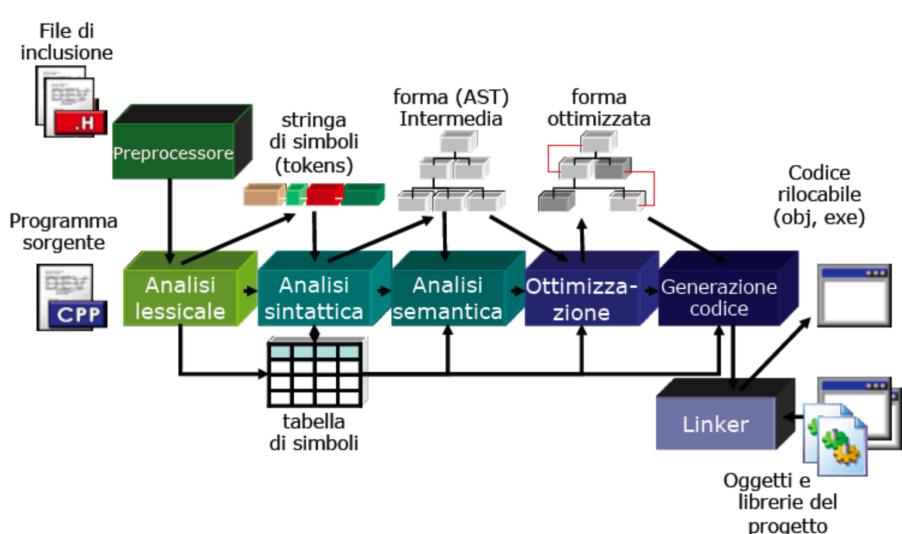
Dall'albero di sintassi viene generato il codice intermedio.

- **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



Sistema di Calcolo

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

- Livello utente
- Livello applicazione
- Livello SO
- Livello fisico

Sistema Operativo

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

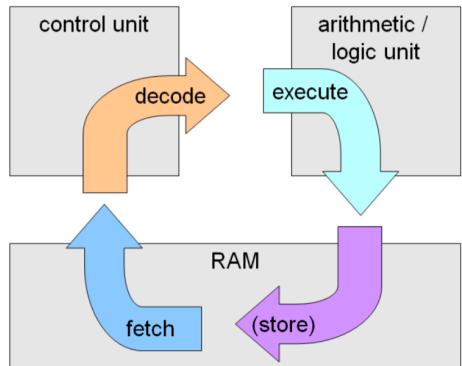
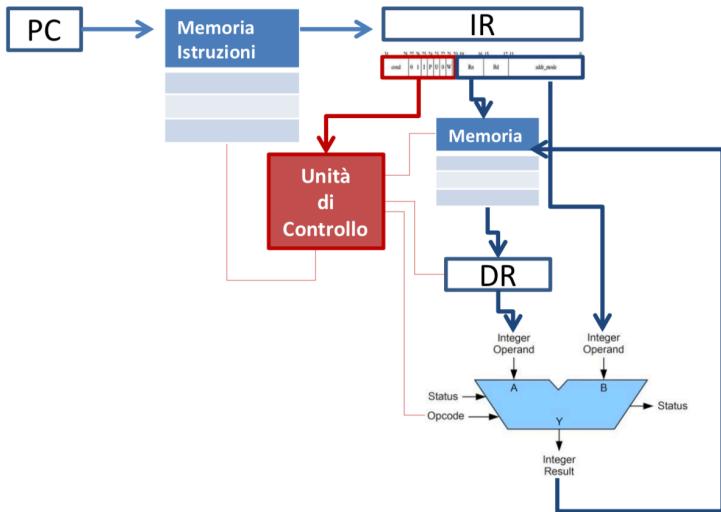
Esecutore

Esecutore: una qualsiasi entità **E** (umana o 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**)

Il Processo di esecuzione delle istruzioni

Una Istruzione ha un ciclo di vita che consta di quattro fasi principali:



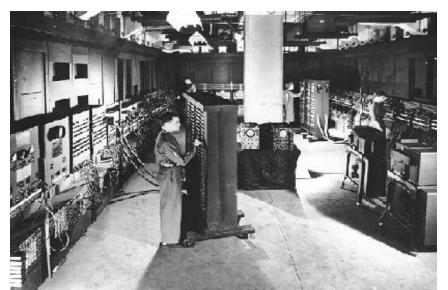
Un po' di Storia...

Primi Calcolatori



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 (Univ. Pennsylvania). Era programmato tramite inserimento di cavi e azionamento di interruttori



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

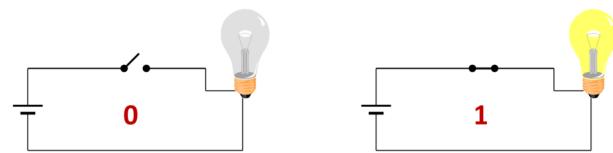
Il Transistor

Il **transistor** o transistore è un dispositivo a semiconduttore largamente usato nell'elettronica analogica.

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

La tensione all'uscita di un transistor può assumere due stati tensione alta (1) o tensione bassa (0)

La cifra binaria è detta **bit**, dall'unione di due elisioni: **binary digit**



Hardware e Software

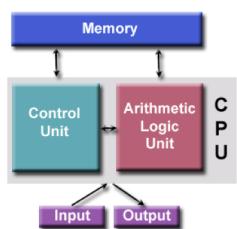
La maggior parte delle persone ha familiarità con le prestazioni entusiasmanti dei computer, e con questo testo apprenderete come ottenerle. È il **software** (cioè le istruzioni che si scrivono per far eseguire ai computer delle **azioni** e prendere **decisioni**) che controlla i computer (denominati spesso **hardware**).

I Computer di Oggi

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

I computer possono eseguire calcoli e prendere decisioni logiche in modo straordinariamente più rapido degli esseri umani. Molti dei personal computer di oggi possono eseguire miliardi di calcoli in un secondo, più di quanto un essere umano possa eseguirne in tutta la sua vita.

I computer elaborano i dati sotto il controllo di sequenze di istruzioni chiamate **programmi per computer**. Questi programmi guidano il computer attraverso



sequenze ordinate di azioni specificate da professionisti chiamati **programmatori**.

- **Legge di Moore**

Generalmente, ogni anno ci si aspetta di pagare almeno un po' di più per la maggior parte dei prodotti e dei servizi. Tutto il contrario avviene invece nel campo dei computer e della telecomunicazione.

Ogni anno o due le capacità dei computer sono quasi raddoppiate senza incrementi dei costi. Questa straordinaria tendenza è chiamata spesso **Legge di Moore**, dal nome della persona che l'ha identificata negli anni Sessanta, Gordon Moore, co-fondatore di Intel, il principale produttore di processori per gli odierni computer e per i sistemi embedded.

- **Organizzazione dei computer**

Malgrado le differenze di aspetto, i computer si possono considerare come suddivisi in varie **unità logiche** o moduli.

Unità di input: Questa unità di “**ricezione**” ottiene informazioni (dati e programmi per computer) da **dispositivi di input** e li mette a disposizione delle altre unità di elaborazione. La maggioranza delle informazioni inserite dagli utenti è acquisita dai computer attraverso tastiere, touchscreen e mouse. Altre forme di input riguardano la ricezione di comandi vocali, la scansione di immagini, la lettura da dispositivi di memoria secondaria (come unità disco, lettori di DVD, unità Blu-ray Disc™ e unità flash USB, chiamate anche “chiavette USB” o “memory stick”),

Unità di output: Questa unità di “**consegna**” prende le informazioni elaborate dal computer e le invia a vari **dispositivi di output** per poterle utilizzare all'esterno del computer. La maggior parte delle informazioni inviate in uscita dai computer viene oggi presentata su schermi, stampata su carta, trasmessa come audio o video su PC, media player (come gli iPod di Apple).

Unità di memoria: Questa unità di “**immagazzinamento**”, ad accesso rapido e con capacità relativamente bassa, conserva le informazioni acquisite attraverso

l’unità di input, rendendole, non appena è necessario, immediatamente disponibili per l’elaborazione. Le informazioni nell’unità di memoria sono *volatili* e vanno normalmente perdute quando l’alimentazione del computer viene staccata. L’unità di memoria è spesso chiamata semplicemente **memoria, memoria primaria o RAM** (Random Access Memory). Le dimensioni della memoria principale su computer di tipo desktop e notebook possono raggiungere 128 GB di RAM,

Unità aritmetica e logica – ALU (Arithmetic and Logic Unit): Questa unità di “**produzione**” esegue *calcoli*, come addizioni, sottrazioni, moltiplicazioni e divisioni. Comprende anche i meccanismi di *decisione* che consentono al computer, ad esempio, di confrontare due elementi contenuti nell’unità di memoria per determinare se sono uguali. Nei sistemi attuali, la **ALU** è solitamente implementata come parte dell’unità logica successiva, la CPU.

Unità centrale di elaborazione – CPU (Central Processing Unit): Questa unità “**amministrativa**” coordina e supervisiona le operazioni delle altre unità.

La CPU:

- dice all’unità di input quando le informazioni devono essere lette e trasferite nell’unità di memoria
- dice alla ALU quando le informazioni presenti nell’unità di memoria devono essere usate nei calcoli
- dice all’unità di output quando inviare informazioni dall’unità di memoria a determinati dispositivi di output.

Unità di memoria secondaria: Questa è l’unità di “**stoccaggio**” a lungo termine e ad alta capacità. I programmi o i dati non utilizzati attivamente da altre unità sono normalmente posti su dispositivi di memoria secondaria (es. il vostro *disco fisso*) finché non servono di nuovo, forse ore, giorni, mesi o persino anni dopo. Le informazioni su dispositivi di memoria secondaria sono **persistenti**: si conservano anche quando l’alimentazione del computer viene disattivata.

• Sistemi operativi

I **sistemi operativi** sono sistemi software che rendono l’uso dei computer più comodo per gli utenti, gli sviluppatori di applicazioni e gli amministratori di sistema. Essi forniscono servizi che consentono di eseguire ogni applicazione con sicurezza, in maniera efficiente e in modo *concorrente* (cioè in parallelo) con altre applicazioni. Il software che contiene i componenti fondamentali del sistema

operativo è chiamato **kernel** (nucleo). Sistemi operativi per desktop molto diffusi sono Linux, Windows e Mac OS X.

• Internet

Alla fine degli anni Sessanta, la Advanced Research Projects Agency (ARPA) del Dipartimento della Difesa degli Stati Uniti mise a punto un progetto per collegare in rete i principali sistemi informatici di circa una dozzina di università e istituti di ricerca da essa finanziati. I computer dovevano essere collegati con linee di comunicazione che operavano a una velocità dell'ordine di 50.000 bit al secondo, una velocità stupefacente in un momento in cui la maggior parte delle persone si connetteva ai computer tramite linee telefoniche alla velocità di 110 bit al secondo. La ricerca accademica era sul punto di fare un gigantesco passo avanti. ARPA procedette a implementare ciò che rapidamente divenne noto come ARPANET, il precursore di quello che oggi è Internet.

Le cose sono andate diversamente dal progetto originario. La principale utilità di ARPANET è stata quella di permettere una forma di comunicazione semplice e rapida attraverso quella che oggi è chiamata posta elettronica (e-mail).

Il protocollo (un insieme di regole) per comunicare su ARPANET divenne noto come **TCP (Transmission Control Protocol)**. Il TCP assicurava che i messaggi, costituiti da parti numerate in sequenza chiamate pacchetti, venissero instradati in modo corretto dal mittente al destinatario, arrivassero intatti e fossero assemblati nell'ordine corretto.

Varie organizzazioni in tutto il mondo implementavano le proprie reti sia per la comunicazione intraorganizzativa che per quella interorganizzativa.

Una sfida era permettere a queste diverse reti di comunicare tra loro. ARPA raggiunse l'obiettivo sviluppando l'**IP** (Internet Protocol), che creava effettivamente una rete di network, l'attuale architettura di Internet. L'insieme combinato di protocolli è ora chiamato **TCP/IP**.

Il World Wide Web

Il **World Wide Web** (detto semplicemente “il web”) è una raccolta di hardware e software associati a Internet che consente agli utenti di computer di trovare e visualizzare documenti multimediali su quasi ogni argomento. Nel 1989, Tim Berners-Lee del CERN (il Centro Europeo di Ricerca Nucleare) iniziò a sviluppare una tecnologia per condividere informazioni tramite documenti testuali collegati da hyperlink.

Berners-Lee chiamò la sua invenzione **HyperText Markup Language (HTML)**. Egli scrisse anche protocolli di comunicazione come l'**HyperText Transfer Protocol (HTTP)** per costituire la spina dorsale del suo nuovo sistema informativo che chiamò **World Wide Web**.

Nel 1994, Berners-Lee ha fondato un organismo, chiamato **World Wide Web Consortium (W3C)**, dedicato allo sviluppo di tecnologie web. Uno degli obiettivi principali del W3C è quello di rendere il web universalmente accessibile, indipendentemente da disabilità, lingua o cultura.

Codice Binario e Rappresentazione

Il Codice Binario

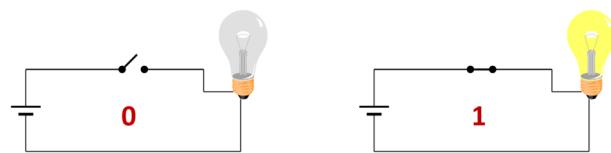
Il **sistema numerico binario** è un sistema numerico posizionale in base 2. Esso utilizza solo due simboli, di solito indicati con 0 e 1. Ciascuno dei numeri espressi nel sistema numerico binario è definito "**numero binario**".

In informatica il sistema binario è utilizzato per la rappresentazione interna dell'informazione dalla quasi totalità degli elaboratori elettronici, in quanto le caratteristiche fisiche dei circuiti digitali rendono molto conveniente la gestione di due soli valori, rappresentati fisicamente da due diversi livelli di tensione elettrica. Tali valori assumono convenzionalmente il significato numerico di 0 e 1 o quelli di vero e falso della logica booleana.

Il Bit

La tensione all'uscita di un transistor può assumere due stati tensione alta (1) o tensione bassa (0)

La cifra binaria è detta **bit**, dall'unione di due elisioni: **binary digit**



Con **n** bit si rappresentano 2^n numeri.

Il Byte

Un **byte** è un insieme di 8 bit (un numero binario a 8 cifre)

$b_7 b_6 b_5 b_4 b_3 b_2 b_1 b_0$

Con un byte si rappresentano i numeri interi fra 0 e $2^8 - 1 = 255$.

Rappresentazione

I sistemi numerali posizionali

Sistemi di numerazione posizionali:

La **base** del sistema di numerazione

Le **cifre** del sistema di numerazione

Il numero è scritto specificando le cifre in ordine ed il suo valore dipende dalla **posizione relativa** delle cifre

Esempio: Il 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

Un numero **intero** N si rappresenta con la scrittura $(c_n c_{n-1} \dots c_2 c_1 c_0)_B$

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

c_n è la **cifra più significativa**, c_0 la **meno significativa**

Un numero **frazionario** N' si rappresenta come $(0, c_1 c_2 \dots c_n)_B$

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

• Sistema Binario

Il **sistema numerico binario** è un sistema numerico posizionale in base 2. Esso utilizza solo due simboli, di solito indicati con 0 e 1. Ciascuno dei numeri espressi nel sistema numerico binario è definito "**numero binario**".

• 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

La corrispondenza in decimale delle cifre oltre il 9 è

$$\begin{array}{ll} A = (10)_{10} & D = (13)_{10} \\ B = (11)_{10} & E = (14)_{10} \\ C = (12)_{10} & F = (15)_{10} \end{array}$$

Esempio:

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

Conversioni

• Da Decimale a Binario (Numeri interi)

Si divide ripetutamente il numero **intero** decimale per 2 fino ad ottenere un quoziente nullo; le cifre del numero binario sono i resti delle divisioni; la cifra più significativa è l'ultimo resto.

Esempio: convertire in binario $(43)_{10}$

$$\begin{array}{r} 43 : 2 = 21 + 1 \quad \text{resti} \\ 21 : 2 = 10 + 1 \\ 10 : 2 = 5 + 0 \\ 5 : 2 = 2 + 1 \\ 2 : 2 = 1 + 0 \\ 1 : 2 = 0 + 1 \quad \text{bit più significativo} \end{array}$$

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

• 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 $(0,21875)_{10}$ e $(0,45)_{10}$

$$\begin{array}{ll} \begin{array}{l} 0,21875 \times 2 = 0,4375 \\ 0,4375 \times 2 = 0,875 \\ 0,875 \times 2 = 1,75 \\ 0,75 \times 2 = 1,5 \\ 0,5 \times 2 = 1,0 \end{array} & \begin{array}{l} 0,45 \times 2 = 0,9 \\ 0,90 \times 2 = 1,8 \\ 0,80 \times 2 = 1,6 \\ 0,60 \times 2 = 1,2 \\ 0,20 \times 2 = 0,4 \text{ etc.} \end{array} \\ \downarrow & \downarrow \\ (0,21875)_{10} = (0,00111)_2 & (0,45)_{10} \approx (0,01110)_2 \end{array}$$

• Da Binario a Decimale

Oltre all'espansione esplicita in potenze di 2 – **forma polinomia...**

$$(101011)_2 = 1 \times 2^5 + 0 \times 2^4 + 1 \times 2^3 + 0 \times 2^2 + 1 \times 2^1 + 1 \times 2^0 = (43)_{10}$$

Si può 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 fino al bit meno significativo.

Esempio: convertire in decimale $(101011)_2$

bit più significativo

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

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

● Da Binario a Esadecimale

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

Esempio: 32 bit corrispondono a 8 cifre esadecimali

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 (es. C e Java) per rappresentare numeri esadecimali

0 c 8 f
0000 1100 1000 1111

Il numero binario ha $4 \times 4 = 16$ bit

Rappresentazioni Numeri con Segno

Per rappresentare **numeri con segno**, occorre utilizzare un bit per definire il segno del numero. Si possono usare 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, **1** per quelli negativi

Se si utilizzano **n** bit si rappresentano tutti i numeri compresi fra

$$-(2^{n-1}-1) \text{ e } +2^{n-1}-1$$

Esempio: con 4 bit si rappresentano i numeri 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:

$$\bar{A} = 2^n - 1 - A$$

Regola pratica:

il complemento a uno di un numero binario A si ottiene cambiando il valore di tutti i suoi bit (**complementando** ogni bit)

– Esempio:

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

Complemento a 2

Il **complemento a 2** di un numero binario $(A)_2$ a **n** cifre è il numero $2^n - (A)_2 =$

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

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

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

In Generale...

- I **numeri positivi** sono rappresentati (come) in modulo e segno
- I **numeri negativi** sono rappresentati in complemento a 2 per la cifra più significativa ha sempre valore 1
- Lo **zero** è rappresentato come numero positivo (con una sequenza di n zeri)

Il campo dei numeri rappresentabili varia da -2^{n-1} a $+2^{n-1}-1$.

Operazioni

Addizione Binaria

Le regole per l'addizione di due bit sono

$$0 + 0 = 0 \quad 0 + 1 = 1 \quad 1 + 0 = 1 \quad 1 + 1 = 0 \text{ con riporto di 1}$$

$$\text{L'ultima regola è... } (1)_2 + (1)_2 = (10)_2 \dots (1+1=2)_{10}$$

Esempio

A binary addition diagram. On the left, there is a vertical column of bits: 1, 1, 1 above the binary number 01011011; followed by a plus sign, another 01011011, and a horizontal line below them. To the right of this column, the word "riporti" is written above a red arrow pointing to the third column from the left. To the right of the arrow is a yellow box containing the addition: 91+90, with a horizontal line separating the first two digits from the result 181.

In complemento a 1 (più semplice da calcolare)...

- Zero ha due rappresentazioni: 00000000 e 11111111
- La somma bit a bit funziona "quasi sempre"

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

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

In complemento a 2...

- Zero ha una sola rappresentazione
- La somma bit a bit funziona sempre

Somma e Sottrazione in Complemento a 2

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

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

Esempio:

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

L'Overflow

L'Overflow si ha quando il risultato di un'operazione non è rappresentabile correttamente con n bit.

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

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

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

Punteggio nei vecchi videogame... sorpresa per i campioni!

$$\begin{array}{r} 0111\ 1111\ 1111\ 1111 + 1 = 1000\ 0000\ 0000\ 0000 \\ 32767 \qquad \qquad + 1 = \qquad -32768 \end{array}$$

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**.

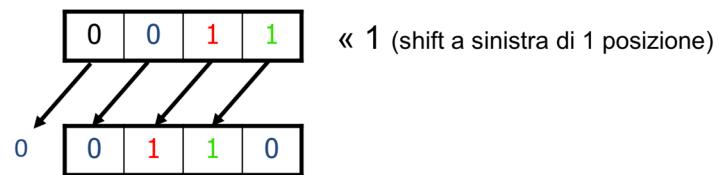
• Shift a sinistra (<<)

Si inserisce come **LSB** un bit a zero. Equivale ad una **moltiplicazione per due**:

$$0011 \ll 1 = 0110 \quad (3 \times 2 = 6)$$

$$0011 \ll 2 = 1100 \quad (3 \times 2^2 = 12)$$

$$0011 \ll 3 = 11000 \quad (3 \times 2^3 = 24)$$

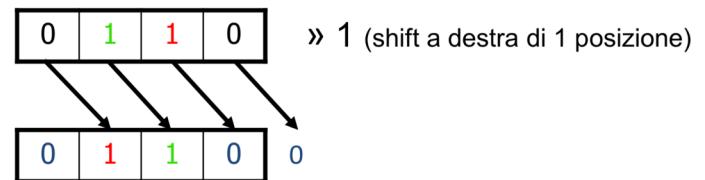


• Shift a destra (>>)

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

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

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



Codifica dei caratteri alfabetici

Oltre ai numeri, molte applicazioni informatiche elaborano caratteri (simboli). Gli elaboratori elettronici trattano numeri.

Si codificano i caratteri e i simboli per mezzo di numeri. Per poter scambiare dati (testi) in modo corretto, occorre definire uno standard di codifica.

La codifica deve prevedere le lettere dell’alfabeto, le cifre numeriche, i simboli, la punteggiatura, i caratteri speciali per certe lingue (æ, ã, è, è,...)

Lo standard di codifica più diffuso è il **codice ASCII**, 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 con il bit più significativo a 1 (quelli con codice dal 128 al 255) rappresentano un'estensione della codifica.

La tabella comprende sia **caratteri di controllo** (codici da 0 a 31) che **caratteri stampabili**. I caratteri alfabetici/numerici hanno codici ordinati secondo l'ordine alfabetico/numerico.

Tabella ASCII

The ASCII code

American Standard Code for Information Interchange

ASCII control characters			ASCII printable characters									Extended ASCII characters											
DEC	HEX	Simbolo ASCII	DEC	HEX	Simbolo	DEC	HEX	Simbolo	DEC	HEX	Simbolo	DEC	HEX	Simbolo	DEC	HEX	Simbolo	DEC	HEX	Simbolo			
00	00h	NULL (carácter nulo)	32	20h	espacio	64	40h	@	96	60h	'	128	80h	ç	160	A0h	á	192	C0h	l	224	E0h	ó
01	01h	SOH (inicio encabezado)	33	21h	!	65	41h	A	97	61h	a	129	81h	ü	161	A1h	í	193	C1h	ł	225	E1h	ń
02	02h	STX (inicio texto)	34	22h	"	66	42h	B	98	62h	b	130	82h	é	162	A2h	ó	194	C2h	ł	226	E2h	ń
03	03h	ETX (fin de texto)	35	23h	#	67	43h	C	99	63h	c	131	83h	â	163	A3h	ú	195	C3h	ł	227	E3h	ń
04	04h	EOT (fin transmisión)	36	24h	\$	68	44h	D	100	64h	d	132	84h	ä	164	A4h	ñ	196	C4h	—	228	E4h	ö
05	05h	ENQ (enquiry)	37	25h	%	69	45h	E	101	65h	e	133	85h	à	165	A5h	ñ	197	C5h	+	229	E5h	ö
06	06h	ACK (acknowledgement)	38	26h	&	70	46h	F	102	66h	f	134	86h	á	166	A6h	»	198	C6h	ä	230	E6h	µ
07	07h	BEL (timbre)	39	27h	'	71	47h	G	103	67h	g	135	87h	ç	167	A7h	°	199	C7h	À	231	E7h	þ
08	08h	BS (retroceso)	40	28h	(72	48h	H	104	68h	h	136	88h	é	168	A8h	¿	200	C8h	Ł	232	E8h	þ
09	09h	HT (tab horizontal)	41	29h)	73	49h	I	105	69h	i	137	89h	ë	169	A9h	®	201	C9h	Ł	233	E9h	ú
10	0Ah	LF (salto de linea)	42	2Ah	*	74	4Ah	J	106	6Ah	j	138	8Ah	è	170	AAh	¬	202	CAh	Ł	234	EAh	ú
11	0Bh	VT (tab vertical)	43	2Bh	+	75	4Bh	K	107	6Bh	k	139	8Bh	í	171	A8h	½	203	CBh	Ł	235	EBh	ú
12	0Ch	FF (form feed)	44	2Ch	,	76	4Ch	L	108	6Ch	l	140	8Ch	í	172	ACh	¼	204	CCh	Ł	236	ECh	ý
13	0Dh	CR (retorno de carro)	45	2Dh	-	77	4Dh	M	109	6Dh	m	141	8Dh	í	173	ADh	í	205	CDh	=	237	EDh	ý
14	0Eh	SO (shift Out)	46	2Eh	.	78	4Eh	N	110	6Eh	n	142	8Eh	Á	174	A Eh	«	206	CEh	+	238	EEh	—
15	0Fh	SI (shift In)	47	2Fh	/	79	4Fh	O	111	6Fh	o	143	8Fh	À	175	A Fh	»	207	CFh	□	239	E Fh	-
16	10h	DLE (data link escape)	48	30h	0	80	50h	P	112	70h	p	144	90h	É	176	B0h	—	208	D0h	ð	240	F0h	÷
17	11h	DC1 (device control 1)	49	31h	1	81	51h	Q	113	71h	q	145	91h	æ	177	B1h	—	209	D1h	ð	241	F1h	±
18	12h	DC2 (device control 2)	50	32h	2	82	52h	R	114	72h	r	146	92h	Æ	178	B2h	—	210	D2h	ð	242	F2h	—
19	13h	DC3 (device control 3)	51	33h	3	83	53h	S	115	73h	s	147	93h	ó	179	B3h	—	211	D3h	—	243	F3h	—
20	14h	DC4 (device control 4)	52	34h	4	84	54h	T	116	74h	t	148	94h	ò	180	B4h	—	212	D4h	ð	244	F4h	—
21	15h	NAK (negative acknowle.)	53	35h	5	85	55h	U	117	75h	u	149	95h	ò	181	B5h	À	213	D5h	—	245	F5h	§
22	16h	SYN (synchronous idle)	54	36h	6	86	56h	V	118	76h	v	150	96h	ú	182	B6h	Á	214	D6h	—	246	F6h	÷
23	17h	ETB (end of trans. block)	55	37h	7	87	57h	W	119	77h	w	151	97h	ú	183	B7h	À	215	D7h	—	247	F7h	—
24	18h	CAN (cancel)	56	38h	8	88	58h	X	120	78h	x	152	98h	ý	184	B8h	©	216	D8h	—	248	F8h	—
25	19h	EM (end of medium)	57	39h	9	89	59h	Y	121	79h	y	153	99h	Ó	185	B9h	—	217	D9h	—	249	F9h	—
26	1Ah	SUB (substitute)	58	3Ah	:	90	5Ah	Z	122	7Ah	z	154	9Ah	Ü	186	B Ah	—	218	D Ah	—	250	F Ah	—
27	1Bh	ESC (escape)	59	3Bh	;	91	5Bh	[123	7Bh	{	155	9Bh	ø	187	B Bh	—	219	D Bh	—	251	F Bh	—
28	1Ch	FS (file separator)	60	3Ch	≤	92	5Ch	\	124	7Ch		156	9Ch	£	188	B Ch	—	220	D Ch	—	252	F Ch	—
29	1Dh	GS (grupo separator)	61	3Dh	=	93	5Dh]	125	7Dh	}	157	9Dh	Ø	189	B Dh	€	221	D Dh	—	253	F Dh	—
30	1Eh	RS (record separator)	62	3Eh	>	94	5Eh	^	126	7Eh	~	158	9Eh	×	190	B Eh	¥	222	D Eh	—	254	F Eh	—
31	1Fh	US (unit separator)	63	3Fh	?	95	5Fh	—				159	9Fh	f	191	B Fh	—	223	D Fh	—	255	F Fh	—

theASCIicode.com.ar

Linguaggio

C

Il Linguaggio C

Sviluppato tra il 1969 ed il 1973 presso gli AT&T Bell Laboratories (Ken Thompson, B. Kernighan, Dennis Ritchie)

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

Nel 1978 viene pubblicato “The C Programming Language”, prima specifica ufficiale del linguaggio.

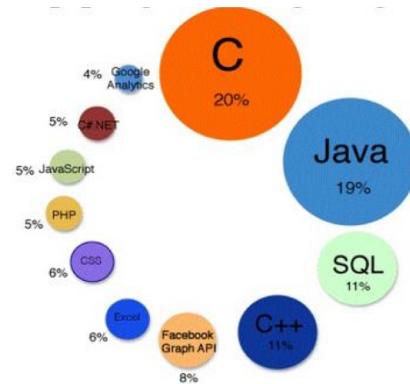
Il C è un linguaggio:

- **Imperativo** ad alto livello
- **Strutturato**
- **Tipizzato** Ogni oggetto ha un tipo
- **Elementare** Poche keyword
- **Case sensitive** Maiuscolo diverso da minuscolo negli identificatori!
- **Portable**
- **Standard ANSI**

I **linguaggi** attualmente più diffusi al mondo sono:

-C, C++(un’evoluzione del C), Java(la cui sintassi è tratta da C++), C#(estremamente simile a Java e C++)).

Il **linguaggio C** è uno dei linguaggi più diffusi. La sintassi del linguaggio C è ripresa da tutti gli altri linguaggi principali



La Struttura di un programma in C



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 il simbolo '#'

Direttive principali:

#include
#define

Esempio:

```
#include <stdio.h>
#include <math.h>
```

```
int main()
{
    // programma vuoto
    return 0;
}
```

Sintassi:

#Include

#include <file> per includere un file di sistema

#include “file” per includere un file definito dal programmatore.

#Define

#define <costante> <valore>

<costante>: Identificatore della costante simbolica (convenzionalmente indicato tutto in maiuscolo)

<valore>: Un 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 I/O

In C, le **operazioni di I/O** non sono gestite tramite vere e proprie istruzioni, bensì mediante opportune funzioni.

Vi sono due Funzioni principali:

- **printf(<formato>,<arg1>,...,<argn>);**
- **scanf(<formato>,<arg1>,...,<argn>);**

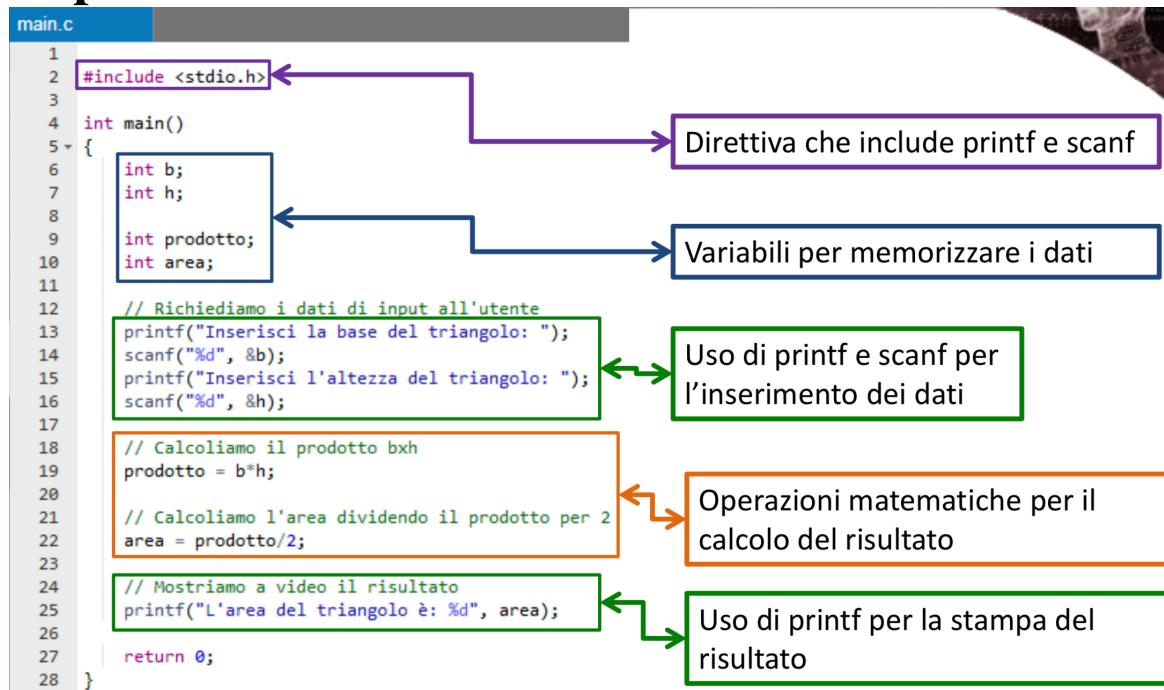
<formato> è una sequenza di **caratteri** che determina il formato di stampa di ognuno dei vari argomenti

nella forma %<carattere>

%**d** intero %**u** unsigned %**s** stringa %**c** carattere
%**x** esadecimale %**o** ottale %**f** float %**g** double

<arg1>,...,<argn> sono le variabili, in cui si vogliono memorizzare i valori, associate alle direttive di formato nello stesso ordine.

Esempio di uso delle Funzioni di I/O



Seguenza di Escape

La tabella ASCII comprende un certo numero di valori ai quali non corrisponde un carattere stampabile. Per rappresentare questi caratteri si utilizza una **sequenza di escape**, dove il carattere **backslash (\)**, detto appunto carattere di escape, assume il ruolo di segnalatore.

Sequenza di escape	Rappresenta
\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

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**
- **Espresioni**
- **Operatori**
- **Simboli o segni speciali**

Sono ignorati:

- **Spazi bianchi, tabulatori, newlines, e commenti**

Gli Identifieri

Identifier è un termine usato dal programmatore per indicare funzioni, variabili, oggetti, costanti, etc.

Ogni identificatore è formato da una sequenza di caratteri di tipo **lettere** o **cifre** o “**_**” (*underscore*)

- Il primo carattere deve essere una **lettera** o **_**
- Caratteri maiuscoli e minuscoli sono diversi
- 31 caratteri
- Identifieri non validi:
 - **un amico** (contiene uno spazio)
 - **un'amica** (contiene un apostrofo)

La Variabile

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.

La Costante

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.

Dichiarazione di Identificatori

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 in memoria**

In C, dichiarare un dato significa, specificarne:

- **Nome** - definisce un identificativo unico per la variabile.
- **Tipo** - l'insieme dei *valori* che può assumere e l'insieme delle *operazioni* che possono applicarsi a tali valori.
- **Modalità di accesso** - valore corrente della variabile.

• Tipi di Base

-char caratteri ASCII ('a', '%') o interi nell'intervallo **[-128,127]**

-int interi (complemento a 2) nell'intervallo **[-2.147.483.648 ; 2.147.483.647]**

-float reali (floating point singola precisione, 7 cifre significative) in valore assoluto in **[1,17 x 10⁻³⁸ e 3,40 x 10³⁸]**

-double reali (floating point doppia precisione, 15 cifre significative) in valore assoluto in **[1,17 x 10⁻³⁰⁸ e 1,79 x 10³⁰⁸]**

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



Qualificatori: **short, long, signed, unsigned**

Specifikatori: **const, extern, static, volatile,...**

L'**Inizializzazione** assegna un valore iniziale ad un identificatore già definito.

ident = valore;
↓
Operatore di assegnazione

La dichiarazione di un identificatore deve precedere il suo primo utilizzo, e può essere in ogni parte del programma

È sempre consigliabile dichiarare tutte le costanti e le variabili **nella parte iniziale** del programma (o della funzione che le utilizza), così è più facile identificare nel programma la **sezione delle dichiarazioni**.

Espressioni di Assegnazione e Operazioni

Espressioni di Assegnazione

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

r-value è una qualsiasi espressione con valore un tipo standard

l-value è una variabile Il tipo di **r-value** deve essere lo stesso o implicitamente convertibile

nel tipo di **l-value** 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:

+ = - = * = / = % =

- Significato: assegnazione + operazione.

Operazioni

Operazioni di incremento e decremento

Per le assegnazioni composte più comuni sono previsti degli operatori esplicativi:

++ --

Significato:

$$\begin{aligned} ++ &\rightarrow +=1 \\ -- &\rightarrow -=1 \end{aligned}$$

Esempi:

```
x++; /* equivale a x = x + 1 */  
valore--; /* equivale a valore = valore - 1 */
```

Operatori Aritmetici

Gli **Operatori Aritmetici** eseguono le principali operazioni matematiche (somma) **+**, (differenza) **-**, (moltiplicazione) *****, (divisione) **/**

Se la divisione è fra due numeri interi, il risultato dell'operazione è ancora un numero intero (troncamento).

Esempio: $27 / 4$ dà come risultato **6** (anziché **6.75**).

Il resto di una divisione fra numeri interi si calcola con l'operatore binario **%**.

Ad esempio, $27 \% 4$ dà come risultato **3**.

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

Operatori Relazionali

Gli **operatori relazionali** sono:

- $>$ (strettamente maggiore)
- \geq (maggiore o uguale)
- $<$ (strettamente minore)
- \leq (minore o uguale)
- $==$ (uguale)
- $!=$ (diverso)

Algebra di Boole

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 la **falsità**.

Variabili e operatori booleani

Variabili in grado di assumere solo due valori:

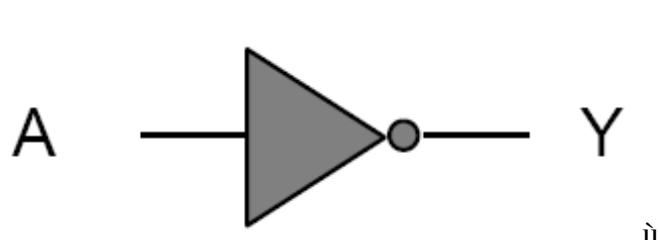
- **VERO**
- **FALSO**

Operatore NOT

L'operatore **NOT** è un operatore logico (operatore booleano) 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. L'operatore **NOT** è indicata con il simbolo \neg oppure con un trattino posto al di sopra della variabile logica

A	\overline{A}
falso	vero
vero	falso



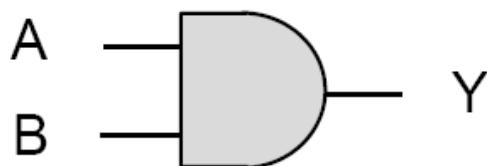
Operatore AND

L'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 soltanto quando entrambe le proposizioni sono vere.

L'operatore **AND** è detto anche congiunzione logica o moltiplicazione logica ed è il simbolo \wedge o il simbolo X

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

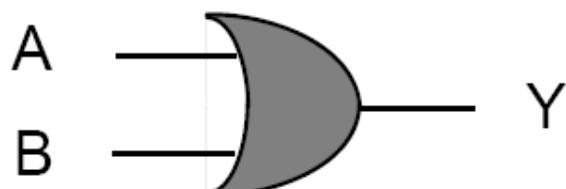


Operatore OR

L'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 ($A \vee B$) è vera. L'operatore **OR** è rappresentato dal simbolo \vee o dal simbolo +

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

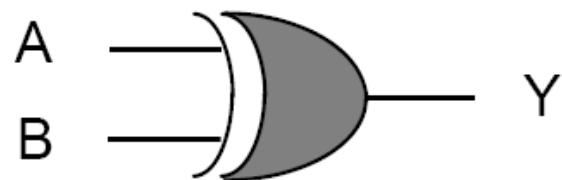


Operatore XOR

L'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 è vera una delle due proposizioni. L'operatore **XOR** è rappresentato dal simbolo \oplus

A	B	$A \oplus B$
falso	falso	falso
falso	vero	vero
vero	falso	vero
vero	vero	falso



Operazione	Operatore	tipo
AND bit a bit	&	Binario
OR bit a bit		Binario
XOR bit a bit	\wedge	Binario
NOT bit a bit (complemento a 1)	\sim	Unario
Shift a sinistra	$<<$	Binario
Shift a destra	$>>$	Binario

Le Strutture Condizionali

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.

Istruzione di controllo **if / else**

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

if (condizione) istruzione;

dove

- **condizione** è una espressione logica
- se la condizione è **true** il programma esegue l'istruzione, altrimenti passa direttamente all'istruzione successiva.

Nel caso di due scelte alternative, all'istruzione di controllo **if** si può associare l'istruzione **else**:

if (condizione) istruzione_1;
else istruzione_2;

dove

- **condizione** è una espressione logica
- se la condizione è **true** il programma esegue **istruzione_1**
- altrimenti esegue **istruzione_2**

Istruzione di controllo 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> ]
}
```

espressione: espressione a valore numerico di tipo **int** o **char**, ma non **float** o **double**

<**costante1**>, <**costante1**>, ... sono costanti dello stesso tipo dell'espressione

<**istruzione 1**>, <**istruzione 2**>, sono sequenze di istruzioni (senza graffe!)

Significato:

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

Strutture di Controllo 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**:

$n! = n \diamond (n - 1) \diamond (n - 2) \dots \diamond 2$ con il caso particolare $0! = 1$.

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**.

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**.

Le strutture di controllo iterative sono corredate dalle istruzioni **break** e **continue**:

L'istruzione **break** nel contesto delle istruzioni di controllo iterative provoca l'immediata terminazione del ciclo

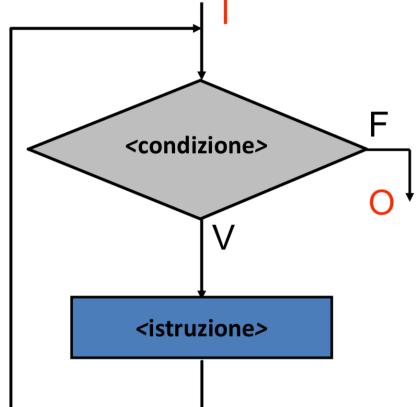
L'istruzione **continue** ha come effetto l'interruzione dell'iterazione corrente. Il controllo di flusso rimane confinato all'interno del ciclo.

Istruzione While

Il costrutto **while** realizza il costrutto **while – do** della programmazione strutturata.
Sintassi:

```
while (<condizione>
      <istruzione>)
```

Finché **<condizione>** è
vera esegue **<istruzione>**
che può essere semplice
o composta.



<condizione> deve essere di tipo logico ed è ricalcolata ad ogni iterazione;

se <condizione> risulta falsa già alla prima iterazione <istruzione> non viene eseguita neppure una volta;

se <condizione> non diventa mai falsa non si esce mai dal loop!

essendo <istruzione> una normale istruzione composta può contenere qualsiasi tipo di istruzione, in particolare altri **while**, dando origine (come per l'if) a **while annidati**.

Istruzione Do-While

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

<condizione> deve essere di tipo logico ed è calcolata ad ogni iterazione;

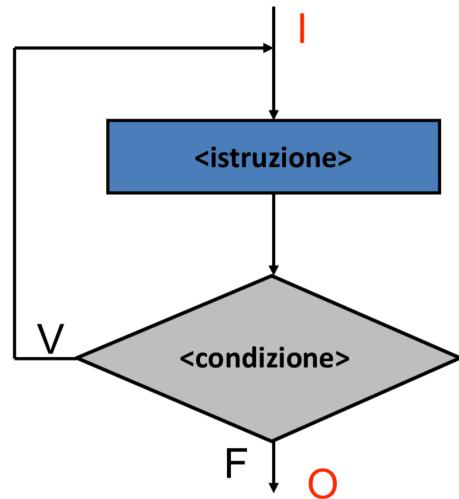
<istruzione> pertanto è sempre eseguita almeno una volta (anche se <condizione> è subito falsa);

se <condizione> non diventa mai falsa non si esce mai dal loop!

Sintassi:

```
do  
  <istruzione>  
  while (<condizione>)
```

Ripeti **<istruzione>**,
che può essere semplice
o composta,
finché **<condizione>**
è vera.



Istruzione For

In altri linguaggi il costrutto **for** permette di eseguire una istruzione, semplice o composta, per un numero prefissato di volte (ciclo a contatore).

Nel linguaggio C è più generale, al punto da poter essere assimilata ad una particolare riscrittura del costrutto **while**.

Sintassi:

```
for (<inizializzazione>; <condizione>; <aggiornamento>)  
  <istruzione>;
```

<condizione> è un'espressione logica; <inizializzazione> e <aggiornamento> sono invece espressioni di tipo qualsiasi. L'istruzione **for** opera secondo il seguente algoritmo:

- viene calcolata <inizializzazione>;
- finché <condizione> è vera (valore non nullo) ;
- viene eseguita <istruzione>;
- viene calcolato <aggiornamento>;

Strutture Informative

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 informativa**.

Le **strutture informative** 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);

Definire una **struttura astratta** 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.

– 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.

Un'altra **classificazione** prevede la distinzione tra

Struttura statica: quando il numero degli 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 (esempio: elenco telefonico, in cui si possono inserire nuovi abbonati, o cancellarne).

I Vettori

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

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

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.

Due tipi di variabili di Vettori:

Variabili scalari: contengono un solo valore

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

Tutti gli elementi condividono lo stesso nome e sono contraddistinti da un indice indicato tra parentesi quadre:

Esempio: int **a[4];**

a[1] a[2] a[3] a[4]

Il Vettore alloca memoria per tutti gli elementi.

tipo nome[num_di_elementi];

Esempio:

int vett[10];

vett :



definisce un vettore di 10 elementi di tipo **int**

Se il vettore non è inizializzato, i valori in esso contenuti sono **indeterminati**

La lista degli inizializzatori può essere indicata tra parentesi graffe dopo un segno '='

int vett[4]={12, 5, 3, 6};

Copia di un Vettore

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

Occorre copiare un elemento per volta dal vettore “sorgente” al vettore “destinazione”, all’interno di un ciclo for

I due vettori devono avere lo stesso numero di elementi, ed essere dello stesso tipo base.

Esempio:

```
/* copia il contenuto di v[] in w[] */  
for( i=0; i<N; i++ ) {  
    w[i] = v[i] ;  
}
```

Nonostante siano coinvolti **due** vettori, occorre **un solo ciclo** for, e **un solo indice** per accedere agli elementi di entrambi i vettori.

Ricerca di un Elemento in un Vettore

Si tratta di una classica istanza del problema di “ricerca di esistenza”

Se l’array non è ordinato ricerca lineare Se l’array è ordinato **ricerca binaria o dicotomica**.

Ricerca Binaria

Sapendo che il vettore è **ordinato** (esiste una relazione d’ordine totale sul dominio degli elementi), la ricerca può essere ottimizzata

– **Vettore ordinato in senso non decrescente:**

se $i \leq j$ si ha $V[i] \leq V[j]$

– **Vettore ordinato in senso crescente:**

se $i \leq j$ si ha $V[i] < V[j]$

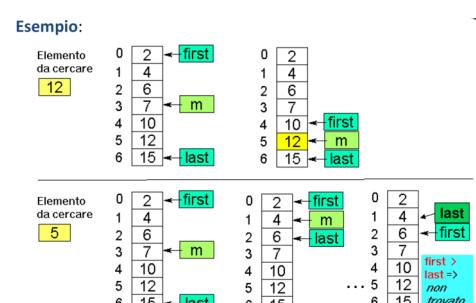
In modo analogo si definiscono l’ordinamento in senso **non crescente** e **decrescente**.

Sia **dim** la dimensione dell’array

– Se l’elemento mediano (posizione **med**) dell’array è l’elemento da cercare **elemento trovato**.

– Se l’elemento mediano dell’array è maggiore dell’elemento da cercare **cercare nella prima metà dell’array** (dalla posizione “0” alla posizione **med-1**).

– Se l’elemento mediano dell’array è minore dell’elemento da cercare **cercare nella seconda metà dell’array** (dalla posizione **med+1** alla posizione “finale”).



La Matrice

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

La definizione di **matrice** ricalca pienamente quella del vettore:

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

tipo_comp può essere un qualunque tipo semplice.

dim1, dim2, ecc.; 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];
```

a	a[0][0]	a[1][0]	a[2][0]	a[3][0]	a[4][0]	a[0]
	a[0][1]	a[1][1]	a[2][1]	a[3][1]	a[4][1]	a[1]
	a[0][2]	a[1][2]	a[2][2]	a[3][2]	a[4][2]	a[2]

Inizializzazione

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

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

Rappresentazione dei Testi

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

Ad ogni diverso carattere viene assegnato, convenzionalmente, un codice numerico corrispondente. Il programma in C lavora sempre con i codici numerici.

Le funzioni di input/output sono in grado di accettare e mostrare i caratteri corrispondenti

La tabella **ASCII** (American Standard Code for Information Interchange) 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.

Il codice ASCII permette di rappresentare un singolo carattere.

Nelle applicazioni pratiche spesso serve rappresentare sequenze di caratteri: **Stringhe**. Ogni stringa è rappresentata dai codici ASCII dei caratteri di cui è composta.

Stringa

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

Si usano vettori di caratteri. La **lunghezza di una stringa** è tipicamente variabile durante l'esecuzione del programma

Occorrerà **gestire l'occupazione** variabile dei vettori di caratteri.

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' ;

Sintatticamente, i **char** non sono altro che degli **int** di piccola dimensione.

Ogni operazione possibile su un **int**, è anche possibile su un **char**.

Operazioni

Le operazioni lecite sui char derivano direttamente dalla combinazione tra:

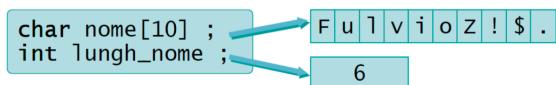
- Le operazioni permesse sugli int
- La disposizione dei caratteri nella tabella ASCII
- Le convenzioni lessicali della nostra lingua scritta

Lunghezza di una Stringa

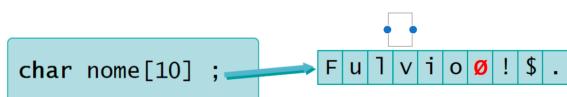
Vi sono due tecniche per determinare la lunghezza di una stringa



1. utilizzare una variabile intera che memorizzi il numero di caratteri validi



2. utilizzare un carattere “speciale”, con funzione di **terminatore**, dopo l’ultimo carattere valido



Il carattere “**terminatore**” deve avere le seguenti Caratteristiche:

- Fare parte della tabella dei codici ASCII
- Non comparire mai nelle stringhe utilizzate dal programma

Per una stringa di N caratteri, serve un vettore di N+1 elementi. Necessario ricordare di aggiungere sempre il terminatore.

Funzioni di lettura e scrittura di stringhe intere:

- **scanf** e **printf**
- **gets** e **puts**

La funzione **gets** è pensata appositamente per acquisire una stringa
Accetta un parametro, che corrisponde al nome di un vettore di caratteri
Esempio:

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

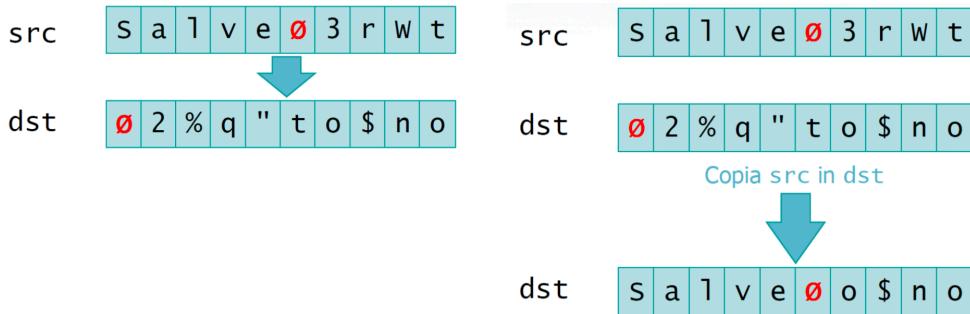
La funzione **puts** è pensata appositamente per stampare una stringa
La variabile da stampare deve essere il nome di un vettore di caratteri
Esempio:

```
printf("Buongiorno, ");
puts(nome);
/* No!! printf("\n"); */
```

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

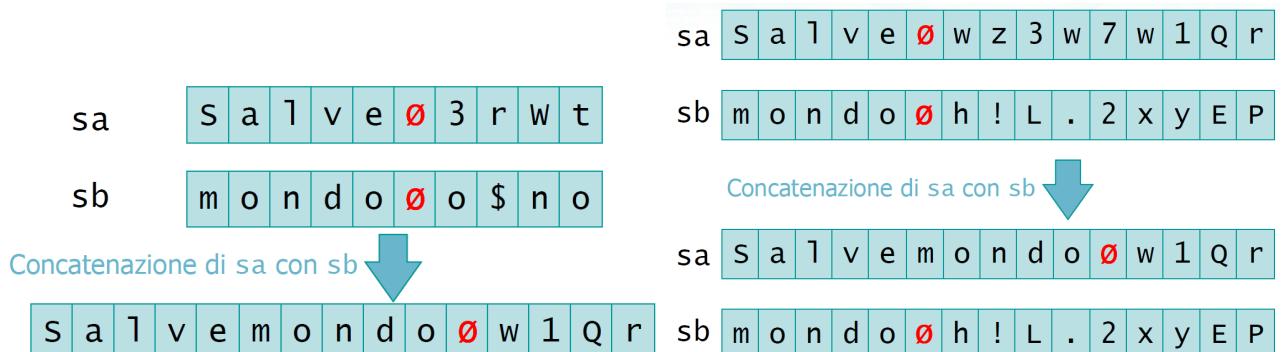
Secondo parametro: stringa sorgente

```
#define MAXS 20  
#define MAXD 30  
  
char src[MAXS+1];  
char dst[MAXD+1];  
int i;  
... /* lettura stringa src */  
for( i=0 ; src[i] != 0 ; i++ )  
    dst[i] = src[i]; /* copia */  
  
dst[i] = '\0'; /* aggiunge terminatore */
```

```
#define MAXS 20  
#define MAXD 30  
  
char src[MAXS+1] ;  
char dst[MAXD+1] ;  
... /* lettura stringa src */  
strcpy(dst, src) ;
```

Concatenazione di Stringhe

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



Nella libreria standard C, includendo `<string.h>`, è disponibile la funzione **strcat**, che effettua la concatenazione di stringhe.

Esempio:

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

Nella prima stringa vi deve essere un numero sufficiente di locazioni libere
 $\text{MAX}+1 \geq \text{strlen}(\text{sa})+\text{strlen}(\text{sb})+1$

Il contenuto precedente della prima stringa viene perso

La seconda stringa non viene modificata

Il terminatore nullo

Deve essere aggiunto in coda alla prima stringa
La **strcat** pensa già autonomamente a farlo

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

```
strcat(sa, sb);  
strcat(sa, sc);
```

È possibile concatenare anche stringhe costanti

```
strcat(sa, "!");
```

Confronto di Stringhe

Il confronto di due stringhe (es.: **sa** e **sb**), mira a determinare se:

- Le due stringhe sono uguali
- Le due stringhe sono diverse
- La stringa **sa** precede la stringa **sb**
- La stringa **sa** segue la stringa **sb**

Nella libreria standard C, includendo `<string.h>`, è disponibile la funzione **strcmp**, che effettua il confronto di stringhe

```
#define MAX 20
```

```
char sa[MAX] ;
```

```
char sb[MAX] ;
```

```
int ris ;
```

```
...
```

```
ris = strcmp(sa, sb) ;
```

```
if(ris<0)
```

```
printf("%s minore di %s\n", sa, sb);
```

```
if(ris==0)
```

```
printf("%s uguale a %s\n", sa, sb);
```

```
if(ris>0)
```

```
printf("%s maggiore di %s\n", sa, 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.

Ricerca in una Stringa

Consideriamo:

s una stringa arbitraria
ch un carattere qualsiasi

Ricerca di un carattere

Nella libreria standard C, includendo <**string.h**>, è disponibile la funzione **strchr**, che effettua la **ricerca** di un **carattere**.

```
#define MAX 10
char s[MAX];
char ch;
...
if(strchr(s, ch)!=NULL)
    printf("%s contiene %c\n", s, ch);
```

Funzioni sulle Stringhe

• Funzioni **isalpha** / **isdigit**

Nome funzione	isalpha
Libreria	#include <ctype.h>
Parametri in ingresso	ch : carattere
Valore restituito	Vero/falso
Descrizione	Restituisce "vero" se il carattere ch è una lettera maiuscola o minuscola (A...Z, a...z), "falso" altrimenti
Esempio	if(isalpha(ch)) { ... }

Nome funzione	isdigit
Libreria	#include <ctype.h>
Parametri in ingresso	ch : carattere
Valore restituito	Vero/falso
Descrizione	Restituisce "vero" se il carattere ch è una cifra numerica (0...9), "falso" altrimenti
Esempio	if(isdigit(ch)) { ... }

• Funzioni isupper / islower

Nome funzione	isupper	Nome funzione	islower
Libreria	#include <ctype.h>	Libreria	#include <ctype.h>
Parametri in ingresso	ch : carattere	Parametri in ingresso	ch : carattere
Valore restituito	Vero/falso	Valore restituito	Vero/falso
Descrizione	Restituisce "vero" se il carattere ch è una lettera maiuscola (A...Z), "falso" altrimenti	Descrizione	Restituisce "vero" se il carattere ch è una lettera minuscola (a...z), "falso" altrimenti
Esempio	if(isupper(ch)) { ... }	Esempio	if(islower(ch)) { ... }

• Funzioni isalnum / isxdigit

Nome funzione	isalnum	Nome funzione	isxdigit
Libreria	#include <ctype.h>	Libreria	#include <ctype.h>
Parametri in ingresso	ch : carattere	Parametri in ingresso	ch : carattere
Valore restituito	Vero/falso	Valore restituito	Vero/falso
Descrizione	Restituisce "vero" se il carattere ch è una lettera oppure una cifra numerica, "falso" altrimenti. Equivalento a <code>isalpha(ch) isdigit(ch)</code>	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(isalnum(ch)) { ... }	Esempio	if(isxdigit(ch)) { ... }

• Funzioni ispunct / isgraph

Nome funzione	ispunct	Nome funzione	isgraph
Libreria	#include <ctype.h>	Libreria	#include <ctype.h>
Parametri in ingresso	ch : carattere	Parametri in ingresso	ch : carattere
Valore restituito	Vero/falso	Valore restituito	Vero/falso
Descrizione	Restituisce "vero" se il carattere ch è un simbolo di punteggiatura (!#\$%&'()*+,.-/:;<=>?@[]^_`{ }~), "falso" altrimenti.	Descrizione	Restituisce "vero" se il carattere ch è un qualsiasi simbolo visibile (lettera, cifra, punteggiatura), "falso" altrimenti.
Esempio	if(ispunct(ch)) { ... }	Esempio	if(isgraph(ch)) { ... }

• Funzioni isprint / isspace

Nome funzione	isprint
Libreria	#include <ctype.h>
Parametri in ingresso	ch : carattere
Valore restituito	Vero/falso
Descrizione	Restituisce "vero" se il carattere ch è un qualsiasi simbolo visibile oppure lo spazio, "falso" altrimenti.
Esempio	if(isprint(ch)) { ... }
Nome funzione	isspace
Libreria	#include <ctype.h>
Parametri in ingresso	ch : carattere
Valore restituito	Vero/falso
Descrizione	Restituisce "vero" se il carattere ch è invisibile (spazio, tab, a capo), "falso" altrimenti.
Esempio	if(isspace(ch)) { ... }

• Funzione iscntrl

Nome funzione	iscntrl
Libreria	#include <ctype.h>
Parametri in ingresso	ch : carattere
Valore restituito	Vero/falso
Descrizione	Restituisce "vero" se ch è un carattere di controllo (ASCII 0...31, 127), "falso" altrimenti.
Esempio	if(iscntrl(ch)) { ... }

• Funzioni toupper / tolower

Nome funzione	toupper
Libreria	#include <ctype.h>
Parametri in ingresso	ch : carattere
Valore restituito	char : carattere maiuscolo
Descrizione	Se ch è una lettera minuscola, ritorna l'equivalente carattere maiuscolo, se no restituisce ch stesso
Esempio	for(i=0; s[i]!=0; i++) s[i] = toupper(s[i]) ;

Nome funzione	tolower
Libreria	#include <ctype.h>
Parametri in ingresso	ch : carattere
Valore restituito	char : carattere maiuscolo
Descrizione	Se ch è una lettera minuscola, ritorna l'equivalente carattere maiuscolo, se no restituisce ch stesso
Esempio	for(i=0; s[i]!=0; i++) s[i] = tolower(s[i]) ;

• Funzioni strcpy / strncpy

Nome funzione	strcpy
Libreria	#include <string.h>
Parametri in ingresso	dst : stringa src : stringa
Valore restituito	nessuno utile
Descrizione	Copia il contenuto della stringa src all'interno della stringa dst (che deve avere lunghezza sufficiente).
Esempio	strcpy(s1, s2) ; strcpy(s, "") ; strcpy(s1, "ciao") ;

Nome funzione	strncpy
Libreria	#include <string.h>
Parametri in ingresso	dst : stringa src : stringa n : numero max caratteri
Valore restituito	nessuno utile
Descrizione	Copia il contenuto della stringa src (massimo n caratteri) all'interno della stringa dst.
Esempio	strncpy(s1, s2, 20) ; strncpy(s1, s2, MAX) ;

• Funzioni strcat / strncat

Nome funzione	strcat
Libreria	#include <string.h>
Parametri in ingresso	dst : stringa src : stringa
Valore restituito	nessuno utile
Descrizione	Accoda il contenuto della stringa src alla fine della stringa dst (che deve avere lunghezza sufficiente).
Esempio	strcat(s1, s2) ; strcat(s1, " ") ;

Nome funzione	strncat
Libreria	#include <string.h>
Parametri in ingresso	dst : stringa src : stringa n : numero max caratteri
Valore restituito	nessuno utile
Descrizione	Accoda il contenuto della stringa src (massimo n caratteri) alla fine della stringa dst.
Esempio	strncat(s1, s2) ;

34

• Funzioni strcmp / strncmp

Nome funzione	strcmp
Libreria	#include <string.h>
Parametri in ingresso	s1 : stringa s2 : stringa
Valore restituito	int : risultato confronto
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) {...} while(strcmp(r, "fine")!=0) {...}

Nome funzione	strncmp
Libreria	#include <string.h>
Parametri in ingresso	s1 : stringa s2 : stringa n : numero max caratteri
Valore restituito	int : risultato confronto
Descrizione	Simile a strcmp, ma confronta solo i primi n caratteri, ignorando i successivi.
Esempio	if(strncmp(r, "buon", 4)==0) (buongiorno, buonasera, buonanotte)

37

• Funzioni atoi / atof

Nome funzione	atoi
Libreria	#include <stdlib.h>
Parametri in ingresso	s : stringa
Valore restituito	int : valore estratto
Descrizione	Analizza la stringa s ed estrae il valore intero in essa contenuto (a partire dai primi caratteri).
Esempio	n = atoi(s) ; n = atoi("232abc") ;

Nome funzione	atof
Libreria	#include <stdlib.h>
Parametri in ingresso	s : stringa
Valore restituito	double/float : valore estratto
Descrizione	Analizza la stringa s ed estrae il valore reale in essa contenuto (a partire dai primi caratteri).
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 **NxM** di caratteri ASCII.

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 vettori di caratteri, cioè **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	\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

I Record

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 sono un 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.

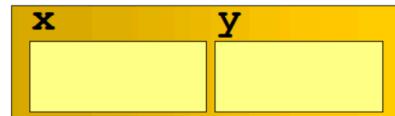
```
struct nome_tag {  
    tipo1 nome_membro1; tipo2 nome_membro2; ...  
};
```

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

Dichiarazioni di **struct** con contenuto identico ma diverso **tag** sono considerate di tipo diverso.

Esempio:

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



dove:

punto è il tag

x e y sono i due membri, scalari di tipo int

Dichiarazioni di Variabili Struct

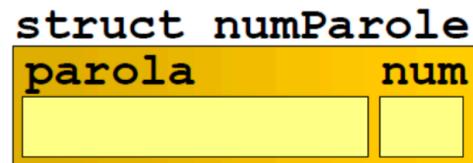
Ha la consueta forma:

tipo var1, var2, var3, ... (salvo che qui il **tipo** è una struct)

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} };
```



Tipo Union

Permette di definire una variabile che può contenere un solo elemento, ma di tipo diverso.

Dichiarazione simile alle strutture, ma un solo membro per volta tra quelli indicati nella dichiarazione può essere in uso

```
union tris {  
    int x;  
    double y;  
    char nome[10];  
} var;
```

Operatori

• Operatore Typedef

Dichiara il nome di un nuovo tipo di dato (in realtà un'abbreviazione) a partire da altri tipi (scalari, aggregati, etc.)

```
typedef tipoEsistente nuovoTipo;
```

La dichiarazione di tipo è identica alla definizione di una variabile, ma è preceduta dalla clausola **typedef**

• Operatore Sizeof

Restituisce il numero di byte di cui è composto un tipo di dato o una variabile (scalare o aggregata).

Sintassi:

```
sizeof(nome_di_tipo)
```

```
sizeof nome_di_variabile
```

I Puntatori

Un **Puntatore** è un tipo variabile che contiene un indirizzo di una locazione di memoria: l'indirizzo di un'altra variabile!

`int* pippo;`

`pippo` è una variabile di tipo **int***,
cioè di tipo **puntatore ad intero**

o anche:

`int *pippo;`

***pippo**

(cioè il valore puntato da pippo)
è una **espressione di tipo intero**

Esempio di come funziona un puntatore

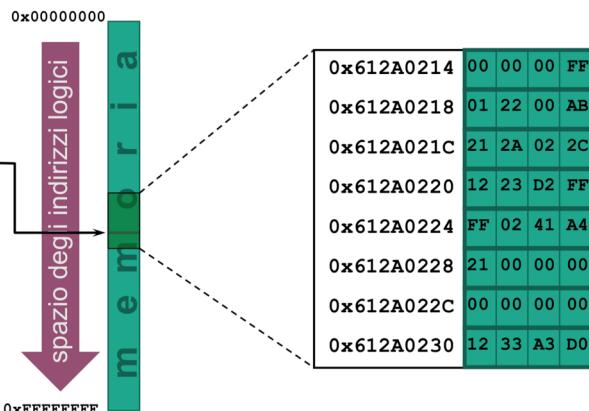
`int pippo;`

il compilatore assegna
alla variabile pippo
una locazione di memoria.

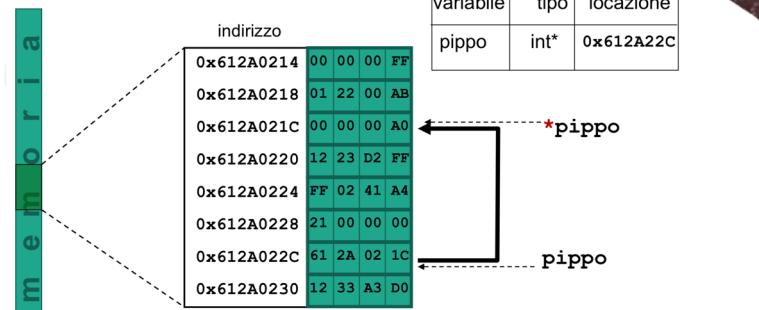
Ad esempio, la
locazione `0x612A22C`

Inoltre, riserva quei
quattro byte per la
variabile pippo.

variabile	tipo	locazione
pippo	int	<code>0x612A22C</code>



`int* pippo;`



`pippo` (il puntatore stesso) vale... `0x612A021C`

`*pippo` (la variabile puntata da pippo) vale... `0x000000A0`

Operazioni sui Puntatori

L'operazione base sui puntatori: somma con un intero

<puntatore ad un tipo T> + <intero>

espressione di tipo puntatore ad un tipo T (T^*)

Esempi:

```
double *p, *q;  
...  
q = p + 3;  
*(p + 3) = 2.0;  
q++;  
q--;  
q+=2;
```

Assegnare i Puntatori

In memoria, un puntatore è un indirizzo di memoria (...di una variabile) (...di cui è noto il tipo)
quale indirizzo?

– **Modo 1:** prendere l'indirizzo di una variabile esistente

• il puntatore punterà a quella variabile

– **Modo 2:** allocare (riservare, prenotare) della memoria libera

- il puntatore punterà ad una nuova variabile, memorizzata nella memoria così riservata.
- la nuova variabile è allocata dinamicamente

Modo 1: prendere l'indirizzo di una variabile esistente il puntatore punterà a quella variabile

Esempio:

Operatore ampersand (&)

il puntatore p
punta
all'indirizzo di
memoria
dove vive la
variabile d

```
double d = 9.0;  
double *p;  
  
p = &d;  
printf("%f", *p);  
*p = 21.5;  
printf("%f", d);
```

Puntatori a Void

Sono puntatori generici e non possono essere dereferenziati (non si può scrivere `*p`), possono essere utilizzati solo come contenitori temporanei di valori di tipo puntatore (a qualsiasi tipo).

`void *h;`

Per dereferenziare il valore di un puntatore a **void** è necessario assegnarlo ad un puntatore al tipo appropriato (non **void**) per poter conoscere la dimensione dell'oggetto puntato

Puntatore Variabile a Valore Costante

`int const *p;` } Sono equivalenti
`const int *p;` }

`p` è una variabile di tipo puntatore-a-costante
(a un oggetto costante di tipo **int**)

Puntatore Costante a Valore Variabile

`int * const p;`

`p` è una costante di tipo puntatore-a-variabile
(a un oggetto variabile di tipo **int**)

Vettori di Puntatori

Gli elementi di questi vettori sono puntatori

`int *vett[10];`
definisce un vettore di 10 puntatori a **int** (le [] hanno priorità maggiore
dell'operatore *)

Esempio di inizializzazione

```
int a, b, c;  
int *vett[10]={NULL};  
vett[0]=&a;  
vett[1]=&b;
```

vett[2] = &c;

I valori da **vett[3]** a **vett[9]** sono tutti NULL in quanto è stato inizializzato il primo elemento (i successivi sono automaticamente a 0 e 0 viene convertito in NULL)

Puntatori a Puntatori

Variabili che contengono l'indirizzo di memoria di un puntatore

Esempio:

```
int a, *b, **c;
```

```
a = 12;
```

```
b = &a;
```

```
c = &b;
```

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 ...

... l'indirizzo del secondo puntatore del vettore **v** (ossia è pari a **&v[1]**, punta a **b**)

Anche **w** è un puntatore ad un puntatore, quando viene incrementato, punta al puntatore successivo, come **v**

Torniamo a come assegnare i vettori...

Ma analizziamo il **Modo 2**:

Modo 2: allocare (riservare, prenotare) della memoria libera

- il puntatore punterà ad una nuova variabile, memorizzata nella memoria così riservata

- la nuova variabile è **allocata dinamicamente**

Allocazione

Esempio

```
int* p;  
p = (int*) malloc(4);
```

con puntatore a Void...

```
void* malloc(unsigned int n);
```

Se non c'è più memoria, l'allocazione fallisce e malloc restituisce il valore speciale **NULL**.

Con i Record...

```
typedef struct {  
    /*blah blah... Campi dell'array...*/  
} NuovoTipo  
  
NuovoTipo * p;  
p = (NuovoTipo *) malloc(sizeof(NuovoTipo));
```

Deallocazione

void free(void* p); libera la memoria che era stata allocata all'indirizzo **p**.

Nota: **p** deve essere il risultato di una **malloc**.

Se si dimentica di deallocare, si ha un cosiddetto **memory leak**.

Allocazione di Vettori

```
int* p;  
p = (int*) malloc(sizeof(int));  
... /* uso *p */  
free(p);
```

```
(void*) calloc(unsigned int n, unsigned int size);
```

calloc = c-contiguous **alloc**-ation

Allocà **n** elementi contigui ciascuno grande **size**

In pratica, alloca un area di memoria grande **n** x

Per il resto funziona come **malloc**

Esempio:

```
int* p;  
(void*) calloc(unsigned int n, unsigned int size);
```

calloc = c-contiguous **alloc**-ation

Allocà **n** elementi contigui ciascuno grande **size**

In pratica, alloca un area di memoria grande **n** x **size**

Per il resto funziona come **malloc**

Esempio:

```
int* p;  
p = (int*)calloc(100000, sizeof(int));
```

Allocà un vettore di 100000 interi.

Differenza 1: dimensione variabile

- se **X** è una variabile intera, si può scrivere:

```
int* v = (int*)calloc(x, sizeof(int));
```

- ma non posso scrivere:

```
int v[x];
```

qui è richiesta una costante

B) vettore allocato dinamicamente:

```
int* v = (int*)calloc(100000,sizeof(int));
```

A) fixed size vector:

```
int v[100000];
```

Differenza 2: bisogna dealloca la memoria

```
int* v = (int*)calloc(100000,sizeof(int));
```

```
... /* usa v */
```

```
free(v);
```

B) vettore allocato dinamicamente:

```
int* v = (int*)calloc(100000,sizeof(int));
```

A) Vettore di dimensione fissa:

```
int v[100000];
```

Differenza 3: dimensione fissa = più efficiente

- il solito prezzo da pagare per l'uso dei puntatori...
- ...maggiorato
- Si supponga che v valga 0xAA000000:

• se fissa v[2]

compilazione →

0xAA000000 + 2 x sizeof(int)
ma precalcolato staticamente

• se dinamico v[2]

compilazione →

READ TEMP 0xAA000000
READ TEMPO 0xAA000000
ADD TEMPO 8
READ TEMP TEMPO

B) vettore allocato dinamicamente:

```
int* v = (int*)calloc(100000,sizeof(int));
```

A) fixed size vector:

```
int v[100000];
```

Differenza 4:

- vengono allocati in zone diverse della memoria...

Esempio

Scrivere un programma che:

- 1) Dichiari un puntatore a puntatore ad intero
- 2) Chieda in input all'utente un intero N
- 3) Usi il puntatore per allocare una matrice di NxN interi
- 4) Ponga nella posizione (i,j) della matrice il valore i+j, Per ogni i,j
- 5) Stampi a video la matrice
- 6) Deallochi la memoria riservata per la matrice

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

int main()
{
    int **Valori = NULL;
    int i,j;
    int N;

    printf("Inserisci la dimensione della matrice:");
    scanf("%d", &N);

    // Allociamo il vettore di righe
    Valori = (int **)malloc(N*sizeof(int));

    // Per ciascuna riga allociamo le colonne
    for(i=0; i<N; i++)
        Valori[i] = (int *)malloc(N*sizeof(int));
    // inizializziamo la matrice con dei valori casuali
    for(i=0; i<N; i++)
        for(j=0; j<N; j++)
            Valori[i][j] = i+j;

    // visualizziamo la matrice
    for(i=0; i<N; i++){
        printf("\n");

        for(j=0; j<N; j++)
            printf("%d ", Valori[i][j]);
    }

    // liberiamo la memoria
    for(i=0; i<N; i++)
        free(Valori[i]);
    free(Valori);
```

```
Inserisci la dimensione della matrice:5
0 1 2 3 4
1 2 3 4 5
2 3 4 5 6
3 4 5 6 7
4 5 6 7 8
```

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 (ad es. calcola la radice quadrata) e a cui è stato dato un nome.

Un programma consta di un **modulo principale** (il **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.

Funzioni

In C i moduli sono Definiti **Funzioni**

Ogni **funzione** è un piccolo programma a sé stante, isolato dalle altre funzioni

All'interno di una funzione possono essere definite delle **variabili locali** (cioè hanno scope locale): le altre funzioni non le “vedono”.

Definizione

```
tipo nomeFunzione (parametri)
{
    definizione_variabili_locali
    istruzioni
    eventuale return
}
```

Corpo della funzione

tipo indica il tipo del valore restituito (es. sqrt restituisce un double)

Se la funzione non restituisce valori (ad esempio StampaCiao visualizza soltanto) bisogna indicare il tipo **void**:

void StampaCiao(....)

Se non si mette nulla viene supposto **int**

Se la funzione non ha parametri (ad esempio StampaCiao), si indica void tra le parentesi:

void StampaCiao(**void**)

Chiamata di una Funzione

Si **chiama una funzione** indicandone il nome seguito da una coppia di parentesi contenenti i valori da elaborare (separati da virgole)

```
eleva(y, 2);
```

Se la funzione non richiede parametri, le parentesi sono vuote, ma devono esserci: **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 di una Funzione

La **funzione** termina (cioè l'esecuzione torna al modulo chiamante) quando viene eseguita l'istruzione **return risultato**;

Una funzione può avere più istruzioni **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**, non si deve indicare risultato, inoltre la **return** che precede la graffa di chiusura (solo questa) può essere **omessa**.

I valori delle variabili locali vengono persi.

Nel **main** la **return** termina il programma.

Per terminare un programma dall'interno di una funzione (e passare lo status al Sistema Operativo) si utilizza la funzione

```
exit(status)
```

dichiarata in **<stdlib.h>**:

```
exit(EXIT_SUCCESS);
```

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 un **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 quindi usare **f2**, **f2** vede **f1**, il **main** vede **f1** e **f2**

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 quindi essere collocato:

- in un punto successivo a dove viene chiamata (nell'esempio seguente, 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 es. stdio.h contiene i prototipi di scanf, printf, getchar, etc.)

Esempio

```
#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;
}
```

Il **prototipo** di una funzione è simile alla definizione della funzione, salvo che:

- manca il corpo
- i nomi dei parametri possono essere omessi (ma i tipi devono essere presenti!)
- ha un punto e virgola alla fine **int eleva(int, int);**

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 a identificatori diversi)
- sono utili per descrivere il significato dei parametri: **int eleva(int base, int esponente);**

Tipi di Parametri

- **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 o espressioni) indicati tra le parentesi alla chiamata di una funzione **eleva(x,2)**

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 ) ;  
}
```

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 (perdendone il valore)

Le variabili locali di **classe di allocazione statica** invece non vengono mai rimosse dalla memoria per cui non perdono il loro valore quando la funzione termina (resta quello che aveva al termine della chiamata precedente)

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 viene automaticamente convertito in NULL)

Esempio

```
char *nomeMese (int n)
{
    static char *nome[] = {"inesistente", "gennaio", "febbraio",
                          ecc...};

    if (n<1 || n>12)
        return nome[0];
    else
        return nome[n];
}
```

Variabili Esterne

Vengono definite (riservando memoria) esternamente al corpo delle funzioni:

- in testa al file, tipicamente dopo le direttive **#include** e **#define**
- oppure tra una funzione e un’altra Sono **visibili** e **condivisibili** da tutte le funzioni che nello stesso file seguono la definizione

Hanno classe di allocazione statica quindi non vengono mai rimosse dalla memoria.

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;
```

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 by- passerebbe la restrizione.

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 ***

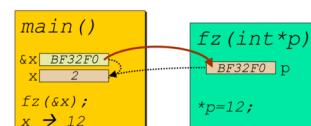
Passaggio dei Parametri (by Reference)

Il passaggio di 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.

```
#include <stdio.h>  
  
void fz(int *);      →     prototipo  
  
int main()  
{  
    int x=2;  
  
    fz(&x);  
    printf("%d\n", x);  
  
    return EXIT_SUCCESS;  
}  
  
void fz(int *p)  
{  
    *p = 12;  
}
```

Nell'esempio:
il main alloca x, gli assegna il valore 2 e chiama fz passandole l'indirizzo di x (&x) presente in una variabile temporanea ("senza nome")
alla chiamata di fz, l'indirizzo di x (che è BF32F0) viene copiato by-value in p
fz accede a x come *p, modificandola in 12
quando fz termina, x vale 12



Passaggio di Vettori

Possono essere passati vettori con dimensioni diverse, ma dello stesso tipo **T**.

Quando si passa un **vettore-di-T**, poiché si indica il nome del vettore, in realtà si passa l'indirizzo di memoria del suo primo elemento.

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 ininfluente)

```
void funz(int matrice[][10])
```

La funzione deve conoscere in qualche modo le dimensioni della matrice (indicarle tra le parentesi quadre non serve a questo scopo)

Poiché 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

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])
```

E' quindi un errore scrivere:

```
void funz (int **matrice)
```

oltre all'errore di tipo, si perde la dimensione delle colonne e quindi non si può determinare la posizione degli elementi della matrice.

Nel Dettaglio...

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,

Inoltre il **Sistema Operativo** alloca spazio di memoria per:

- il **code segment** (CS)
- il **data segment** (DS)
- lo **stack** e lo **heap** (condiviso)

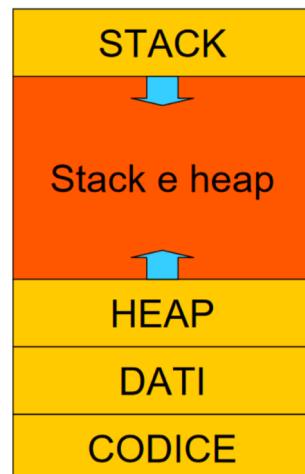
Lo **stack** contiene inizialmente le variabili locali della funzione **main()**.

Lo **heap** inizialmente è vuoto e serve per contenere i blocchi di memoria allocati dinamicamente con funzioni **malloc()** (trattate in altre slide).

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 la chiamata

Nell'**Activation Record** viene anche memorizzato l'indirizzo di ritorno dalla funzione: la locazione di memoria che contiene l'istruzione del chiamante da cui continuare dopo che la funzione è terminata



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.

Puntatori a Funzioni

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 a una **funzione**, restituito da una **funzione**.

Definizione di Variabili

- **tipo (*nome) (parametri) ;**

definisce la variabile **nome** come puntatore a una funzione che restituisce un valore del **tipo** indicato e richiede i **parametri** indicati

- **double (*fp) () ;**
definisce **fp** come variabile puntatore a 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 a una funzione che ha come parametri due **double** e restituisce un **double**.
- **int (*fpv[3]) (long) ;** definisce **fpv** come vettore di 3 elementi, ciascuno è un puntatore a una funzione che ha come parametri un **long** e restituisce un **int**.

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)**

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 dalla stessa funzione comportamenti diversi.

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) effettua la somma di 2 numeri interi
int meno(int a,int b) effettua la differenza di 2 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
    ...
}
```

Ordinamenti

Quello di ordinare in modo crescente o decrescente dei numeri, o delle parole in ordine alfabetico diretto o inverso, è uno dei problemi più frequenti della programmazione.

Per ordinare dei dati esistono due tecniche principali, dette rispettivamente **ordinamenti interni** e **ordinamenti esterni**.

- Gli *ordinamenti interni* si usano quando la lista dei dati non è troppo lunga e può essere memorizzata per intero nella memoria del computer, di solito in un vettore.
- Gli *ordinamenti esterni* 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.

Classificazione Algoritmi di Ordinamento

Si puo procedere alla classificazione di algoritmi di ordinamento, considerando:

-**La Complessità**, notazione **O** grande per indicare la complessità, notiamo che

- un algoritmo **O(lg n)** può impiegare 20 microsecondi
- un algoritmo **O(n1.25)** può impiegare 33 secondi
- un algoritmo **O(n2)** può impiegare fino a 12 giorni.

-**L'efficienza**, un'altra classificazione degli algoritmi di ordinamento si basa sulla loro efficienza o economia di tempo. Una buona misura dell'efficienza si ottiene contando numeri di confronti tra chiavi e di movimenti (trasposizioni) necessari per il riordino.

Metodi di Ordinamento

Metodi diretti:

- Tecniche semplici ed ovvie
- Adatti ad illustrare le caratteristiche dei maggiori principi di ordinamento
- Facili da capire e da realizzare

Metodi avanzati:

- Tecniche più complesse e meno intuitive
- Richiedono, generalmente, un numero di confronti nell'ordine di $n \cdot \log n$

Alcuni dei **metodi diretti** sono:

–Ordinamento per selezione

- Altresì detto ordinamento per minimi (o massimi) successivi

–Ordinamento per inserimento

- Basato sul concetto dell'*inserzione ordinata*

–Ordinamento per scambi

- Altresì noto come *bubble sort*

Ordinamento per Selezione (*Selection Sort*)

Si applica al caso di una lista di **n** elementi memorizzati in un array.

Ha per obiettivo il riordinamento degli elementi mediante spostamento fisico

Algoritmo

(Prende il numero più piccolo e lo sostituisce con il primo valore del vettore, prende il secondo numero più piccolo e lo sostituisce con il secondo valore del vettore e così via... fino ad ordinare il vettore.)

- Prende in esame la generica sotto-lista **a[i]...a[n]**
- Determina l'elemento minimo **min** e la sua posizione **jmin**
- Esegue lo scambio tra **a[i]** ed **a[jmin]**
- Il procedimento viene ripetuto per le successive sotto-liste, cioè per $i=1..n-1$

```
#include <stdio.h>

void Swap(int *a, int *b);
void SelectSort(int A[], int n);

int V[] = {72, 23, 12, 5, 0};

int main(int argc, char *argv[])
{
    int i=0;
    int n=5;

    printf("Vettore non ordinato: ");
    for(i=0; i<n; i++)
        printf("%d ", V[i]);

    SelectSort(V, n);

    printf("Vettore ordinato: ");
    for(i=0; i<n; i++)
        printf("%d ", V[i]);
}

return 0;
}
```

```
void SelectSort(int A[], int n)
{
    int i, j, min, t;

    for(i=0; i<n; i++){
        min = i;

        for (j = i; j < n; j++)
            if (A[j] < A[min])
                min = j;

        Swap(&A[min], &A[i]);
    }
}

void Swap(int *a, int *b){

    int temp;

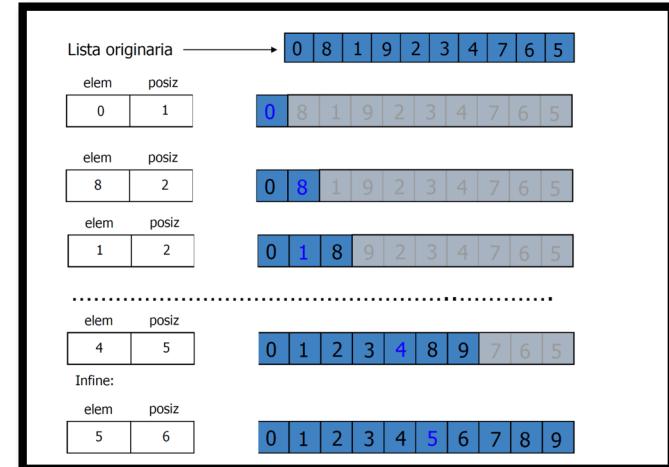
    temp = *a;
    *a = *b;
    *b = temp;
}
```

Ordinamento per Inserimento (*Insertion Sort*)

Si applica al caso di una lista di **n** elementi memorizzati in un array

- Ha per obiettivo il riordinamento degli elementi mediante spostamento fisico
- L'algoritmo utilizzato si basa sull'operazione di **push** di un elemento **elem** in un vettore (utilizzata per costruire liste ordinate):

- Ricerca della posizione **i** del vettore ove inserire **elem** (primo $V[i] > elem$)
- Spostamento in avanti di un posto di tutti gli elementi $V[j]$, $j=i..nelem$, **nelem** numero di elementi presenti in **V**
- Incremento di **nelem** di un'unità
- Inserzione di **elem** in **V**



Consideriamo Array $a[0]..a[n-1]$

Algoritmo

(prende in considerazione il secondo elemento dell'array e, se è minore del primo, lo scambia con esso poi considera il terzo elemento e lo inserisce nella posizione corretta rispetto ai primi due elementi e così via...)

Caso a) → Inserzione ordinata di elementi:

Si inserisce $a[0]$

Per $i=1..n-1$

{

-si cerca il primo elemento $a[j] > a[i]$ nella sottolista $a[0]..a[i-1]$ l'elemento **nuovo** va aggiunto in posizione j

-gli elementi $a[i-1]..a[j]$ vanno spostati a destra di una posizione

-l'elemento nuovo va assegnato ad $a[j]$: $a[i] \rightarrow a[j]$

}

Algoritmo Caso b) →

Riordinamento di un vettore preesistente

Come nel caso precedente...

-Al ciclo i -mo viene prelevato $a[i]$

- $a[i]$ va posizionato al posto giusto in $a[0]..a[i]$

-gli elementi ancora da ordinare restano nella posizione iniziale

```
#include <stdio.h>
void InsertionSort(int A[], int n);
int main(int argc, char *argv[])
{
    int i=0;
    int n=5;

    printf("Vettore non ordinato: ");
    for(i=0; i<n; i++)
        printf("%d ", V[i]);

    InsertionSort(V, n);

    printf("Vettore ordinato: ");
    for(i=0; i<n; i++)
        printf("%d ", V[i]);

    return 0;
}
```

```
void InsertionSort(int A[], int n) {
    int i,j,k;
    int temp;

    for(i=1;i<n;i++) {
        temp=A[i]; // Salva V[i] in temp

        // Ricerca il I elemento > di V[i] in V[0]..V[i-1]
        j=0;
        while((A[j] < A[i]) && (j < i))
            j++;

        // V[i] va inserito in posizione j...
        // Prima, sposta V[j]..V[i-1] in avanti
        for (k=i-1;k>=j;k--)
            A[k+1]=A[k];

        // Inserisci temp in V[j]
        A[j]=temp;
    }
}
```

Ordinamento per Scambi (*Bubble Sort*)

Si applica al caso di una lista di **n** elementi memorizzati in un array

Ha per obiettivo il riordinamento degli elementi mediante spostamento fisico

L'algoritmo utilizzato:

– Prevede l'ordinamento attraverso scambi tra coppie successive

- Si esamina la prima coppia (**a[1],a[2]**) e si esegue uno scambio nel caso non sia ordinata

- Si prosegue in maniera analoga con la coppia successiva (**a[2],a[3]**) e così via fino alla coppia (**a[n-1],a[n]**)

- Al termine del ciclo, l'elemento massimo occupa la posizione **n**, alla quale sarà pervenuto attraverso un certo numero di scambi (da qui il paragone con la bolla che **gorgoglia** verso l'alto)

– Il procedimento viene ripetuto con riferimento alla sotto-lista **a[1]...a[n-1]** e così via fino a pervenire alla sotto-lista **a[1],a[2]**

L'ultimo scambio effettuato in un ciclo determina la posizione a partire dalla quale la lista è ordinata:

(1)

Lista originaria → [9 | 8 | 4 | 5 | 6 | 2 | 3 | 7 | 1 | 0]

Ciclo # 1:

coppia relazione scambio
1 9>8 $a_1 \leftrightarrow a_2$ [8 | 9 | 4 | 5 | 6 | 2 | 3 | 7 | 1 | 0]

coppia relazione scambio
2 9>4 $a_2 \leftrightarrow a_3$ [8 | 4 | 9 | 5 | 6 | 2 | 3 | 7 | 1 | 0]

coppia relazione scambio
9 9>0 $a_9 \leftrightarrow a_{10}$ [8 | 4 | 5 | 6 | 2 | 3 | 7 | 1 | 0 | 9]

(3)

Ciclo # 4, al termine:

coppia relazione scambio
[] [] $a_6 \leftrightarrow a_7$ [4 | 2 | 3 | 5 | 1 | 0 | 6 | 7 | 8 | 9]

Ciclo # 8, al termine:

coppia relazione scambio
[] [] $a_1 \leftrightarrow a_2$ [1 | 0 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9]

Ciclo # 9

coppia relazione scambio
[] [] $a_1 \leftrightarrow a_2$ [0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9]

(2)

Ciclo # 2:

coppia relazione scambio
1 8>4 $a_1 \leftrightarrow a_2$ [4 | 8 | 5 | 6 | 2 | 3 | 7 | 1 | 0 | 9]

coppia relazione scambio
8 8>0 $a_8 \leftrightarrow a_9$ [4 | 5 | 6 | 2 | 3 | 7 | 1 | 0 | 8 | 9]

Ciclo # 3:

coppia relazione scambio
3 6>2 $a_3 \leftrightarrow a_4$ [4 | 5 | 2 | 6 | 3 | 7 | 1 | 0 | 8 | 9]

coppia relazione scambio
4 6>3 $a_4 \leftrightarrow a_5$ [4 | 5 | 2 | 3 | 6 | 7 | 1 | 0 | 8 | 9]

coppia relazione scambio
6 7>1 $a_6 \leftrightarrow a_7$ [4 | 5 | 2 | 3 | 6 | 1 | 7 | 0 | 8 | 9]

coppia relazione scambio
7 7>0 $a_7 \leftrightarrow a_8$ [4 | 5 | 2 | 3 | 6 | 1 | 0 | 7 | 8 | 9]

Consideriamo Array a[0]..a[n-1]

Algoritmo

(consiste nell'effettuare diverse passate lungo l'array. A ogni passata vengono confrontate le successive coppie di elementi (l'elemento 0 e l'elemento 1, poi l'elemento 1 e l'elemento 2, ecc.). Se una coppia è in ordine crescente (o se i valori sono identici), si lasciano i valori come sono. Se una coppia è in ordine decrescente, i valori vengono scambiati nell'array.)

Per $i=1..n-1$ {

-si considera la sottolista $a[0]..a[n-i]$

-si effettua una scansione della lista dal primo all'ultimo elemento

-se un elemento precede un elemento <, lo si scambia col successivo

}

```
void BubbleSort(int V[], int n) {

    int i,j;
    int nswaps=0; // numero di scambi
    int ncycles=0; // numero di iterazioni
    boolean scambi=TRUE; // TRUE se è stato effettuato almeno
    i=1; // uno scambio
    while((i<=n-1) && (scambi==TRUE)) {
        scambi=FALSE;
        ncycles++;
        for(j=0;j<n-i;j++) { // Analizza il sottoinsieme V[0]..V[n-i]
            if(V[j]>V[j+1]) {
                nswaps++;
                swap(&V[j], &V[j+1]);
                scambi=TRUE;
            }
        }
        i++;
    }
    printf("Sono stati eseguiti %d scambi\n", nswaps);
    printf("Sono stati eseguiti %d scambi\n", ncycles);
}
```

Le Liste

Definire tipi ricorsivi:

```
typedef struct Persona {  
    char nome[20];  
    char cognome[20];  
    int peso;  
    struct Persona *padre;  
};  
struct Persona a, b;  
a.padre = &b;
```

Una **lista** è una **struttura dati ricorsiva** (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.

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

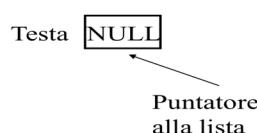
```
{  
    int info;  
    elem * succ;  
};
```

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**.

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.

Es.:
`Lista_SL Testa;
Testa.next = 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.

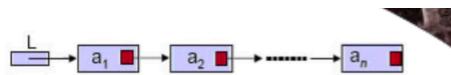
Queste liste possono aumentare e diminuire di dimensioni dinamicamente in base al flusso del programma, e quindi devono essere gestite in un modo più efficiente dell'allocazione di memoria permanente sotto forma di array

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.

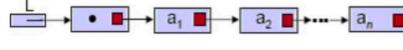
Se la struttura possiede un solo membro puntatore a se stessa, la lista è detta **single-linked** (o **monodirezionale**), se ne possiede due, è detta **double-linked** (o **bidirezionale**).

Lista monodirezionale (può essere visitata in un solo senso -> presenza di un puntatore per nodo)



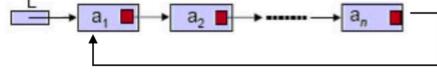
Lista monodirezionale con sentinella

(ha una cella in più detta **sentinella**, che è direttamente indirizzata da L, ma non contiene i valori di alcun elemento).

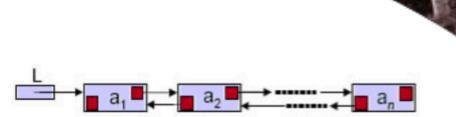


Lista monodirezionale circolare

(l'ultimo puntatore non ha valore **NULL**, ma punta al primo nodo)

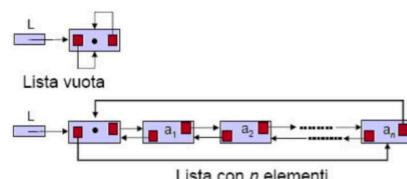


Lista bidirezionale (può essere visitata nei due sensi -> presenza di due puntatori per nodo)



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 sulle 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

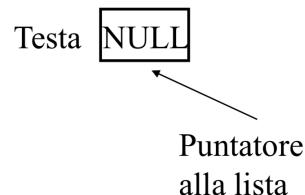
Per creare una lista, basta definirla, ovvero è sufficiente creare il modo di riferirsi ad essa.

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** poiché all'inizio (creazione della lista) non punta a niente in quanto non ci sono elementi.

Es.:

```
Lista_SL Testa;  
Testa.next = NULL;
```

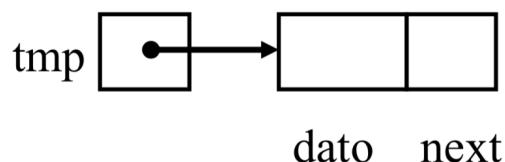


Creazione di un Nuovo Nodo

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**)

Es.:

```
Nodo_SL *tmp = (Nodo_SL *)malloc(sizeof(Nodo_SL));
```



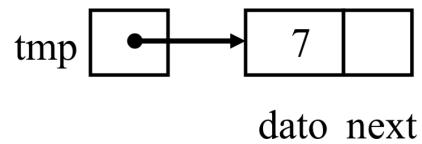
Assegnazione di Valori ai campi dati

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;
}
```



Inserimento di un Elemento

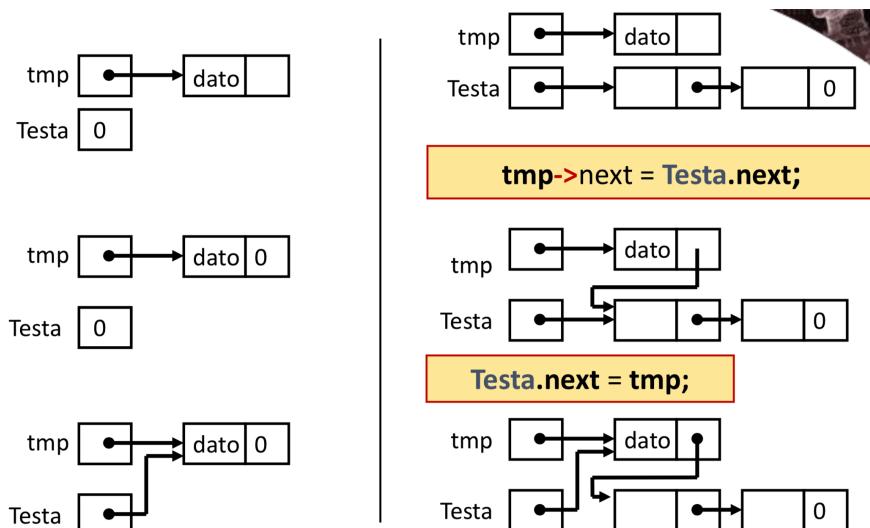
- Le operazioni di inserimento di un elemento (ed analogamente quelle di cancellazione) possono avvenire secondo diverse modalità, (ovvero in diverse posizioni della lista). In ogni caso l'inserimento di un nuovo elemento nella lista prevede sempre i seguenti passi:
 - 1) Creazione di un nuovo nodo (allocazione dinamica)
 - 2) Assegnazione di valori ai campi dati
 - 3) Collegamento del nuovo elemento alla lista esistente

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!

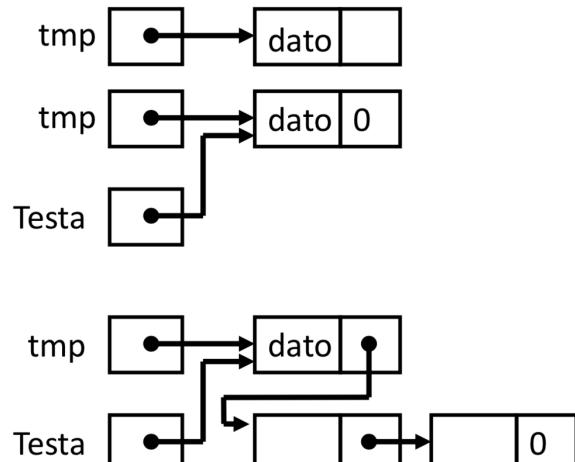


```
void InserisciInTesta_SL(Lista_SL *Testa, int dato)
{
    Nodo_SL *temp = NULL;

    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;
}
```



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.

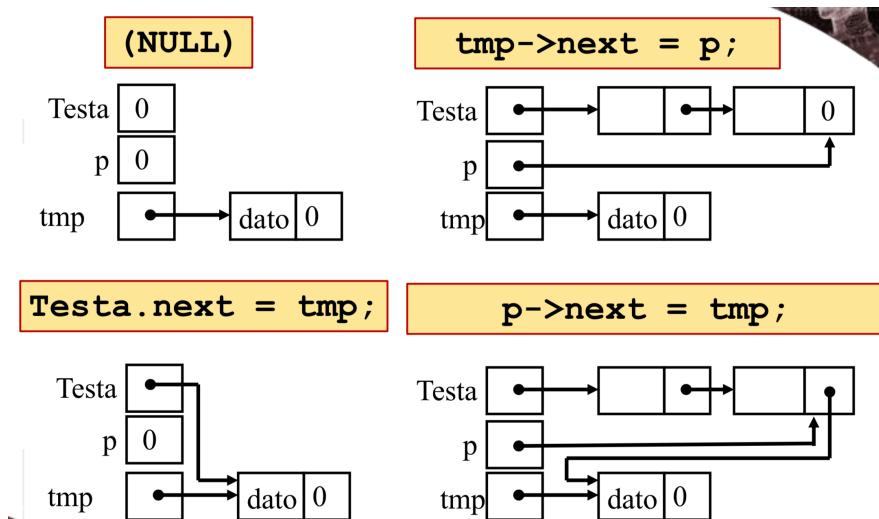
Supponiamo di averlo trovato e che sia il puntatore **p**: Il campo **next** del nuovo nodo punterà a **NULL** (in quanto è l'ultimo)

Il campo **next** dell'ex ultimo nodo punterà al nuovo nodo

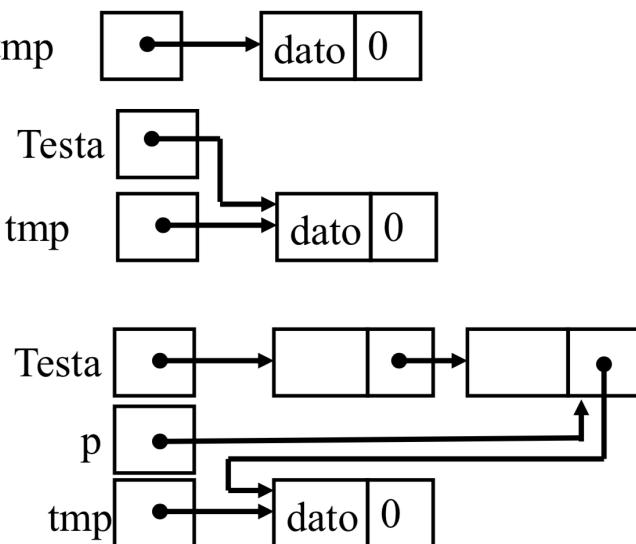
Es.:

```
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

Il **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);
}
```

Liste Semplicemente Concatenate

Ricerca di un elemento qualsiasi nella Lista

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.

Es:

```
Nodo_SL *q = Testa.next;
Nodo_SL *p = Testa.next;
while (q !=NULL && q->dato<dato) {
    p=q;
    q=q->next;
}
```

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 in Lista

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 {
        nuovo = CreaNodo_SL(dato);
        temp = CercaElemento_SL(*Testa, predecessore);

        if(tmp){
            nuovo->next = temp->next;
            temp->next = nuovo;
        }
    }

    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.

Bisogna aggiornare il puntatore alla testa **Testa.next** che dovrà puntare al nodo successivo a quello da eliminare.

-salvataggio del nodo da eliminare:

Node_SL *tmp = Testa . next;

-aggiornamento della lista:

Testa . next = tmp->next;

-distruzione del nodo:

free (tmp) ;

Liste Doppiaamente Concatenate

Le **Liste Doppiaamente Concatenate** estendono la rappresentazione delle liste concatenate, introducendo un puntatore al nodo precedente.

A differenza delle liste concatenate semplici, le liste doppiaamente concatenate possono essere attraversate facilmente in entrambe le direzioni.

Le implementazioni di alcune funzioni sono semplificate dall'introduzione del nuovo puntatore.

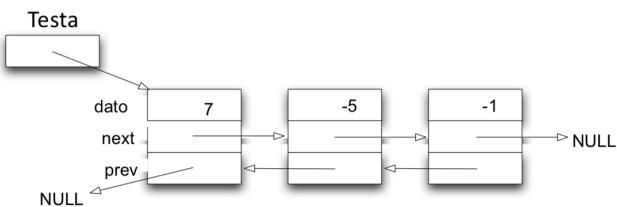
Possiamo rappresentare il tipo di dato lista come puntatore a struttura contenente:

- 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;
```

Da un nodo è possibile accedere sia al nodo successivo (campo next) che al nodo precedente (campo prev) nella lista



Struttura della lista:

```
typedef struct Lista_DL {  
    Nodo_DL *next;  
} Lista_DL;
```

Creazione di una Lista

Per creare una lista, basta definirla, ovvero è sufficiente creare il modo di riferirsi ad essa.

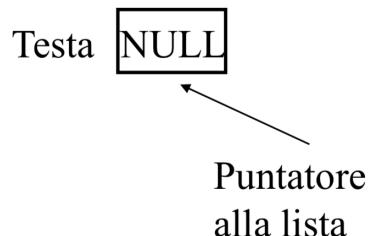
L'unica cosa che esiste sempre della lista è la sua **testa** (o **radice**) ossia il puntatore al suo primo elemento.

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.

Es.:

```
Lista_DL Testa;
```

```
Testa.next = NULL;
```

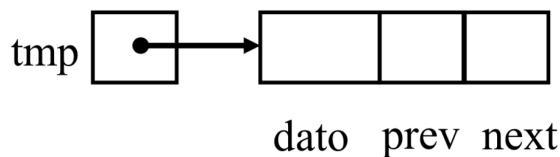


Creazione di un Nuovo Nodo

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**)

Es.:

```
Nodo_DL *tmp = (Nodo_DL *)malloc(sizeof(Nodo_DL));
```



Creazione Nodo

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 liste concatenate. Il nodo di una lista concatenata contiene il puntatore al nodo precedente, oltre al puntatore 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**.
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

```
void InserisciInTesta_DL(Lista_DL *Testa, int dato)
{
    Nodo_DL *temp = NULL;

    if(!Testa->next){
        temp = CreaNodo_DL(dato);

        if(temp){
            Testa->next = temp;
            return;
        }
    } else {
        temp = CreaNodo_DL(dato);

        if(temp){
            temp->next = Testa->next;
            Testa->next->prev = temp;
            Testa->next = temp;
        }
    }

    return;
}
```

Inserimento in Coda

Se la lista è vuota, si inserisce il nuovo nodo come unico nodo puntato da **Testa.next**.
Se la lista non è vuota, troviamo il nodo di coda scorrendo la lista a partire da **Testa.next**.

```
void InserisciInCoda_DL(Lista_DL *Testa, int dato)
{
    Nodo_DL *temp = NULL;
    Nodo_DL *q = Testa->next;

    if(!Testa->next){
        temp = CreaNodo_DL(dato);

        if(temp){
            Testa->next = temp;
            return;
        }
    } else {
        temp = CreaNodo_DL(dato);

        while(q->next)
            q = q->next;

        q->next = temp;
        temp->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.

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 *temp = NULL;
    Nodo_DL *nuovo = NULL;

    if(Testa->next == NULL)
        return;
    else {
        temp = CercaElemento_DL(*Testa, predecessore);

        if(temp){
            nuovo = CreaNodo_DL(dato);
            nuovo->prev = temp;
            temp->next->prev = nuovo;
            temp->next = nuovo;
            nuovo->next = temp->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.

Bisogna aggiornare il puntatore alla testa **Testa.next** che dovrà puntare al nodo successivo a quello da eliminare.

salvataggio del nodo da eliminare:

Node_DL *tmp = Testa . next;

aggiornamento della lista:

Testa . next = tmp->next; tmp->next->prev = NULL;

distruzione del nodo:

free (tmp) ;

• Eliminazione del Nodo in Testa

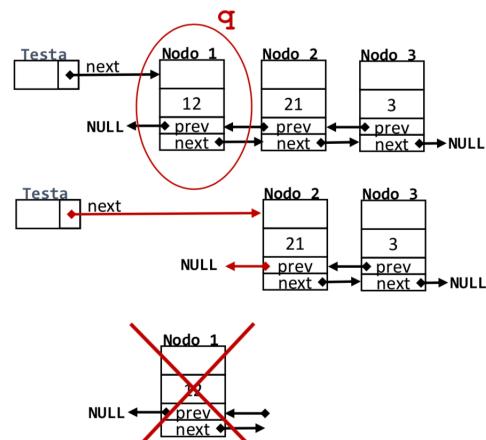
```
void CancellaInTesta_DL(Lista_DL *Testa)
{
    Nodo_DL *q = Testa->next;

    if(!q)
        return;
    else {

        if(q->next){
            Testa->next = q->next;
            Testa->next->prev = NULL;
        }else

            Testa->next = NULL;

        free(q);
    }
}
```



• Eliminazione del Nodo in Coda

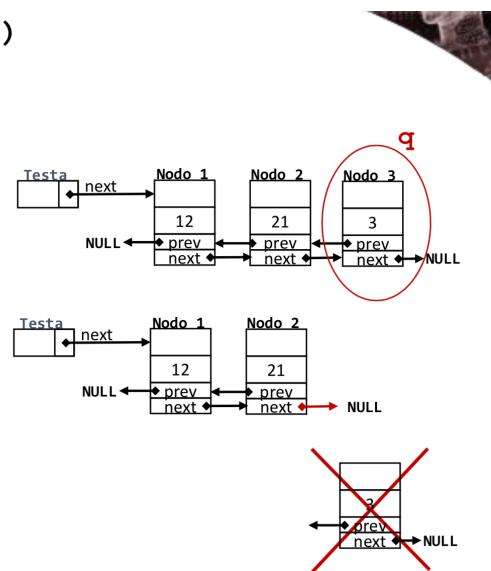
```
void CancellaInCoda_DL(Lista_DL *Testa)
{
    Nodo_DL *q = Testa->next;

    if(!q)
        return;
    else {

        while(q!=NULL && q->next!=NULL)
            q=q->next;

        if(q->prev)
            q->prev->next = NULL;

        free(q);
    }
}
```

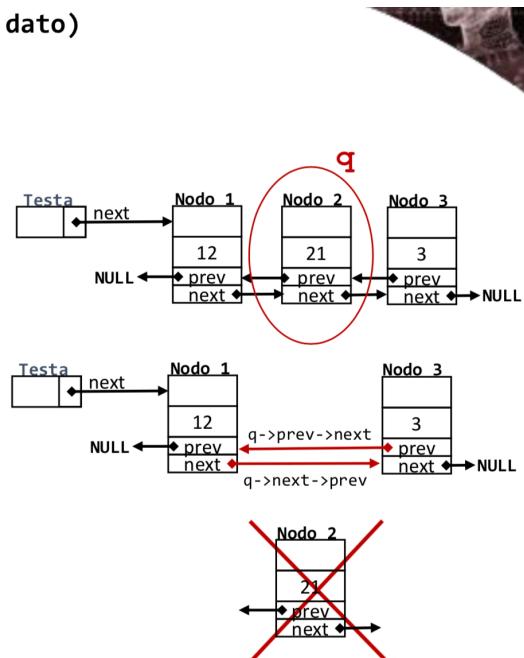


• Eliminazione di un nodo Generico

```
void CancellaElemento_DL(Lista_DL *Testa, int dato)
{
    Nodo_DL *temp = NULL;

    if(Testa->next == NULL)
        return;
    else {
        temp = CercaElemento_DL(*Testa, dato);

        if(temp){
            if(temp->next==NULL)
                CancellaInCoda_DL(Testa);
            else if(temp->prev==NULL)
                CancellaInTesta_DL(Testa);
            else {
                temp->prev->next = temp->next;
                temp->next->prev = temp->prev;
                free(temp);
            }
        }
    }
    return;
}
```



La Ricorsione

Una funzione matematica è definita **ricorsivamente** quando nella sua definizione compare un riferimento a se stessa

La **Ricorsione** consiste nella possibilità di definire una funzione mediante se stessa
È basata sul principio di **induzione matematica**:

- se una **proprietà P** vale per $n=n_0$ (**CASO BASE**)
e si può provare che, assumendola valida per n , allora vale per $n+1$, allora **P** vale per ogni $n \geq n_0$

Operativamente, risolvere un problema con un approccio ricorsivo comporta:

-identificare un **caso base** ($n=n_0$) in cui la soluzione sia nota
-riuscire a esprimere la soluzione nel caso generico n in termini dello stesso problema in uno o più casi più semplici ($n-1$, $n-2$, etc).

Esempio: il fattoriale di un numero naturale

fact(n) = n!

$n! : N \rightarrow N$

$n! \text{ vale } 1 \text{ se } n == 0$

$n! \text{ vale } n * (n-1)! \text{ se } n > 0$

Funzioni Ricorsive

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);
}
```

Servitore & Cliente: **fact** è sia servitore che cliente (di se stessa)

```
main() {
    int fz,f6,z = 5;
    fz = fact(z-2);
}
```

Vediamo il Codice nei Dettagli...

1)

```
int fact(int n) {  
    if(n<=0)  
        return 1;  
    else  
        return n*fact(n-1);  
}
```

Si valuta l'espressione che costituisce il parametro attuale (nell'environment del main) e si trasmette alla funzione fact una copia del valore così ottenuto (3).

```
main() {  
    int fz,f6,z = 5;  
    fz = fact(z-2);  
}
```

2)

```
int fact(int n) {  
    if(n<=0)  
        return 1;  
    else  
        return n*fact(n-1);  
}
```

La funzione fact lega il parametro n a 3. Essendo 3 positivo si passa al ramo else. Per calcolare il risultato della funzione è necessario effettuare una nuova chiamata di funzione fact(2)

```
main() {  
    int fz,f6,z = 5;  
    fz = fact(z-2);  
}
```

Per calcolare il risultato della funzione è necessario effettuare una nuova chiamata di funzione; n-1 nell'environment 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'environment di fact vale 0 quindi viene chiamata fact(0)

3)

```
int fact(int n) {
    if(n<=0)
        return 1;
    else
        return n*fact(n-1);
}
```

```
main() {
    int fz,f6,z = 5;
    fz = fact(z-2);
}
```

Il controllo passa al **main** che assegna a **fz** il valore 6

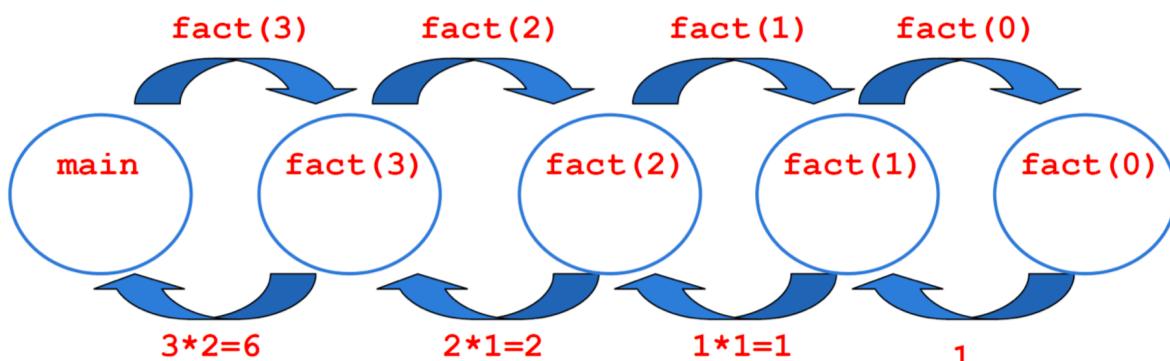
Il nuovo servitore lega il parametro n a 0. La condizione $n \leq 0$ è vera e la funzione **fact(0)** restituisce come risultato 1 e termina.

Il controllo torna al servitore precedente **fact(1)** che può valutare l'espressione $n * 1$ (valutando n nel suo environment 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 environment 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 environment dove vale 3) ottenendo come risultato 6 e terminando.

4)



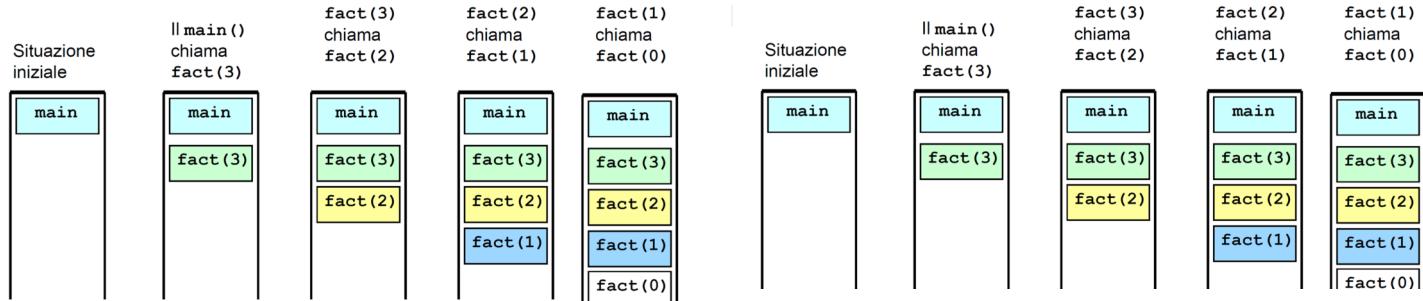
main fact(3) fact(2) fact(1) fact(0)

Cliente di fact(3)	Cliente di fact(2)	Cliente di fact(1)	Cliente di fact(0)	Servitore di fact(1)
Servitore del main	Servitore di fact(3)	Servitore di fact(2)	Servitore di fact(1)	

Cosa Succede Nello Stack?

(1)

Seguiamo l'evoluzione dello stack durante l'esecuzione



(2)

Seguiamo l'evoluzione dello stack durante l'esecuzione

(3)

Seguiamo l'evoluzione dello stack durante l'esecuzione

`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`.

