FONDAMENTI DI INFORMATICA

Prof. Valerio Persico – A.A. 2022/23

INDICE DEGLI ARGOMENTI

RAPPRESENTAZIONE DELL'INFORMAZIONE

1. L'INFORMATICA (p.3)
2. CODIFICA (p.4)
3. RAPPRESENTAZIONE DIGITALE E IL CODICE BINARIO (p.5)
4. SISTEMA DI NUMERAZIONE BINARIO (p.8)
5. CODIFICA DEI NUMERI REALI (p.12)
6. RAPPRESENTAZIONE DEI TESTI (p.16)
7. I DATI MULTIMEDIALI (p.17)

ARCHITETTURA

8. OPERATORI BOOLEANI (p.19)
9. GLI ALGORITMI E GLI ESECUTORI (p.20)
10. IL MODELLO DI VON NEUMANN (p.21)
11. LA MEMORIA (p.22)
12. LA CPU (p.23)
13. I BUS E IL CLOCK (p.26)
14. LE ISTRUZIONI (p.26)
15. INTERRUZIONI E CACHE, EVOLUZIONE DEL MODELLO DI VON NEUMANN (p.28)
16. FIRMWARE E SOFTWARE (p.29)

PROGRAMMAZIONE

17. LA PROGRAMMAZIONE E I LINGUAGGI DI PROGRAMMAZIONE (p.31)
18. IL LINGUAGGIO C/C++ (p.31)
19. REGOLE LESSICALI E SINTATTICHE (p.32)
20. GESTIONE DEGLI ERRORI (p.32)
21. STRUTTURA E QUALITÀ DEI PROGRAMMI (p.33)
22. VARIABILI E IDENTIFICATORI (p.34)
23. ARITMETICA IN C/C++ (p.36)
24. COSTANTI (p.37)
25. STRUTTURE DI CONTROLLO (p.38)
26. SOTTOPROGRAMMI E MODULARITÀ (p.44)
27. VISIBILITÀ (SCOPE) DELLE VARIABILI (p.48)
28. NUMERI PSEUDOCASUALI (p.49)
29. TIPI DI DATO STRUTTURATI (p.50)
30. INPUT E OUTPUT SU FILE (p.55)
31. I PUNTATORI (p.56)
32. LE LIBRERIE E LA COMPILAZIONE SEPARATA (p.58)
33. ALLOCAZIONE DINAMICA (p.59)
34. TIPI DI DATO ASTRATTI (p.61)
35. ALGORITMI DI ORDINAMENTO (p.61)

RAPPPRESENTAZIONE DELL'INFORMAZIONE

L'INFORMATICA

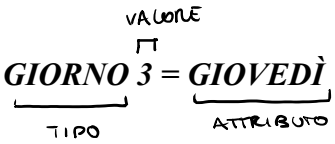
La parola informatica nasce dalla crasi delle parole **informazione** e **automatica**; infatti, l'informatica è lo studio di come, partendo da un **ingresso**, si possa produrre un **risultato** attraverso determinate procedure (raccolte in un **algoritmo**) che permettono il **passaggio dell'informazione**. Lo strumento utilizzato per il passaggio (automatico) delle informazioni è il **calcolatore** che, essendo semplicemente uno strumento, **non è dotato di intelligenza** ma esegue unicamente le istruzioni che gli vengono impartite.

È così già definito i tre step fondamentali per poter trasmettere un'informazione:

1. Codifica delle **informazioni**;
2. Codifica delle **strutture**;
3. Codifica degli **algoritmi**.

Generalmente l'informatica si definisce come "*la Scienza della gestione e della elaborazione dell'informazione*" ma tra tutte le definizioni che vi sono state applicate sorge un comun denominatore: **l'informazione**. Il termine informazione deriva da *informare*, cioè dare forma, e perciò è collegato al concetto di **incertezza**: un'informazione è un concetto astratto che viene comunicato (in qualsiasi forma) **riducendo l'insieme di possibili risposte ad una domanda**. Ad esempio, alla domanda "*Che giorno è oggi?*" l'informazione riguardo il giorno corrente riduce il campo di possibili risposte da sette ad una. **L'informazione** è spesso associata al concetto di **messaggio** nonostante questo ha solo il compito di **rappresentarla** e **trasportarla**.

Per essere tale un'informazione non deve solo ridurre l'incertezza ma deve anche essere dotata di tre caratteristiche fondamentali: il **tipo** (l'insieme in cui l'informazione può essere codificata, ad esempio i giorni o i chilometri di un'auto), il **valore** (il valore specifico appartenente al tipo) e un **attributo** (l'interpretazione del dato)



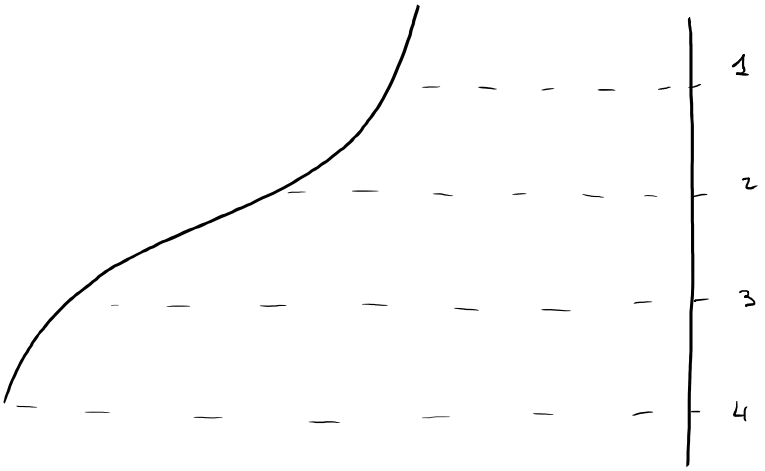
Un'informazione, per poter essere usata, deve essere **adeguatamente rappresentata**; nel quotidiano si usano due tipologie di rappresentazioni che risalgono a migliaia di anni fa:

- La **lingua parlata**, che permette di **comunicare** l'informazione tramite i suoni;
- La **scrittura**, che permette di **conservare** l'informazione tramite un insieme di simboli.

Affinché l'informazione venga **percepita e usata** è necessario che tutte le parti coinvolte nello scambio di questa informazione **condividano le stesse modalità di rappresentazione**, altrimenti ciò che per il mittente significa qualcosa può **essere alterato** dal modo di rappresentare diverso del ricevente. Generalmente i tipi di rappresentazione sono due:

RAPPRESENTAZIONE ANALOGICA: la rappresentazione cambia **analogamente** (insieme) alla grandezza fisica reale. Con esse le proprietà del fenomeno sono **omorfe** alla forma della rappresentazione, restituendo così una misura continua e precisa

RAPPRESENTAZIONE DISCRETA: sfrutta un **insieme di rappresentazioni distinte** che vengono messe in relazione con **alcuni elementi dell'universo da rappresentare**. Essa è **un'approssimazione** della rappresentazione analogica in quanto si basa su un insieme finito, in cui tra due rappresentazioni non se ne trovano altre, mentre nella rappresentazione analogica ne troveremo sempre infinite.



In questo esempio si nota la differenza tra le due rappresentazioni: a sinistra, la rappresentazione analogica, può assumere infiniti valori tra 1 e 4 mentre quella a destra, la rappresentazione discreta, ammette come possibili solo i valori interi tra 1 e 4 (1, 2, 3, 4) e perciò non è accurata. Questa **approssimazione non sempre è svantaggiosa**: i calcolatori hanno **risorse** di memoria **limitate** e sarebbe impossibile rappresentare analogicamente un'informazione in quanto necessiterebbe di infinito spazio; tuttavia, non va sottovalutato l'**errore dell'approssimazione** e va programmato il calcolatore in modo che essa sia **minore possibile**.

Inoltre, **campionando** la misurazione **frequentemente si può ricostruire una rappresentazione discreta alquanto accurata** e vicina a quella analogica ma una discreta **non si può quantizzare nella rappresentazione analogica** originale.

CODIFICA

La **codifica** è l'insieme di regole con cui **un'informazione è trasformata nella sua rappresentazione**, è composta da **regole di composizione** e **posizionamento** che determinano il modo in cui tale informazione è percepita. **Non esiste una codifica univoca** né ci sono svariati modi di rappresentare un'informazione, **l'importante è che la codifica sia condivisa da mittente e ricevente**. Se bisogna comunicare su un pacco l'apertura della scatola scriverlo in una lingua specifica come l'italiano mi permette di comunicare con un numero ristretto di persone (solo coloro che conoscono l'italiano), una lingua comune come l'inglese con la maggior parte delle

persone (escluso chi non conosce la lingua) mentre un simbolo comune è piuttosto universale e permette di avere una buona copertura.

Il codice è un sistema di simboli e regole che permette la composizione di una rappresentazione. Ogni codice è costituito da

SIMBOLI: gli elementi atomici che, uniti secondo le regole del linguaggio, costituiscono le parole codice.

ALFABETO: l'insieme di simboli a cui fare riferimento nella composizione delle parole codice; la lunghezza *n* dell'alfabeto è detta **cardinalità** (*n*).

PAROLE CODICE (o stringhe): le sequenze ammissibili con cui i simboli possono essere combinati; le stringhe possono avere una **lunghezza** (*l*) caratteristica che determina il numero di simboli da combinare.

LINGUAGGO: le regole di composizione delle parole codice.

Durante la codifica di un'informazione va ponderata bene la **scelta della lunghezza** delle parole codice, in quanto essa **determina il numero di informazioni massime che si possono codificare**. Se il numero di rappresentazioni e la cardinalità dell'alfabeto sono determinati dal contesto, **la scelta della lunghezza è arbitraria** e dipende da chi sta codificando il messaggio.

Una lunghezza **non sufficiente non permette di codificare tutte le informazioni** mentre ad una lunghezza **eccessiva corrisponde un codice ridondante**, in cui molte parole chiave non codificano nessuna rappresentazione e si traduce nello spreco di risorse dell'elaboratore.

Si prende in considerazione un insieme V di informazioni da codificare di cardinalità *m*, un alfabeto di cardinalità *n* e una lunghezza non definita *l*, il **numero di parole codice distinte** (definito come n^l) deve essere al più maggiore di *m* e almeno uguale ad *m*.

$$n^l \geq m$$

Tuttavia, la lunghezza delle stringhe può anche non essere fissa ma variare. Nel caso di **stringhe a lunghezza fissa** si ha un insieme finito di parole codice che si può rappresentare e vengono gestite meglio le risorse dell'elaboratore, mentre nel caso di **stringhe a lunghezza variabile** le parole codice sono molte di più ma è difficile gestire al meglio le risorse dell'elaboratore. Non esiste una scelta giusta ma tutto dipende dal **contesto**.

La relazione tra le parole codice e le informazioni da rappresentare deve essere una **corrispondenza biunivoca,** cioè non ci devono essere più parole codice che codificano un'informazione e non ci devono essere più informazioni rappresentate da una parola codice.

RAPPRESENTAZIONE DIGITALE BINARIA E IL CODICE BINARIO

Ad oggi spesso viene usata la **rappresentazione digitale binaria**, basata su un alfabeto di cardinalità due e i cui simboli vengono definiti **bit** (binary digit), cioè le unità minime di

rappresentazione e memorizzazione digitale. I simboli utilizzati sono arbitrari, convenzionalmente si usano **0** e **1** ma si possono usare qualsiasi simboli che fanno riferimento ad una scelta fra due (**Vero** e **Falso**, **Positivo** e **Negativo**).

I supporti di memoria fisici che sfruttano la rappresentazione binaria sono chiamati **Flip-Flop** in quanto agiscono solo su **due stati possibili**, con il vantaggio di **semplificare la memorizzazione** e l'elaborazione e di subire **meno disturbi** rispetto ad una rappresentazione analogica; la rappresentazione digitale binaria **ha avuto così tanto successo proprio per la semplicità** con cui si passa dalla rappresentazione analogica a quella discreta con una **buona approssimazione** (x ± qualcosa è x) ma anche per la semplicità con cui possono essere riscontrate nel **mondo fisico** le **scelte fra due** (la polarità di un disco, il passaggio o meno di corrente).

Il termine **digitale** indica qualsiasi rappresentazione basata sui bit, cioè su una scelta fra due, ma il suo significato è stato ampliato e oggi il termine viene usato per **indicare un'informazione codificata attraverso una rappresentazione discreta**, non analogica.

Dunque, il **codice binario** è composto da un alfabeto di **cardinalità 2** (A = {0;1}), da **stringhe di lunghezza $\boldsymbol{l}$** (dove l è anche il numero di bit da implementare) e da $\mathbf{2}^{\boldsymbol{l}}$ **parole codice diverse**. Viceversa, se si vuol sapere **quanti bit servono per poter codificare k informazioni diverse** si deve invertire la formula e quindi:

$$l \geq \lceil \log_2 k \rceil$$

Dove $\lceil x \rceil$ indica la funzione **ceiling** che restituisce il primo intero maggiore o uguale dell'argomento.

Le informazioni sono spesso rappresentate usando multipli del bit (b), il **byte** (l = 8, B) e i suoi multipli. Quando una stringa ha **lunghezza superiore ad 8** si parla non più di byte ma di **words**, anche se solitamente esse sono **composte da multipli del byte** (quindi a 16, 32, 64, 128 bit).

Lavorando con un sistema a base due i multipli del byte non saranno più determinati, come i multipli di una grandezza fisica, sulla base di dieci ma sulla base di due, andando contro il Sistema Internazionale. Infatti

1KB = $2^{10}B = 1024\ B = 8192\ b$

1MB = $2^{20}B = 1048576\ B = 8388608\ b$

1GB = $2^{30}B$ ...

1TB = $2^{40}B$ ...

Tuttavia, è sbagliato usare i multipli del SI perché faremo riferimento a potenze di 10, per risolvere questo problema nel commercio comune si usano le classiche unità di misura (KB, MB, GB, TB) ma per indicare il valore in binario si aggiunge una i tra il multiplo e l'unità di misura (KiB, MiB, GiB, TiB).

Con **un byte** si può rappresentare solo **256 informazioni**, se però non sono sufficienti 8 bit si può ricorrere ai multipli del byte sopra mostrati per rappresentare le K informazioni distinte. La lunghezza in questo caso sarà:

$$K \leq 2^{8 \cdot b}$$

In tal caso però si codificano comunque **stringhe a lunghezza finita**; pertanto, viene di conseguenza che i numeri gestiti siano a **precisione finita**, ossia siano quelli rappresentati da un numero finito di cifre o appartengano ad un **intervallo fissato [min; max]** e che **la più piccola differenza tra due numeri sia fissata**. Tuttavia, con la stessa lunghezza si possono individuare due intervalli: quello **da 0 a $2^l - 1$** e quello che comprende anche i numeri relativi, cioè **da $+(2^{l-1} - 1)$ a $-(2^{l-1} - 1)$**.

Anche per la codifica dei numeri relativi ci sono **diverse strategie** ma l'importante è condividerle tra mittente e ricevente. Una di queste strategie, quella **segno e modulo**, consta nell'adibire il MSB alla codifica del segno (0 per i positivi e 1 per i negativi). Tale strategia porta inevitabilmente alla codifica di due numeri 0 (-0 = 10000… e +0 = 00000…).

In un sistema di calcolo a precisione finita bisogna tener conto dell'**overflow**, il fenomeno che si verifica quando in seguito ad un'operazione tra due stringhe **il risultato è maggiore del più grande numero rappresentabile o minore del più picco**lo, in genere quando non è compreso nell'insieme dei valori rappresentabili pur essendo nell'intervallo [min; max].

Un esempio di overflow è la somma di **700+300** o la divisione **2:3** nell'intervallo di numeri interi **[-999; 999]**: nel primo caso il risultato è **1000**, che è più grande del massimo valore, nel secondo caso il risultato **non è un numero intero** e perciò non appartiene all'insieme dei valori rappresentabili pur essendo compreso tra gli estremi.

Nei sistemi a **precisione finita cadono anche alcune regole dell'algebra convenzionale**, ad esempio non sempre è possibile rispettare la proprietà associativa della sottrazione ($a + (b - c) = (a + b) - c$). Ad esempio $100 + (900 - 600)$ si può fare mentre $(100 + 900) - 600$ va in overflow. Pertanto, si dice che **l'algebra dei numeri a precisione** finita non sempre è la stessa dell'**algebra convenzionale**.

Un modo di risolvere questo problema è rappresentare **l'intervallo di definizione** come un **sistema periodico**: i valori esterni a tale intervallo vengono ricondotti ad essi prendendo il resto della divisione del valore per il periodo. Ad esempio

Intervallo	Periodo	Valore	Divisione	Resto	Valore ricondotto
[-7; 8]	16	7+2=9	9:16	1	1
[0; 60]	60	58+9=67	67:60	7	7

SISTEMA DI NUMERAZIONE BINARIO

In informatica il **sistema di numerazione binario** ha un'importanza capitale perché ci permette di rappresentare le informazioni con la sola alternanza di 0 e 1, cifre che i calcolatori sono in grado di percepire e trasmettere.

Un **sistema di numerazione** è un insieme di simboli (cifre) e regole che associano ad una e una sola sequenza di cifre uno e un solo valore numerico. I sistemi di numerazione possono essere di due tipologie:

- **Posizionale**, in cui la posizione delle cifre ha un impatto sul valore numerico (la numerazione araba);
- **Non posizionale**, in cui la posizione delle cifre non impatta il valore numerico (la numerazione romana).

In un sistema numerico posizionale le cifre sono definite dalla base (o radice) *b* su cui si basa il sistema, in particolare avremo *b* cifre che vanno **da 0 a *b*-1**. Numericamente la **definizione di sistema numerico posizionale pesato** è:

Data una **base** o radice ***b*** e un **insieme di cifre C di cardinalità *b***, un numero può essere espresso come:

$$N = c_i \cdot b^i + c_{i-1} \cdot b^{i-1} + c_{i-2} \cdot b^{i-2} + \cdots + c_2 \cdot b^2 + c_1 \cdot b^1 + c_0 \cdot b^0$$

Dove *i* va da 0 a *b*. Nel caso volessimo aggiungere anche le **cifre decimali continueremo con gli indici minori di 0**.

Se volessimo passare **da una numerazione in una base ad un'altra** in base diversa non ci resta altro che **applicare la definizione** di sistema numerico posizionale pesato.

$$[10100101]_2 = 1 \cdot 2^7 + 0 \cdot 2^6 + 1 \cdot 2^5 + 0 \cdot 2^4 + 0 \cdot 2^3 + 1 \cdot 2^2 + 0 \cdot 2^1 + 1 \cdot 2^0$$
$$= 128 + 0 + 32 + 0 + 0 + 4 + 1 = 165$$

Anche il sistema di numerazione binario è un sistema posizionale, infatti vi è un insieme di cifre a cui corrisponde un peso in funzione della posizione in cui si trovano, in ordine crescente da destra a sinistra. Si definiscono così:

LSB (Least Significant Bit): è la cifra (o bit) più a destra, cioè quella con peso 0 e meno importante;

MSB (Most Significant Bit): è la cifra (o bit) più a sinistra, cioè quella con più peso e più importante.

1	0	1	0	0	1	0	1
Peso 7	Peso 6	Peso 5	Peso 4	Peso 3	Peso 2	Peso 1	Peso 0

Per convertire un numero da binario a decimale basta applicare la definizione di sistema numerico posizionale pesato, di seguito è mostrato l'algoritmo per convertire un numero decimale in numero con base *b*.

ALGORITMO PER IL CAMBIAMENTO DI BASE (iterativo)

1. Dividere la parte intera del numero *d* per la base *b*;
2. Scrivere il resto della divisione;
3. Se il quoziente è diverso da 0 si reitera l'algoritmo dall'istruzione 1 usando il quoziente come numero intero *d*;
4. Se il quoziente è zero scrivere le cifre ottenute in ordine inverso.

Per quanto riguarda la parte decimale di un numero si procede con un algoritmo diverso, che prende solo la parte decimale (con la parte intera 0).

1. Moltiplicare la parte frazionaria *d* del numero per la base *b*;
2. Scrivere la parte intera del prodotto;
3. Se la nuova parte frazionaria è diversa da zero o non si ripete periodicamente o non sono state ancora raggiunte le cifre binarie prefissate, si reitera l'algoritmo dal punto 1;
4. Se la nuova parte frazionaria è zero o si ripete periodicamente o sono state raggiunte le cifre binarie prefissate si scrive il risultato rispettando l'ordine di calcolo.

d	: b	r	p
165	82	1	7
82	41	0	6
41	20	1	5
20	10	0	4
10	5	0	3
5	2	1	2
2	1	0	1
1	0	1	0
0	NN	NN	NN

d	· b	i	p
0,25	0,5	0	-1
0,5	1,00	1	-2
1,00	NN	NN	NN

Quindi $[165{,}25]_{10} = [1010010101]_2$ dove le ultime due cifre sono la parte decimale.

Per scrivere tutti i possibili numeri binari distinti con *l* bit si alternano gli 0 e gli 1 ogni 2^i volte, dove *i* è la posizione che si sta mappando.

Ad esempio, tutti i possibili numeri binari in 3 bit:

P2	P1	P0
0	0	0
0	0	1

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

Le numerazioni **ottale** ed **esadecimale**, **in base 8 e in base 16** (rispettivamente la **terza** e la **quarta potenza di due**), hanno alcune caratteristiche che, in relazione alla numerazione in base 2, **facilitano il lavoro dell'elaboratore** e **rendono la rappresentazione molto più leggibile all'uomo**. Infatti, per rappresentare **8 cifre** ci vogliono **3 bit** e per rappresentarne **16** ci vogliono **4 bit**; pertanto, si può mettere a confronto le cifre ottali o esadecimali con la loro rappresentazione binaria. Basta, infatti, risalire alla seguente tabella per convertire le due numerazioni senza ricorrere ad alcun algoritmo.

Tabella di conversione per la numerazione ottale:

Ottale	Binario		
0	0	0	0
1	0	0	1
2	0	1	0
3	0	1	1
4	1	0	0
5	1	0	1
6	1	1	0
7	1	1	1

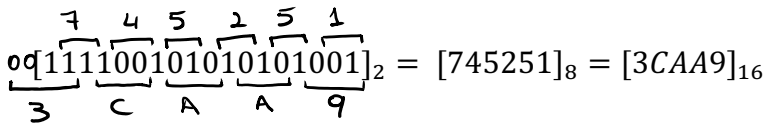
Tabella di conversione per la numerazione esadecimale:

Ottale	Binario			
0	0	0	0	0
1	0	0	0	1
2	0	0	1	0
3	0	0	1	1
4	0	1	0	0
5	0	1	0	1
6	0	1	1	0
7	0	1	1	1
8	1	0	0	0
9	1	0	0	1
A	1	0	1	0
B	1	0	1	1
C	1	1	0	0
D	1	1	0	1

E	1	1	1	0
F	1	1	1	1

La numerazione decimale è sostituita con quella ottale o esadecimale nel caso serva una **maggior leggibilità di un'informazione**, come ad esempio negli indirizzi MAC di un sito web, nella codifica dei colori o in quella dei permessi dei file nei sistemi UNIX.

Per tradurre una stringa da binario ad ottale/esadecimale non si deve fare altro che dividere a gruppi di 3/4 il numero binario e trovare la corrispondenza nelle tabelle riportate di sopra. Bisogna però partire sempre dalla posizione di peso minore in quanto può capitare che la lunghezza della stringa binaria non sia un multiplo di 3/4, in tal caso dovremo aggiungere tanti zeri quanti necessari ma solo al MSB.



Oltre alla rappresentazione segno e modulo esiste un altro modo di rappresentare i numeri relativi, che prende il nome di **complemento a due**. Secondo questo sistema di rappresentazione le stringhe che hanno come **MSB 0**, nell'intervallo $[0;\ 2^{l-1}-1]$, rappresentano **sé stesse** mentre quelle con **MSB 1** rappresentano i **numeri negativi**, nell'intervallo $[\ 2^{l-1};\ 2^{l}-1]$, ottenuti traslando di 2^l l'intervallo di rappresentazione; pertanto, i rappresentanti dei numeri negativi saranno in $2^l-|x|$ l'intervallo $[\ -2^{l-1};-1]$.

Esempio, l=3:

000	001	010	011	100	101	110	111
0	1	2	3	4	5	6	7
0	1	2	3	-4	-3	-2	-1
-4	-3	-2	-1	NN	NN	NN	NN

[0;3] si codificano così come sono, [4;7] sono i numeri negativi traslati di 2^l

Quella mostrata di sopra è la formula per calcolare il **complemento a due di un valore** negativo:

$$x" = 2^l - |x|$$

Per trovare il complemento a due di -4 si applica la regola appena mostrata: $2^3 - |-4| = 8 - 4 = 4$, il rappresentante di -4 sarà il quarto valore codificabile, 100.

Esiste anche un altro modo di complementare un numero, attraverso le sue cifre: si definisce complemento alla base *b* di una cifra *c*:

$$c" = b - 1 - c$$

Trovato così il complemento di ogni cifra nella numerazione di base b si prende il modulo del numero negativo che si vuol trovare e si complementano tutte le cifre sommando alla fine 1.

Si prenda ad esempio $\mathbf{3 = 011}$, il complemento di 0 è 1 e di 1 è zero. Si ottiene così $\mathbf{-3 = 100 + 001 = 101}$.

Esiste ancora un altro modo di convertire i numeri negativi nel sistema complementare a due; infatti, partendo da destra si lasciano tutte le cifre invariate fino al primo 1 (compreso), complementando solo quelle successive.

$3 = 011$ diventa $-3 = 101$

Questo tipo di rappresentazione è utile in quanto **il calcolatore è facilitato nell'eseguire determinate operazioni**, come la sottrazione: non viene costruito un circuito dedicato per la sottrazione ma si ricicla quello dell'addizione calcolando il negativo dell'addendo che ci interessa.

$$x - y = x + (-y)$$

Rispetto alla codifica segno e modulo **la codifica a complemento di due riesce a codificare un numero in più** (infatti **si toglie sempre il bit per il segno**, l-1, **ma non quello per il "secondo" zero**, il -1 sta solo in un estremo dell'intervallo non in entrambi) riducendo ad una sola la rappresentazione dello zero, al costo di un intervallo di rappresentazione non simmetrico.

NUMERI POSITIVI: $[0;\ 2^{l-1} - 1]$;

NUMERI NEGATIVI: $[-2^{l-1};\ -1]$;

INTERVALLO COMPLETO DI RAPPRESENTAZIONE: $[-2^{l-1};\ 2^{l-1} - 1]$

Questo sistema di codifica **non mette in discussione il sistema posizionale pesato**, semplicemente il **MSB non avrà più peso positivo** ma peso negativo (-2^{l-1}) in quanto **se è 1** (il numero è negativo) esso deve essere bilanciato dalle altre posizioni per ritornare un numero negativo:

$$-9 = 11110111$$

1	1	1	1	0	1	1	1
$-128 \cdot 1$	$64 \cdot 1$	$32 \cdot 1$	$16 \cdot 1$	$8 \cdot 0$	$4 \cdot 1$	$2 \cdot 1$	$1 \cdot 1$
-128	$64 + 32 + 16 + 4 + 2 + 1 = 119$						

$$-128 + 119 = -9$$

CODIFICA DEI NUMERI REALI

L'insieme dei numeri reali è **ovunque denso**, ciò significa che in ogni intervallo **tra due numeri esistono infiniti altri numeri** appartenenti a quell'insieme. Tuttavia, la rappresentazione digitale è una **rappresentazione discreta**, prende in considerazione intervalli continui e finiti; per ovviare

a questo problema si divide l'**insieme dei numeri reali** in **intervalli di fissata dimensione**, ogni valore **x appartenente a uno di quelli intervalli viene assimilato all'estremo inferiore di quell'intervallo**:

$$\forall x \in [x_i; x_{i+1}] : x \approx x_i$$

Questa strategia porta inevitabilmente all'**approssimazione**, infatti se si assimila un valore ad un intervallo di valori di cui esso fa parte, si escludono infiniti valori di intermezzo. Il **calcolo numerico** è quella disciplina che studia e cerca algoritmi per implementare una rappresentazione dei numeri reali il più accurata possibile e risolvere problemi che fanno uso ad ampia scala dei numeri reali.

Ad esempio, il numero $0{,}00347$ viene approssimato, in un'aritmetica a quattro cifre decimali, a $0{,}0035$, commettendo un **errore** di $3 \cdot 10^5$, espresso in funzione della cifra con peso meno significativo del numero da rappresentare, $0{,}3 \cdot 10^4$. Seguendo questo criterio, cioè **preso -*m* il peso della cifra meno significativa l'errore massimo** che si commette è:

$$\varepsilon = \frac{1}{2} \cdot 10^{-m}$$

Definito anche valore di **upperbound**.

Si definisce anche il valore **epsilon macchina**, cioè **quel minimo valore diverso da zero di cui non si possono dare altre cifre decimali**. Questo valore dipende fortemente da *-m*, in quanto è l'ultima cifra significativa che si può rappresentare e se si aggiunge **una quantità che sia più piccola** di 10^{-m} ovviamente **verrà interpretata come 0 dal calcolatore**.

Dato un numero finito di cifre è possibile rappresentare un solo **numero razionale che sia l'approssimazione di un intervallo di valori reali**. Nella rappresentazione di questi numeri esistono due tipologie di notazioni: la notazione a **virgola fissa** e a **virgola mobile**.

NOTAZIONE A VIRGOLA FISSA: si decide a priori la quantità di cifre da dedicare alla parte intera e alla parte decimale XXX.YY.

NOTAZIONE A VIRGOLA MOBILE (FLOATING POINT): vengono dedicate delle cifre alla rappresentazione dell'esponente a cui si eleva la base che indica l'ordine di grandezza del numero rappresentato.

I numeri reali vengono così rappresentati:

$$r = mb^e$$

Dove *m* è la **mantissa**, che determina il numero di cifre significative e quindi la **precisione**, b la base in cui andremo ad esprimere il numero ed è **l'esponente** a cui si eleva la base, che determina **l'ampiezza** dell'intervallo di rappresentazione.

I vantaggi dell'uso di questa notazione scientifica sono:

- **Indipendenza** dalla posizione della virgola (floating point);

- Possibilità di **trascurare tutti gli zeri** che precedono la prima cifra significativa grazie alla normalizzazione della mantissa;
- Possibilità di rappresentare con poche cifre numeri **molto grandi** o **molto piccoli**.

Ad esempio, si prenda la codifica di tre cifre per la mantissa e due per l'esponente, in notazione a virgola fissa sarebbe:

$XXX.YY$, cioè $x \in [-999.99; 999.99]$;

In virgola mobile:

$\pm X.XX \cdot 10^{YY}$, cioè $x \in [-9.99 \cdot 10^{99}; +9.99 \cdot 10^{99}]$;

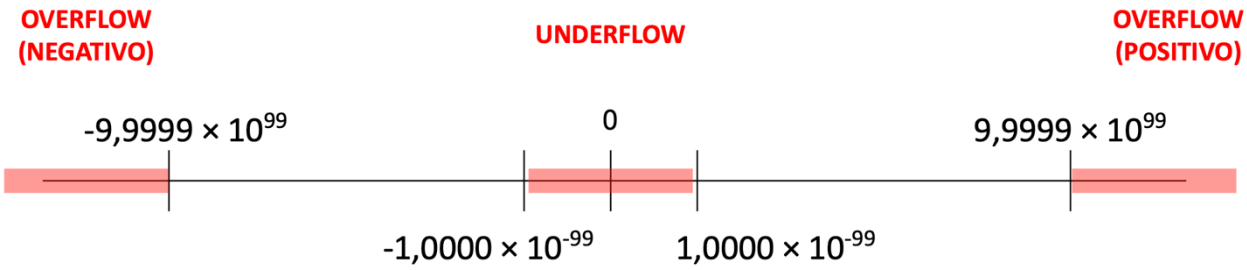
Si noti come la notazione in virgola mobile ci permetta di rappresentare con le stesse cifre molti numeri e molto grandi.

Esistono **infinite coppie** (mantissa; esponente) con cui **rappresentare un numero rea**le in notazione floating point, spostando la virgola della mantissa dove è più comodo, ma la **rappresentazione normalizzata preserva il maggior numero di cifre significative** ed è considerata la più ottimale. La forma normalizzata di un numero reale ci impone che la **prima cifra sia diversa da zero** e che **la parte intera della mantissa sia minore della base**.

Quando il numero rappresentato va **oltre il maggior numero rappresentabile o il minore**, si va in condizione di **overflow** (positivo o negativo), mentre quando il valore che si cerca di rappresentare è **più piccolo del più piccolo intervallo che si può rappresentare** (epsilon macchina), si parla di **underflow**.

$|x| > 9.9999 \cdot 10^{99}$ condizione di overflow;

$|x| < 1.0000 \cdot 10^{-99}$ condizione di underflow.



Gli intervalli di rappresentazione $[x_i; x_{i+1}]$ non sono tutti uguali ma **variano al variare della mantissa e delle sue cifre significative**. Ciò significa che quando ci si **avvicina alla condizione di overflow l'intervallo è molto grande**, con un **errore** altrettanto **grande**, mentre quando ci si **avvicina alla condizione di underflow** l'**errore** è molto **piccolo**. Tuttavia, questa discrepanza di errore assoluto non si riflette nella pratica in quanto calcolando i **rispettivi errori relativi** si nota che essi sono simili e gestibili.

$$[1.000 \cdot 10^{-99};\ 1.001 \cdot 10^{-99}]\ e\ [9.998 \cdot 10^{99};\ 9.999 \cdot 10^{99}]$$

Per quanto riguarda le **operazioni in virgola mobile**, se da un lato per la moltiplicazione e la divisione servono operazioni separate su mantisse ed esponenti, per la sottrazione e la somma serve **l'allineamento dell'esponente**, cioè si fa spostare la mantissa del numero con esponente più piccolo in modo da avere lo stesso esponente del più grande e si opera. Tale processo porta alla **perdita di cifre significative**, con un conseguente **errore maggiore** nell'operazione.

$$1.9909 \cdot 10^1 + 5.9009 \cdot 10^4 = 0.0001 \cdot 10^4 + 5.9009 \cdot 10^4 = 5.9010 \cdot 10^4$$

In questa operazione si sono perse ben quattro cifre significative, $.9909$. Mentre per la moltiplicazione (e divisione):

$$1.9909 \cdot 10^1 \cdot 5.9009 \cdot 10^4 = (1.9909 \cdot 5.9009)(10^{1+4})$$

Come si può dedurre, **sottraendo numeri di valore quasi uguale** le cifre più significative si eliminano (**cancellazione**), con un risultato che ha perso quasi tutte (se non tutte) **le cifre significative**; invece, se si **divide per un numero molto piccolo è facile ottenere un valore che va oltre il valore di overflow**.

Per quanto riguarda la rappresentazione binaria, con base $b = 2$, **è necessario** che la parte intera della **mantissa sia 1** e non 0; pertanto tutti i numeri reali binari in notazione scientifica, a prescindere dal segno, saranno descritti con la parte intera 1. Seguendo questo cavillo della rappresentazione normalizzata per rappresentare un numero reale vengono registrati in memoria la **parte decimale della mantissa** (come se fosse un intero) e l'**esponente** a cui si eleva tale numero:

$$(xxxx; yy) \rightarrow 1.xxxx \cdot 2^{yy}$$

Lo **standard 754 IEE** definisce tre formati numerici principali per descrivere un numero in virgola mobile:

SINGOLA PRECISIONE (32 bit): vengono dedicati **1 bit al segno, 8 all'esponente e 23 alla mantissa**;

DOPPIA PRECISIONE (64 bit): i bit per il **segno** restano **invariati**, per l'**esponente** se ne dedicano **11** e per la **mantissa 52**;

PRECISIONE ESTESA (80 bit): è un formato poco usato.

Formato	segno	esponente	mantissa
32 bit	1 bit	8 bit	23 bit
64 bit	1 bit	11 bit	52 bit

Nel passaggio tra 32 a 64 bit **il vero cambiamento sta nella mantissa**, che guadagna molti più bit dell'esponente. Ciò permette, pertanto, si potranno codificare **maggiori intervalli**.

Nella maggior parte dei linguaggi di programmazione la descrizione dei numeri a virgola mobile in 32 e 64 bit non hanno la stessa denominazione: i primi sono richiamati con la funzione *float* e i secondi con *double*.

Generalmente per codificare l'esponente si usa la **codifica per eccesso**.

Ci sono quattro casi particolari nella descrizione dei numeri reali attraverso questo sistema:

MANTISSA TUTTI 0, ESPONENTE TUTTI 0: viene rappresentato come 0 (con segno);

MANTISSA 0, ESPONENTE DIVERSO DA TUTTI 0: sono rappresentati i numeri denormalizzati, con esponente pari al più piccolo degli esponenti in singola precisione (-126 e non -127) e la mantissa che sottintende 0 e non 1;

ESPONENTE TUTTI 1 E MANTISSA TUTTI 0: viene rappresentato infinito (con segno);

ESPONENTE TUTTI 1 E MANTISSA DIVERSA DA TUTTI 0: si codifica la condizione di NaN (Not a Number), cioè un numero indefinito (per segnalare operazioni come le radici di numeri negativi).

RAPPRESENTAZIONE DEI TESTI

Il **testo** è uno degli oggetti digitali più importanti e usati nel mondo digitale, permettendoci di ricevere informazioni in un **linguaggio comprensibile** senza fare conversioni. Il testo viene codificando associando ad **ogni lettera un codice binario** secondo uno **standard prefissato**.

I primi **standard di codifica delle parole** seguivano le esigenze di coloro che lo avevano inventato, gli statunitensi, per cui non si teneva affatto conto delle **esigenze linguistiche dei paesi orientali** o di tutti quei caratteri che, in lingue come l'italiano o il francese, non sono presenti nell'alfabeto inglese. La prima codifica che ebbe un impatto significativo nel mondo informatico fu la codifica **ASCII a 7 bit**, *American Standard Code for Information Interchange*, sviluppato alla fine degli **anni Sessanta** con lo scopo di facilitare la comunicazione tra computer. Lo standard ASCII non codifica solo lettere, ma anche **segni di punteggiatura** e alcuni **caratteri speciali non stampabili** (come il carattere di a capo). Pochi anni dopo ASCII fu aggiornato, permettendo la codifica dei caratteri accentati di alcune altre lingue attraverso la codifica di un bit in più, da 7 a 8.

Il problema del **multilinguismo** fu superato solo grazie allo standard **Unicode**, codifica che permise la codifica delle scritture a **livello universale** assegnando univocamente un numero ad un simbolo. Lo standard Unicode utilizza la formattazione **UTF** (Unicode Transformation Format); il più usato è **UTF-8**, che utilizza da 1 a 4 byte per carattere: i primi 128 valori (che iniziano con lo 0) codificano ASCII, i successivi utilizzano **2 byte per la codifica di altre lingue** e 3**-4 byte per altri caratteri**.

Un **testo** è una sequenza di caratteri e stringhe che viene letto con un ordine preciso e **sequenziale**; gli **ipertesti**, invece, sono dei documenti nei quali dei **frammenti di testo, i nodi**, vengono collegati attraverso dei **riferimenti, i link**, in modo da saltare da testo a testo in funzione di cosa ci serve, senza dover percorrere tutti i frammenti di testo sequenzialmente. Il campo di applicazione più conosciuto degli ipertesti è nella struttura del web, formattata con il linguaggio **HTML** che fa largo uso degli ipertesti (*Hyper Text* Markup Language).

I DATI MULTIMEDIALI

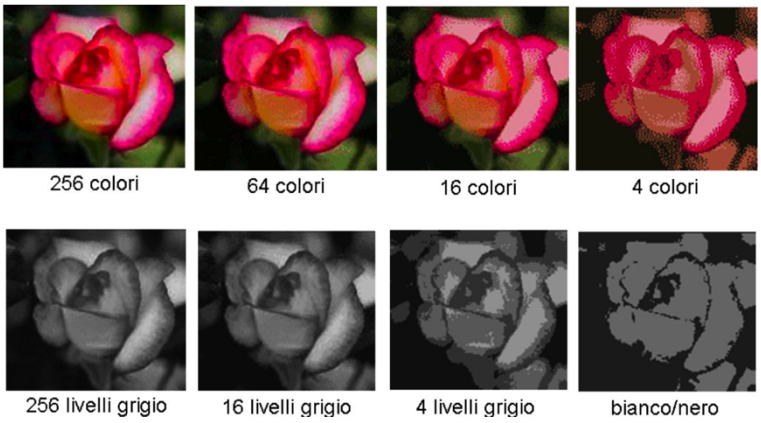
Una rappresentazione **analogica** può essere sempre vista come una **funzione continua** nel tempo, mentre quelle **digitali** discrete sono sempre **approssimazioni**. Per **passare da** una rappresentazione **analogica a** una **digitale**, essenziale per rappresentare l'informazione, si procede con delle operazioni di **campionamento** e **quantizzazione**. Per campionamento si indica la memorizzazione del valore della rappresentazione analogica ad **intervalli di tempo regolari**, individuando dei **campioni**; per quantizzazione si indica l'**approssimazione dei campioni** ad un certo numero prefissato di livelli. Ovviamente un **campionamento preciso** è un campionamento che avviene ad **intervalli di tempo più stretti** e richiede risorse di memoria maggiori.

Un esempio di quantizzazione si ha campionando la frequenza di un'onda sonora a 157 Hz ma registrarla come 160 Hz in quanto il mio calcolatore ha come livelli di rappresentazione solo 150 Hz e 160 Hz.

Per codificare un'**immagine**, che è un insieme continuo di informazioni, bisogna **scomporla** in un insieme finito di elementi che verranno codificati con **sequenze di bit**. Un esempio di codifica delle immagini è la codifica **BITMAP**, che mappa e suddivide l'immagine in tanti **pixel** (*Pic*ture *El*ement) associando **una sequenza di bit ad ogni elemento del reticolo**. Per le **immagini colorate** si sfruttano i principi della **colorimetria**, lo studio della rappresentazione dei colori. I colori possono essere rappresentati a partire da **tre colori primari** che, combinati con metodi **additivi** o **sottrattivi**, possono generare un'ampia palette di colori, associabili ad una sequenza di bit. La codifica più usata per i colori è quella **RGB**, che sfrutta il **rosso**, il **verde** e il **blu** come colori primari mischiandoli in diverse quantità per ottenere colori diversi:

$$colore = aR + bG + cB$$

Dove a, b e c sono i coefficienti che esprimono la "quantità di colore" da mischiare. Codificando ogni base con **8 bit**, otterremo **24 bit** per ogni colore e circa **16 milioni di colori** codificabili. Più sono i colori, più accurata e più pesante è l'immagine.



Le immagini possono essere codificate anche con un metodo **vettoriale**, cioè descrivendo gli **elementi grafici di alto livello** (archi, linee, colori…) che la compongono, ogni figura è codificata con un **identificatore** e dei **parametri di posizione** (punti, lunghezze dei segmenti...). Il metodo vettoriale ci permette di effettuare **modifiche di alto livello** sull'immagine e di renderla

indipendente dalla risoluzione e dal dispositivo di visualizzazione; tuttavia, un'immagine **non è sempre scomponibile in elementi primitivi** e, per poterla visualizzare, bisogna avere lo stesso programma di generazione.

Per la codifica BITMAP i formati più comuni sono **JPEG** (Joint Photographers Expert Group), **BMP** (Microsoft Bit Map) e **GIF** (Graphics Interchange Format), mentre per la codifica vettoriale **PDF** (Portable Document Format) e **DXF** (Drawing eXchange Format).

Per quanto riguarda i **video**, la codifica digitale prevede di **ingannare il cervello umano** e posizionare **una serie di immagini in sequenza** che ci illudono il movimento; ci vogliono infatti **24 fps** (frame per second) per dare l'**idea del movimento**. Lo standard **MPEG** (Moving Picture Expert Group) codifica ogni **frame fisso**, **insieme all'audio**, con delle tecniche particolari di compressione. La **compressione** in un video è **fondamentale** per risparmiare risorse e memoria; la compressione agisce in diversi modi: si parla di **compressione entropica** quando **si riducono i bit** necessari a codificare un'informazione, **compressione differenziale e semantica** quando **si riducono le informazioni da memorizzare** o trasmettere. Generalmente la compressione può conservare o meno il contenuto originale dell'informazione in funzione dei metodi di compressione: si parla di **compressione reversibile (lossless)**, quando si comprime sfruttando le ridondanze del dato senza perdere informazione, e di **compressione irreversibile (lossy)**, quando si sfruttano le ridondanze nella percezione dell'informazione perdendo parte dell'informazione.

La compressione **lossless** viene applicata con una serie di algoritmi che riescono a **ricavare l'informazione originale da quella compressa**, a costo di una **compressione minore**. La compressione **lossy**, invece, **riduce significativamente la memoria** necessaria a conservare l'informazione e sfrutta **illusioni che il cervello può subire senza accorgersene** come la rimozione di parti che possono non essere percepite dall'occhio umano nella codifica dei formati GIF e JPEG.

Anche per quanto riguarda l'**audio** bisogna sfruttare un **difetto del corpo umano**, infatti l'orecchio può percepire fedelmente un audio originale se la sua **frequenza non è inferiore a 30KHz**. Infatti, per la codifica dell'audio, che è **un'onda analogica continua** formata da infinite vibrazioni, si campiona la frequenza originale in campioni nell'ordine delle **decine di KHz**. In funzione della profondità e della frequenza di campionamento si può avere audio più o meno pesanti o audio **mono** (ad un solo canale) o **stereo** (a più canali, per un audio multidimensionale).

I **metadati** sono l'ultima famiglia di informazione che si può codificare e per capire il loro utilizzo bisogna fare una digressione sul concetto di dato. Un **dato** è una **rappresentazione di informazioni concrete in modo formale**, al fine di permettere l'**elaborazione automatica** della stessa; la differenza tra **dato** e **informazione** è il **contesto**: **il dato ne è privo**; infatti, non ha senso se non lo si inserisce in un contesto e, quando si fa ciò, si ottiene un'informazione. **I metadati** (dal latino e dal greco *Metà datum* "dati che parlano di dati") sono particolari dati che **descrivono le caratteristiche o i modi di gestione di altri dati** (metadati descrittivi e metadati gestionali). Un esempio di dato e metadato è la **libreria**: **i libri sono i dati e l'etichettatura è il metadato**.

ARCHITETTURA

GLI OPERATORI BOOLEANI

Sulle stringhe è possibile effettuare delle operazioni **bit a bit** (**bitwise operator**), cioè prendendo ogni singolo bit che compone tale stringa e manipolarlo. Essi sono chiamati **operatori booleani** e i principali sono:

AND, PRODOTTO LOGICO: dati due bit, restituisce 1 solo se entrambi i bit sono 1, altrimenti restituisce 0, è un operatore binario;

OR, SOMMA LOGICA: dati due bit, restituisce 0 solo se entrambi i bit sono 0, altrimenti restituisce 1, è un operatore binario;

NOT: dato un bit, se è 0 restituisce 1 e se è 1 restituisce 0, è un operatore unario.

In ordine di precedenza: NOT, AND e OR.

Se si codificano **1** come **vero** e **0** come **falso**, si ottiene il corrispettivo degli **operatori logici matematici**.

Esiste un altro operatore booleano che si può esprimere in funzione dei precedenti ed è l'**OR disgiuntivo**, **XOR** (oppure **disgiunzione esclusiva**). Esso **restituisce 1** solo se **uno solo** dei due bit in entrata è 1, in tutti gli altri casi è 0. Lo XOR si può scrivere, in funzione degli operatori booleani di base, come:

$$[a\ AND\ (NOT\ b)]OR[(NOT\ a)\ AND\ b]$$

Per verificare che una proposizione come questa sia vera è necessario fare la **tabella di verità**:

a	b	a AND b	a OR b	a XOR b	NOT a
0	0	0	0	0	1
0	1	0	1	1	1
1	0	0	1	1	0
1	1	1	1	0	0

Si può infatti dimostrare così che la tabella di verità di $a\ XOR\ b$ è uguale a quella di $[a\ AND\ (NOT\ b)]OR[(NOT\ a)\ AND\ b]$.

Uno strumento utile nella logica booleana che ci permette di scrivere un operatore in funzione degli altri due è il **Teorema di De Morgan**, secondo cui:

$$\overline{a+b} = \bar{a} \cdot \bar{b}$$

$$\overline{a \cdot b} = \bar{a} + \bar{b}$$

Per operazione bit a bit si intende:

0	0	1	0	1	1	1	0	AND

1	1	1	0	1	0	1	0	=
0	0	1	0	1	0	1	0	

Le applicazioni principali di questi operatori sono nelle **bitmask,** insieme di valori booleani, ciascuno detto **flag**, utili a complementare una stringa (XOR), a resettare tutti i bit a 0 (AND) o a valutare una stringa (AND)...

Come già detto, se si assume 1 come vero e 0 come falso, si può **assimilare gli operatori booleani ad operatori logici**, con la possibilità di instaurare anche **proposizioni logiche**. Una **proposizione logica** è una qualsiasi proposizione che si può dire **vera o falsa**. Per costruire proposizioni complesse o validarne altre è possibile utilizzare gli **operatori di relazione** (non sono operatori logici ma possono essere usati come argomento di essi).

Nella costruzione di predicati complessi con diverse proposizioni, è fondamentale confrontare e validare le sue componenti con le **tabelle di verità**.

GLI ALGORITMI E GLI ESECUTORI

Dalla definizione di informatica (studio sistematico dei processi con cui si può trasmettere automaticamente un'informazione) possono essere estrapolati **i tre processi principali per risolvere un problema**:

1. **Analisi** del problema;
2. **Progettazione** di una sequenza definita di passi che possa risolvere il problema;
3. **Controllo e verifica** della correttezza dell'algoritmo sviluppato.

Prendendo in considerazione questa triade si può anche considerare l'**informatica come lo studio sistematico degli algoritm**i e come essi si comportano nei calcolatori. I **calcolatori** sono solo lo strumento su cui vengono eseguiti gli algoritmi, **non sono dotati di intelligenza propria** e vanno **appropriatamente istruiti** per eseguire un compito.

Il **computer** è un calcolatore progettato appositamente per essere istruito a compiere **automaticamente** e in **tempi utili** delle operazioni prestabilite dal programmatore, che gli fornisce un algoritmo da eseguire. Di conseguenza non è necessaria solo la progettazione dell'**algoritmo**, che può avere **gradi di astrazione differenti** (come il linguaggio scritto o lo pseudocodice), ma bisogna anche **comunicarlo al calcolatore** in maniera adeguata e in modo che esso possa comprenderlo ed eseguirlo. Il linguaggio in cui l'algoritmo è scritto per poter essere compreso dal computer è detto **linguaggio di programmazione**. Infine, l'algoritmo codificato in un linguaggio di programmazione deve essere conservato in un **programma** al fine di essere eseguito dal calcolatore; un programma non solo deve accettare un **input** e restituire un **output**, ma deve contenere anche la **specifica**, cioè la **descrizione in linguaggio naturale** di quale sia il problema da risolvere nel programma. L'inserimento della specifica nei programmi **non è utile al fine del calcolatore ma per coloro che toccheranno in futuro il programma**, in modo che essi sappiano cosa il programma fa effettivamente.

Algoritmo e **calcolatore** non sono altro che una **generalizzazione** dei concetti di **processo** e **processore**: il processo è il compito da eseguire e il processore colui che lo esegue. Sono una generalizzazione perché il processore è un componente del calcolatore (la CPU) e il processo una parte di un algoritmo (una particolare sequenza che compie una determinata operazione propedeutica).

In un algoritmo possono essere distinti anche due tipi di sequenze: le **sequenze statiche**, cioè le sequenze in cui le **istruzioni** sono **scritte**, e le **sequenze dinamiche**, cioè la sequenza con cui le **istruzioni** sono **eseguite**.

IL MODELLO DI VON NEUMANN

Ogni computer ha una **propria architettura** e un proprio modo di esser costruiti per cui per trovare un modo generale di descrivere la loro composizione bisogna usare il metodo **black box**, ossia non si studia l'interno di un computer ma come **funzionano** e si **interfacciano** le sue **componenti fondamentali**; per fare ciò si ricorre ai **modelli**, cioè delle **astrazioni** più o meno accurate per descrivere l'architettura dei computer.

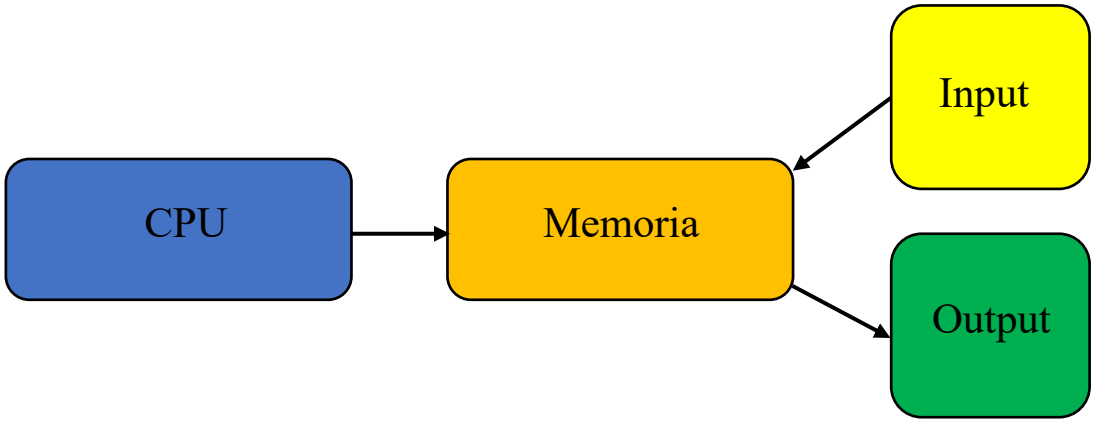
Uno dei modelli più accurati è il **modello di Von Neumann**, sviluppato dall'omonimo ricercatore nel 1945, in esso si può notare:

CENTRAL PROCESS UNIT (CPU): il centro di controllo che gestisce i processi fondamentali;

MEMORIA: contiene un programma con le istruzioni da seguire e i dati prodotti da quel programma;

DISPOSITIVI DI I/O: che si interfacciano con la memoria per fornire e produrre dati.

Il modello di Von Neumann è di tipo **Stored Procedure**, cioè a procedura memorizzata: **le istruzioni del programma sono registrate insieme ai dati nella memoria**, garantendo una notevole **flessibilità** in quanto macchine nate per fare dei calcoli **possono essere impiegate per la risoluzione di problemi di natura diversi**, come impieghi amministrativi o gestionali.



LA MEMORIA

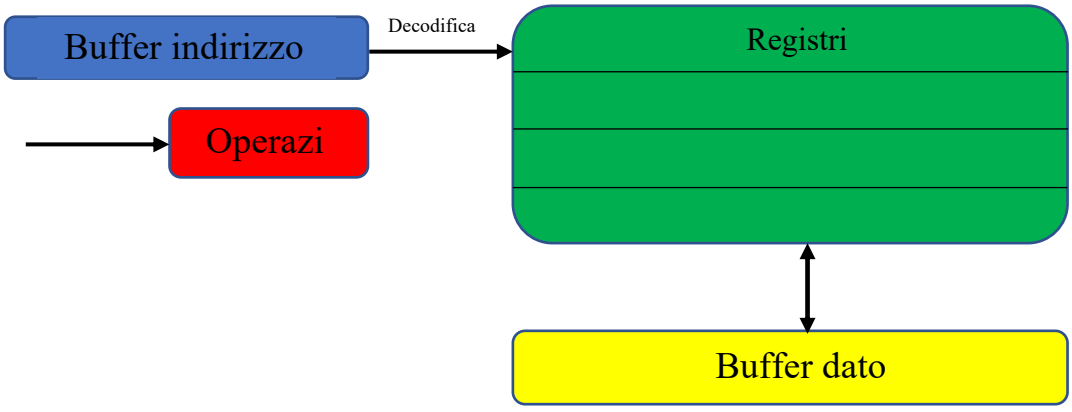
La **memoria** può essere interpretata come un **insieme di contenitori** di dimensione finita (profondità), i **registri**, a cui è associato un identificativo univoco e specifico per quel contenitore, **indirizzo**. La **profondità** dei registri si misura **in bit**, nonostante la maggior parte delle volte sia un multiplo del **byte**, e, indipendentemente da come sono costruiti, i dispositivi di scrittura e lettura devono essere in grado di **interpretare o modificare lo stato in funzione dei bit** (ad esempio la polarità positiva è 0 e quella negativa è 1 di un disco ottico, 5V è 1 e 0V è 0 in un circuito elettronico e così via...). Anche gli **indirizzi** sono **identificati in bit** ma la loro **quantità dipende dal numero di registri della memoria**.

Come già anticipato, sulla memoria si possono effettuare due operazioni: **loading** (caricamento, lettura dell'informazione senza distruggerla) e **storing** (conservazione, scrittura dell'informazione distruggendo la precedente). La memoria è un sistema che permette di **conservare** il dato e di **fornirlo**; tutto ciò avviene attraverso l'uso dei **buffer**, dei **componenti intermedi** che permettono la comunicazione di informazioni come gli indirizzi o i dati con la CPU.

Il **processo di lettura/scrittura** avviene nelle seguenti fasi:

1. Il processore indica preventivamente l'indirizzo del registro da abilitare alla scrittura e l'operazione da compiere attraverso un buffer dedicato;
2. Si esegue l'operazione:
    a. Nel caso di scrittura si preleva il dato da scrivere dal buffer dato, un buffer dedicato alla conservazione di informazioni diverso dal buffer indirizzo, e lo si scrive all'interno del registro abilitato;
    b. Nel caso di lettura si preleva il dato da leggere dal registro e lo si scrive nel buffer dato in modo da poter essere trasportato alla CPU.

Le **tempistiche** per questi due processi sono diverse e **dipendono dalle tecnologie** utilizzate per la costruzione dei registri e dei componenti accessori come i buffer. Il **buffer dato ha la stessa lunghezza dei registri mentre il buffer indirizzo ha lunghezza** $\log_2(\#\textbf{registri})$. Generalmente la **dimensione totale della memoria** è data dal **prodotto della profondità dei registri**, in byte, e **dalla quantità totale di registri** che possiede la memoria.



Esistono diversi tipi di memoria, generalmente si distinguono:

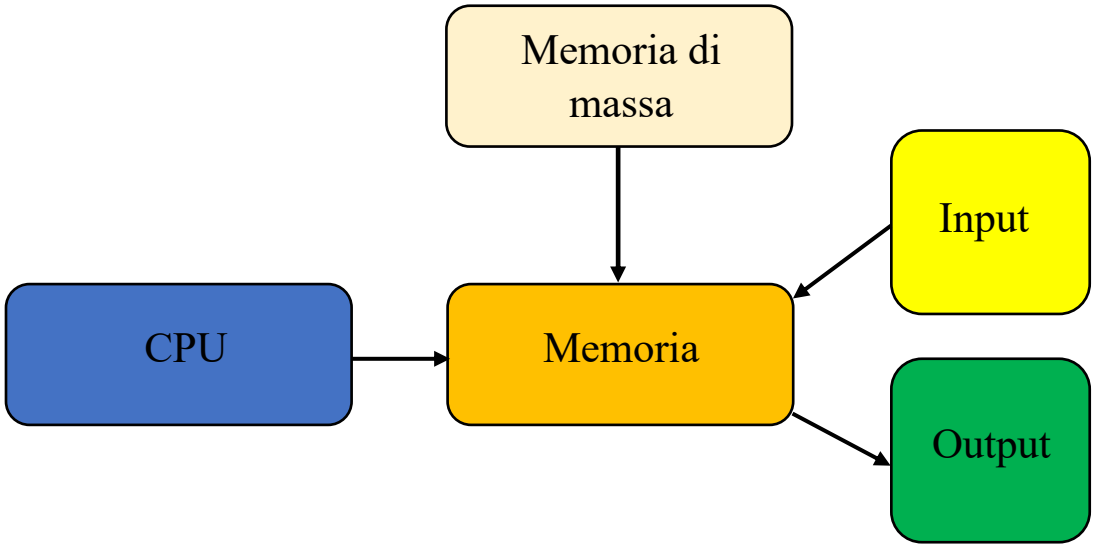
MEMORIA AD ACCESSO CASUALE: il tempo di accesso **non dipende dalla posizione** del registro (RAM, Random Access Memory);

MEMORIA AD ACCESSO SEQUENZIALE: il tempo di accesso **dipende dalla posizione** del registro.

Si può costruire una memoria che sia in grado di **proibire la scrittura**, magari per conservare l'integrità di un dato fondamentale (**ROM**, Read Only Memory) oppure memorie permanenti e volatili. Le **memorie permanenti** sono quelle nelle quali si conservano tutti i dai che devono essere operati dal calcolatore, le **memorie volatili** sono quelle che si caricano di informazioni all'accensione e si scaricano allo spegnimento del calcolatore.

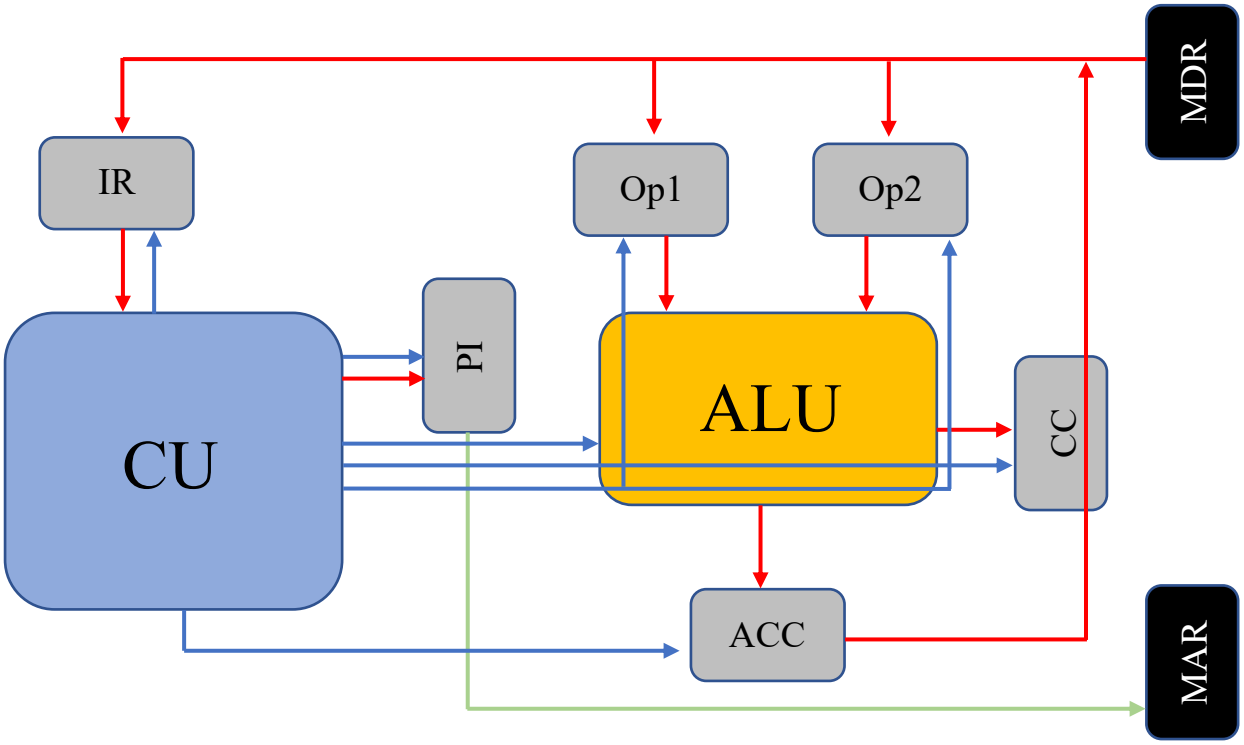
La **memoria RAM** comunica con la memoria di massa per prelevare dei dati e con la CPU per effettuare le operazioni viste prima. L'impiego di quest'altro terminale è dovuto al fatto che esse sono **molto veloci** e permettono uno **scambio** di informazioni con la CPU e le periferiche di I/O **più efficiente**. Tuttavia, esse sono **volatili** e **poco capienti**, pertanto per conservare al meglio le informazioni, si utilizzano le **memorie di massa**, più **capienti** ma **non abbastanza veloci** da sostituire le RAM nel loro compito.

Il modello di Von Neumann si aggiorna e si rappresenta nel seguente modo:



LA CPU

La **Central Process Unit**, o CPU, è il processore che permette all'elaboratore di eseguire delle istruzioni al fine di risolvere un problema. Nel suo diagramma black box, la CPU è composta da tre componenti interni fondamentali, la **C.U.**, la **A.L.U.** e i **registri interni** in un diagramma che si può semplificare come segue:



Scambio dati; Indica posizione; Segnali di controllo

CONTROL UNIT: la **CU** è quella componente **responsabile del prelievo delle informazioni** e dei dati dalla memoria e **dell'esecuzione di tali istruzioni** (se non sono di tipo aritmetico, infatti in quel caso vengono predisposti dalla CU la componente ALU e gli operatori da impiegare); solo una volta eseguita un'istruzione si può passare alla successiva in un **rigido schema sequenziale**;

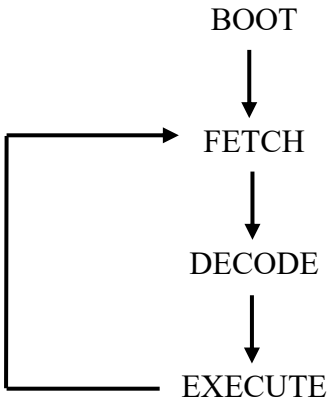
ARITHMETIC AND LOGIC UNIT: l'**ALU** è **responsabile di tutti i calcoli aritmetici**, fa impiego degli operatori (che devono essere predisposti dalla CU) e contiene tutti i circuiti necessari allo svolgimento delle operazioni aritmetiche o bitwise. L'esito delle operazioni viene comunicato al registro CC (Condition Code).

Ad oggi **ALU e CU vengono aiutati da altri componenti interni**: nel caso della ALU interviene un microprocessore matematico che contiene i circuiti necessari a codificare la logica specifica della virgola mobile, mentre nel caso della CU essa è aiutata da alcuni registri interni che sono molto più veloci e accessibili di quelli della memoria centrale ma che di contro hanno una profondità limitata.

REGISTRI INTERNI: permettono la **comunicazione di memoria, CU e ALU** e sono:

- **IC** (Instruction Control), contiene l'istruzione prelevata dalla memoria ed eseguita dalla CU;
- **PI** (Prossima Istruzione), contiene il registro della memoria in cui è conservata l'istruzione da eseguire e che la CU deve andare a prelevare;
- **ACC** (ACCumulatore), è utile alla ALU al fine di conservare gli operandi (Op1 e Op2 nel caso fossero integrati al componente) e il risultato dell'operazione;
- **CC** (Condition Code), una stringa bit-a-bit che contiene le condizioni che si sono verificate durante l'esecuzione delle istruzioni.

Queste componenti interagiscono ed operano secondo uno schema ben preciso che prende il nome di **ciclo del processore**:



Infatti, dopo l'avviamento (Boot), che deve predisporre la prima operazione da eseguire, si passa al **prelievo** e alla **codifica delle istruzioni e dei dati** dalla memoria (Fetch), per poi **prelevare gli operandi**, sempre dalla memoria (Operand), per infine **eseguirli** (Execute). Alla fine di questo processo si prosegue con **l'istruzione successiva**, ritornando al momento di Fetch.

Fase di Fetch: per avviare un programma si inserisce l'indirizzo della prima istruzione nel registro PI (Prossima Istruzione) inviandolo poi al buffer indirizzo della memoria (MAR) che accederà al registro. Una volta ottenuta l'istruzione dal buffer dato (MDR) della memoria la si copia nel registro IR (Instruction Register) mentre il registro PI si è automaticamente aggiornato all'indirizzo successivo.

Fase di Decode: prima di essere eseguita, l'istruzione viene decodificata prelevando, se necessario, altre informazioni al fine di fornire alla CU un'istruzione completa su cui operare. In ogni caso il registro PI punterà alla prossima istruzione da prelevare.

Fase di Execute: nel caso in cui l'istruzione preveda un'operazione aritmetica, i registri verranno caricati con gli operandi e la CU comunicherà alla ALU l'operazione da eseguire. Dopo averla eseguita verrà aggiornato il registro CC (Condition Code) e il risultato conservato in un registro.

I BUS E IL CLOCK

Registri e componenti sono collegati tra loro attraverso i **bus**, dei canali di comunicazione costituiti da **uno o più fili integrati** nel wafer e che possono trasportare contemporaneamente uno o più bit. In funzione del loro lavoro hanno diversi nomi: **bus dati**, **bus indirizzi** e **bus di controllo** o **di comando**.

I **data bus** servono alla CU per **indicare ai dispositivi cosa essi devono fare** e sono impiegati in operazioni come quelle di lettura e scrittura, permettendo ai dati di fluire da CPU a registri di memoria selezionati e viceversa.

L'**address bus** permette alla CU di **comunicare al dispositivo o al registro di memoria che lo si vuole selezionare**; infatti, attraverso essi passa l'informazione relativa all'indirizzo. In questa ottica **ogni dispositivo** (che deve avere anch'esso un indirizzo) **o registro di memoria deve essere capace di riconoscere sul bus il proprio indirizzo**. Solo la CPU è in grado di scrivervi indirizzi, tutti gli altri dispositivi o registri possono solo leggerla e, in funzione dell'istruzione, comunicare i dati in uscita al data bus. I registri di memoria potranno prelevare e inserire dati nel data bus, mentre i dispositivi di input/output solo inserire/prelevare.

I bus possono essere **seriali**, se costituiti da un solo filo e in quel caso passa un solo bit alla volta, o **paralleli**, se costituiti da n fili e in quel caso possono passare contemporaneamente n bit. **L'address e il data bus sono bus paralleli**, la dimensione del primo indica la **capacità di indirizzare** della CPU (n fili possono far codificare una scelta fra 2^n) ossia la capacità di gestire il numero di dispositivi di I/O e la dimensione della memoria, mentre la dimensione del secondo indica la **velocità di scambio** delle informazioni (n fili fanno passare n bit contemporaneamente).

Tutte le attività di un elaboratore sono sincronizzate con un **orologio interno** che scandisce periodicamente i ritmi di lavoro. Tale orologio è il **clock del processore** ed è un **segnale periodico** a **periodo fisso**. Tale segnale sarà quindi dotato di **periodo T** e di **frequenza *f***, misurata in Hertz.

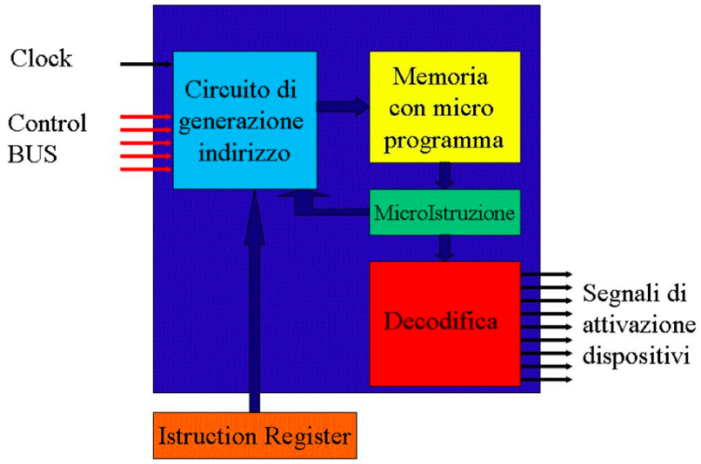
La **velocità di elaborazione** di una CPU è **direttamente proporzionale** alla **frequenza** del suo clock; infatti, esso scandisce il numero di operazioni elementari che la CU può eseguire in un'unità di tempo (può anche accadere che un'operazione elementare abbia bisogno di più cicli di clock).

LE ISTRUZIONI

Le **istruzioni** sono **operazioni semplici** che trasferiscono informazioni **da registro a registro** con un controllo delle condizioni riportate nel registro CC o verificabili con il confronto tra registri. Il trasferimento di informazioni può avvenire tra registri della stessa memoria, da memoria a RAM (o viceversa), da memoria ad output o da input a memoria. Le istruzioni **vengono eseguite dall CU o dall'ALU**, se sono operazioni aritmetiche.

L'esecuzione di un'istruzione da parte della CU consiste nell'inoltro di una **sequenza di abilitazioni** dei dispositivi il cui effetto corrisponde all'operazione richiesta. Le prime CU erano costruite con dei circuiti a **logica programmata**, cioè si evolvevano in tanti modi quante erano le

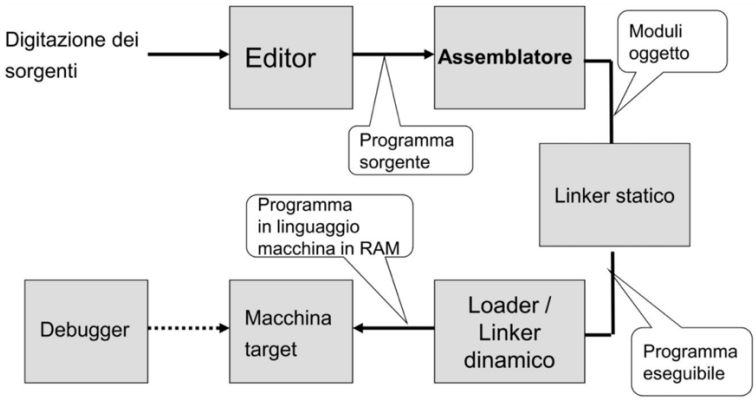
istruzioni che era in grado di eseguire e se si richiedeva un'istruzione non presente fisicamente bisognava riprogrammare l'hardware del processore. Ad oggi invece si preferisce una **logica microprogrammata**, in cui ad ogni istruzione corrisponde una sequenza di **microistruzioni** conservate nella memoria interna della CU. Lo scopo delle microistruzioni è di **comporre l'abilitazione** necessaria all'attuazione dell'istruzione originale; a tal fine un circuito interno alla CU **genera gli indirizzi necessari** per individuare tali microistruzioni che **un decodificatore trasforma poi in segnali di abilitazione**. L'istruzione nel registro IR determina la posizione della prima microistruzione.



L'impiego della logica microprogrammata permette di **scrivere dei programmi in un linguaggio molto simile al linguaggio macchina**, il quale abilita il programmatore ad accedere e usare le microistruzioni memorizzate nella CU. Questi linguaggi prendono il nome di **linguaggi Assembly** e sono particolarmente **legati al processore** su cui i rispettivi programmi gireranno, in quanto un linguaggio Assembly relativo ad un processore può non contenere le microistruzioni di un altro linguaggio relativo ad un altro processore. In merito a ciò esistono due filosofie di progettazione di un processore:

- **CISC** (Complex Instruction Set Computer), possiedono un ampio repertorio di istruzioni, le quali sono più potenti e permettono la scrittura in linguaggio macchina più semplice ma al costo di una maggiore complessità di progettazione;
- **RISC** (Reduced Instruction Set Computer), possiedono un repertorio di istruzioni più ridotto ed essenziale grazie ad un hardware più semplice e una complessità di progettazione ridotta ma al costo di un software più complesso.

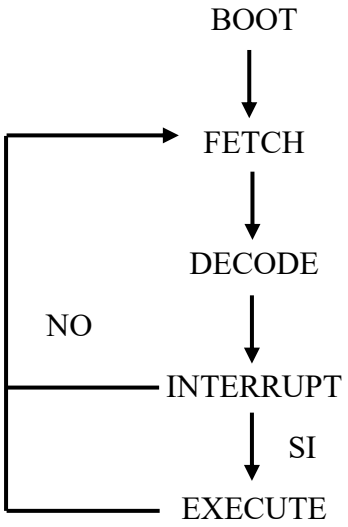
Il **ciclo di un programma in Assembly** è simile a quello di un linguaggio di programmazione compilato ma al posto del compilatore si fa riferimento ad un **assemblatore** che mette insieme le varie microistruzioni in una sola istruzione.



INTERRUZIONI E CHACHE, EVOLUZIONE DEL MODELLO DI VON NEUMANN

Nel modello di Von Neumann non era possibile **sovrapporre più processi**, la CPU era sempre impegnata e poteva svolgere un compito alla volta. Per ovviare a questo problema si è pensato di adibire dei sistemi dedicati, dei **canali dedicati**, allo svolgimento di determinate operazioni e di introdurre un segnale hardware, il **segnale di interruzione**, per attirare l'attenzione della CPU per quelle operazioni non eseguibili da questi componenti. Ad esempio, nell'elaborazione di un'immagine non lavora direttamente la CPU ma un componente apposito che richiama la CPU solo in caso di necessità.

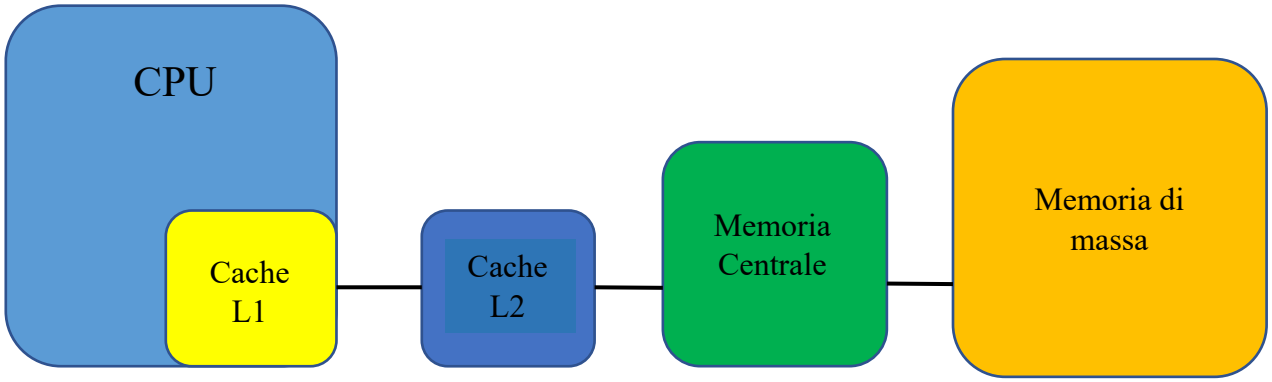
Secondo questo modello la CPU si **disinteressa** delle proprie attività, le **delega** per poter svolgere altre operazioni e riprende il focus su quelle delegate nel momento in cui i componenti a cui sono state assegnate richiedono l'attenzione. In funzione del concetto di interruzione si aggiorna il **ciclo del processore**:



Nel momento in cui viene creato un segnale per l'interruzione, si deve creare un omonimo per **carpire** tale interruzione sulla CU. **Quando un'interruzione viene sollevata** tale circuito

consegna un **bit 1 alla CU**, la quale al termine dell'istruzione verifica o meno la sua presenza. Se il **bit è 0 la CU preleva la prossima istruzione e continua indisturbata**, se è 1 esegue un programma del sistema operativo, **ISR** (Interrupt Service Routine), che trova la causa dell'interruzione e decide quale eseguire per prima in base alla priorità o importanza.

Per ridurre i tempi di trasferimento tra memoria e CPU è stata inserita tra i due una memoria molto più veloce che fa da buffer per il prelievo di informazioni dalla memoria centrale, la **cache**. Durante il suo ciclo, **la CU preleva** le istruzioni e gli operandi dalla cache e quando ciò non è possibile (cioè quando la cache è vuota) ordina un nuovo **travaso** in essa. Esistono due tipologie di cache, la cache di **livello 1 (L1)**, che è fisicamente più vicina al processore, e la cache di **livello 2 (L2)** che è leggermente più lenta e distante.



FIRMWARE E SOFTWARE

Chiamato **programma** un insieme di **istruzioni**, si definisce **firmware** l'insieme di **microprogrammi** composti dalle microistruzioni memorizzate nella memoria interna alla CU, l'insieme di tutti i programmi del computer è detto **software**, anche se in generale software è tutto quanto può essere preteso dall'**hardware** (basta mettere in memoria un programma diverso per cambiare attività). Tra tutte le macchine automatiche i computer sono **sistemi polifunzionali**, in quanto può eseguire e risolvere infinite funzioni e problemi a patto che si progetti un programma per essi.

Nel software si può distinguere il **software di base**, che serve a tutti gli utenti ed è composto da sistema operativo, utilità e traduttori di linguaggi di programmazione, e il **software applicativo**, che risolve un problema specifico e particolare.

Si può distinguere anche il **middleware**, cioè **un'astrazione di programmazione che nasconde all'utente l'eterogeneità di ciò che sta sotto un'applicazione** (reti, hardware, sistemi operativi, linguaggi di programmazione...) **e il suo interfacciarsi in rete**. Ad esempio, il middleware è ciò

che permette all'utente di vedere una foto pubblicata senza mostrare il come e il dove essa è conservata.

Il **Sistema Operativo** (SO) è un **insieme di programmi adibito alla gestione delle risorse hardware** al fine di semplificare l'uso da parte dell'utente. I primi computer non avevano SO e per poter passare da un programma all'altro bisognava manualmente provvedere ad un nuovo caricamento in memoria; con gli SO questo processo è automatico e la CPU si trova ad eseguire in alternanza i programmi del SO e quelli applicativi.

Il **SO offre una serie di funzionalità base allo sviluppatore** come, ad esempio, **l'accesso ai dispositivi**, tra cui i driver del dispositivo, la parte di codice che comanda un modello di dispositivo I/O (un'astrazione dei dettagli dell'hardware). L'accesso ai dispositivi è disponibile grazie a dei moduli software raggruppati in librerie di sistema tipiche per ogni linguaggio di programmazione.

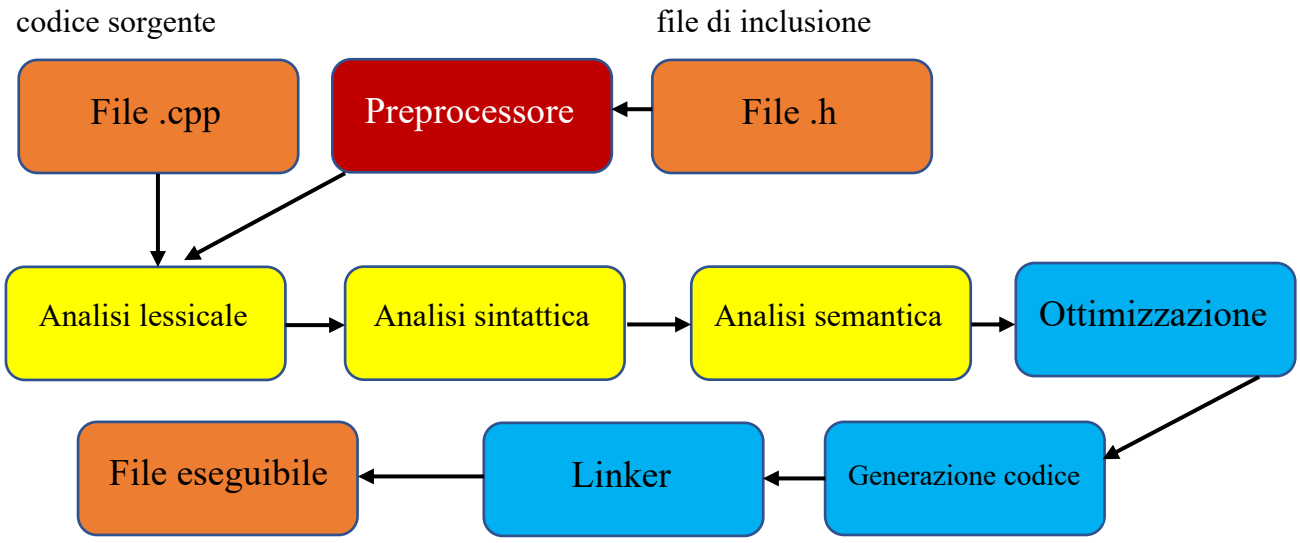
In questa ottica si fa la distinzione tra **linguaggio di alto livello** e **linguaggio di basso livello**: i linguaggi assembly sono di basso livello e servono per creare del codice che interagisca direttamente con l'hardware o per la creazione di dispositivi specializzati (senza SO), mentre i linguaggi di alto livello sono più vicini al modo di pensare comune e sono ad un livello di astrazione maggiore (anche i SO sono scritti in linguaggi di alto livello, tipo con C).

Per eseguire un programma scritto usando linguaggi di alto livello è necessaria una **traduzione delle istruzioni** in istruzioni scritte in linguaggi di basso livello, **così da poterle comunicare meglio all'hardware**; questo processo avviene, in base al linguaggio di programmazione in uso, con un **interprete** o un **compilatore**. Il ciclo di compilazione avviene in due fasi principali, analisi e sintesi.

FASE DI ANALISI: in questa fase il compilatore verifica la correttezza sintattica, semantica e lessicale del programma (simboli, regole e vincoli contestuali);

FASE DI SINTESI: prima della generazione del codice si allocano i necessari spazi di memoria e i necessari registri, si ottimizza il codice e, infine, il generatore di codice trasla la forma intermedia in un linguaggio assembler o macchina

Di seguito è proposto il ciclo dei programmi C++:



PROGRAMMAZIONE

LA PROGRAMMAZIONE E I LINGUAGGI DI PROGRAMMAZIONE

Programmare significa istruire un computer, cioè **progettare un algoritmo** (una sequenza di passi) e **codificare** tale algoritmo, andando a **tradurlo in un linguaggio comprensibile al computer**. Esistono molteplici linguaggi di programmazione, ognuno con le proprie caratteristiche, i propri punti di forza e le proprie debolezze. **Ogni linguaggio di programmazione**, tuttavia, **ha tre componenti** costanti:

1. **Lessico**, l'insieme di parole che costituisce il linguaggio;
2. **Sintassi**, il modo di combinare le parole del lessico;
3. **Semantica**, il significato delle parole e della sintassi.

Ovviamente un programma può essere soggetto ad **errori**, che possono essere ricondotti ad una di queste categorie. In ordine crescente cresce anche la **difficoltà dell'errore**, un errore lessicale può essere corretto subito mentre un errore di semantica può portarci alla riprogettazione dell'algoritmo.

Esistono due tipologie di linguaggi di programmazione, i **linguaggi compilati** e i **linguaggi interpretati**. I linguaggi compilati fanno uso del **compilatore**, un programma che legge le istruzioni e le traduce in un linguaggio comprensibile al processore in modo che le esegua (esso rileva errori lessicali e sintattici ma non di semantica) e sono i più efficienti, mentre i linguaggi interpretati usano un **interprete** che legge le istruzioni riga per riga.

Per scrivere un programma in un linguaggio di programmazione è utile usare un **ambiente di sviluppo integrato**, IDE (Integrated Development Environment), un programma con una serie di strumenti che ci permette di scrivere ed eseguire del codice.

IL LINGUAGGIO C/C++

C nasce nel 1972 nei laboratori di AT&T da **Dennis Ritchie** con lo scopo di facilitare l'istruzione di un computer con un linguaggio che fosse **molto più maneggevole e di alto livello** rispetto ai precedenti (come Fortran, Assembly e altri), tuttavia **non implementava la programmazione orientata ad oggetti** (OOP, Object Oriented Programming). Fu **Bjarne Stroustrup** ad **implementare questo tipo di programmazione** in C, dando vita ad un nuovo linguaggio, **C++**, **nel 1983**.

Gli step che vanno dalla codifica dell'algoritmo in un programma alla sua esecuzione sono:

1. **Editor**, il programma viene scritto in un IDE o in un Text Editor;
2. Viene **prodotto un codice sorgente**, prog.c o prog.cpp;
3. Il programma viene **dato al compilatore che lo esegue**, verificando la bontà della scrittura sintattica e lessicale, sollevando eventuali errori;
4. **Si genera il codice binario** (prog.o, l'estensione è tipica del sistema operativo);

5. **Link**, si collegano al programma una serie di librerie e funzioni pre-fatte che ci evitano di "ricostruire la ruota";
6. Si genera un **output eseguibile**;
7. Si va al **loader** che esegue il programma.

REGOLE LESSICALI E SINTATTICHE

In un programma C/C++ si possono riempire le righe con due tipologie di scrittura: le **istruzioni** e i **commenti**. Le istruzioni sono i passi che il processore deve eseguire, mentre i commenti sono delle frasi o delle parole che il compilatore sa di dover ignorare.

I commenti non hanno finalità nel programma ma è buona abitudine inserirli per rendere il codice più leggibile. Esistono due tipi di commenti, a singola riga o a più righe; per il primo tipo si usa il carattere `//commento`, mentre per quelli a più righe `/*commento*/`.

Con il simbolo **#** si indicano le **istruzioni di precompilazione**, spesso seguite da `include` come **direttiva**, e servono a dire al compilatore **cosa eseguire** o **cosa inserire** nel programma **prima di compilarlo**. Un esempio di istruzione di precompilazione è `#include <iostream>`, che specifica al compilatore di includere nel programma la libreria (racchiusa sempre tra **<>**) iostream, che regola il flusso I/O.

Oltre le istruzioni di precompilazione è importante che **ogni istruzione in C/C++ termini con ;** in modo da specificare al compilatore che **l'istruzione è terminata e che può passare alla seguente**. Se il **;** manca il compilatore proseguirà all'infinito senza passare all'istruzione successiva, sollevando l'errore lessicale.

Con l'identificatore `cout` il compilatore può **stampare a schermo** un dato, una stringa o una variabile che si passa alla funzione. Per poter inserire l'argomento all'identificatore, bisogna racchiuderlo tra i delimitatori **<<**, caratteristici per questa funzione. Analogamente con l'identificatore `cin` seguito da **>>** si può inserire un **input** dall'utente.

GESTIONE DEGLI ERRORI

Gli errori che si possono commettere nella scrittura di un programma sono di tre tipi: **lessicali**, **sintattici**, **semantici** e **logici**, anche se ne esistono di altri. I primi due sono rilevati dal compilatore, che ferma il processo di compilazione e non restituisce output, mentre l'errore logico viene rilevato a **tempo di esecuzione**, cioè quando il programma è già compilato (quindi non ci sono stati errori lessicali o sintattici) ed è in esecuzione.

ERRORI LESSICALI: corrispondono agli "errori di ortografia" del linguaggio comune ed occorrono quando si scrive in maniera errata le parole chiave;

ERRORI SINTATTICI: occorrono quando si scrive bene le parole chiave ma le si usano in maniera errata, non seguendo le regole sintattiche del linguaggio di programmazione;

ERRORI SEMANTICI: occorrono quando si scrivono bene la sintassi ma si effettuano operazioni illecite, come il confronto di un intero con una stringa;

ERRORI LOGICI: è l'errore più difficile da individuare e correggere in quanto non viene sollevato dal compilatore; è un errore che occorre quando nella fase di progettazione dell'algoritmo si commette un errore e, nonostante il programma venga eseguito, non restituisce il risultato che si aspetta, l'unico modo per rilevarli è effettuare delle verifiche con degli input di cui si conosce il risultato e confrontarli con l'output del programma;

WARNING: sono errori meno gravi che, in base a come è stato costruito il compilatore, possono essere automaticamente risolti senza intervento del programmatore;

ERRORI IN FASE DI COLLEGAMENTO: errori che occorrono quando si usa un oggetto in maniera diversa da come è stato progettato (ad esempio, utilizzo una funzione con tre argomenti quando nella progettazione richiedeva solo due parametri); questi errori sono rilevati e sollevati solo dal linker in quanto ne ha piena visibilità;

ERRORI IN FASE DI CARICAMENTO: vengono sollevati quando il programma richiede più memoria di quella allocata o disponibile;

ERRORI IN FASE DI ESECUZIONE: vengono sollevati quando occorre un errore durante l'esecuzione di un'istruzione che rispetta le regole lessicali, sintattiche e semantiche (ad esempio la divisione per zero).

In base a questi errori si hanno diversi casi di sollevamento dell'errore da parte del compilatore, in ordine crescente di difficoltà di rilevamento:

1. **Errore direttamente causato da codice errato**, individuato **alla riga errata**;
2. **Conseguenza indiretta di codice errato**, individuato **dopo la riga errata**;
3. **Errore concettuale**, nessun errore sintattico o lessicale sollevato ma output diverso da quello aspettato;
4. **Mancato rispetto degli standard**, non rileva nessun errore ma **può rendere il programma non compatibile con altri compilatori** o ambienti di sviluppo.

STRUTTURA E QUALITÀ DEI PROGRAMMI

Ogni programma C/C++ contiene il **modulo** anticipato dalla frase `int main()`, che indica il blocco di codice che va eseguito per primo (cioè quello racchiuso tra le `{}` che seguono). Un programma può anche essere composto da più "**sottoprogrammi**", altri moduli simili a `int main()` che hanno nomi diversi e istruzioni diversi; tuttavia, verranno sempre eseguiti dopo il `main`.

Per poter scrivere codice che anche altri programmatori possono comprendere e analizzare vanno seguite alcune **regole formali**, come la regola di **indentazione**: le istruzioni allo **stesso livello gerarchico** vanno **scritte allo stesso livello** e quelle che invece **sono contenute in altri blocchi** di codice vanno **scritti con un rientro maggiore** di quello precedente, mentre ogni riga di codice

deve contenere una e una sola istruzione. Va specificato che **il compilatore esegue le istruzioni** (della stessa gerarchia) in **ordine sequenziale**, dalla riga precedente a quella successiva. Infine, va opportunamente scelto il **nome delle variabili** in modo che esse **indichino adeguatamente il loro utilizzo** (una variabile che contiene il pi greco non va identificata con "pippo" ma magari con "pi_greco").

Anche i **commenti** sono elementi importanti per la scrittura di programmi puliti perché permettono di **comunicare a coloro che leggono il codice sorgente**. Pertanto, inserire commenti che indichino cosa una particolare istruzione fa è buon uso.

VARIABILI E IDENTIFICATORI

In un qualsiasi linguaggio di programmazione di alto livello esistono quattro tipologie di frasi:

DICHIARAZIONI: in C/C++ sono le direttive di precompilazione e sono istruzioni che riceve il compilatore; pertanto, non vengono codificate in linguaggio macchina;

ISTRUZIONI: sono le vere e proprie operazioni che il processore esegue una volta che gli viene dato il programma in linguaggio macchina dopo esser stato tradotto dal compilatore;

STRUTTURE DI CONTROLLO: definiscono l'ordine di esecuzione delle istruzioni;

COMMENTI: servono a chi deve leggere il codice.

Un **identificatore** è una frase, di lunghezza arbitraria e significato esaustivo di cosa deve fare, che viene assegnata ad una variabile, ad una costante, ad un'etichetta, a un tipo definito dal programmatore o ad una funzione. Tuttavia, nonostante si ha la possibilità (non conveniente) di scrivere identificatori lunghi quanto pare, bisogna seguire alcune **regole**. L'identificatore deve:

- Iniziare con una lettera o con '_';
- Contenere solo numeri, cifre o '_';
- Non contenere simboli speciali;
- Non essere una parola chiave del lessico;
- Non iniziare con una cifra;
- Differenziare le lettere maiuscole dalle minuscole.

Un identificatore può essere utilizzato solo dopo essere dichiarato; la **dichiarazione** serve al compilatore per indicare di che tipo è quell'elemento. Infatti, nel caso delle variabili le si dichiara nel seguente modo:

```
nome_tipo nome_variabile;

nome_tipo variabile1, variabile2, variabile3;
```

Si possono dichiarare **più variabili nella stessa istruzione di dichiarazione** solo se esse sono dello **stesso tipo**. Dopo aver dichiarato una variabile la si può inizializzare nel processo di

inizializzazione, cioè associando un valore (temporaneo, perché per definizione la variabile cambia) attraverso il simbolo = (che non ha lo stesso significato dell'omonimo matematico).

```
nome_variabile = espressione;
```

L'inizializzazione può essere inserita contestualmente alla dichiarazione, in una scrittura compatta del seguente tipo:

```
nome_tipo nome_variabile = valore_variabile;
```

Una **variabile** è una **cella di memoria** nella quale si inserisce un singolo valore di un determinato tipo. Quando si dichiara la variabile la si va anche a definire, in quanto **si crea la variabile allocando un concreto spazio di memoria ad essa**.

Dopo averla dichiarata e inizializzata la variabile può essere **usata**. Generalmente, una variabile ***non può*** **essere usata prima di essere stata definita** (altrimenti non viene riconosciuta, in quanto non esiste ancora) e ***non deve*** **essere usata prima di esser stata inizializzata** (altrimenti il codice fa altro)

per sua definizione (l'etimologia aiuta) **una variabile non contiene un valore fisso** e nel tempo può cambiare. **Quando una variabile non cambia** essa è definita **costante** ed è anticipata dalla parola chiave `const`, o in alternativa con la direttiva `define`.

Ogni variabile appartiene ad un **tipo**, cioè un'etichetta che ci dice come il contenuto della variabile deve essere interpretato. Esistono **tipi semplici** (intero, reale, booleano o carattere e sono `int`, `float`, `double`, `bool` e `char`) e **strutturati**, questi ultimi derivano dalla **manipolazione di tipi semplici o strutturati**. Il linguaggio permette di dare nomi anche ai tipi definiti dal programmatore o a quelli predefiniti con la parola dichiarazione `typedef`:

```
typedef tipo nome_tipo;
```

Questa dichiarazione **non occupa memoria** in quanto si sta solo istruendo il compilatore a sostituire con il nostro nome quello del tipo ogni volta che viene usata.

L'occupazione di memoria delle variabili dipende dal tipo e dal compilatore ma è possibile modificare tale occupazione con i modificatori **long**, **short** e **unsigned**. Per verificare la lunghezza di un tipo è necessario usare la funzione `sizeof()` passando come argomento o la variabile del cui tipo si vuole sapere la lunghezza o il tipo stesso.

Lo standard C/C++ garantisce le **dimensioni minime**; per quanto riguarda i **modificatori**, avremo sempre una situazione del genere:

$$r_{short} \leq r_n \leq r_{long}$$

Infatti, la modifica del tipo **dipende dal compilatore** e non sempre si possono avere effettivi cambiamenti (tipo int è uguale a long int).

Questi valori, ottenibili con la funzione `sizeof()`, sono inseriti come costanti nella libreria `<limits.h>`.

ARITMETICA IN C/C++

Gli **operatori aritmetici** in C/C++ restituiscono **valori dello stesso tipo** delle componenti dell'operazione se essi sono dello stesso tipo, altrimenti **si conserva il tipo dell'operatore con maggiore astrazione**. Ad esempio, il tipo double ha un'astrazione maggiore di float, per cui un'addizione float + double sarà di tipo double. Gli operatori aritmetici principali sono **+, -, *,** / e %; quest'ultimo è l'operatore modulo, che restituisce il resto di una divisione tra interi.

Esiste anche un **modulo dedicato per fare operazioni matematiche** più specifiche e complesse, il modulo `<math>`.

La rappresentazione dell'output di un'operazione aritmetica può essere di diversi tipi. **Decimale** se è preceduta da una cifra diversa da 0, **esponenziale** se tra la mantissa e l'esponente si inserisce e/o E, **ottale** se si fa precedere il dato dallo 0, **esadecimale** se da 0x o 0X e **binario** se da 0b o 0B.

Esistono altri tipi di operatori che si possono implementare in C/C++:

OPERATORI LOGICI (NAO): ! (NOT), && (AND) e || (OR). Tali operazioni possono essere effettuate sia con numeri binari (0 e 1) sia con i valori di default di C/C++ `True` e `False`; generalmente qualsiasi valore diverso da 0 è `True`;

OPERATORI BITWISE (CAXO): ~ (Complemento bitwise), & (And bitwise), ^ (XOR bitwise) e | (OR bitwise);

OPERATORI RELAZIONALI: == (uguale matematico), != (diverso), <, <=. > e >= (maggiori o minori); restituiscono sempre un valore `True` o `False`;

OPERATORI DI INCREMENTO: ++i, i++, --i, i-- e sono utili quando si deve incrementare una variabile di una quantità definita; quando la i è prima degli operatori essa viene prima assegnata e poi la variabile si incrementa, quando è dopo si incrementa prima essa stessa e poi la variabile.

Ad esempio:

```
int i = 1;

int secondi = 0;

secondi += i++;    // in questo caso si aggiunge 1

secondi += ++i     //in questo caso prima i = i + 1 e poi sec + 1
```

In C/C++ si possono anche **comporre gli operatori**, cioè aggiungere ad una variabile il proprio valore più qualcosa:

```
var op espressione;

op = op + espressione;
```

Queste due diciture sono uguali, infatti si può effettuare queste operazioni analogamente:

```
A = A + 1;
```

```
A += 1;
A = A + (++I);
A += ++I;
```

E così via...

Infine, è implementato `?` come **operatore condizionale**. Infatti, quando lo si inserisce prima di una condizione e gli si fanno seguire due "**scelte**" esso restituirà la prima condizione in caso di `True` e la seconda in caso di `False`.

```
condizione ? scelta1 : scelta2;
```

Gli operatori appena descritti seguono delle precise **regole di precedenza**:

Priorità	Operatore	Descrizione
1	++ --	Incremento/ decremento postfisso
2	++ --	Incremento/decremento prefisso
2	+ - !	Operatori unari di segno e complemento
3	(tipo)	Conversione di tipo
4	* / %	Moltiplicativi
5	+ -	Additivi
6	<< >>	Shift
7	== != < > <= >=	Relazionali
8	&	AND bitwise
9	^	XOR bitwise
10	|	OR bitwise
11	&&	AND logico
12	||	OR logico
13	? :	Condizionale
14	= *= /= %= += -= >>= <<= &= ^= |=	Assegnazione semplice o composta

COSTANTI

Le costanti sono delle variabili che non possono essere alterate nel tempo e per le quali **non si può effettuare l'operazione di riassegnazione**. Esse possono essere **numeriche** (intere e reali) o **alfanumeriche** (caratteri e stringhe) e sono codificabili in due modi:

- Con la parola chiave `const` prima della dichiarazione;
- Con la direttiva di precompilazione `#define nome valore`.

La differenza tra questi due metodi sta nel fatto che **la prima è un'istruzione** e **la seconda una direttiva**, quando il compilatore trova la variabile nel codice nel primo caso rimanda all'istruzione

relativa e sa di non doverla modificare, mentre nel secondo caso conosce a priori che quel valore è costante.

Sia nelle variabili che nelle stringhe si possono inserire i **valori ASCII** dei caratteri corrispondenti utilizzando il \ e il valore esadecimale o ottale relativo. Il carattere \ serve anche a **separare due stringhe su due diverse righe** e a effettuare le cosiddette **sequenze di escape**, cioè caratteri speciali che effettuano operazioni particolari o ignorano una keyword del linguaggio.

Carattere speciale	Descrizione
\n	Newline o anche ritorno a capo
\r	Carriage return o ritorno a inizio rigo
\t	Tab
\v	Vertical tab
\b	Backspace
\f	Form feed
\a	Alert o beep
\'	Singolo apice
\"	Singole virgolette
\?	Punto interrogativo
\\	Backslash

Sia nelle variabili che nelle costanti il contenuto deve corrispondere al tipo specificato in via di definizione. Se però tra stringhe e caratteri la compatibilità è nulla, con i valori numerici basta che il tipo del valore sia contenuto nel tipo della variabile (float > int, quindi si può scrivere un int in una variabile float ma non il contrario). **È possibile cambiare il tipo di una variabile** effettuando il **casting** (o **coercizione**) con una delle istruzioni seguenti:

```
tipo1 variabile;

variabile = (tipo2)espressione;
```

Oppure

```
variabile = tipo2(espressione);
```

STRUTTURE DI CONTROLLO

Il **paradigma di programmazione** definisce il modello adottato da un linguaggio di programmazione per definire il concetto di **programma**. Il **paradigma imperativo** definisce i programmi come **sequenze di istruzioni di trasformazione dei dati** che vengono eseguite **sequenzialmente**. Infatti, per sviluppare un programma con questo tipo di paradigma c'è bisogno di definire i **dati** che saranno oggetto di trasformazioni e il **flusso trasformazionale**, o **di controllo**, cioè l'insieme di istruzioni che trasformano i dati.

$$\text{Programma} = \text{dati} + \text{flusso di controllo}$$

Secondo questo paradigma il programma si sviluppa **eseguendo le istruzioni nell'ordine in cui sono scritte** e richiamando istruzioni precedenti con il comando `goto`. Infatti, è possibile iterare una sequenza di codice o verificare una condizione servendosi di questo tipo di istruzione, che accetta una condizione e un registro dove è contenuta la prima istruzione del blocco da iterare, e del registro CC, che verifica le condizioni proposte. Tuttavia, questa struttura non basta, o meglio crea un **codice molto complesso e articolato** che prende il nome di ***spaghetti code*** ed è caratterizzato da un'elevata **illeggibilità**.

Per ovviare a questa difficoltà è stato sviluppato un altro paradigma di programmazione, la **programmazione strutturata**, che si basa sul **teorema di Böhm-Jacopini**, il quale afferma che **qualunque algoritmo può essere implementato usando tre sole strutture: la struttura sequenza, la struttura selezione e la struttura iterazione**, a patto che esse individuino sempre un **punto di ingresso** e un **punto di uscita** e che permettano di usare la **ricorsione** e l'**annidamento**.

Dal momento in cui punto di ingresso e di uscita sono univoci, l'annidamento deve innestare il punto di entrata e di uscita della struttura innestata con quello di uscita e di entrata della struttura su cui si innesta.

La **struttura sequenza** è individuata dal **blocco**. Il blocco è un insieme di istruzioni, con diversi gradi di complessità, da eseguire **sequenzialmente**; esso è individuato da **parentesi graffe** e può contenere o essere contenuto in un altro blocco. In tal caso, per distinguere vari blocchi annidati si usa una diversa **indentazione**, che individua anche un ordine gerarchico.

La **struttura selezione** è individuata da quattro strutture di controllo: **if**, **if-else**, **if-else-if** e **switch**, tutte semplici varianti della prima.

IF: la struttura di controllo if è costituita da una condizione logica che restituisce un valore booleano. Al suo verificarsi il programma esegue un'istruzione, se invece restituisce falso il programma non fa nulla e continua ad eseguire le istruzioni successive;

IF-ELSE: la struttura if-else è costituita da un if a cui è aggiunto un blocco di codice nel caso in cui la condizione sia falsa;

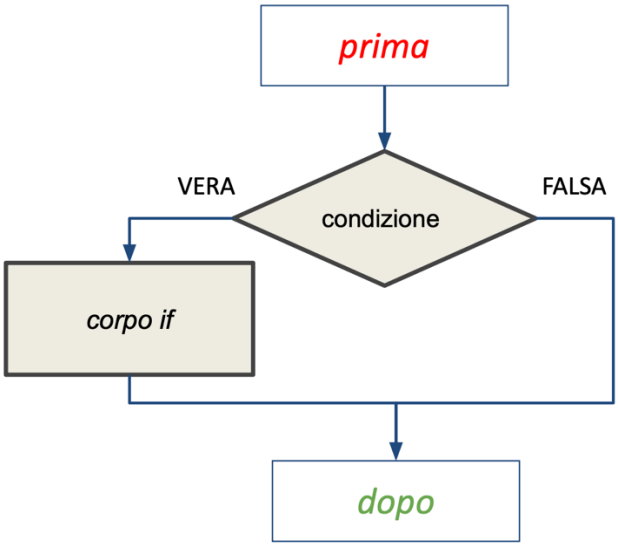
IF-ELSE-IF: la struttura if-else-if è costituita da un blocco di codice che si verifica nel momento in cui la condizione originaria è falsa ma che è preceduta da una seconda condizione e che viene eseguita solo al suo verificarsi. Come per l'if-else, i vari blocchi si escludono a vicenda, cioè non esiste alcuna condizione che permetta l'esecuzione di tutti i blocchi della struttura;

SWITCH: è una variazione dell'if-else-if che permette di eseguire codice diverso in base a diversi valori che la condizione assume; per eseguire un codice bisogna specificare il caso in cui la condizione si presenta e dopo chiudere il blocco con la parola `break`, altrimenti esegue le istruzioni in tutti i casi. È comodo anche inserire una condizione di default, cioè il blocco di codice da eseguire nel caso la condizione non assuma nessun valore tra quelli codificati.

Questo costrutto nasce dall'esigenza di rendere più elegante l'annidamento di più if-else-if.

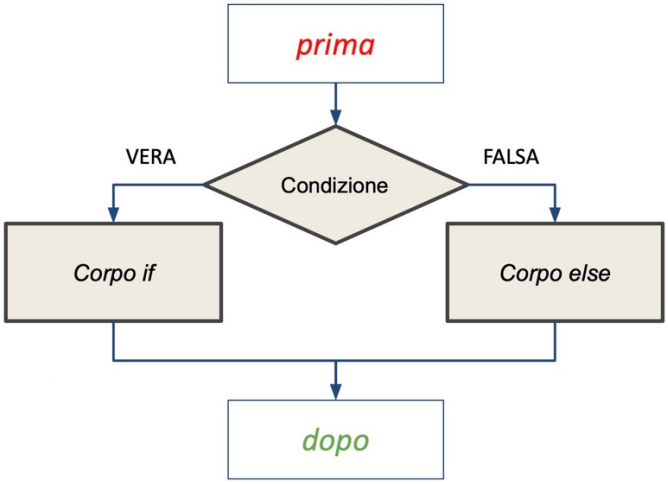
Struttura If:

```
if (condizione){
     blocco;
}
```



Struttura If-Else:

```
if(condizione){
     Blocco 1;
}else{
     Blocco 2;
}
```



Struttura If-Else-If:

```
if(condizione){
     Blocco 1;
} else if (condizione){
     Blocco 2;
} else {
     Blocco 3;
}
```

Struttura Switch:

```
if(condizione){
     Blocco 1;
} else if (condizione){
     Blocco 2;
} else if (condizione){
     Blocco 3;
} else if ...
```

Ma il metodo più elegante è:

```
switch(condizione){
     default:
          blocco default;
          break;
     case k1:
          blocco 1;
          break;
```

```
    case k2:
        blocco 2;
        break;
    ...
}
```

La **struttura iterazione** è individuata da tre strutture di controllo: **while**, **do-while** e **for**, e sono detti anche **loop**. L'**iterazione** permette di ripetere uno stesso blocco di codice finché non si verifica una condizione o per un determinato numero di volte.

WHILE: l'esecuzione del blocco è ripetuta finché la condizione non diventa false, cioè non si verifica la condizione di terminazione. La verifica della condizione avviene prima di eseguire il blocco di codice e, pertanto, esso viene eseguito da 0 a n-1 volte.

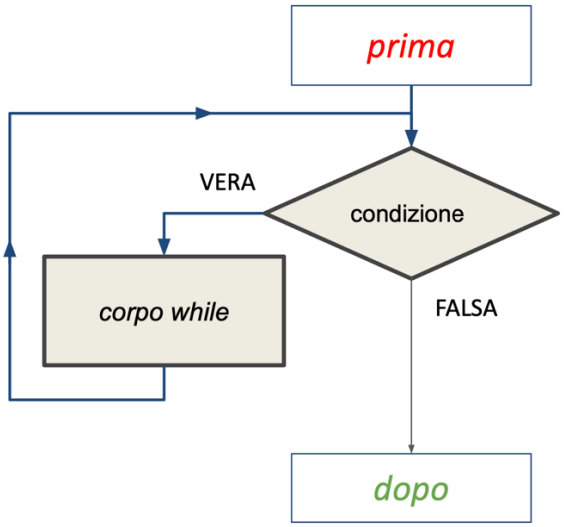
DO-WHILE: l'esecuzione del blocco è ripetuta, come nel while, finché la condizione non diventa false ma in questo caso viene eseguito prima il blocco e poi si verifica la condizione; pertanto, il blocco è eseguito da 1 a n volte.

FOR: anche il for esegue un blocco di codice finché la condizione specificata non è falsa ma esso prescrive l'esecuzione delle istruzioni indicate come inizializzazione, il calcolo della condizione e l'esecuzione in sua funzione, l'aggiornamento della variabile di conteggio dichiarata nell'inizializzazione e l'eventuale ripetizione del ciclo. In un ciclo for possono essere inserite anche più variabili di controllo, però ad ogni variabile di controllo deve seguire necessariamente un'istruzione di aggiornamento.

Generalmente è buon uso specificare una **variabile di conteggio** per tenere traccia delle **iterazioni** e della **condizione** ma non sempre questa pratica è **necessaria**; ad esempio, nel while molto spesso si utilizzano **condizioni logiche booleane** e, pertanto, quando è necessaria la variabile di conteggio è più naturale usare un **ciclo for**. Ciononostante, un ciclo for può sempre essere descritto come un ciclo while (non sempre è valido il contrario).

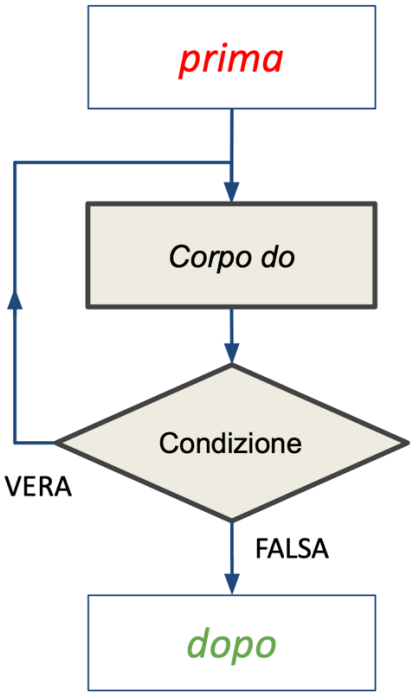
Struttura While:

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



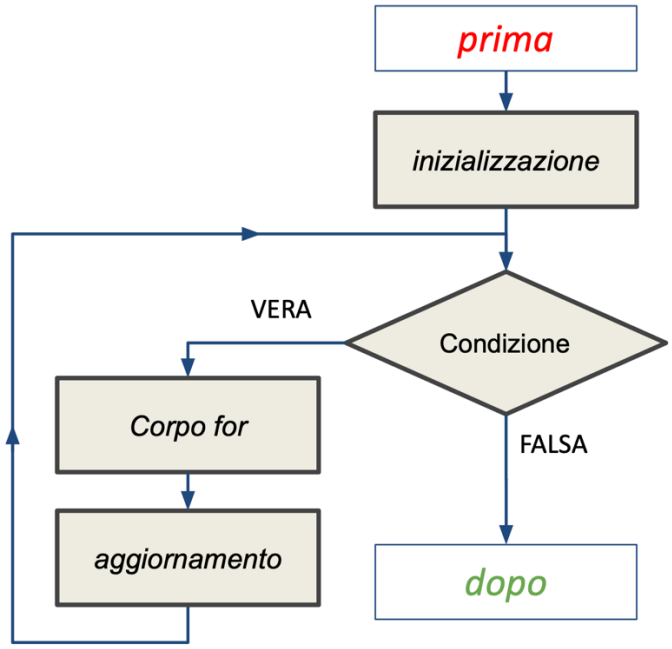
Struttura Do-While:

```
do{
     blocco;
     aggiornamento;
}while(condizione);
```



Struttura For:

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



Esistono delle *keyword* del linguaggio C++ che permettono di effettuare **istruzioni di salto** che introducono eccezioni nello svolgimento del flusso di controllo strutturato. Sono *keyword* particolari e da eseguire solo in casi straordinari ed è preferibile evitarle:

- `goto <Label>`, o salto condizionato, permette di spostare l'esecuzione arbitrariamente al punto di codice sorgente che è etichettato con Label, cioè un'istruzione (`label: istruzione;`);
- `break`, solitamente è usata solo nello switch ma permette di interrompere arbitrariamente un costrutto iterativo, corrisponde ad un goto all'iterazione successiva;
- `continue`, salta arbitrariamente le istruzioni successive in un blocco di un costrutto iterativo passando all'iterazione seguente senza interrompere il ciclo, corrisponde ad un goto alla verifica della condizione.

SOTTOPROGRAMMI E MODULARITÀ

La **modularità** è l'organizzazione in parti, i **moduli**, di un sistema in modo da renderlo **più efficiente** e **facile da comprendere e manipolare**; un programma può essere diviso in **moduli** capaci di svolgere diverse funzioni. Ogni modulo deve essere costituito da una parte di codice **ben**

distinta e deve essere facilmente **identificabile**, il loro utilizzo è conveniente nel momento in cui si deve migliorare la **leggibilità** del software e si **usano più volte le stesse parti** di codice.

Un **modulo software** è un **componente** a cui viene assegnato un **nome** e che **risolve uno specifico problema**; infatti, è un'**astrazione**, la quale è indicata nell'**interfaccia** del modulo stesso. Pertanto, si può distinguere:

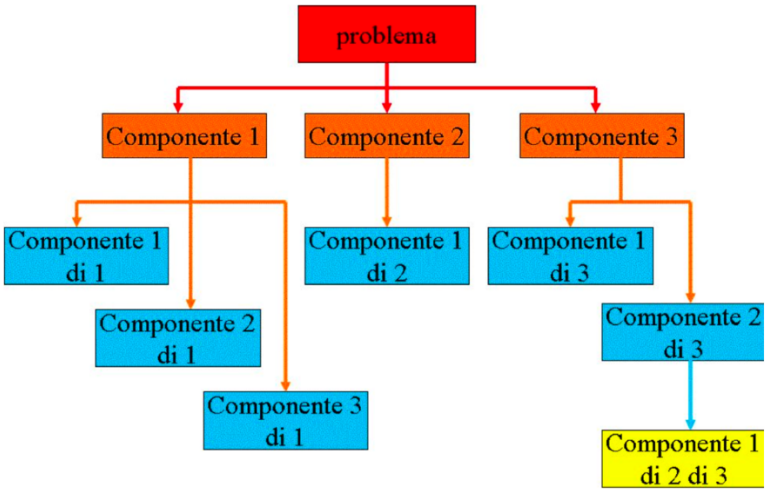
- L'**interfaccia** di un modulo, che è visibile dall'esterno e specifica l'astrazione, il ***cosa*** deve fare;
- Il **corpo**, che è nascosta e protetta dall'esterno e indica il ***come*** deve essere realizzata l'astrazione.

L'astrazione può avvenire in due modi.

ASTRAZIONE SUL CONTROLLO: concerne l'astrazione di una data funzionalità dalla sua implementazione ed è supportata dai tradizionali linguaggi di programmazione;

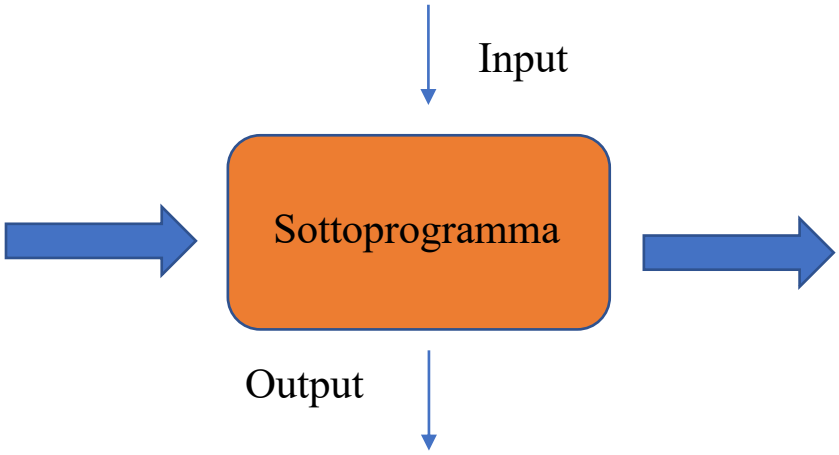
ASTRAZIONE SUI DATI: concerne l'estrazione di entità (oggetti) che costituiscono il sistema, descritti in termini di strutture dato e operazioni su esse possibili, e sono supportate dai linguaggi di programmazione orientati ad oggetti.

È possibile effettuare un analisi **top-down** del problema per determinare i moduli da implementare nel codice. La **radice** costituisce **il problema** stesso, i **nodi le varie decisioni di progetto** e le **foglie la descrizione della soluzione** in modo comprensibile all'esecutore (istruzioni di linguaggio o moduli software).



Il vantaggio della modularità per la costruzione di programmi complessi sta nella **possibilità di individuare al loro interno dei moduli funzionali**, detti **sottoprogrammi**. Ad ogni sottoprogramma sono affidate determinate **responsabilità**, come la soluzione di un **sottoproblema**; hanno un **nome**, possono essere attivati durante l'esecuzione del programma "zero" con la specifica del nome assegnato e gli è passato un **flusso di controllo in esecuzione**; infatti, essi hanno **un solo punto di ingresso e un solo punto di uscita**.

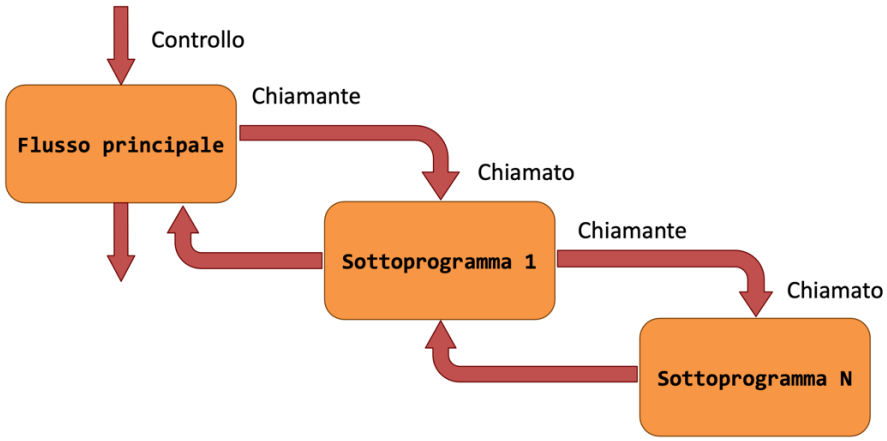
Il sottoprogramma scambia informazioni con l'esterno, in **input** o **output**. Se non scambia informazioni con l'esterno è detto **procedura**, se scambia sia con l'interno che con l'esterno allora è una **funzione** (per la similitudine con una funzione matematica).



La **definizione di un sottoprogramma** necessita di un'**intestazione** (o titolo o signature) e di un **corpo**. Nell'intestazione si inserisce il nome del sottoprogramma, l'elenco dei **parametri for**mali e il tipo del valore restituito, mentre nel corpo il blocco sequenziale di dichiarazioni e istruzioni.

```
tipoRitorno nomeFunzione(tipoP1 P1, ..., tipoPn Pn){
     corpo funzione
}
```

La **forma parametrica**, appena descritta, consente di specificare delle variabili generali (i **parametri formali**) che verranno associate ai dati sul quale il modulo svolge il suo compito (i **parametri effettivi/attuali** o **argomenti**) solo quando il modulo viene chiamato.



Si può **richiamare un sottoprogramma** specificando **come istruzione il suo nome e gli argomenti** (se presenti); pertanto, in quanto istruzione, l'**invocazione** del sottoprogramma produce l'**esecuzione** delle istruzioni di cui il sottoprogramma è costituito. Dal momento in cui un programma è composto da un insieme strutturato di moduli, il programma principale, o **main**, è deputato a fornire il **flusso di controllo principale** ed è l'unico che può **richiamare** tutti i sottoprogrammi.

Nella forma parametrica si fa distinzione tra parametri formali e parametri effettivi:

PARAMETRI FORMALI: sono quelli che vengono dichiarati nell'interfaccia del sottoprogramma e sono definiti con un valore solo all'invocazione del chiamante;

PARAMETRI EFFETTIVI: sono quelli che vengono associati ai parametri formali al momento dell'invocazione.

I parametri effettivi devono essere dello **stesso tipo** di quelli formali e devono essere specificati nello **stesso ordine**.

Quando un **sottoprogramma termina l'esecuzione** delle sue istruzioni può **passare al chiamante al più un valore** (che può essere conservato in una variabile), dello stesso tipo dichiarato nell'interfaccia e specificato dalla keyword `return`.

Dal momento in cui in ogni sottoprogramma C++ è necessario specificare un **tipo di ritorno**, nel caso di **procedure che non producono alcun output** si può usare il tipo **void per evitare di dover ritornare qualcosa**. Tuttavia, void **è usato anche nel caso in cui il sottoprogramma produce più dati in output**, questi infatti vengono **riportati tra gli argomenti della funzione** come parametri di uscita ma vengono **passati per riferimento**.

Come appena visto, **una procedura non restituisce alcun dato** in output ed è di **tipo void,** mentre **una funzione restituisce una variabile di tipo**. Tuttavia, le funzioni possono restituire **una sola variabile con l'istruzione `return`**, per passare più valori bisogna introdurre il concetto di passaggio dei parametri.

Il programma chiamante e il sottoprogramma hanno necessità diverse nell'operare con i parametri: **il sottoprogramma deve conoscere i parametri** che gli devono essere passati **e comunicare i**

dati al chiamante, mentre quest'ultimo deve **fornire i valori dei parametri e ottenere in cambio i risultati**. C++ mette a disposizione due meccanismi per l'associazione dei parametri effettivi con quelli formali:

- Passaggio per valore;
- Passaggio per riferimento.

Con il **passaggio per valore**, si **copia** il contenuto del parametro effettivo nel parametro formale, operando su **allocazioni di memorie diverse**, in modo da non alterare la variabile o l'oggetto che era passato come parametro effettivo. Al termine del programma le variabili locali o i parametri formali svaniscono e **la locazione di memoria viene liberata**.

Nel **passaggio per riferimento** al parametro formale viene associato l'**indirizzo di memoria da cui prelevare il contenuto della variabile**. In questo modo, a patto che il parametro formale sia una variabile, il sottoprogramma **opera direttamente su quella allocazione di memoria**, **modificando il parametro effettivo**, scrivendo o leggendo dati fuori dal suo scope. Per passare per riferimento una variabile la si fa precedere da `&`.

Il passaggio per riferimento **non ha lo scopo di passare in output più dati** ma se si inseriscono dei parametri formali per riferimento nella signature della funzione è possibile restituirli come dai di output, ovviando alla limitazione del `return` che ammette un solo dato.

```
void funzione(int a, int &b, int &c);
```

In questo modo il parametro *a* è solo di input, ma *b* e *c* sono sia di input che di output (se si vuole renderli solo di input li si dichiara come costanti).

In quanto **lavora su un solo spazio** (e non anche in una copia), il passaggio per riferimento è utile quando si lavora con dati di grandi dimensioni.

In C++ si possono passare come parametri formali anche **variabili di tipo strutturato**, con la necessità che i parametri effettivi siano dello stesso tipo. Nel caso degli **array** viene passato come parametro effettivo **un riferimento (puntatore) al primo elemento dell'array**, in modo che il parametro formale assuma automaticamente il significato di puntatore al tipo degli elementi dell'array (con le matrici la stessa cosa ma si deve specificare il massimo di colonne).

VISIBILITÀ (SCOPE) DELLE VARIABILI

È possibile **dichiarare** variabili sia **nel programma principale** che **in un sottoprogramma**; in tal caso intervengono le **regole di visibilità**, cioè l'ambito in cui opera un identificatore dichiarato in un programma, **fissando la porzione di codice che può usarlo** ed **evitando interferenze** con identificatori che non rientrano in uno specifico contesto. Generalmente un identificatore può essere usato solo dalle istruzioni che **seguono la sua dichiarazione** o, se è definito in un blocco di codice, **solo in quel blocco di codice** (non è visibile all'esterno).

In C++ **la visibilità è limitata** (locale) **al blocco più interno** nel quale è dichiarato un identificatore, essi **prendono vita all'attivazione e scompaiono alla chiusura del blocco**. I blocchi interni hanno visibilità degli identificatori esterni al blocco ma non vale il contrario (**locale a globale** ma non globale a locale). Con questo schema di funzionamento si giustifica la possibilità di chiamare, in blocchi diversi, un oggetto con lo stesso identificatore pur lasciando diverso il contenuto (cambia l'allocazione di memoria).

In C++ sono tre i possibili livelli di visibilità:

- **Globale**, visibile a tutto il file (unità di compilazione);
- **Locale**, visibile solo all'interno di un blocco;
- **Parametri formali**, visibili solo all'interno di una funzione.

Con la parola chiave `extern` si può fare riferimento a variabili globali (esterne, appunto) in altre unità di compilazione, inoltre in C++ i **namespace** permettono di isolare e gestire in gruppo insiemi di identificatori per evitare conflitti di nomi.

È definito **ciclo di vita di una variabile** l'intervallo compreso tra la sua **creazione** e allocazione in memoria e la sua **eliminazione** e liberazione dello slot occupato.

Con il fenomeno dell'**aliasing** si possono dichiarare in blocchi diversi variabili con lo **stesso identificatore**. All'interno del blocco più profondo le variabili prese in considerazione saranno quelle dichiarate dentro allo stesso blocco (surclassando quelle globali) ma al di fuori esse non hanno alcun peso.

NUMERI PSEUDOCASUALI

In C++ è possibile utilizzare i **numeri pseudocasuali**. Un numero non sarà **mai al 100% casuale** perché viene **restituito da un algoritmo** che esegue passi ben precisi e prevedibili; tuttavia, la **casualità** dei numeri pseudocasuali deriva dal fatto che l'algoritmo sfrutta un **seed** che rende unico il processo e dunque **garantisce l'incertezza** nell'estrazione del numero. Per inizializzare un numero pseudocasuale in C++ si sfruttano le librerie **cstlib** e **ctime**. La prima fornisce gli strumenti effettivi per estrarre il numero e la seconda serve per inizializzare il seed. Infatti, utilizzando il sistema **Unix Epoch**, si può ottenere il tempo corrente (yy/mm/dd/h/m/s) come un unico numero rappresentati dai **secondi passati sin dal 1/1/1970**; questo escamotage permette di avere un **seed che cambia in continuazione**. La funzione **rand()** restituisce un intero casuale senza alcuna restrizione, per circoscrivere l'estrazione ad un insieme di numeri n si divide il numero ottenuto per il modulo di n+1 in modo da poter avere come **risultato da 0 a n**.

I passi per estrarre un numero pseudocasuale sono:

```
#include <cstlib>
#include <ctime>
```

```
[...]
int seed = time(NULL);
srand(seed); //Inizializza la funzione rand() con il seed
int numeroGrande = rand();
int numero = numeroGrande % n;
```

TIPI DI DATO STRUTTURATI

Un **tipo di dato strutturato** è un tipo di dato che nasce dalla **combinazione** di più **tipi di dati primitivi**. Uno dei più usati è l'***array***. Un array è un **insieme di variabili/strutture/oggetti dello stesso tipo, di dimensione costante e finita**. La dichiarazione dell'array specifica il tipo di dato che vi deve essere inserito:

```
tipo nome[dimensione];
```

Gli elementi di un array vengono allocati tutti in **locazioni di memoria contigue**, a partire dall'etichetta associata a tutto l'array, anche se il contenuto degli elementi dell'array non è definito (come nel caso qui sopra). Per inizializzare un array si può inserire **ogni elemento in parentesi graffe** dopo la dichiarazione o si può **accedere al singolo elemento e modificarlo** in scrittura.

```
int vettore[3] = {0, 1, 2};
```

Per accedere ad un elemento di un array è necessario **specificare l'indice di posizione** di tale elemento; in un array di **dimensione n** gli **indici vanno sempre da 0 a n-1**. L'**accesso** ad un elemento di un vettore **può essere effettuato sia in lettura che in scrittura**, a seconda di dove è posizionata l'istruzione di accesso (cioè l'identificatore dell'array affiancato dall'indice dell'elemento).

Lettura: `valoreDaLeggere = vettore[indice];`

Scrittura: `vettore[indice] = valoreDaScrivere;`

Per indicare un **indice** si possono usare **variabili o espressioni** a patto che esse siano di **tipo intero** e **non superino la dimensione** dell'array. Quest'ultima prerogativa deve essere attentamente **evitata dal programmatore**, piuttosto che dall'utente.

Spesso nella pratica **non si è noto a priori della quantità di elementi** da inserire in un array, per ovviare al problema si specifica una **dimensione** che possa accogliere tanti elementi quanti ne bisogna allocare nel peggiore dei casi (**worst case**) e **si tiene traccia della quantità di elementi da effettivamente inserire** con una variabile di conteggio (**riempimento**), il cui aggiornamento è a carico del programmatore. Questo tipo di allocazione è definito **allocazione statica** (inserire una dimensione maggiore di quella necessaria) ma è possibile anche un **approccio dinamico** (modificare all'occorrenza la dimensione).

Di seguito è proposto un esempio di array con variabile di conteggio:

```
#define MAXDIM 100
int main(){
     int vett[MAXDIM];
     int riemp = 0;

     vett[riemp] = 2;
     riemp++;

     vett[riemp] = 5;
     riemp++;

     vett[riemp] = 4;
     riemp++
}
```

P0	P1	P2	P3	...	P99
2	5	4		...	

Nel caso in cui si voglia modificare il valore precedente:

```
riemp--;
vett[riemp] = 9;
riemp++;
```

In questo modo si aggiorna l'array:

P0	P1	P2	P3	...	P99
2	5	9		...	

Per **ricercare un valore** specifico in un array si può **iterare** nello stesso per **n volte** (con un ciclo for) e restituire il valore nel caso esso sia presente o restituire niente se non è presente.

Un tipo strutturato simile all'array è l'***array multidimensionale***, o **matrice**. Una matrice è una struttura di dato utilizzata quando per identificare un elemento c'è bisogno di **più posizioni** (come un piano cartesiano) e si configura come **più array in sequenza**; infatti, in memoria le matrici si

registrano sempre in **allocazioni contigue** ma, in funzione degli indici inseriti, dal compilatore vengono presi e **raggruppati in più righe**.

La dichiarazione di una matrice è analoga a quella di un array; tuttavia, non si inserisce una sola dimensione ma più, in base alla necessità:

```
tipo nome[dim1][dim2][dimN];
```

L'inizializzazione della matrice può seguire la sua dichiarazione e gli elementi possono essere inseriti sia in maniera contigua che raggruppata:

```
int mat[2][3]={1, 2, 3, 4, 5, 6};
```

```
int mat[2][3]={{1, 2, 3}, {4, 5, 6}};
```

Come gli array, per **accedere in lettura e in scrittura** ad un elemento è necessario specificare la posizione dell'elemento ma, con le matrici, la posizione è espressa da **più indici**:

```
mat[i][j];
```

L'accesso ad un elemento della matrice, in memoria, equivale ad accedere ad un array alla posizione `i+colonne+j`.

Le ***stringhe***, nella loro implementazione più elementare, sono un **array di caratteri** e, pertanto, sono definite come un **tipo strutturato**. Le due espressioni che seguono sono equivalenti:

```
string nome = "Ciao";
```

```
char nome[5] = {'C', 'i', 'a', 'o', '\0'};
```

L'ultimo carattere (`'\0'`), nonostante non sia visibile, è sempre presente nelle stringhe; esso è definito **carattere tappo** (o carattere terminatore, ASCII NULL) e serve a codificare la **terminazione di una stringa** in un array di caratteri, pertanto occupa la **posizione dim-1**. Quando si inizializza un array di caratteri vanno sia elencati i singoli caratteri che il carattere tappo, in questo modo le funzioni che manipolano la stringa sanno **fino a dove far scorrere l'array**.

Se nella dichiarazione dell'array si inserisce una **dimensione più grande** di quanto necessario non verrà sollevato alcun errore ma si occuperanno solo **spazi di memoria non previsti**; l'importante è che sia **sempre presente il carattere tappo**.

Un'altra peculiarità delle stringe come array di caratteri è che **le operazioni di input e output possono essere effettuate sulla stringa** invece che sull'array:

```
char nome[10] = "Ciao";
```

```
nome = "Luca";
```

```
cout << nome << endl;
```

```
cin >> nome;
```

Grazie alla **libreria** `cstring` è possibile effettuare alcune operazioni grazie a delle **funzioni già costruite**:

Nome Funzione	Funzionalità
`strlen`	Calcola la lunghezza della stringa
`strcpy`	Copia una stringa in un'altra
`strncpy`	Copia n caratteri da una stringa ad un'altra
`strcat`	Accoda i caratteri di una stringa ad un'altra
`strncat`	Accoda n caratteri di una stringa ad un'altra
`strcmp`	Compara due stringhe
`strncmp`	Compara i primi n caratteri di due stringhe

```
Funzione(stringaSuCui, stringaDaCui, eventualeN);
```

Con le attuali nozioni di stringa e di input non è possibile fa inserire all'utente una **stringa che contenga spazi**; infatti, l'operatore `cin >>` registra nella variabile **tutto ciò che si trova prima del carattere spazio**. Questo problema è ovviabile attraverso l'utilizzo del metodo `cin.getline()` che permette di registrare in una variabile **tutto ciò che si trova prima del carattere `\n`**. Tuttavia, la convenienza viene ad un costo: dopo che il metodo `cin.getline()` viene invocato e registra l'input, **non pulisce il buffer dal carattere `\n`** e, pertanto, c'è bisogno di **liberare quello spazio** prima di registrare una nuova variabile in input, altrimenti l'utente scriverebbe `\n valore input \n`. In questo modo il compilatore, usando la funzione `cin.getline()` **ignora tutto il buffer**, perché inizia con il carattere a capo. Tale problema si risolve **pulendo il buffer** con la funzione `cin.ignore()` prima di inserire il comando di input.

Per generalizzare il concetto di tipo strutturato si usa la collezione ***record***. Un record è **una collezione di dati eterogenea a cui viene dato un unico nome** e i cui elementi, i **campi** o **membri**, possono essere dello **stesso tipo** o di **tipo diverso**. Si parla di **generalizzazione** perché **ogni struttura dati nota è un record ma non tutti i record sono strutture dati note**.

Per dichiarare un record si specifica il nome, i campi e il tipo dei campi; tuttavia, a differenza di come è stato visto fino ad ora, **la dichiarazione non alloca alcuno spazio in memoria** ma, analogamente alla direttiva `define`, specifica solo al compilatore che quando incontrerà quell'identificativo si farà riferimento a quella **struttura astratta**.

La definizione di un record può avvenire sia insieme alla dichiarazione che lontano:

```
struct nomeRecord{
     tipo1 nome1;
     tipo2 nome2;
     ...
};
struct nomeRecord nomeIstanza = {var1, var2, ...};
```

Oppure:

```
struct nomeRecord{
     tipo1 nome1;
     tipo2 nome2;
     ...
} nomeIstanza;
```

In particolare, in questo secondo caso (ma il concetto può essere applicato generalmente a tutte le varianti di record) per **accedere ai singoli membri** del record si usa la **dot notation**, cioè si specifica a destra del punto l'elemento da cui prendere l'elemento specificato a sinistra del punto:

```
nomeIstanza.nome1;
```

Due dichiarazioni `struct` anche se contengono gli stessi membri, introducono variabili che vengono **considerate di tipo diverso**; solo se vengono specificate **nella stessa dichiarazione** `struct` **sono dello stesso tipo**. Infine, sui record si possono effettuare solo le operazioni definite sui singoli campi e la copia di un intero record in un altro dello stesso tipo.

È possibile **combinare due tipi di dato strutturati**, ad esempio un vettore e un record. Nel caso banale uno dei campi del record può essere un vettore ma in particolare interessa il caso in cui gli elementi di un vettore siano record. In tal caso bisogna dichiarare il vettore con il record di riferimento e ogni elemento potrà essere considerato come un record a sé stante:

```
struct elemento{
    char nome[40];
    int valore;
}
Struct elemento vettore[MAXDIM];
vettore[i].nome = "Luca";
```

Il vettore in tal caso avrà **tutte le proprietà dei vettori** e ogni suo elemento **tutte le proprietà del record di riferimento**.

In genere per **definire un nuovo tipo** si utilizza lo strumento `typedef`, che prende un particolare tipo di variabile e gli **associa un nome univoco**, in modo che ogni variabile che è del tipo definito da `typedef` farà riferimento al tipo definito nella dichiarazione. **Non si creano veri e propri tipi di dato come con i record** ma si **cambia il nome del tipo particolare** per semplicità e pulizia del codice.

```
typedef tipoEsistente nuovoTipo;
typedef char[40] nome;
```

```
typedef struct persona studente;
```

INPUT E OUTPUT SU FILE

In C++ è possibile gestire **diverse tipologie di input e output**, finora è stato utilizzato l'esempio elementare di **input da tastiera e output su schermo**. Questo tipo di operazioni permettono la visualizzazione e l'inserimento di informazioni che sono **volatili**, cioè alla chiusura del programma vengono eliminate; **per conservare le informazioni** registrate in un programma si può effettuare **un'operazione di scrittura o lettura su un file**, in modo da conservare una prova tangibile del programma, da poter riprendere per effettuare altre operazioni. Questo metodo permette anche l'**intercomunicabilità** dei programmi, infatti un file su cui si conserva l'output di un programma può essere un file su cui si registra l'input di un altro programma.

Per effettuare queste operazioni ci si serve della libreria `fstream` e dell'inizializzazione di tre oggetti che si occupano di eseguire le operazioni di input e output:

```
fstream myFile;

ifstream inFile;

ofstream outFile;
```

Un file si apre con il comando `open()`, preceduto dall'oggetto corrispondente all'operazione da eseguire, e vi si passa il **percorso del file** da scrivere/leggere e l'**operazione** da eseguire (opzionale). Esistono **diverse operazioni possibili**, le più usate sono:

Modalità	Descrizione
ios::app	Aggiunge i dati alla fine di un file
ios::ate	Apre il file per l'output e sposta le operazioni di lettura e scrittura alla fine del file
ios::in	Apre il file per la lettura
ios::out	Apre il file per la scrittura
ios::trunc	Pulisce i vecchi dati di un file prima di aprirlo

Con un semplice controllo di stato è possibile verificare se il file è stato aperto:

```
myFile.open(...);

if(!myFile){

    cout << ”File non creato” << endl;

}
```

Dopo aver aperto un file si può scrivere o leggere con i seguenti comandi:

```
inFile >> x;

outFile << x;
```

Ma **ogni volta che si apre un file, è necessario chiuderlo** con il comando `file.close()`. Esistono altre operazioni che si possono eseguire, come la **verifica dell'apertura** con `file.is_open()` oppure il fallimento di un'operazione con `file.fail()` o **l'avvenuta terminazione** con `inFile.eof()`.

Per scrivere su un file è necessario sapere la sua **posizione**, quindi il suo **percorso** o **path**. Un path è una **sequenza di caratteri** che indica precisamente **i nodi del filesystem**, che ha una **struttura ad albero**, da visitare che conducono al file (o alla cartella) da raggiungere. Un path può essere di due tipi:

- **Relativo** e fa riferimento alla directory di lavoro, infatti viene omessa tutta la parte che conduce dalla radice alla directory e ciò permette di lasciare il percorso inalterato nel caso essa venga spostata. Ha il vantaggio di essere più breve;
- **Assoluto**, non dipende dalla directory di lavoro e specifica dalla radice del filesystem al file da raggiungere.

I PUNTATORI

Una variabile di tipo puntatore contiene un indirizzo di memoria. I puntatori in C++ possono essere di qualsiasi natura, semplici o strutturati, e adempiono a diversi scopi, come per il **riferimento a funzioni**, per i **vettori**, per i **parametri formali di funzione** o per il **riferimento a specifiche allocazioni di memoria**.

Un puntatore ad una variabile T contiene l'indirizzo di memoria in cui è allocata T ed è **fortemente tipizzato**, cioè **dipende** strettamente **dal tipo** della variabile che punta; infatti, nella dichiarazione di un puntatore va necessariamente inserito il tipo della variabile che punta. Per dichiarare una variabile di tipo puntatore si aggiunge * dopo il tipo della variabile puntata:

```
nomeTipo* nomePuntatore;
```

A questa variabile puntatore **è possibile associare solo indirizzi di memoria che codificano per variabili di tipo `nomeTipo`**. Viceversa, è necessario il casting delle variabili:

```
pintero = (int*)preale
```

Ai puntatori possono essere associati i due operatori unari & e *.

- **Operatore di referenziazione** (o riferimento) &, prende una variabile e restituisce l'indirizzo di memoria in cui è memorizzata;
- **Operatore di dereferenziazione** *, prende un puntatore e restituisce il valore della variabile che punta.

```
int valore = 5;

int* puntatore = &valore;

valore == *puntatore && puntatore == &valore;
```

Ciò indica che i due operatori sono **complementari** e codificano **operazioni inverse**.

Per **azzerare un puntatore** lo si punta ad un'**area di memoria indefinita** che può essere codificata o con lo 0 (puntatore nullo) oppure ad una costante definita nella libreria `stddef.h`, ovvero `NULL`.

Le **espressioni aritmetiche** che si effettuano su variabili comuni non sono le stesse rispetto a quelle effettuate **sui puntatori**; infatti, esse sono fortemente **influenzate dal tipo** delle variabili che gli operandi puntano. In particolare, sono definite:

- Operazioni di **somma e differenza** tra puntatori (+ o -);
- Operazioni di **incremento e decremento** tra puntatori (++ o --, post o pre).

La differenza sostanziale sta nell'**unità presa in considerazione**: quando si incrementa un puntatore di *i* volte non si aggiungono *i* bit al suo indirizzo ma si sposta il puntatore di *i* posizioni rispetto al tipo della variabile puntata. Ad esempio, se si punta una variabile intera e il tipo int occupa 4 bit, l'incremento di una posizione aumenta l'indirizzo di 4 bit. Generalizzando, **si aggiungono `sizeof(puntatore)` bit per ogni unità**.

Questa tecnica è particolarmente utile nei **vettori** e nelle **strutture di dati omogenee**, ma se la si usa con variabili comuni si rischia di **accedere ad aree di memoria che codificano variabili di tipo diverso**, accedendo solo ad una parte dell'allocazione e con una codifica diversa, ottenendo un **risultato alterato**.

Un puntatore può essere associato a qualsiasi tipo, anche ad un altro puntatore, ma è di elevata utilità il caso dei **vettori** e dei **record**. In C++ **un array è un caso particolare di puntatore**; infatti, il nome del vettore senza parentesi quadre è un **puntatore al primo elemento**, cui seguono in allocazioni contigue gli altri del vettore. Pertanto, le seguenti scritture sono equivalenti:

```
v[3] == *(v+3);
```

```
v = &v[0];
```

Nel caso di array multidimensionali per accedere ad un elemento **si moltiplica la riga a cui si trova l'oggetto per il numero totale di colonne e si aggiunge la colonna a cui si trova l'oggetto**.

```
A[i][j] == *(A+(i*colonne)+j;
```

I **puntatori a record** sono puntatori che puntano al registro di memoria in cui è allocato il record e sono utili nella realizzazione di tipi dinamici come liste concatenate o doppiamente concatenate.

Con i puntatori a record possono essere fatte tutte le operazioni che si eseguono normalmente con un record; tuttavia, diventa più semplice l'accesso ai campi di una struct:

```
struct persona{
   char[40] nome;
   int età;
```

```
  bool haPatente;
} persona1;


struct persona* pPersona = &persona1;
pPersona->nome == persona1.nome;
```

Un generico puntatore, quando passato come **parametro formale** ad una funzione, può essere una **valida alternativa al passaggio per riferimento** (nonostante quest'ultimo sia più potente perché il compilatore lo tratta automaticamente con la tecnica dell'indirizzamento indiretto). L'esistenza di due tecniche molto simili come queste deriva dal fatto che **nel linguaggio C**, da cui deriva C++, **non esiste il passaggio per riferimento** e per passare ad una funzione un indirizzo bisognava necessariamente passare un puntatore. Inoltre, è possibile **restituire da una funzione un valore di tipo puntatore** (quindi un indirizzo) che generalmente punta ad un'**area associata ad uno dei parametri di ingresso**. Quando una funzione ha come parametro un puntatore che è dello stesso tipo del puntatore restituito è possibile effettuare anche l'**esecuzione concatenata** della funzione:

```
T* f(... T*) -> f(...f(...p))
```

LE LIBRERIE E LA COMPILAZIONE SEPARATA

Il **programma principale** è quella unità di programma **unica** (ne esiste una sola) che **si interfaccia direttamente con il sistema operativo** e, in C++ come in molti altri linguaggi, si chiama `main`. Il `main` è il punto in cui **inizia la compilazione**, indipendentemente da dove esso è posizionato nel progetto, e può prendere dei parametri in ingresso per interagire con il sistema operativo; tuttavia, un `main` **restituisce sempre un intero**, che è 0 nel caso di esecuzione corretta.

Nello stesso `main` file possono essere creati dei sottoprogrammi o delle funzioni; tuttavia, è buon uso implementare i sottoprogrammi in **librerie** esterne che possono essere **invocate dal** `main` e le cui funzioni usate. In C++ per includere una libreria si manda una **direttiva al precompilatore** di questo tipo:

```
#include <nomeLibreria>
```

Una volta fatto ciò si può mettere mano ad ogni oggetto definito in tale libreria. L'utilizzo delle librerie, sempre come convenzione, prevede la **costruzione di tre file distinti**:

1. Il **file di intestazione** (`file.h`) che contiene le **dichiarazioni** di variabili globali, costanti e tipi e le dichiarazioni dell'interfaccia di una funzione;
2. Un **file di implementazione** (`file.cpp`) che contiene l'**implementazione** di tutte le funzioni e variabili dichiarate nel file di intestazione;
3. Un **file di testing** (`main.cpp`) che contiene il programma principale che **esegue e verifica** la correttezza delle funzioni sviluppate.

Questa pratica prende il nome di **compilazione separata**, in quanto ogni file viene compilato come file a sé stante, nonostante ci sia comunicazione tra i tre blocchi.

Va specificato che nella creazione del file di intestazione, è buona pratica inserire le dichiarazioni precedentemente menzionate tra le direttive `#ifndef` e `#define`.

ALLOCAZIONE DINAMICA

La memoria è divisa in due porzioni distinte e separate, una **memoria stack** e una **memoria heap**. Nella memoria **stack** prevale il concetto di **gestione automatica delle variabili** gestite con una **politica LIFO** (Last In First Out): quando in un blocco vengono dichiarate delle variabili esse sono **allocate in posizioni contigue**, spostando man mano lo **stack pointer** che indica la **prossima allocazione** in cui inserire un valore; quando il blocco termina, le variabili in esso contenute sono **deallocate automaticamente dal compilatore**, che allo stesso tempo **riduce di una posizione lo stack pointer** per ogni variabile. La politica LIFO gestisce proprio l'ordine con cui le variabili vengono deallocate: **le variabili allocate per ultime vengono deallocate per prime e viceversa**.

Nella memoria **heap** questo processo non è implementato e, come suggerisce il nome "mucchio", le variabili sono **allocate in spazi di memoria casuali**, non necessariamente contigui. Tuttavia, per tenere traccia di una variabile si ricorre al suo **puntatore**, il quale viene **allocato come variabile** nella memoria **stack** e tiene traccia dell'indirizzo in cui è inserito il valore nella heap. Praticamente, per allocare una variabile nella memoria heap si ricorre all'istruzione `new` che **dichiara un puntatore** e **inizializza il valore** da esso puntato:

```
int* puntatore = new int;
```

Poiché le variabili dinamiche non funzionano col concetto di gestione automatica delle variabili, **la deallocazione non avviene automaticamente** da parte del compilatore ma è **a carico del programmatore**, il quale si deve occupare di inserire la seguente istruzione:

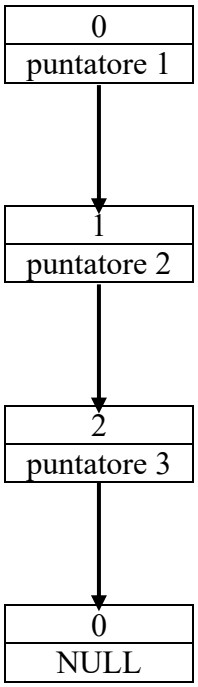
```
delete puntatore;
```

L'istruzione **non elimina il puntatore** ma dealloca il valore puntato dal puntatore.

Con l'allocazione dinamica bisogna prestare attenzione a **non cambiare o modificare il puntatore** che punta alla variabile nella memoria heap, altrimenti si può inciampare in un **memory leak**. Un memory leak è un errore che deriva dall'**allocazione di più memoria di quanta ce ne sia a disposizione**, rendendo impossibile allocare nuove variabili; **perdendo il puntatore** a quell'elemento si disperde il valore della variabile in memoria e non è possibile **né riprenderlo né utilizzare quello spazio**. Tale errore moltiplicato per la dimensione del programma può condurre ad un memory leak.

È possibile dichiarare anche un vettore di elementi con l'istruzione `new`; tuttavia, i singoli elementi non verranno allocati in posizioni contigue come per un comune vettore ed è perciò che entra in gioco il concetto di **linked list**.

Una linked list, o **lista a puntatori**, è un **vettore** che fa uso dell'**allocazione dinamica** per poter allocare una **quantità di elementi che va di pari passo con l'esigenza del programma**. Fino ad ora è stata usata sempre una **versione statica** di un vettore, inizializzato con la dimensione pari al caso peggiore, **allocando più memoria di quanto il programma potrebbe usufruirne**. Per evitare questo spreco di memoria si utilizzano le liste a puntatori. Una lista a puntatori si compone di un **blocco essenziale** costituito da una **variabile allocata dinamicamente** e un **puntatore** che punta al prossimo elemento della lista:



Questo vettore equivale al seguente allocato staticamente:

0	1	2	3

Si possono già notare delle differenze:

- Per la lista a puntatori **si allocano il doppio delle variabili** ma si ha la possibilità di aumentare la dimensione del vettore;
- Per il vettore statico si alloca una **quantità finita di variabili** ma non si allocano i loro puntatori, che possono essere estrapolati dalla testa del vettore.

Bisogna notare che la **testa della lista a puntatori** è identificata da un **blocco che non è puntato** da niente e la **coda da un blocco che non punta a niente**.

Nella pratica ogni elemento della lista sarà una struct di questo tipo:

```
struct elemento{
    int valore;  // Valore
    struct elemento* prossimo; // Puntatore
}
```

Le **liste a puntatori** possono eseguire, sebbene con un grado di difficoltà maggiore, **tutti i compiti** che possono eseguire i **vettori statici**.

TIPI DI DATO ASTRATTI

Per **implementare un algoritmo** risolutivo è necessario effettuare la **specifica sulle operazioni e sui dati**, cioè scegliere e determinare quali passi e con che tipo di dati risolvere un determinato problema. È stato già detto infatti che:

ALGORITMI + STRUTTURE DI DATI = PROGRAMI

Dunque, **la scelta sul tipo di dato da utilizzare è fondamentale** al fine di sviluppare un algoritmo esaustivo a risolvere il problema; tuttavia, **è necessario un adeguato livello di astrazione che separi il *come* dal *cosa*** e che tenga **nascosti alcuni livelli non essenziali** alla comprensione del sistema stesso (ad esempio per usare una lista a puntatori è necessario sapere che esistono determinate operazioni ma a fini pratici può essere nascosto il meccanismo con cui funzionano).

Un **tipo di dato astratto** (ADT) è lo strumento che permette di esprimere i dati in maniera più **indipendente** possibile **dalle caratteristiche del linguaggio di programmazione**, è un **modello matematico** che permette di descrivere tutte quelle **strutture di dati** che mostrano un **comportamento comune** ed è uno dei cardini su cui si basano i moderni **linguaggi di programmazione orientati agli oggetti**.

Il tipo di dato astratto è **indirettamente definito dalle operazioni che si possono compiere su di esso e dai vincoli che tali operazioni devono rispettare**; infatti, definire un ADT significa **definirne le proprietà e le operazioni ammissibili**, ovvero **astrarre una specifica implementazione** rendendo il dato accessibile solo attraverso un'**interfaccia**. L'oggetto deve essere **manipolabile solo attraverso le operazioni** su esso definite e **non è possibile accedere all'oggetto nella sua interezza** in maniera diretta.

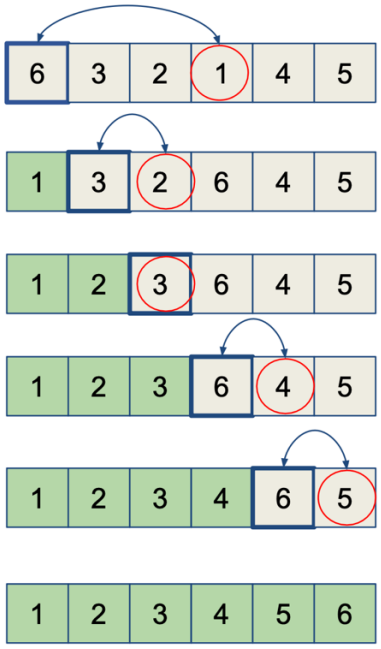
Tale processo viene definito **incapsulamento** e permette il cosiddetto **information hiding**, cioè la possibilità di **nascondere** e **proteggere** alcune informazioni superflue all'uso dell'ADT.

ALGORITMI DI ORDINAMENTO

Per **algoritmo di ordinamento** si intende una sequenza finita di passi che, **dato un insieme di numeri ordinabili, restituisce lo stesso insieme ordinato in maniera crescente**. Esistono diversi algoritmi di ordinamento ma i più importanti sono il **selection sort** e l'**insertion sort**.

Il selection sort è molto semplice e intuitivo e si compone dei seguenti passi:

1. Si cerca l'**elemento più piccolo** del vettore;
2. Si **scambia** tale elemento **con il primo** del vettore;
3. Si **ripetono** i passi precedenti sul sotto-vettore rimanente.



L'**insertion sort** è lo stesso algoritmo usato per **ordinare** quotidianamente un **mazzo di carte**, infatti la **k-esima iterazione** dell'algoritmo vede **i primi k elementi già ordinati** tra loro, rimuovendo l'elemento dal **sotto-vettore non ordinato** e posizionandolo nella corretta posizione di quello ordinato.

Per implementare tale algoritmo s**i prende in considerazione il primo elemento del vettore come sotto-vettore ordinato**, utilizzando **due puntatori** (uno che punta all'**elemento da ordinare** e uno che punta al suo **precedente**): se l'elemento da ordinare è **minore** del secondo, i due si **scambiano**, **altrimenti** i due puntatori passano al **prossimo** elemento.