PROGRAMMAZIONE I Stefan Andrei

Programmazione I (M-Z) - A.A. 2019/20

(Obiettivi e programma del corso, modalità esami e prove itinere, testi consigliati) H.M. Deitel, P. J. Deitel, C++ Fondamenti di programmazione - Maggioli Editore (2014). Eckel, Thinking in C++, Vol. I, 2°Ed. (disponibile gratuitamente online) Horstmann, C++ for everyone, 2°Ed. - Wiley Stroustrup, Programming: Principles and Practice Using C++.

https://www.dmi.unict.it/farinella/Prog1/

ALGORITMI
- Codifica dell'algoritmo

DIAGRAMMI DI FLUSSO
- Tipi di blocchi

COSTRUTTI DELLA NOTAZIONE LINEARE STRUTTURATA (NLS)

ARRAY
- Monodimensionali
- Bidimensionali
- K-dimensionali

LINGUAGGI

PARADIGMI DI PROGRAMMAZIONE

LINGUAGGI DI PROGRAMMAZIONE

TRADUZIONI
- Compilatori: PRO e CONTRO
- Interpreti: PRO e CONTRO
- Java Virtual Machine (JVM)
- Just in Time Compiler (JIT)
- Innovazioni in C++
- C vs C++

CONCETTI BASE DELLA OOP

VARIABILE IN C++
- **Assegnamento in C++**
- Somma in C++
- Incremento/decremento prefisso/postfisso
- Commenti su righe // e fra le righe /* */

SISTEMI DI NUMERAZIONE
- Troncamento
- Overflow e underflow

CODIFICA STANDARD (32 bit)
- Decimale → Floating Point
- Floating Point → Decimale

RANGE DEI TIPI DI VARIABILE
- sizeof()

#include <climits>

STRINGHE
FLUSSI STANDARD
CONTROLLI CONDIZIONALI IF/ELSE
    Operatori relazionali
SINTASSI vs SEMANTICA
OPERATORE CONDIZIONALE
CONFRONTI LESSICOGRAFICI
COSTRUTTO SWITCH
HAND TRACING
OPERATORI LOGICI C++
COSTRUTTI DI CICLO C++
    Costrutto for
    Costrutto do while
    Istruzione break
    Istruzione continue
GESTIONE ERRORI DI I/O C++
    Metodo basic_ios.fail()
        Operatore "!"
    Metodo basic_ios.clear()
    Metodo basic_ios.ignore()
ARRAY IN C++
ARRAY MULTIDIMENSIONALI
FILE I/O
    Apertura di un file
        Aprire un file in lettura con ifstream(metodo costruttore)
        Aprire un file in lettura con ifstream (metodo open())
        Aprire un file in scrittura con ofstream
        Aprire un file in lettura e/o scrittura con fstream
        Aprire un file in lettura/scrittura con fstream con bitwise '|'
    Scrivere su un file + controllo errore
        Scrivere su un file mentre controllo errori
    Lettura di un file + controllo errore
        Lettura da file mentre controllo errori
    Chiusura del file
GENERAZIONE NUMERI PSEUDO-CASUALI
    Generazione numeri casuali in un intervallo [a,b]
        Generare numeri fra [0,1] in virgola mobile
        Generare numeri fra [min,max] double
PUNTATORI IN C++
    Array vs puntatori
    Puntatori mediante indice come gli array
    Aritmetica dei puntatori
ACCESSO AI VALORI DI UN ARRAY
    Errore aritmetica dei puntatori
    Addizione\sottrazione e assegnamento
    Inizializzazione di puntatori
VALORE VS INDIRIZZO
PUNTATORI A COSTANTI

PUNTATORE COSTANTE
PUNTATORE COSTANTE AD UNA COSTANTE
FUNZIONI IN C++
PROTOTIPO VS DEFINIZIONE DI FUNZIONE
    Compilazione di moduli
PARAMETRI FORMALI VS PARAMETRI ATTUALI(R)
AREA STACK (pila) e RECORD DI ATTIVAZIONE
PARAMETRO ATTUALE COME INDIRIZZO O VALORE
ALLOCAZIONE AUTOMATICA
ALLOCAZIONE DINAMICA
ALLOCAZIONE STATICA
PASSAGGIO DI PARAMETRI ARRAY
    Array multidimensionali
    Indirizzo o contenuto di una cella di memoria ed equivalenze
ALLOCAZIONE IN MEMORIA (r)
    Operatore new
FUNZIONI CHE RESTITUISCONO UN PUNTATORE
malloc() e free()
OGGETTI STRINGA
    Array di caratteri → Tipo numerico
        `<cstdlib>`
        `sscanf`
        `ssprintf()`
        `<cctype>`
    Operatore [] vs metodo at()
PROGRAMMAZIONE AD OGGETTI
    Stato
    Comportamento
    Ciclo di vita
CLASSI VS OGGETTI
    Programmazione procedurale vs programmazione ad oggetti
AGGREGATI E COLLEZIONI DI OGGETTI
    Relazioni part-of
    Array di oggetti
IMPLEMENTAZIONE CLASSI
    Modificatore di accesso
    Flusso di invocazione di metodi
    Passaggio per valore
    Passaggio per puntatore
    Passaggio per riferimento
    Argomenti nella funzione main
FUNZIONE INLINE E CONSTEXPR
MODIFICATORI DI ACCESSO PER I MEMBRI DI UNA CLASSE
IMPLEMENTAZIONE COSTRUTTORI
REFERENCES
STATIC CONST E FRIEND
    Static
    Const

Friend
NAMESPACE E OVERLOADING DEI METODI
CREAZIONE,COPIA,DISTRUZIONE DI OGGETTI
Creazione
Copia
Distruzione
OVERLOADING DEGLI OPERATORI
Esempi di overload di alcuni operatori
EREDITARIETÀ
Overriding
Polimorfismo
RTTI: Run Time Type Identification
FUNZIONI TEMPLATE
CLASSI TEMPLATE
RICERCA, ORDINAMENTO E RICORSIONE
Ricerca dicotomica
Ordinamento
Bubble Sort
Selection Sort
Insertion Sort
Ricorsione

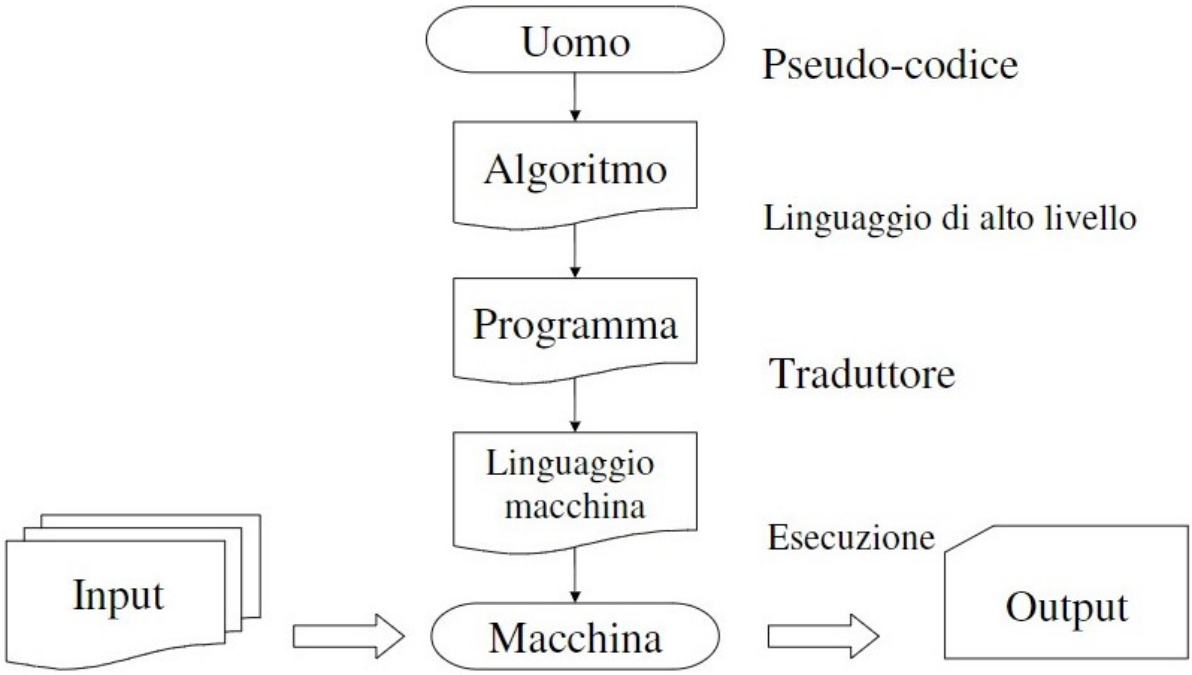
ALGORITMI

Un **algoritmo** è una **sequenza di passi** concepita per essere eseguita automaticamente da una macchina in modo da risolvere un problema dato.

Caratteristiche:

- sequenza di passi **FINITA;**
- viene eseguito sempre **IN ORDINE** e in un **TEMPO FINITO;**
- se è *risolvibile* viene chiamato **COMPUTABILE;**
- azioni **eseguibili** e non **ambigue;**
- **Determinismo;**
- **Terminazione**.

Codifica dell'algoritmo



Ogni problema può essere diviso in *sotto-problemi* e l'approccio può essere:

1. **TOP-DOWN**: si costruisce una visione generale del problema senza scendere nei dettagli;
2. **BOTTOM-UP**: si prendono i sotto-problemi e vengono uniti tra loro per formare parti di problema più grandi;
3. **IBRIDO**: Unione dei punti 1 e 2.

ESEMPIO:

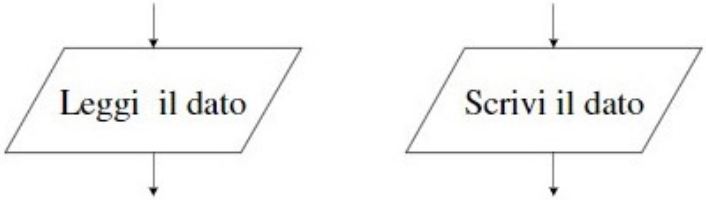
> *Stampare tutti i nomi di persona presenti in un testo.*
> 1. (TD) Leggere il testo, riga per riga, separando le singole parole.
>   • (BU) Usare la funzione **getLine()**.
>   • (BU) Usare la funzione **getWords()** sull'intera riga.
> 2. (TD) Memorizzare ogni nome di parola quando esso viene letto.
> 3. (TD) Stampare tutti i nomi di parola memorizzati.
>   • (BU) Usare la funzione print() su tutte le parole.

DIAGRAMMI DI FLUSSO

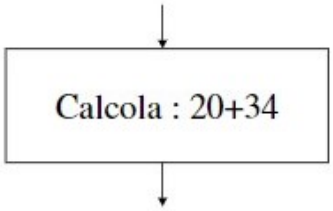
Tipi di blocchi

- Inizio/fine;

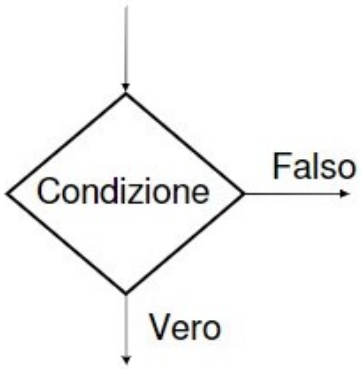
- Input/output;



- Istruzioni imperative e/o operazioni;



- Istruzione condizionale.



COSTRUTTI DELLA NOTAZIONE LINEARE STRUTTURATA (NLS)

Sequenza: Equivalente ad uno o più blocchi di operazioni che si susseguono.

```
Istruzione 1
Istruzione 2
```

```
Istruzione 3
```

Selezione: Si applica una ***condizione***. Se tale condizione è vera si prosegue su un'istruzione, altrimenti, se è falsa, si prosegue su una seconda istruzione, finendo comunque sullo stesso punto.

```
if (condizione) then
    Istruzione 1
else
    Istruzione 2
end if
```

Iterazione (o ciclo): Si applica, anche qui, una ***condizione***. Se tale condizione è vera si entra nel ciclo e si eseguono le istruzioni. Alla fine **si torna** a valutare la **condizione**. Si esce dal ciclo quando la condizione è falsa.

```
while (condizione) do
     Istruzione 1
end while
```

TEOREMA di Böhm-Jacopini): I tre costrutti fondamentali della NLS sono sufficienti a descrivere ***qualunque*** algoritmo.

ARRAY

Monodimensionali

Un array è una struttura di dati **omogenea** che funge da **contenitore** di elementi dello stesso tipo, per esempio solo numeri, solo stringhe ecc..

Gli indici ***i*** partono da 0 (ne linguaggio C\C++) e finiscono a ***DIM-1***.

Per richiamare un elemento all'interno di un array A si usa la notazione A**[i]**.

> *Per esempio A[2] si riferisce all'elemento presente all'indice 2, cioè al **terzo** elemento*

Bidimensionali

Hanno 2 dimensioni, ovvero **RIGHE** e **COLONNE**. Si indica solitamente con **NxM** dove:

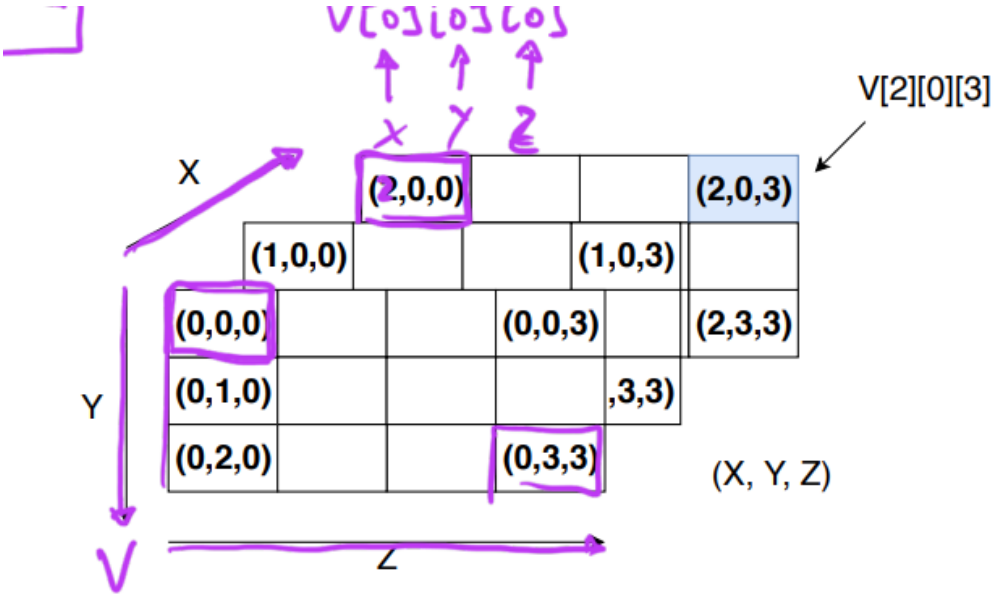
N=righe

M=colonne

Assegnamento di un valore ad una cella: ***A[3][4] = 3;***

K-dimensionali

Operazione modulo %. Fare una divisione tra 2 numeri e considerare il valore del resto



Un array **k-dimensionale** è una struttura dati che **si estende su k dimensioni**.

Nel caso sopra abbiamo un array di tipo X x Y x Z dove:

- X è il numero di **piani**
- Y è il numero di **righe**
- Z è il numero di **colonne**

Per prendere in considerazione un elemento di un array k dimensionale A, si usa la notazione **A[X][Y][Z]**

> *Solitamente si fanno 3 cicli while annidati quando bisogna lavorarci su.*

LINGUAGGI

I linguaggi sono classificati in *3 livelli*:

- linguaggio `macchina` (eseguito dal calcolatore);
- linguaggio `assembly` ;
- linguaggio di alto livello e si evolvono verso: `astrazione` , `semplificazione` e una `similarità con il ragionamento umano` .

PARADIGMI DI PROGRAMMAZIONE

Un paradigma di programmazione non è altro che la ***filosofia con cui si scrivono i programmi***. Essa caratterizza;

- **metodologia** con cui si scrivono i programmi (strutture di controllo o procedure);
- **la computazione.**

Ogni linguaggio, generalmente, si basa su un paradigma di programmazione.

Un paradigma di programmazione può essere di diversi **tipi**:

1. Programmazione `funzionale`: serie di **funzioni matematiche** che richiamano dei pezzi di codice. E' più facile la verifica della correttezza del codice, meno errori però poco vicino al nostro modo di concepire i sistemi
2. Programmazione `logica`: descrive la **struttura logica** del problema. Il programma si concentra sugli **aspetti logici del problema**.

E questi sono poco diffusi.

Altri tipi di paradigmi di programmazione sono:

1. Programmazione `imperativa`: sequenza di **istruzioni impartite** al calcolatore, per esempio "leggi A", "Stampa A", "A=0"
2. Programmazione `strutturata`: si basa sul fatto che per scrivere qualsiasi programma abbiamo bisogno dei **tre costrutti** *sequenza*, *selezione* e *iterazione*;
3. Programmazione `procedurale`: il codice sorgente è suddiviso in una serie di blocchi identificati da un nome e sono usati per risolvere il problema. **Side-effects**= data una variabile x, per passare il valore di x, **la passo per via *dell'indirizzo*** del dato stesso **e non per *valore***. Intervengo su quell'area di memoria mediante l'indirizzo.
4. Programmazione `modulare`: i moduli sono separati dalle **interfacce**. Possono essere visti come dei **blocchi** che formano dei sotto-problemi di macro-problemi. Si usano questi moduli cambiando semplicemente i dati in input;
5. Programmazione `orientata agli oggetti` **OOP**: **include** aspetti di programmazione *imperativa*, *strutturata*, *procedurale* e *modulare*. Migliora **l'efficienza** del processo di **produzione** e **mantenimento** del software, favorisce la **modularità** e il riutilizzo delle componenti.

LINGUAGGI DI PROGRAMMAZIONE

I linguaggi di programmazione sono stati introdotti per ***facilitare la scrittura dei programmi.***
Sono definiti da un insieme di regole formali, le regole grammaticali o **sintassi**.
La sintassi include la specifica di tutti i ***simboli da usare*** durante la scrittura di un programma.

La **semantica** di un'istruzione definisce il **significato logico dell'istruzione**.

TRADUZIONI

Il linguaggio di alto livello è quello `comprensibile` e piu vicino al linguaggio umano.

linguaggio di basso livello è `meno comprensibile` (assembly) più vicino al linguaggio macchina.

```
Load ACC, var1
Add ACC, var1
Store tot, ACC
```

Grazie alla traduzione si ha un evoluzione dei linguaggi verso sistemi simbolici piu **espressivi.** Si è piu agevoli nel scrivere programmi vista anche l'indipendenza dal sistema sul quale si scrive il codice.

Il traduttore genera un linguaggio macchina a partire da un codice scritto di alto livello

I `traduttori` sono di due tipi:

- **interpreti**, il programma non viene memorizzato. E' "**monouso**";
- **compilatori**, il programma rimane tradotto in memoria.

> *Alcuni linguaggi compilati sono: C++, Pascal, Cobol*

- L'interprete prende il codice sorgente e traduce la singola istruzione generando il corrispondente codice macchina, viene eseguito e poi si passa all'istruzione successiva

Compilatori: PRO e CONTRO

`PRO`

1. **efficienza**, occupano meno spazio;
2. **performance**, il codice sorgente è già tradotto;
3. **non invasivo** perchè **non installato** su macchine di produzione.

`CONTRO`

Il codice è generato per una **determinata architettura** e non per architetture diverse.

Interpreti: PRO e CONTRO

`CONTRO`

1. **performance**, viene **tradotto riga per riga** ogni volta;
2. **efficienza**, occupa **molte risorse** sulla macchina;
3. **invasivo** perchè **è installato** sulla macchina di produzione.

`PRO`

1. **Non è importante** l'architettura della macchina.

Java Virtual Machine (JVM)

Il codice il linguaggio Java (**bytecode**) viene scritto dal programmatore, viene letto da JVM ed esso eseguito sulla macchina di produzione.

L'interpretazione del Bytecode comporta un overhead (sovraccarico) rispetto all'esecuzione del codice macchina.

Just in Time Compiler (JIT)

Il compilatore Just in Time JIT è stato concepito per ***ridurre l'overhead*** di JVM.

Un metodo eseguito dalla JVM viene compilato in codice macchina dal JIT per poter essere seguito in seguito senza interpretazione.
Per fare questa operazioni il JIT fa uso di molte risorse.

Innovazioni in C++

> ***Prototipazione*** *del software è quella che va prima della produzione, tipo il test del prototipo.*

`ASSEMBLY`:

1. Sostituzione numeri con le parole;
2. Il programmatore deve conoscere l'architettura della macchina;
3. I dati si spostano dalla memoria ai registri della CPU;

`PROCEDURALI`:

1. Viene nascosta la complessità delle operazioni sui dati;
2. Si modella un diagramma di flusso che rappresenta il movimento dei dati tra le varie funzioni;
3. Le funzioni sono su blocchi di dati

`OOP`:

1. Nascondono i dati e le complessità del programma. Per richiamare determinati dati bisogna usare una determinata sintassi;
2. Il sistema è formato da oggetti(definiti da delle caratteristiche, per esempio l'oggetto 'persona' o la classe 'persona');
3. Scambio di messaggi fra oggetti attraverso le interfacce.

C vs C++

Con c++ la progettazione di interfacce grafiche è più agevole

• In C non esiste reale **incapsulamento**, quindi la
modifica dei dati influenza anche altre parti del programma, introducendo maggiore probabilità di errore.
• C++ favorisce la modularità, creare blocchi che tra loro creano il programma;
• C++ permette di scrivere programmi maggiormente mantenibili.

CONCETTI BASE DELLA OOP

Una **classe** è una caratteristica di alcuni dati accomunati da elementi **comuni**.

I membri della classe possono essere pubblici o privati. così che dall'esterno non possono essere richiamati.

L'**incapsulamento** permette di **nascondere** alcuni dati e metodi agli utenti utilizzano una determinata classe.

Caratteristiche:

1. `Ereditarietà di classi` prendere una classe generale già esistente ed estenderla su delle classi derivate più specifiche ad essa. In questo caso le caratteristiche della classe derivata avranno automaticamente le caratteristiche della classe di base;
2. Riutilizzo di codici già scritti
3. non c'è bisogno di riscrivere l'intero codice

Polimorfismo: Fare più di una cosa con un pezzo di codice singolo.

VARIABILE IN C++

> **Una variabile è una sorta di contenitore di dati identificato da un nome all'interno di un programma.**

Una variabile è associata ad un indirizzo specifico nella memoria del calcolatore nel quale si esegue il programma

```
DEFINIZIONE + INIZIALIZZAZIONE
int numero.di.ordini = 10; //definizione inizializzata
int numero.di.ordini; //definizione non inizializzata
int v1, v2, v3; //dichiarazione di più variabili
-----------------------------------------------------
TIPI
int A; //variabile di tipo Intero positivo o negativo
short int a=c; //variabile di tipo Intero Corto
const int i=0;//variabile costante che non cambia più
/*
const indica che la variabile non verrà mai modificata durante l'intero programma
*/
char c='c'; // variabile di tipo Carattere
boolean a=true; //variabile booleana (true o false)
double a=1.0; //variabile con la virgola
string a="acqua"; //variabile di tipo stringa
cin >> A;//il valore digitato in input verrà assegnato ad A
cout << A;//verrà stampato A
```

Con il carattere ***;*** termina la singola istruzione.

ERRORI

```
int codice= "AA10"; //errore perchè l'assegnazione non è intera
int var;
int double;
int int;
/*
non si possono usare nomi di variabili uguali ai nomi delle
funzioni del programma.
*/
```

Regole dei nomi delle variabili:

1. devono **iniziare** con una **lettera** oppure con un _
2. i rimanenti caratteri possono essere lettere o numeri e non devono esserci spazi
3. per fare **spazi** fra parole si può mettere _
4. **NON** si possono usare parole tipo **double**, **int**, e altri nomi di funzioni

Assegnamento in C++

Sintassi:

```
i=0;
i=a+b;
i=(a+b)/2;
```

Somma in C++

Sintassi: $a = a + b \iff a+ = b$

Incremento/decremento prefisso/postfisso

```
ESEMPIO con a=1
b=a++; b=2
b=++a; b=1
```

`Forma Prefissa ++a` Incremento → Assegnamento

`Forma post fissa a++` Assegnamento(senza incremento) → Incremento

Commenti su righe // e fra le righe /* */

```
// COMMENTO SU RIGA

/*

COMMENTO FRA RIGHE

*/
```

SISTEMI DI NUMERAZIONE

Conversione da base 2 a base 10. Numeri interi

$$\mathbf{10101010}_2 = 1 \times 2^7 + 0 \times 2^6 + 1 \times 2^5 + 0 \times 2^4 + \\ + 1 \times 2^3 + 0 \times 2^2 + 1 \times 2^1 + 0 \times 2^0 = 170_{10}$$

Da base 2 a base 10. Numeri con parte frazionaria

Conversione in base 10 del numero $\mathbf{101.0101}_2$

$$\mathbf{101}_2 = 1 \times 2^2 + 0 \times 2^1 + 1 \times 2^0 = 5_{10}$$

$$\mathbf{0101} = 0 \times 2^{-1} + 1 \times 2^{-2} + 0 \times 2^{-3} + 1 \times 2^{-4} = \\ = \mathbf{0.3125}_{10}$$

$2^{-1} = \frac{1}{2}$ $2^{-2} = \frac{1}{4} = \frac{1}{2^2}$

Conversione da base 10 a base 2. Numeri interi

$136_{10} = \mathbf{10001000}_2$.

136 : 2	=	68	r=**0**
68 : 2	=	34	r=**0**
34 : 2	=	17	r=**0**
17 : 2	=	8	r=**1**
8 : 2	=	4	r=**0**
4 : 2	=	2	r=**0**
2 : 2	=	1	r=**0**
1 : 2	=	0	r=**1**

Da base 10 a base 2. Numeri con parte frazionaria

Il numero $28.125_{10} = 11100.001_2$

Parte intera (si converte nel solito modo): $\mathbf{28_{10} = 11100}$

Parte frazionaria:

0.125×2	$=$	0.250	riporto **0**
0.250×2	$=$	0.500	riporto **0**
0.500×2	$=$	1.000	riporto **1**
0.000×2	$=$	0.000	FINE

$\mathbf{0.125_{10} = 0.001_2}$

Troncamento

Ci sono numeri la quale parte frazionaria non è rappresentabile in bit se essa è infinita. Per questo motivo si applica il **troncamento all'ottava cifra**.

Per rappresentare i numeri interi si usa il **valore assoluto** ed un eventuale segno. Per rappresentare il segno si **sacrifica un bit,** che verrà rappresentato con **0**, se il valore è **positivo**, e con **1**, se il valore è **negativo**. Quando si rappresenta anche il segno, il modulo (cioè i numeri rappresentabili) è dimezzato perchè metà saranno con il segno +, quindi positivi, e metà con il segno -, quindi negativi.

Numeri interi senza segno

Codifica di Numeri interi **senza segno** con 16 bit:
Intervallo numerico: $[0, 2^{16} - 1] = [0, 65.535]$ 65 536

Codifica di numeri interi **senza segno** con 32 bit:
Intervallo numerico: $[0, 2^{32} - 1] = [0, 4294967295]$

Numeri interi con segno

Codifica di numeri interi **con segno** a 16 bit:
Intervallo numerico $\pm(2^{15}-1) = 32767$ (1 bit per il segno)

Codifica di numeri interi **con segno** a 32 bit:
Intervallo numerico $\pm(2^{31}-1) = 2.147.483.647$ (1 bit per il segno)

Overflow e underflow

Se si esce fuori dagli intervalli "permessi", limiti di rappresentazione, si parla di overflow o underflow, rispettivamente se supera il limite per eccesso o per difetto.

CODIFICA STANDARD (32 bit)

In notazione scientifica i numeri con la virgola possono essere rappresentati anche come numeri di tipo 0.515 per la base elevata a una determinata potenza. Per esempio *5.15* decimale può essere scritto come $0.515 \cdot 10$.

Esempio in base 10

a) $96.103 = 0.96103 \times 10^{+2}$
b) $2.96 = 0.296 \times 10^{+1}$
c) $2.96 = 29.6 \times 10^{-1}$

> *E' possibile trasformare un numero con la virgola in un numero di tipo* $0.165651 \cdot 10^2$\

Da qui si distinguono **3 parti** di un numero in **notazione scientifica**:

- **mantissa(m)** è tutta quella parte dopo la virgola ed è codificata con 23 bit;
- **esponente(e)** è l'esponente del fattore con la base del numero ed è codificato con **8 bit**;
- **segno(s)** è il segno del numero in notazione scientifica ed è codificato con **1 bit**.

> *Lo standard IEEE 754 definisce il formato per la rappresentazione dei numeri in virgola mobile:*

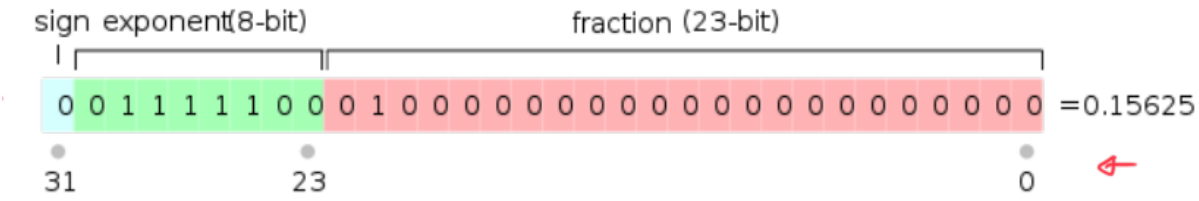
> *• 1 bit per la rappresentazione del segno (s);*
> *• 8 o 11 bit per la rappresentazione dello esponente (E);*
> *• 23 o 52 bit per la rappresentazione del significando o mantissa (M);*

Decimale → Floating Point

> *Esercizio:*
> Convertire -11.25 in floating point
> ***s=1 (numero negativo)***
> *11=1011 (rappresentato in valore assoluto)*
> *0,25=01*
> *Numero in binario è 1011.01*
> *In notazione scientifica binaria (forma normale):* $1.01101 \cdot 2^3$
> ***m=01101 (+ gli 0 fino ad arrivare 23 bit)***
> *e'=e+127 = 130*
> *e' = in binario* ***10000010 (8 bit)***
> *In Floating Point è 1 10000010 01101(00000000000000000)*

Floating Point → Decimale

> Esercizio:
> Trasformare in decimale il numero che è in floating point *1 01111110 10 (00000000000000000000)*
> *s=1*
> *e'=01111110*
> *m=10*
> *in notazione scientifica è*
> $s, m \cdot 2^e$
> *e'=e-127=126-127=-1*
> $1,1 \cdot 2^{-1}$
> *Il numero in binario è -0.11=-1,1 x2^-1*
> *In decimale sarebbe 2^-1 + 2^-2= 0.5+0.25=****-0.75***



RANGE DEI TIPI DI VARIABILE

Dimensione e range sono valori tipici!

Tipo	Range (tipico)	Precisione	Dimensione
int	$\pm 2.147.483.647$	–	4 bytes
unsigned	$[0, 4.294.967.295]$	–	4 bytes
long	$\pm(2^{63}-1)$	–	8 bytes
unsigned long	$[0, 2^{64}-1]$	–	8 bytes
short	± 32768	–	2 bytes
unsigned short	$[0, 65535]$	–	2 bytes
double	$\pm 10^{308}$	15 cifre	8 bytes
float	$\pm 10^{38}$	6 cifre	4 bytes

sizeof()

In C/C++ esiste l'operatore **sizeof()**, che si usa come una fuinzione con argomento il particolare tipo o anche una variabile di un certo tipo. Serve per avere, come risultato dell'operazione, la **dimensione in bit dei vari tipi di variabili.**

#include <climits>

Intervalli numerici e precisione

C++ eredita dal C gli header `float.h` e `limits.h`

```
#include <climits> // header da includere
INT_MIN, INT_MAX // più piccolo/grande valore intero
LONG_MIN, LONG_MAX // più piccolo/grande valore long
```

```
#include <cfloats> //header da includere
// più piccolo/grande valore float positivo
FLT_MIN, FLOAT_MAX
// no.  di cifre significative rappresentabili
// (precisione!)
FLT_DIG, DBL_DIG
```

STRINGHE

#include <string> ← Libreria da includere

`.substr(x,y)` x=indice punto inizio sottostringa, y=indice fine sottostringa
Se la y viene omessa, il comando prenderà in considerazione tutto quello che segue la x.
`.length()` dà come valore la lunghezza della stringa, cioè il numero di caratteri, inclusi spazi.

FLUSSI STANDARD

1. Standard error `cerr` è associato al video. L'errore va visualizzato a video.
2. Standard input `cin`, permette di inserire da tastiera un valore della variabile.

`cout`, `cin`, `cerr` sono oggetti ed hanno *scope globale*: il programmatore può usarli all'interno del suo programma senza doversi preoccupare di inizializzare alcunchè.

L'operatore + sulle stringhe in cout è:

```
string all_names = name + "and" + your name;
cout << "My name and your name is " << \
all_names << endl;
```

```
//Sintassi Input/output
#include <iostream>

using namespace std;
```

```
int main () {
  int numero;
      cout << "Inserisci un numero: " << endl;
      cin >> numero;
}
```

CONTROLLI CONDIZIONALI IF/ELSE

Il costrutto **if/else** in C++ analizza la condizione e fa un'operazione in base alla veridicità della condizione.

```
// Sintassi IF
if (...) {
  cout << ...;
}
else {
  cout << ...;
}
```

Operatori relazionali

```
In C++ la sintassi degli operatori relazionali è:
< minore
<= minore o uguale
> maggiore
>= maggiore o uguale
== uguale
!= diverso
```

SINTASSI vs SEMANTICA

La **sintassi** riguarda la **compilazione del programma**. Se il programma compila allora la sintassi corretta.
La **semantica** riguarda il risultato del compilatore, cioè il **significato logico** del risultato ottenuto dopo la compilazione del programma. Se il programma compila e dà un risultato non aspettato, allora la semantica è errata.

OPERATORE CONDIZIONALE

Sintassi `(cond1? expr1 : expr 2);`

se **cond1** è **vera**

allora valuta **expr1**

altrimenti valuta **expr2**

```
cout << "Max (x,y)" << (x>y ? x : y) << endl;
//è lo stesso di
if (x>y) {
    cout << "Max=" << x << endl;
}
else {
    cout << "Max=" << y << endl;
}
```

CONFRONTI LESSICOGRAFICI

Questi tipi di confronti vanno a confrontare **l'ordine alfabetico** di una stringa o di un carattere char.

Questo confronto **NON** confronta la lunghezza della stringa o del carattere ma confronta quale stringa ha la **precedenza nell'ordinamento dei caratteri**.

Date 2 stringhe S1 ed S2, il compilatore analizza le stringhe prendendo il primo carattere della prima stringa e confrontandolo con il corrispettivo carattere della seconda stringa. (S2[1] vs S2[1] → S2[2] vs S2[2] → ecc..). La terminazione di questo confronto avviene quando trova delle lettere diverse fra loro e queste vengono messe a confronto.
Se trova un carattere in S1 minore di un carattere corrispettivo in S2, allora S1<S2.

Se avessimo s1=gianpaolo e s2=gian, la minore sarà considerata s2 perchè è più corta.

```
string myname="Pippo";
string yourname="Paperino":
string selected;
if (myname<yourname)
  selected=myname;
else
  selected=yourname;
OUTPUT: selected=Paperino perchè Paperino viene prima di Pippo.
```

Quando si confrontano due caratteri fra loro, si confronta il codice in codice ASCII.

> Per esempio `('A'<'a')` con `A=65` e `a=97` allora A<a **VERO**
> *vale anche ("123prova" < "Abaco") è VERO perchè i numeri, in ASCII, hanno un codice di codifica minore*

COSTRUTTO SWITCH

```
1   if (digit == 1) { digit_name = "one"; }
2   else if (digit == 2) { digit_name = "two"; }
3   else if (digit == 3) { digit_name = "three"; }
4   else if (digit == 4) { digit_name = "four"; }
5   else if (digit == 5) { digit_name = "five"; }
6   else if (digit == 6) { digit_name = "six"; }
7   else if (digit == 7) { digit_name = "seven"; }
8   else if (digit == 8) { digit_name = "eight"; }
9   else if (digit == 9) { digit_name = "nine"; }
10  else { digit_name = ""; }
```

Tanti if annidati possono creare problemi per quanto riguarda la leggibilità e l'eleganza del programma.

Per fare quest'operazione si può utilizzare uno switch come segue:

```
1   switch (digit)
2   {
3     case 1:
4       digit_name = "one";
5       break;
6     case 2:
7       digit_name = "two";
8       break;
9     // altri case ...
10    default:
11      digit_name = "NO DIGIT";
12      break;
13  }
```

Lo switch **lavora** nel seguente modo:

1. valuta **l'argomento** che ha tra parentesi;
2. confronta l'argomento con **case 1**

3. se l'argomento e il case **sono uguali**, allora vengono eseguite le istruzioni presenti sotto il caso indicato
4. se l'argomento e il case non sono uguali, si va al case successivo
5. se non trova un case uguale all'argomento di partenza va a eseguire le istruzioni nella categoria **default**.
6. dopo aver eseguito un case, per uscire dalla continua verifica dei case si usa il **brake**.

HAND TRACING

E' una tecnica che consiste nello scrivere una tabella con le variabili note in un programma e, seguendo il codice, seguire i cambiamenti di quest'ultime. Serve per capire il funzionamento del codice.
E' come se si facesse un **debug** manuale.

OPERATORI LOGICI C++

- AND in C++ è && (operatore binario)
- OR in C++ è || (operatore binario)
- NOT in C++ è ! (operatore unario)
- NAND(A,B)
- NOR(X,Y)
- XOR(A,B) è falso quando A e B sono entrambi falsi o veri ed è vero quando uno dei due è VERA. Questo operatore conta la disparità del valore vero.

```
int a;
if (a!=2 && a!=3) // AND
if (a==2 || a==3) // OR
```

- In un AND la valutazione a corto circuito o di McCarthy si arresta quando è sufficiente la prima expr1 a determinare il valore logico (true o false).
- In un OR la valutazione a corto circuito o di McCarthy si arresta quando l'operando a sinistra dell'OR expr1 a determinare il valore logico (true o false).

Per l'AND e per l'OR valgono le leggi di De Morgan

> **1. !(A && B) ⇔ !A || !B**
> **2. !(A || B) ⇔ !A && !B**
> Queste due leggi sono dimostrabili tramite due tabelle di verità.

COSTRUTTI DI CICLO C++

```
while (condizione){ // se è vera, si esegue l'istruzione
  Istruzione
}
```

> *a=a+b = a+=b*

> Esercizi 11.1 slide
> https://www.dmi.unict.it/farinella/Prog1/Lezioni/11_loop2021.pdf

Costrutto for

```
for (Inizializzazione; condizione; aggiornamento contatore){
  Istruzione
```

Si può utilizzare quando si ha un **numero di iterazioni conosciuto** a priori visto che c'è la parte d'istruzione contenente l'inizializzazione di una variabile che farà da contatore del for.
(l'inizializzazione può essere fatta anche prima del for e all'interno di esso si può fare solo un'istruzione di assegnamento.
L'inizializzazione della variabile all'interno del ciclo for è determinata anche dalla specificazione del tipo di variabile che si sta inizializzando.

La variabile inizializzata **vale solo nel ciclo for**, dopodichè viene eliminata dal programma visto che ha ***scope(visibilità) solo nel ciclo for***.

```
for (int i=0; i<5; i++)
{
  istruzioni;
}
// Esegue l'istruzione 5 volte
```

Costrutto do while

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

In questo costrutto l'istruzione all'interno del do while viene eseguita ALMENO UNA VOLTA.

Esercizi slide H 11.4 ecc https://www.dmi.unict.it/farinella/Prog1/Lezioni/11_loop2021.pdf

Istruzione break

Break permette di uscire da un ciclo se ci dovesse essere un ciclo infinito

```
double capitale=1000.0;
for (int anno=0; anno<N; anno++)
  capitale+=capitale*TASSO_INTERESSE;
  if(capitale>TARGET)
    break;
}
// Se il capitale supera il target prima che anno<N, allora si esce subito dal ciclo perchè si ha
// il break.
```

Istruzione continue

Permette di saltare al prossimo ciclo e saltando tutte le istruzioni che si trovano dopo di esso

```
const int N=20;
for (int i=0; i<N; i++) {
  if (i%2==0)
    continue;
      cout << i << endl;
}
// Stampa solo i numeri dispari
```

GESTIONE ERRORI DI I/O C++

Se l'utente inserisce un valore sbagliato da input, ovvero mette un tipo di input diverso da quello richiesto(mette una stringa al posto di un numero intero richiesto o un float)

E' possibile gestire l'errore con `std:ios_base.fail()`

Metodo basic_ios.fail()

(basic_ios è cin o altri elementi di 'iostream')

Sintassi:

```
float x;
std::cin >> x;
if (cin.fail()){ // ERRORE I/O
  cerr<<"Inserito input non valido"!" << endl;
    return -1;
}
```

`cin.fail()` solitamente è **falso**. Quindi se si ha un errore, esso diventa vero e quindi si verifica il VERO nell'if.

Operatore "!"

```
float x;
if (!(std::cin >> x)) { //restituisce un bool
```

```
  std::cerr<<"Inserito un input non valido" << endl;
  return -1;
}
```

Il metodo basic_ios.fail() dà il valore true quando:

- eofbit end of file
- failbit errore di io, ovvero formattazione o estrazione
- badbit (altri errori)
- goodbit (nessun errore)

Metodo basic_ios.clear()

Si scrive solo dopo aver indicato l'errore con `cin.fail()`, serve per riportare i flag allo stato iniziale. Mette a zero tutti i flag di errore dello stream.

```
if (cin.fail()) {
  cout << "Errore";
cin.clear(); // reset flags
  // ...
}
```

Metodo basic_ios.ignore()

Dopo aver segnalato un errore di input, all'interno della memoria **restano i caratteri inseriti dall'utente che non sono stati richiesti**. Pertanto, per liberare quella piccola porzione di memoria si utilizza il metodo `basic_ios.ignore()`.

`cin.ignore()` serve per scartare caratteri rimasti nello stream a seguito di errore o perchè l'utente ha inserito più stringhe separate da spazi.
La sintassi è: `cin.ignore(numeric_limits<streamsize>::max(), '\n').`

ARRAY IN C++

In un array V si accede con V[3] al dato in posizione numero 3 dell'array.
Un array si definisce nel seguente modo:

```
#define DIM 10
const int dim = 10;

short mydim= 10;
int V1[10];
float V2[DIM]
double V3[dim];
long V4[mydim];

V[7]=4;
V2[0]=6.7;
```

```
// oppure

int V[10]= {1,2,3,4,5,6,7,8,9,10}; //inizializzazionoe completa
int W[10]= {1,2,3,4,5}; //inizializzazione parziale
int Z[]={1,2,3,4,5}; // dimensione array determinata implicitamente!

//oppure

int V[1000]={0}; //inizializzazione tutti a zero
int W[1000]={}; // tutti a zero
int Z[1000]; // non inizializzati
```

ARRAY MULTIDIMENSIONALI

Sintassi della definizione di un array bidimensionale:

```
#define N 3
#define M 4

// matrice di dimensione N x M
int V[N][M] = {0};
int W[N][M] = {};
int Z[N][M] = {1,2,3,4,5,6,7,8,9,10,11,12}; // inizializzazione delle RIGHE.
                        //Ogni N elementi si crea una nuova riga per M volte.

// errore di compilazione per mancanza di M
int V[][]= {1,2,3,4,5,6,7,8,9,10,11,12};

// OK!
int V[][M]= {1,2,3,4,5,6,7,8,9,10,11,12};
```

In generale, per un array bidimensionale è denotato con `[L/M]` → dove L è la lunghezza della lista di inizializzazione, ed M il numero di colonne specificato

https://www.dmi.unict.it/farinella/Prog1/Lezioni/14_IO2021.pdf

FILE I/O

Classi di input\output a caratteri con i file:

- std::`ofstream` → scrittura;
- std::`ifstream` → lettura;
- std::`fstream` → lettura e/o scrittura.

Apertura di un file

Aprire un file in lettura con ifstream(metodo costruttore)

```
#include <fstream>
#include <iostream>
```

```
std::ifstream myfile("test.txt"); //costruttore
//controllo se il file è stato aperto correttamente
if  ( !myfile.is_open() )
  cerr << "Errore apertura file!" << endl;
```

Aprire un file in lettura con ifstream (metodo open())

```
#include <fstream>
#include <iostream>

std::ifstream myfile();
myfile.open("test.txt");

if(!myfile.is_open){
  cerr << "Errore apertura file";
```

Aprire un file in scrittura con ofstream

```
#include <fstream>
#include <iostream>

std::ofstream myfile();
myfile.open("test.txt");

if(!myfile.is_open){
  cerr << "Errore apertura file";
```

Aprire un file in lettura e/o scrittura con fstream

Per `fstream` la **modalità di apertura file va specificata.**

```
#include <fstream>
#include <iostream>
std::fstream myfile();
  myfile.open("test.txt", std::fstream::out); //scrittura sul file

if(!myfile.is_open()){
  cerr << "Errore apertura file";
```

Aprire un file in lettura/scrittura con fstream con bitwise '|'

L'operatore **bitwise** è rappresentato con `|` e si mette nelle modalità di apertura di `fstream`.

```
#include <fstream>
#include <iostream>
std::fstream myfile();
myfile.open("test.txt", std::fstream::out | std::fstream::in); //scrittura/lettura

if(!myfile.is_open()){
  cerr << "Errore apertura del file";
```

Scrivere su un file + controllo errore

E' possibile scrivere su un file aprendolo con **fstream** e metodo costruttore. Per scrivere sul file si utilizza l'operatore di inserimento '`<<`'. Dopo quest'operazione è possibile verificare se la scrittura è avvenuta con successo controllando eventuali errori con il metodo `std::basic_ios.fail()`

```
#include <fstream>
#include <iostream>

fstream myfile("test.txt"), std::fstream::out); //apertura file in scrittura
myfile << "Pippo test"; //scrivo la stringa "Pippo test" sul file "myfile"
//controllo vari errori con
if(myfile.fail())
  cerr << "Errore scrittura sul file";
```

Scrivere su un file mentre controllo errori

E' possibile scrivere sul file e controllare l'errore **contemporaneamente** mediante l'operatore di inserimento '`<<`' e l'operatore '`!`'.

```
#include <fstream>
#include <iostream>
std::fstream myfile();
myfile.open("test.txt", std::fstream::out); //scrittura

if( ! ( myfile << "pippo test" )){ //controllo se la scrittura ha valore logico true
  cerr << "Errore scrittura sul file";
```

Lettura di un file + controllo errore

Quando si fa una lettura dal file, è da tenere in considerazione che avviene la lettura del **PRIMO ELEMENTO** all'interno del file. Per leggere gli **elementi successivi al primo** serve ripetere l'operazione.

```
#include <iostream>
#include <fstream>

fstream myfile("test.txt", std::fstream::in);
string s; //dichiaro una stringa per scrivere dal file
myfile >> s;

if (myfile.fail()) //controllo errore di lettura file
  cerr<<"Errore lettura dal file";
```

Lettura da file mentre controllo errori

E' possibile leggere da un file e, contemporaneamente, verificare se ci sono errori di lettura mediante l'operatore di estrazione '`>>`' + l'operatore '`!`' in un if.

```
#include <iostream>
#include <fstream>

fstream myfile("test.txt", std::fstream::in);
string s; //dichiaro una stringa per scrivere dal file


if ( ! (myfile >> s;) ) //controllo errore di lettura file
  cerr<<"Errore lettura dal file";
```

Chiusura del file

Un file si chiude con metodo close() di std::fstream

```
fstream myfile ("test.txt", std::fstrem::out);
if (!myfile<<"test"))
  cerr<< "Errore scrittura su file" << endl;
myfile.close();
```

GENERAZIONE NUMERI PSEUDO-CASUALI

Generare un numero pseudo-casuale può equivalere a p*escare un numero da una urna*.

Il generatore deve avere delle caratteristiche:

1. **randomicità**;
2. **controllabile**, deve generare la stessa sequenza su diversi input;
3. **portabile**, su differenti architetture;
4. **efficiente**, in termini di risorse di calcolo occupate per la sua esecuzione.

`#include <cstdlib>` per `rand()` e `srand()`
`#include <ctime>` per la funzione `time()`

`srand()` fissa la sequenza pseudo casuale. Quindi in più esecuzioni si avrà sempre la stessa sequenza di numeri random '**etichettata**' con quella determinata **sequenza prescelta**. Se dovessimo cambiare sequenza nel srand, chiaramente i numeri generati cambieranno rispetto alle generazioni precedenti.
`srand(time(0))` farà stampare numeri random in sequenza diversa per ogni esecuzione del programma. (*va in base all'ora che cambia ogni secondo*)

```
srand (111222333); //seme
//oppure
srand (time(0));
```

> *time(0)*= tempo in secondi decorsi dal 1 gennaio 1970. ogni volta che si avvia il programma, il numero time(0) cambia.

rand() produce un numero compreso fra [0,RAND_MAX] e genera un *unsigned int*

Generazione numeri casuali in un intervallo [a,b]

Dati due numeri naturali a e b con a < b, ci sono **(b − a + 1) +a** valori nell'intervallo [a, b].

> es [2,4] = 4-2+1=3 → 3 numeri in questo intervallo

`rand()%(b-a+1)+a` genera un numero esattamente nell'intervallo [a,b] e si può usare:

```
#include <cstdlib>
#include <ctime>
//dato l'intervallo [4,10] genero numeri random all'interno di esso
int r=rand()%(10-4+1)+4;
```

Generare numeri fra [0,1] in virgola mobile

Per generare numeri in floating point bisogna **forzare** una divisione in virgola mobile.

```
double r=rand()/RAND_MAX*1.0; //così r è un floating point
```

oppure

```
double r= (double) (rand() / RAND_MAX); //utilizzando un cast (double)
```

Generare numeri fra [min,max] double

```
double r = min + (double) ((rand()) /( (double)((RAND_MAX/(max-min)))))
```

PUNTATORI IN C++

Il puntatore è una variabile che **contiene un indirizzo di memoria,** in particolare l'indirizzo di memoria dell'elemento puntato.

La cella puntata conterrà un numero nel formato indicato dal tipo della variabile puntatore dichiarata precedentemente.
Il puntatore all'intero **indirizzerà al primo elemento di tipo intero** presente a quell'indirizzo.
Indicare il tipo di puntatore è utile per capire la quantità di byte che esso deve occupare e **non è possibile** far puntare un puntatore ad un indirizzo che non ha quel **TIPO** di contenuto.

```
int num = 12000;
int *p; //dichiara p come puntatore a int
p= &num; //assegna a p l'indirizzo di num
*p=10; //modifica il dato puntato da p
```

`&` si chiama **operatore di referenziazione** estrae l'indirizzo di memoria di una certa variabile.
`*` si chiama **operatore di dereferenziazione** o **indirezione,** ed esso:

- definisce variabili;
- modifica il dato puntato dal puntatore stesso.

Array vs puntatori

Un array non è altro che un puntatore che punta SEMPRE, di default, alla prima cella dell'array.

```
double v[]={1.2, 10.7, 9.8};
cout << v; //stampa un indirizzo, ES: 0X11223344
cout << *v; //stampa 1.2
cout << v[0]; //stampa 1.2
//Il puntatore *v punta sempre alla prima cella dell'array

double w[]= {3.4, 6.7, 9.8};
v=w //errore di compilazione
```

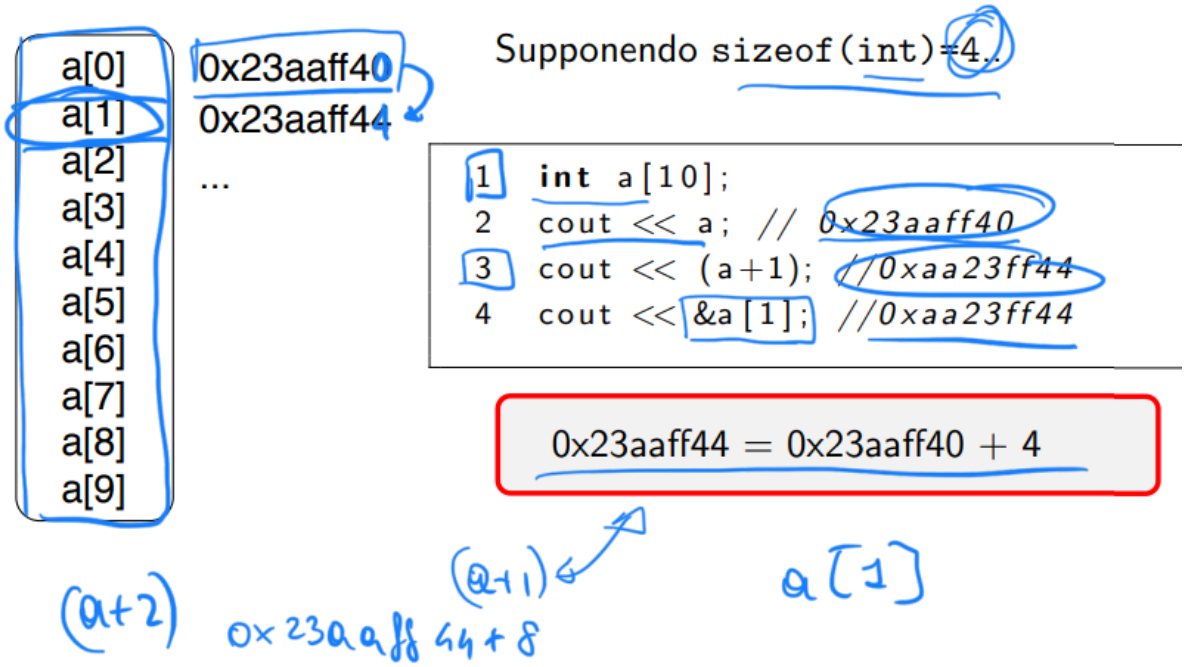
Puntatori mediante indice come gli array

```
double v[]={1.2, 10.7, 9.8};
double *ptr=v;
cout << ptr[1]; //stampa 10.7
cout << ptr[2]; //stampa 9.8
```

```
double v[]={1.2, 10.7, 9.8};
double *ptr=v;
cout << *(ptr+1); //stampa 10.7 //estrae il contenuto della seconda cella
cout << *(ptr+2); //stampa 9.8 //estrae il contenuto della terza cella
cout << *(v+2); // stampa 9.8
```

Aritmetica dei puntatori

```
int a[10];
cout << a; //0x23aaff40
cout << (a+1); //0xaa23ff44
cout << &a[1]; //0xaa23ff44
```



ACCESSO AI VALORI DI UN ARRAY

Metodo di accesso	Esempio
Nome array e [ ]	v[2]
Puntatore e [ ]	ptr[2]
Nome array e aritmetica dei puntatori	*(v+2)
Puntatore e aritmetica dei puntatori	*(ptr+2)

Errore aritmetica dei puntatori

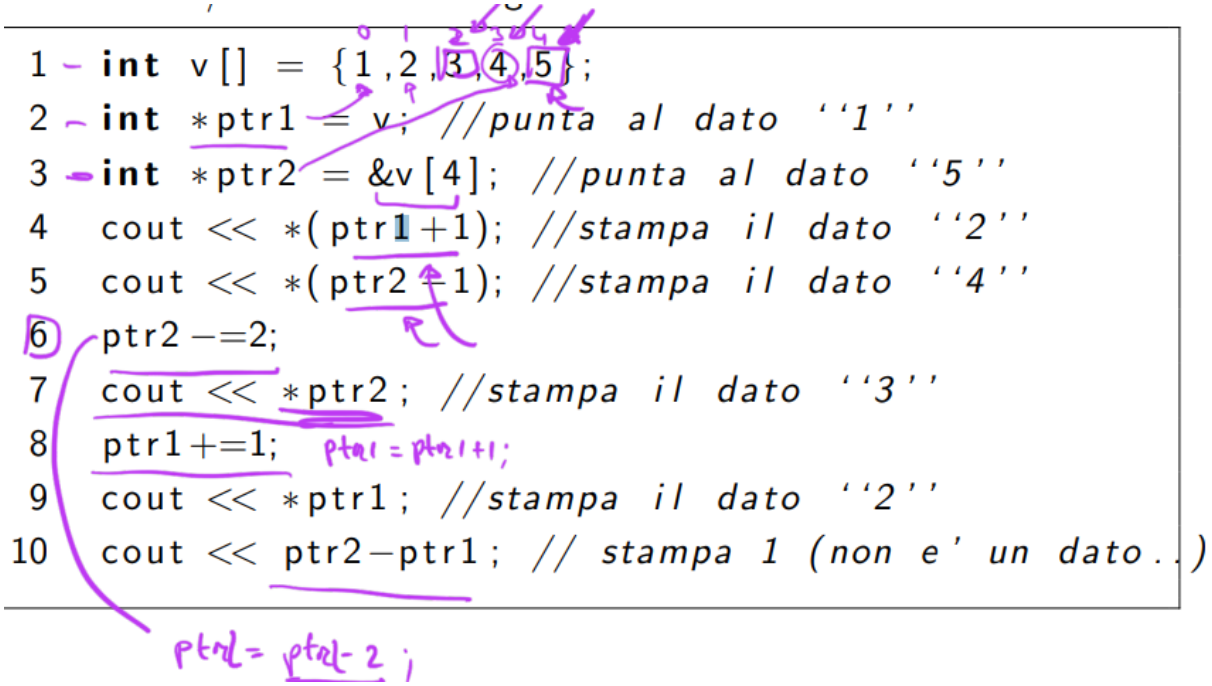
```
v[]={1,2,3};
int *ptr=v;
*(ptr+7); // errore run time
v[4]; //errore run time
```

> L'**errore run-time** avviene quando il programma è compilato ed eseguito ma dopo crasha(smette di funzionare).

Nella maggior parte dei casi il SO invia un segnale di kill all'applicazione a causa del tentativo di accesso a locazioni di memoria non assegnate al processo.

Operatori consentiti per aritmetica dei puntatori.
• Operatori **unari** di **incremento** ++/-- applicati ad una variabile puntatore.
• Operatori binari di **addizione** e **sottrazione** +/- e di **assegnamento** +=, -=, +, -, in cui un membro è un intero e l'altro membro è un puntatore.
• Operatore di **sottrazione** - applicato a due puntatori. Ovvero il valore di un puntatore può essere sottratto al valore di un altro puntatore.

Addizione\sottrazione e assegnamento



Inizializzazione di puntatori

E' possibile inizializzare un puntatore che punta a nulla.
Se il puntatore **punta a NULL**, assume un valore **false.**
Se il puntatore **punta a qualcosa**, assume un valore **true**.

```
int *ptr = NULL; // macro c/c++ // vale come "false"
int *ptr1 = nullptr; // C++11

if (!ptr) { //test se ptr è valido, cioè se punta a un valore non diverso da 0
```

VALORE VS INDIRIZZO

Per confrontare **due indirizzi** mediante puntatori: `if (ptr1==ptr2)`

Per confrontare **due valori** mediante puntatori: `if (*ptr1==*ptr2)`

```
int num;
int *ptr1 = &num; //punto all'indirizzo di num
int *ptr2 = &num;

// confronta indirizzi
if (ptr1==ptr2) { //sarà vero
   // ...
}
//confronto valori
if (*ptr1==*ptr2) { // sarà vero
  // ...
}
```

PUNTATORI A COSTANTI

Sintassi: ***const double *ptr = v;***

Non si può cambiare il valore dell'area puntata dal puntatore(cella di memoria) ma posso far scorrere il puntatore e farlo puntare a qualsiasi cosa, quindi posso cambiare l'indirizzo puntato. La variabile del puntatore, invece, può essere costante oppure non costante.

```
double d1 = 10.9;
double d2 = 4.5;
const double *ptr1 = &d1;
*ptr1=56.9; //errore!
ptr1=&d2 ; //OK
```

ptr1 è un puntatore a costante di tipo double.

In sintesi **è possibile cambiare l'indirizzo** del puntatore ma non il **valore puntato**.

PUNTATORE COSTANTE

Sintassi: ***double const *ptr = v;***

Non è possibile cambiare **l'indirizzo** del puntatore ma è possibile cambiare il **valore puntato**.

```
double d1=10.9;
double d2=4.5;

double*const ptr2=&d2;

ptr2=&d1; //errore!
*ptr2=10.5; //OK
```

PUNTATORE COSTANTE AD UNA COSTANTE

Sintassi: ***const double const *ptr = v;***

```
double d1 = 10.9;
double d2 = 4.5;
const double * const ptr3 = &d2;
ptr3 = &d1; //errore!
*ptr3 = 10.5; //errore!
```

Dichiarazione	Istruzione	Corretta?
`const double *ptr`	`*ptr = 45.9`	NO
	`ptr = &x`	OK
`double * const ptr`	`*ptr = 45.9`	OK
	`ptr = &x`	NO
`const double * const ptr`	`*ptr = 45.9`	NO
	`ptr = &x`	NO

FUNZIONI IN C++

Una funzione rappresenta un **blocco di codice identificato da un nome**.

Il codice può essere strutturato in un modo che il programma principale possa richiamare delle funzioni che ci aiutano alla risoluzione dell'algoritmo. In questo modo si può '**modularizzare**' il programma.

> *Per esempio creo una funzione che mi calcoli il fattoriale e questa funzione la posso richiamare ogni qualvolta mi serve utilizzarla.*

> *In C++ esistono le classi dove si hanno i metodi, cioè funzioni membro che permettono di avere un incapsulamento.*

In C/C++ una funzione è costituita da:
• **nome** della funzione;
• tipo di **ritorno**;
• lista di argomenti o **parametri formali**;
• **corpo** della funzione (istruzioni) tra parentesi graffe;

```
char func( string s, int i) {
  return s[i]; //restituisce l'i-esimo carattere di una stringa
}
```

Esempio di utilizzo

```
char func( string s, int i) {
  return s[i]; //restituisce l'i-esimo carattere di una stringa
}

char ret = func ("pippo", 3);
cout << ret; // OUTPUT: la terza p, pipPo
```

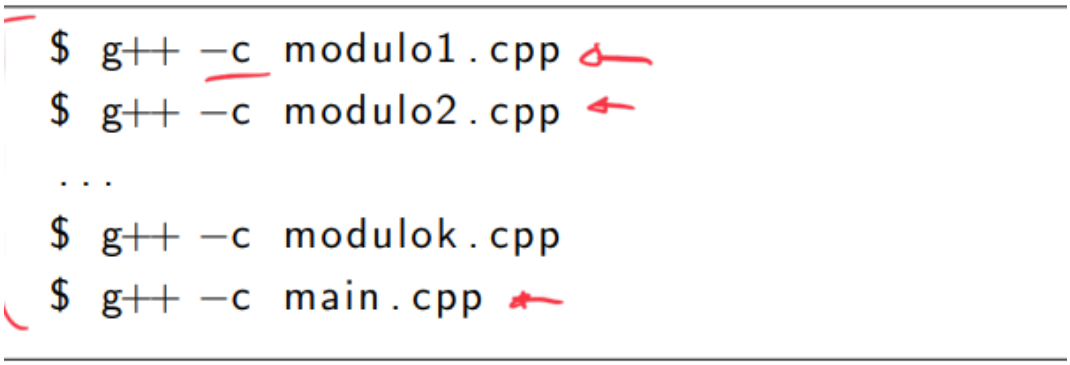
Una funzione che ha un ritorno di tipo **void** vuol dire che non verrà restituito nulla.

PROTOTIPO VS DEFINIZIONE DI FUNZIONE

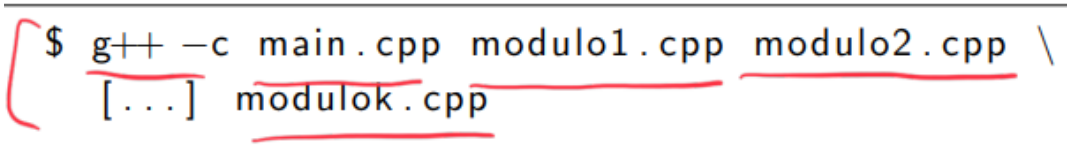
E buona pratica:

- **raccogliere le dichiarazioni dei prototipi** delle funzioni in appositi **file header** (ES: ***modulo1.h***). Un header contiene in genere:
  1. **direttive** #define e altre direttive #include
  2. dichiarazione di **costanti globali**
  3. **prototipi di funzioni**
  4. **dichiarazione** di **classi**
- **raccogliere la definizione delle funzioni** (e metodi) in un **file sorgente**, ES: *modulo1.cpp*;
- **includere la direttiva** `#include "modulo1.h"` in ogni file sorgente (*main*) in cui si fa uso di tali funzioni.

Compilazione di moduli



oppure



Per ottenere un output dopo la compilazione dei moduli si mette ***“-c”*** .Nella cartella si troveranno una serie di file oggetto in output dopo tale operazione:

- *modulo1.o*
- *modulo2.o ecc..*

Infine si possono **assemblare** (**fase di linking** che produce un eseguibile)

```
$ g++ main.o modulo.o modulo2.o ... moduloN.o
```

PARAMETRI FORMALI VS PARAMETRI ATTUALI(R)

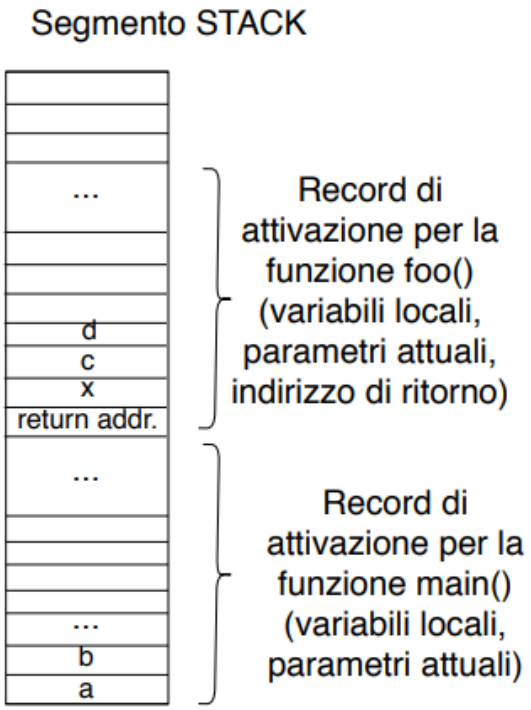
La lista di argomenti presenti nella segnatura di una funzione è una lista di **parametri formali**, quindi relativa alla **definizione della funzione**.

Il **parametro attuale** è il **valore** che si passa alla **funzione** quando essa viene **utilizzata.**

AREA STACK (pila) e RECORD DI ATTIVAZIONE

Ogni volta che si inizia un programma si crea uno stack. Lo stack è una pila che è di tipo ***Last In First Out (LIFO),*** cioè **l'ultimo dato viene inserito (push)** in cima alla pila ed esso stesso sarà il **primo dato ad uscire (pop)** quando verrà richiamato.

Lo stack permette di inserire varie variabili all'interno di un programma e, inoltre, si viene conservato un **indirizzo di ritorno**, che permette di tornare al programma principale dopo aver eseguito le operazioni nello stack.




```
1   void foo(int x){
2     double c, d;
3     //...
4   }
5   int main(){
6     int a,b;
7     //...
8     foo(a);
9     //..
10  }
```

PARAMETRO ATTUALE COME INDIRIZZO O VALORE

Se si ha una funzione alla quale vi è un **parametro formale di tipo "valore"**, si avrà un codice del tipo:

```
void foo(int x){ //x parametro formale
      x=90; //quest'istruzione non avrà alcun effetto su a
}
int main(){
    int a=10;
  foo(a);
  cout << a //stamperà 10
    }
```

Se si ha una funzione alla quale vi è un **parametro formale puntatore**, **richiederà** come **parametro attuale** un **indirizzo**. In tal caso **è possibile modificare il valore della variabile indicata nella funzione**.

```
void foo(int *p){
  *p=90;
}
int main(){
    int a=10;
    foo(&a);
    cout << a; //stamperà 90
}
```

ALLOCAZIONE AUTOMATICA

- La memoria viene allocata mediante una dichiarazione di una **variabile locale** ad una funzione
- Ci sarà uno **scope**/visibilità della variabile **limitato al blocco di codice** in cui è stata dichiarata.
- Area di memoria usata è detta **STACK**
- Ciclo di vita del blocco allocato **termina** con la **fine dell'esecuzione del blocco** in cui viene usato

```
void foo(){
    int a=0; // a visibile solo in foo()
}
```

ALLOCAZIONE DINAMICA

- Effettuata mediante operatore `new` in qualsiasi punto del programma.
- Area di memoria usata è denominata **HEAP**.
- Ciclo di vita del blocco di memoria termina con invocazione di operatore `delete` sul puntatore.

```
int *p = new int (2); //cella int, valore iniziale 2
// ...
delete p; //deallocazione della cella puntata da p
```

ALLOCAZIONE STATICA

- Dichiarazione di variabili **al di fuori** da qualunque blocco.
- Segmento di memoria ospitante è detto segmento **DATA**.
- Ciclo di vita / **scope globale**: inizia e termina **con il programma stesso**.

```
// ESEMPIO:
#include <iostream>

using namespace std;

//allocazione dinamica di intero (HEAP)
//allocazione statica di variabile puntatore (DATA)
//Scope del puntatore global globale!!
int *p_global = new int(2);

//allocazione statica di intero (DATA)
// Scope globale!!
int global = 10;
```

```
void foo(){

  cout << "*p_global: " << *p_global << ", global:" << global;

  //Si provi a decommentare la seg. linea di codice..
  //cout << "*dynamic: " << *dynamic << ", local: " << local << endl;
}

int main(){

  cout << "*p_global: " << *p_global << ", global: " << global << endl;

  //allocazione automatica (STACK)
  //scope della variabile local locale!!
  int local = 1;

  //allocazione dinamica (HEAP)
  //scope del puntatore dynamic locale!!
  int *dynamic = new int(3);
  foo();
}
```

PASSAGGIO DI PARAMETRI ARRAY

Il passaggio di un array come parametro di una funzione avviene **sempre per indirizzo**.

> *Il nome di uno array è un puntatore costante al primo elemento dello array*

```
void init(int *v, int n){
   // ...
    for (int i=0; i<n; i++){
      v[i] =0;
    }
}
int main(){
    int x[10];
    init(x, 10);
}
```

Array multidimensionali

Per il passaggio di array multidimensionali allocati sul segmento DATA o sullo STACK, nel prototipo della funzione che riceve il dato, **vanno specificate tutte le dimensioni** e può **esserne omessa SOLO UNA** che è quella che può essere **automaticamente calcolata**.

Indirizzo o contenuto di una cella di memoria ed equivalenze

```
(1) &v[i][j] // dà l'indirizzo della cella i j

EQUIVALE

(2) (*(v+i) + j); //aritmetica dei puntatori
```

```
(3) (v[i] + j); //aritmetica dei punt. e indici

(4) v[i][j] // da il contenuto della cella i j

equivale

(5) *(*(v+i) + j); //aritmetica dei punt.
(6) *(v[i] + j); //arit. punt. + operatore []
```

Se al parametro formale viene passato un **valore**, quando la funzione verrà usata **lavorerà su una copia della variabile.**
Se al parametro viene passato un **indirizzo**, quando la funzione verrà usata **lavorerà sulla variabile direttamente** e sarà possibile cambiare il valore della variabile nel main mediante la funzione stessa.

ALLOCAZIONE IN MEMORIA (r)

https://www.dmi.unict.it/farinella/Prog1/Lezioni/19__memory_allocation2021.pdf

Un puntatore sta sullo stack

la cella puntata sta sull' heap

Operatore new

```
1 int *a=new int;
2 float *f = new float(0.9) ;
3 *a = 10;
4 double *arr= new double [10] ;
```

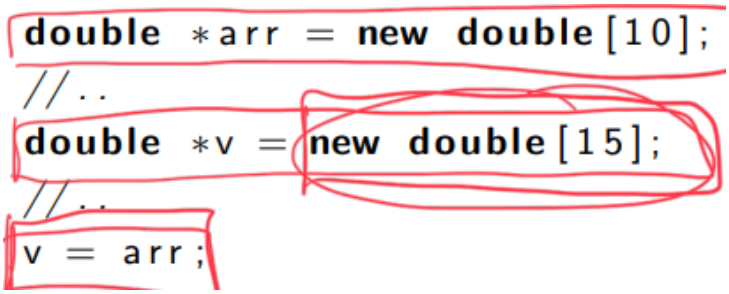
L'operatore new permette di operare sull'**allocazione dinamica.**

A differenza dell'allocazione automatica, la memoria allocata dinamicamente va successivamente liberata mediante l'operatore `delete`.

```
1  int *a = new int;
2  //...
3  delete a; //deallocazione
```

```
1  double *arr = new double[10];
2  //..
3  delete [] arr;
```

Se un puntatore punta ad un'allocazione dinamica, dopo l'operazione `delete` esso conterrà ancora il valore dell'indirizzo puntato precedentemente al delete. Solitamente si usa mettere assegnare il puntatore a **NULL** o **nullptr**.



A seguito della copia dell'indirizzo di arr su v, si perde l'indirizzo al blocco di memoria di 15 elementi double.

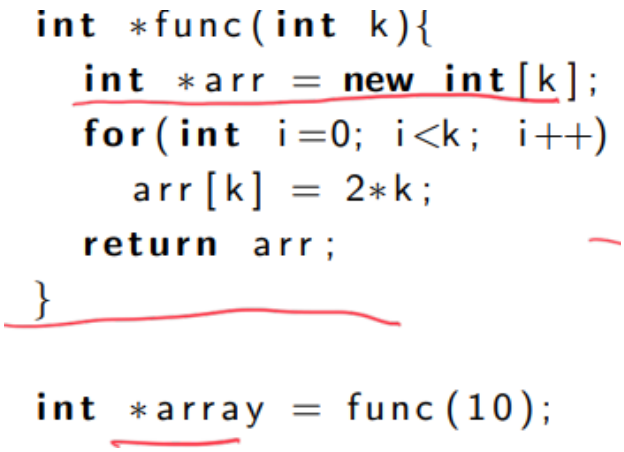
Non è più possibile deallocare il blocco di memoria con l'operatore delete.

In caso di un utilizzo doppio di delete di una stessa allocazione di memoria si avrà un comportamento indefinito e possibilmente **disastroso.** Potrebbe venire deallocata una memoria utilizzata già nel programma e tutti i dati verrebbero persi.

Per evitare un doppio delete si usa un **if**:

```
if (arr){ //sarà vero se punterà a qualcosa
  delete [] arr;
  arr=nullptr;
     // ... altri tentativi di deallocazione
  if (arr){ //sarà falso perchè arr punterà a nullptr
  // ...
  }
}
```

FUNZIONI CHE RESTITUISCONO UN PUNTATORE



Se è necessario avere un puntatore come return dentro una qualsiasi funzione, sarà **necessario allocare tale puntatore dinamicamente sullo HEAP** in modo da poterlo "passare" al main o a qualsiasi altra funzione dopo la "distruzione" della funzione o comunque alla fine di essa.

malloc() e free()

`malloc` serve per allocare e riservare delle locazioni di memoria.

per deallocare si usa `free()` come se fosse un delete.

```
#include <cstdlib>
double *arr = (double *) malloc (sizeof(double)*10);
//..
free(arr); //deallocazione
```

L'argomento di malloc() è di dimensione in **byte** (quindi si usa **sizeof**)

Restituisce un puntatore generico, ovvero di tipo void *, per questo bisogna operare un **type casting** al tipo desiderato.

OGGETTI STRINGA

Una stringa può essere vista come un **array di caratteri**.

```
char s1[]={'m, 'y', ' ', 's, 't', 'r', 'i', 'n', 'g', 0};
char s2[]={'m, 'y', ' ', 's, 't', 'r', 'i', 'n', 'g', '\0'};
```

Un letterale stringa è un **puntatore costante** ad un array di caratteri: in quanto il compilatore memorizza il letterale in memoria e conserva il puntatore.

la libreria `<cstring>` contiene funzioni di:

- **copia di una stringa** su un array di caratteri con la funzione `strcpy (*destinaz,sorgente);`
- **comparare in modo lessicografico** con `strcmp("s1","s2")`, e restituisce un int>0 se s1>s2, int<0 se s2>s1, int=0 se s1=s2
- **ricercare** sottostringhe `strstr`(), restituisce un puntatore alla prima occorrenza trovata

```
#include <cstring>

char name[15];
strcpy(name, "pippo"); //copia "pippo" all'interno dell'array di caratteri name
-------------------------------------
strcmp ("pippo",”paperino”);// restituisce un int>0
strcmp (”paperino”,”pippo”);//restituisce un int<0
strcmp (”pippo”,”pippo”);//restituisce zero
-------------------------------------
string s1="caio";
string s2="io";
char *found = strstr(s1,s2); //cerca s2 in s1 e dà un puntatore alla prima occorrenza
```

E' possibile usare l'operatore `==` fra due stringhe ma non fra due array di caratteri char.

Array di caratteri → Tipo numerico

`<cstdlib>`

Questa libreria contiene delle funzione che permettono di **convertire** stringhe a interi (int → long → long long).

Se la stringa **inizia con un numero**, la funzione restituisce il numero, altrimenti 0 se la stringa non inizia con un numero.

```
int main(){

  cout << atoi("012") << endl;  // 12
  cout << atoi("aa012") << endl; // output: 0. No controllo errore...
  cout << atoi("0") << endl;  //0
  cout << atoi("-12") << endl;  //0

  cout << atof("0.14") << endl; //0.14
  cout << atoi("aa0.12") << endl; // output: 0. No controllo errore...
  cout << atof("0") << endl; //0
  cout << setprecision(10) << atof("-12.123456") << endl; //0
}
```

`sscanf`

E' un'istruzione analoga a `atoi/l/ll()`.

```
int y ;
const char*str="123456789" ;
sscanf(str,"%6d",&y);// %6d vuol dire che stampa 6 cifre intere, &y è la destinazione di tale \
                        conversione
cout << y << endl ; // stampa 123456
```

`ssprintf()`

```
  int a=10;
  float x = 43.567;
  double alpha = 123.456789123;
  char str[100];
  sprintf(str, " Intero a: %d, x: %.6f, alpha: %.12f", a, x, alpha);
  cout << endl << str << endl; //Output=Intero a: 10, x: 43.567001, alpha: 123.456789123000
```

`<cctype>`

Questa libreria permette di operare sulle stringhe per verificare se il carattere preso in considerazione è maiuscolo, un numero, minuscolo, un simbolo, uno spazio ecc..

Si utilizzano gli oggetti di tipo string al posto di array di caratteri di char e si include `<string>`

```
#include <cctype>
```

funzione	Valore di ritorno
isalpha	true se l'argomento è una lettera, false altrimenti
isalnum	true se l'argomento è lettera o numero, false altrimenti
isdigit	true se l'argomento è numero, false altrimenti
ishex	true se l'argomento è composto di cifre per rappresentazione esadecimale: $0-9$, a-f, A-F

```
#include <cctype>
```

funzione	Valore di ritorno
isprint	true se l'argomento è un carattere stampabile, false altrimenti
ispunct	true se l'argomento è un carattere di punteggiatura, false altrimenti
islower	true se l'argomento è minuscolo, false altrimenti
isupper	true se l'argomento è maiuscolo, false altrimenti
isspace	true se l'argomento è uno spazioo, false altrimenti

Operatore [] vs metodo at()

L'operatore `[]` **non fa nessun controllo** sulla lunghezza della stringa.

Invece `at()` fa il controllo sulla lunghezza di caratteri di una stringa.

```
1 string s = "0123456789"; //10 caratteri
2 // nessun controllo sulla lunghezza della stringa!
3 cout << s[10] << endl;
4
5 //at() solleva ''eccezione'' se indice >= s.length()
6 cout << s.at(10) << endl;
```

> ***ESAME*** *sono spesso presenti le stringhe. vedere se una sottostringa è presente e quante volte.*

PROGRAMMAZIONE AD OGGETTI

Gli oggetti sono capaci di **scambiare messaggi fra loro**. Si immagina che l'oggetto comunichi con l'esterno ritornando dei valori richiesti. Si tratta di una sequenza di scambi di messaggi. Alcuni oggetti avranno all'interno una serie di funzioni che servono per gestire questi scambi di messaggi. Chiaramente è possibile scambiare messaggi anche fra oggetti.

Un oggetto ha:

1. **uno stato**, caratterizzato dalle variabili e dai corrispondenti valori di esse;
2. **un comportamento**, cioè tutte quelle operazioni eseguibili e metodi su un oggetto, come, per esempio, **cambiare lo stato dell'oggetto** oppure **inviare messaggi ad altri oggetti**.

A questo oggetto è possibile accedere mediante questi metodi.

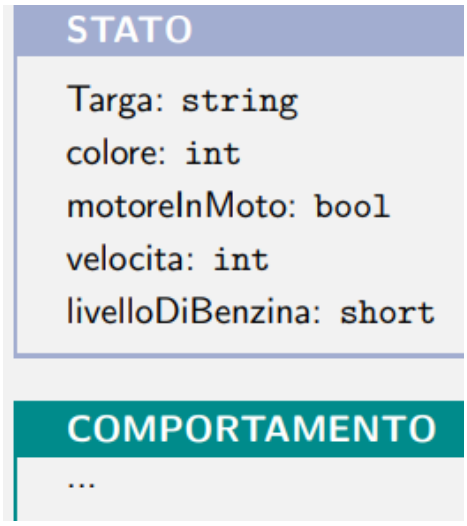
Stato

Lo stato è formato da **attributi** che rappresentano una **proprietà definita** in modo astratto ed è spesso **mappato su una variabile**.

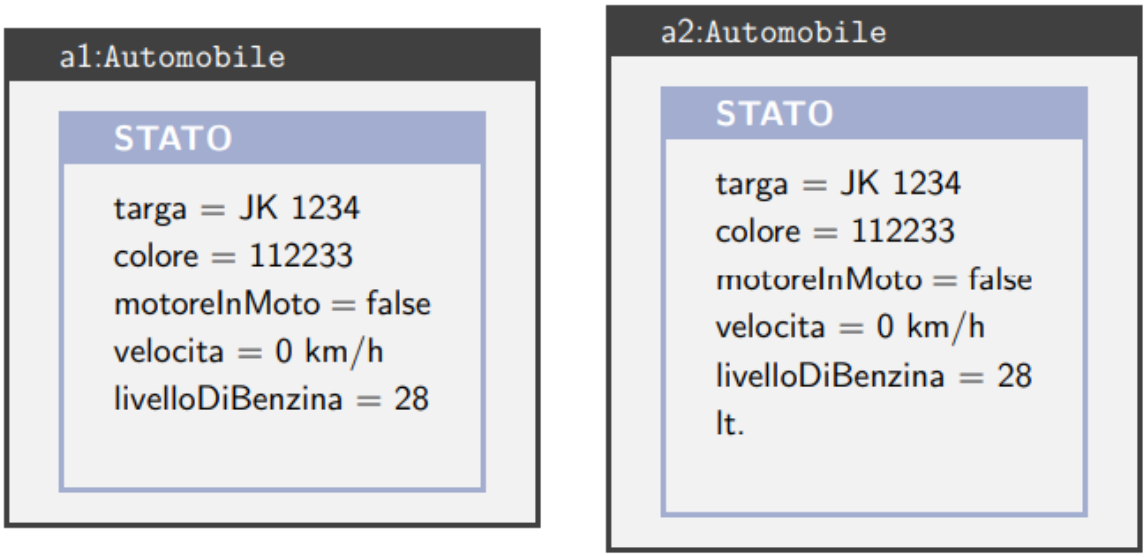
Comportamento

Esso è modellato e descritto da un insieme di metodi, o **funzioni membro**.

Esempio di utilizzo di un oggetto **AUTOMOBILE**:



Questa è la definizione di una classe di oggetti, in questo caso oggetto Automobile.



Se due oggetti sono inizializzati con lo **stesso stato**, si parla di **identità**.

Ogni oggetto ha una propria identità, quindi sono distinguibili fra oggetti dello stesso tipo.

Finchè non si termina il programma, gli oggetti mantengono le informazioni al proprio interno per un **tempo indefinito.**

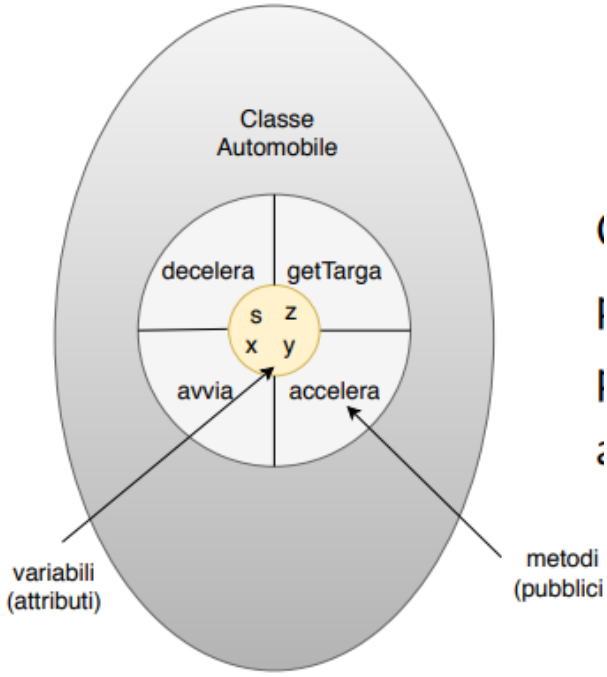
Ciclo di vita

Un oggetto durante l'esecuzione del programma, è soggetto alle seguenti fasi:

1. **creazione** (stato iniziale). Quando non vengono forniti parametri all'oggetto, si avrà un'inizializzazione 'standard'. Come la string che inizializza una stringa vuota se non riceve stringhe durante la definizione.
2. **utilizzo**, cambiando lo stato dell'oggetto, il valore delle variabili oppure utilizzare il valore di quelle variabili;
3. **distruzione** dell'oggetto permettendo la liberazione della memoria.

CLASSI VS OGGETTI

Una classe **definisce** in modo **formale** un codice e dice cosa deve fare un oggetto di quel tipo. Rappresenta la descrizione di come devo no essere costruiti gli oggetti istanza della classe. La classe descrive la struttura degli oggetti.



Un oggetto espone i metodi pubblici, che operano prevalentemente sugli attributi.

Le classi vengono progettate mediante software mediante varie fasi:

- fase di Analisi, ovvero analisi dei requisiti e del dominio dell'applicazione
- progettazione delle classi (modellazione o design)
- Implementazione, cioè codifica delle classi utilizzando un linguaggio di programmazione orientata ad oggetti.

Un oggetto si dice istanza di una classe, ovvero si intende **un esemplare** di quella determinata classe.

Quindi un oggetto è un contenitore di variabili(attributi) che descrive lo stato e di metodi per elaborare le variabili.

> *Una classe costituisce una determinata rappresentazione delle caratteristiche di una entità del mondo reale. Un oggetto è un'istanza della classe: la sua struttura ed il suo comportamento sono conformi alla descrizione della classe.*
> *Tuttavia ogni oggetto ha la sua identità e generalmente un proprio valore per la variabili di stato o attributi.*

Per definire le classi si utilizza `class` . In fase di design della classe si deve capire lo stato e il comportamento, quindi bisogna implementare i metodi e le altre varie funzioni.

Lo scambio di **messaggi fra oggetti** avviene tramite invocazione dei metodi e questi ultimi servono a:

- prelevare informazioni
- causare cambiamenti di stato
- avviare un'attività

Sintassi:

```
<obj>.<metodo>([p1][p2]...);
```

`obj` denota l'oggetto destinatario

`metodo` denota il nome del metodo

`[p1] [p2]` indicano la lista di eventuali parametri

```
// ESEMPIO //
1 Automobile m1;
2 m1.setMotore(”elettrico”); // cambiamento di stato
3 m1.avviaTergicristallo(); // attività
4 m1.accendiMotore(); // cambiamento di stato
5 m1.accelera(); // attività
6 m1.getVelocita(); // invio di informazioni al chiamante
```

Un **metodo** è una funzione associata ad una determinata classe. Esso rappresenta la parte dell'implementazione del comportamento dell'oggetto.

Quando si invoca un metodo si ha un'esecuzione del codice di tale metodo. Alla fine dell'esecuzione, cioè in corrispondenza di un `return` o in presenza **dell'ultima istruzione del metodo**, si ritorna al programma principale seguendo da dove si è interrotto.

Esistono diversi tipi di messaggi per i metodi. Tra questi si distinguono i tipi:

- **informativo**: `setTarga()`. Informa l'oggetto che qualcosa è cambiato e che deve aggiornare il suo stato;
- **interrogativo** `getTarga()`: Chiede informazioni in merito allo stato dell'oggetto;
- **imperativo** `avviaMotore():` Chiede all'oggetto di avviare una o più attività.

Programmazione procedurale vs programmazione ad oggetti

Programmazione strutturata/procedurale: una sequenza di invocazioni di funzioni.
Programma ad oggetti: l’esecuzione di un programma ad oggetti flusso di messaggi tra oggetti, ovvero una sequenza di invocazione di metodi.

Il **metodo costruttore** serve ad inizializzare (in maniera di **default** o con i **parametri** che gli vengono passati) lo stato dell’oggetto appena creato.
La chiamata al costruttore è **automatica**, avviene contestualmente alla creazione dello oggetto.

Ogni volta che si definisce un nuovo oggetto, avviene la chiamata automatica al costruttore di quell'oggetto.
Il compilatore “forza” la chiamata automatica al costruttore.
In questo modo non possono esistere oggetti istanziati ma non inizializzati. L'inizializzazione di un oggetto è possibile modificarla all'interno della classe di quell'oggetto stesso.

AGGREGATI E COLLEZIONI DI OGGETTI

Se una entità da modellare ad oggetti è complessa, è conveniente scomporla in un insieme di costituenti:

1. **codifica**: lo sviluppatore concepirà diverse classi per la modellazione della singola entità;
2. **esecuzione del programma**: un oggetto sarà composto da più oggetti, istanze delle classi che insieme modellano l’entità da rappresentare.

E' utile scomporre il codice perchè così il codice è riutilizzabile e vi è una maggiore manutenibilità.

Relazioni part-of

Una classe è definita da diverse parti. L'oggetto composto scambia messaggi con le varie parti di esso(messaggi informativi, informativi e interrogativi).

Per capire al meglio le relazioni part-of ci si può riferire al classico esempio è l’oggetto di tipo Automobile in relazione con Motore e Persona.

> *Gli oggetti di tipo automobile vengono creati/distrutti contestualmente alla creazione/distruzione dell'oggetto stesso.*

La relazione **automobile-motore** è una **relazione stretta, o di composizione**, perchè l'auto non può esistere senza motore.

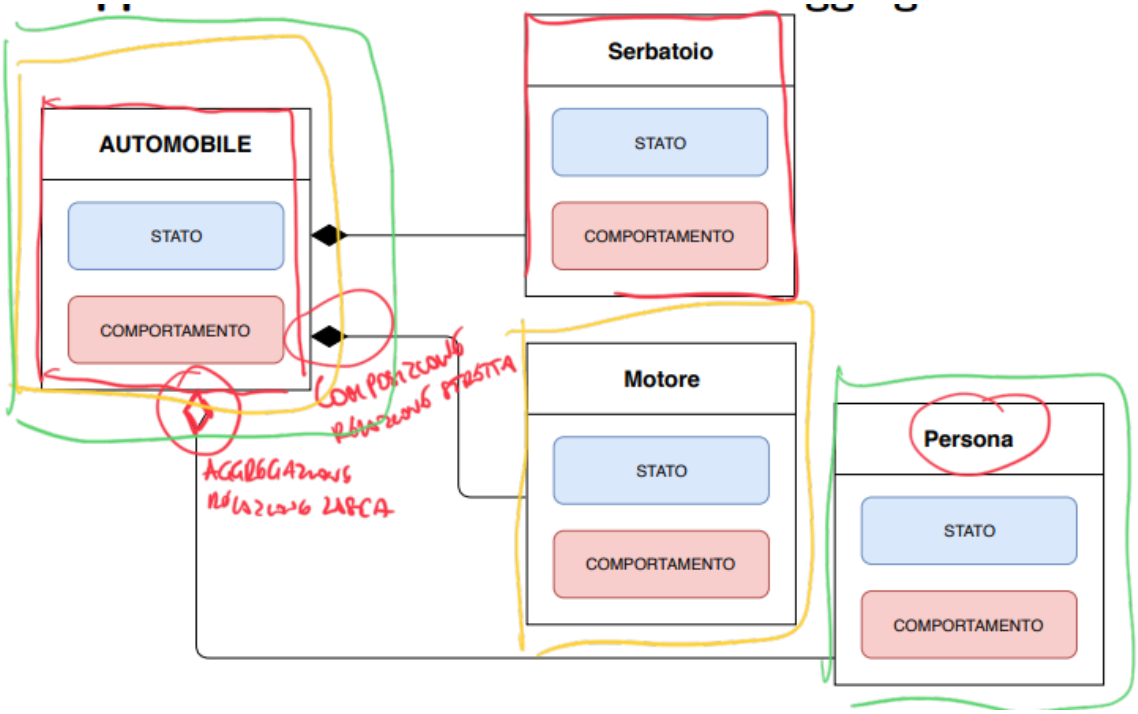
Nella relazione di composizione, **l’oggetto contenuto**
(ES: motore o serbatoio) **viene creato e distrutto**
insieme all’oggetto contenitore. (Es: Automobile)

La relazione **automobile-persona** non è una relazione stretta perchè la persona non è una componente dell'automobile.

Questa è una **relazione di aggregazione** e non è indispensabile all'automobile. Questi due oggetti *possono stare in relazione* ma ***non necessariamente***.

La relazione di *aggregazione* si indica con un ***rombo vuoto***.

La relazione di *composizione* si indica con il ***rombo pieno***.



In generale, nella **relazione di composizione**:

- l'oggetto "contenuto" non ha una vita propria;
- si realizza con un oggetto "contenuto" interno al contenitore;
- l'oggetto "contenitore" è responsabile della costruzione e distruzione del contenuto;
- Il coordinatore della composizione deve:
- creare l'oggetto contenitore;
- fornire eventuali valori per l'inizializzazione del contenuto tramite costruttore e/o metodi dell'oggetto contenitore.

Invece nella **relazione di aggregazione**:

- il ciclo di vita degli oggetti contenuto e contenitore indipendenti;
- si realizza con un oggetto "contenuto" interno al contenitore;
- contenitore non responsabile della costruzione e distruzione del contenuto.

Il **coordianatore** della aggregazione deve:

- creare l'oggetto contenuto;
- creare un contenitore passandogli un puntatore allo oggetto contenuto.

Array di oggetti

```
const int NUM FRECCE = 20;
Bersaglio b(10);
Freccia F[NUM FRECCE] ;
```

```
int tot=0, i=0;
  while(i < NUM_FRECCE){
    F[i].lancia(b);
    i ++;
  }
i=0;
  while(i<NUM_FRECCE)
    tot+=b.punteggio(F[i++]);
```

IMPLEMENTAZIONE CLASSI

La classe è la descrizione generale di come devono essere costruiti gli oggetti corrispondenti. In altre parole una classe è una sorta di progetto con cui ogni particolare oggetto di quel tipo dovrà essere costruito. Un oggetto costruito sulla base della descrizione contenuta nella classe X si dice **istanza della classe X.**

Una classe viene definita:

```
class A {
. . . // variabili  o campi o attributi (STATO)
. . . // metodi
};
```

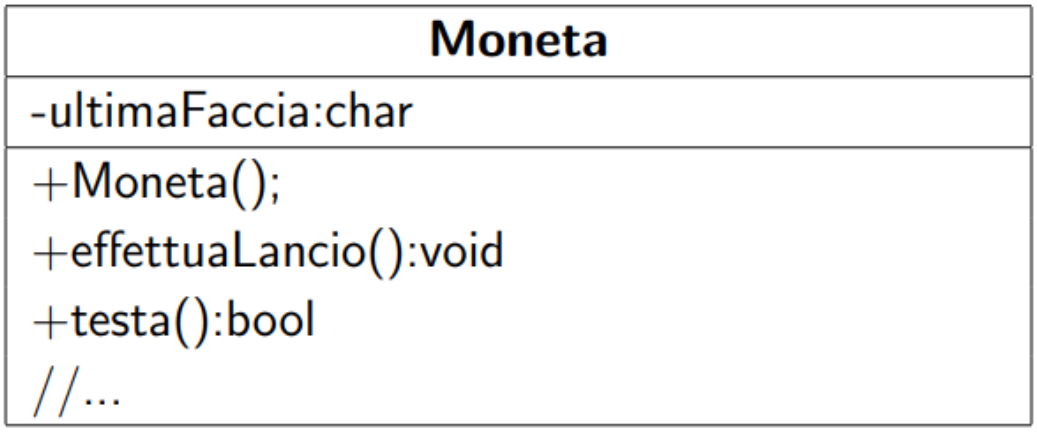
Modificatore di accesso

Ad alcuni metodi ci si può accedere dall'esterno mentre ad altri no. Ciò è realizzabile mediante i modificatori di accesso **public**, **private** e **protected**.

Il **simbolo** `-` indica che vi sarà un attributo di una classe ed è **privato**. E può essere modificato solo da metodi all'interno della classe stessa

Il **simbolo** `+` indica che il metodo è pubblico e può essere usato all'esterno di quella classe.

Il **simbolo** `#` indica che il metodo è di tipo protected e può essere usato solo all'interno di una gerarchia di classi.



```
+) public:
-) private:
#) protected:
```

Esempio dichiarazione della classe moneta → file moneta.h

```
class Moneta{
  public :
    Moneta();
    void effettuaLancio( ) ;
    bool testa( ) const ;
    bool croce( ) const ;
    char getFaccia ( ) const ;
  private:
    char ultimaFaccia;
};
```

Eempio definizione dei metodi di Moneta → file moneta.cpp

```
tipo_ritorno nome_classe::<nome_metodo>

bool Moneta::testa(){
  return(ultimaFaccia=='T');
}
```

Si può dichiarare la classe Moneta e si possono definire il comportamento dei metodi all'interno del file stesso.

```
class Moneta{
  private:
    char ultimaFaccia;
  public:
    bool testa(){
     return(ultimaFaccia=='T');
    }
};
```

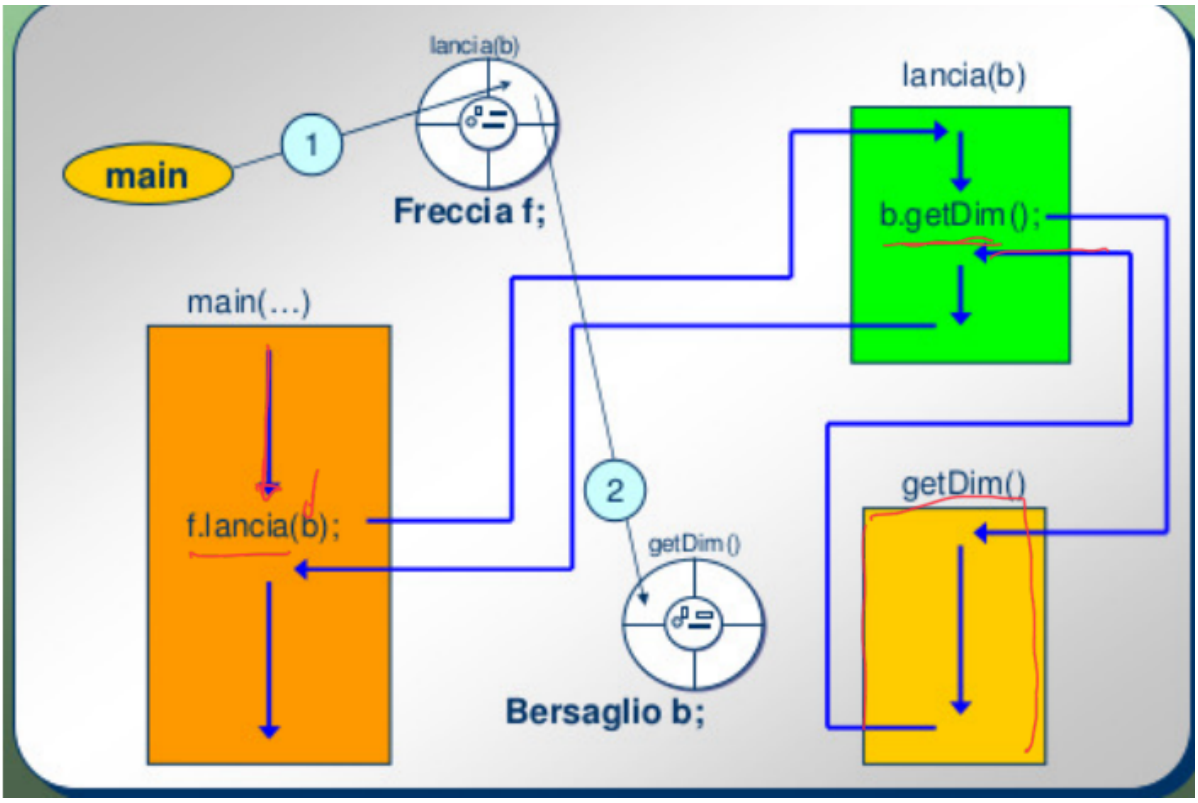
Flusso di invocazione di metodi

Quando un metodo viene invocato, il flusso di controllo “passa” a tale metodo:

- saranno eseguite le istruzioni del metodo, dalla prima all’ultima, fino alla fine o fino ad un’istruzione *return*.
- Dopo l’esecuzione dell’ultima istruzione del metodo, oppure in corrispondenza di una istruzione return, il flusso “ritorna” al punto in cui e partita la chiamata:
- sarà eseguita l’istruzione successiva alla chiamata del metodo.

Quindi la programmazione a oggetti è un **flusso di invocazione di metodi**.

Esecuzione di metodi in cascata vuol dire che si ha un metodo ne invoca un altro a cascata.



Quando si parla di invocazione di metodi in cascata **(oggetti distinti)** un metodo ne invoca un altro ma di oggetti distinti e non della stessa classe.

La dichiarazione del metodo definisce il suo prototipo e appartiene ad una ben determinata classe.

L'invocazione di un metodo è un'istruzione che avvia l'esecuzione del codice relativa al corpo di quel metodo.

All'interno dei metodi è possibile referenziare dati locali al metodo quindi i parametri di tale metodo esistono solo all'interno di quel metodo.

Tutte le variabili dichiarate all'interno del metodo stanno sullo stack, quindi alla fine di tale esecuzione esse verranno distrutte.

Le variabili locali hanno uno scope limitato al metodo stesso e un ciclo di vita relativo alla fine del metodo.

A differenza delle funzioni, è possibile referenziare variabili di istanza della classe in cui il metodo è definito

```
class Moneta{
  private:
    char valoreUltimoLancio{
    // . . .
    }
```

```
bool Moneta::testa(){
  return(valoreUltimoLancio=='T');
}
```

L'operatore ***this*** è un puntatore alla classe stessa. E' usato, spesso, quando si hanno degli attributi di una classe che hanno lo stesso nome di alcuni parametri formali di qualche metodo. All'interno di un metodo gli attributi di classe sono 'oscurati' dalle variabili locali che la funzione utilizza. Pertanto, se si vuole usare un attributo della classe stessa, è necessario rivolgersi ad essa mediante l'operatore this.

Passaggio per valore

Un metodo può avere dei parametri formali passati per valore. In questo caso le modifiche del parametro formale non si riportano all'esterno di una funzione perchè il parametro sta sullo stack perchè viene eseguita una copia della variabile istanza (nel main).

Passaggio per puntatore

Il codice del metodo può modificare il valore della variabile usata come parametro attuale, mediante l'operatore di indirezione o referenziazione.

Passaggio per riferimento

Il parametro formale diviene alias del parametro attuale. Ogni modifica ad un parametro formale all'interno del corpo del metodo si riflette al parametro attuale presente all'istruzione di chiamata di una funzione.

Passaggio by reference: usi tipici/consigliati

1. il metodo **deve modificare uno o più dei suoi argomenti**
2. il metodo deve restituire più di un valore, ad esempio:

```
1  int sum(int a, int b, int &result){
2    if(a > 0 && b >0){
3      result = a+b;
4      return 0; // valore di controllo
5    }
6    else
7      return -1; // valore di controllo
8  }
```

Argomenti nella funzione main

Main può prendere in input degli argomenti: **argc** (numero parametri) e **char *argv[]** che è un array di caratteri)

```
int main(int argc, char *argv[]){
  //...
}
```

```
$ g++ mio_prog.cpp -o mio_prog
$ ./mio_prog arg1 arg2
```

---

la stringa "arg1" sarà accessibile da argv[1]. la stringa "arg2" sarà accessibile da argv[2].

mentre la stringa "mio_prog" sarà memorizzata in argv[0].

In un metodo i parametri formali vanno inizializzati a partire da quello piu a destra.

```
int foo(int a=1, double b,char c='Z') ; //errore perchè l'argomento in mezzo\
non è specificato
```

FUNZIONE INLINE E CONSTEXPR

La ***keyword inline*** serve per ridurre di molto il tempo di compilazione. Non è un comando ma solo una keyword.

Se presente la keyword inline, il compilatore sostituisce ad ogni chiamata alla funzione (ES: linea 5) il codice del corpo di f().
Si utilizza per una maggiore efficienza! Inoltre il programmatore continua a ragionare in termini di funzioni/metodi.

```
inline double f ( double y ){
  return y*y;
}
  // . . .
  f(0.5);
  //...
```

In questo caso avviene una sostituzione senza la chiamata alla funzione stessa.

constexpr vuol dire che il compilatore deve valutare l'espressione a tempo di compilazione. Piuttosto che compilare il programma per cosi e com'è, valuta la funzione in fase di compilazione. In fase di esecuzione non verrà chiamata nemmeno la funzione. La chiamata alla funzione viene effettuata quando il codice è compilato.

MODIFICATORI DI ACCESSO PER I MEMBRI DI UNA CLASSE

public: visibile sia all'interno che all'esterno della classe. In particolare, i metodi public costituiscono l'interfaccia della classe.
private: visibile solo all'interno della classe. In particolare, i metodi private si dicono **metodi di servizio.**

protected: all'esterno della classe sono a tutti gli effetti private. Ma differisce dai membri private nel contesto dell'ereditarietà. Sarà visto in seguito..
non specificato: equivalente a private (default a private).

IMPLEMENTAZIONE COSTRUTTORI

Un costruttore è un **metodo** e prende il nome della stessa classe e non dichiara alcun tipo di ritorno. Serve, sostanzialmente, per istanziare gli attributi della classe, quindi crea l'oggetto.

REFERENCES

Le reference sono utili per creare **ALIAS** di un determinato oggetto. Dopo la sua azione è possibile modificarne il valore come se fosse l'oggetto stesso al quale si riferisce. non è possibile utilizzare la aritmetica dei puntatori perché se facessimo questa operazione, incrementare mo l'oggetto al quale si riferisce ovvero l'alias.

STATIC CONST E FRIEND

Static

Si utilizza per "aumentare lo scope" di una determinata variabile definita in una classe. Una variabile static non può essere inizializzata all'interno di una classe, o, per essere più precisi, non può essere inizializzata quando viene dichiarata per la prima volta nel blocco di codice *private*.

Tutte le istanze della classe (nel main) condividono una unica copia in memoria della variabile static. Eventuale modifica si riflette su tutti gli altri oggetti ed essa va inizializzata a livello globale, fuori dallo scope della classe.

Tutti gli oggetti della stessa classe possono vedere la variabile ***static*** quindi è un'**attributo comune a tutti gli oggetti creati nel main di quella classe in particolare**.

L'attributo static va **DICHIARATO** come *private* o *public* ma va **INIZIALIZZATO fuori dalla classe**, per esempio `int Classe::variabileStatic=0`;

Una variabile “static int a=0;“ **DICHIARATA** all'interno di un metodo, sopravvive anche all'estinzione di quel metodo e quindi non è più una variabile locale ma una variabile che viene condivisa/aggiornata ogni qualvolta viene invocato il metodo. Può essere usata per capire quante volte è stata invocata una funzione.

Const

Esempio: *int getX() const {}*

La funzione:

- non può modificare lo **stato dell'oggetto**(i suoi attributi);
- non può modificare **variabili membro**;
- non può invocare **metodi non const**.

Friend

Una funzione dichiarata come “friend” **non è membro della classe** nella quale è stata dichiarata ma serve per far *accedere agli attributi pubblici e privati della classe di appartenenza trovandosi fuori dallo scope della classe stessa*.

Questi metodi sono spesso usati quando vengono eseguiti gli overload degli operatori e operazioni simili.

```
class A {
  private:
    int a;
  public:
    friend getA();
};

A::getA(){
  return a;
}
```

NAMESPACE E OVERLOADING DEI METODI

Name clash si ha quando una funzione si chiama allo stesso modo in 2 file “.h”.

I **namespace** includono un livello di scope più alto e possono essere considerati dei veri e propri contenitori di nomi.

Si ha overloading di funzioni quando queste hanno lo stesso nome ma fanno cose diverse.

Overloading é quando le **STESSE FUNZIONI CON LO STESSO NOME** prendono in input diversi parametri.

L’overloading serve per evitare di chiamare una funzione che fa la stessa cosa delle altre, utilizzando moltissimi nomi di funzioni.

Il compilatore capisce quale funzione deve considerare in base al numero e tipi di parametri passati in input, ovvero in base ai parametri attuali nel main.

In caso ambiguitá con altre funzioni caso vengono fatte conversioni, promozioni.

Se si vuole creare una funzione che prende in input un numero di parametri variabile, è possibile realizzarla ed essa viene chiamata **ELLIPIS**.

1. si include #include <cstdarg>;
2. si usa **va_list** list;
3. **va_start** (nCount, list) // da il numero degli elementi della lista elementi che vengono passati alla funzione;
4. **va_arg**(list,int) é possibile consumare gli argomenti della lista.

CREAZIONE,COPIA,DISTRUZIONE DI OGGETTI

Creazione

Alla creazione dell’oggetto viene invocato il **metodo costruttore** se esso è definito all’interno della classe.

Un costruttore può prendere in input dei valori che serviranno per **inizializzare i vari attributi** dell’oggetto. Se tale costruttore è definito, alla creazione dell’oggetto mediante chiamata a costruttore, è necessario **specificare i vari parametri** (se non esiste un costruttore di default).

Il **costruttore di default** non prende in input nessun parametro e inizializza gli attributi con dei valori predefiniti dall’utente.

Il costruttore che prende in input dei valori standard è quel costruttore che ha come **parametri dei valori settati** già all’interno dei parametri formali.

Chiaramente è anche possibile fare overload di costruttori affinchè la costruzione dell’oggetto diventi più flessibile.

Con un costruttore è possibile definire la **lista di inizializzazione** che serve per creare l’oggetto chiamando altri costruttori di altre classi (gerarchia delle classi). Questa lista di inizializzazione può essere omessa se nella superclasse è presente il costruttore di default.

Copia

Il costruttore di copia permette di copiare due oggetti fra loro secondo dei nostri criteri. La sua sintassi è:

```
class X{
  X(int,float) ;
```

```
  X(const X &); // costruttore di copia
};
```

Se il costruttore di copia non è specificato, in caso di un suo utilizzo all'interno del main, verrà eseguita **una copia membro a membro** e a volte può essere sconveniente perchè non si è mai sicuri di **COME** è avvenuta la copia.

Distruzione

Durante la creazione di un oggetto è molto probabile che **vengano allocate in maniera dinamica** degli attributi. Alla fine del programma, tutto quello che è stato allocato in memoria viene distrutto in maniera automatica. Non verrà liberata la parte di memoria occupata dinamicamente, pertanto è possibile avere casi di ***memory leak***. Per risolvere questo possibile problema, si definisce il metodo distruttore in questo modo:

```
class X{
  private:
    int* x;
  public:
    X(int,float);
    X (const X &);
      ~X(){
        delete x; //distruttore
      }
};
```

OVERLOADING DEGLI OPERATORI

Gli **operatori di base** (somma, differenza, moltiplicazione, parentesi tonde/quadre ecc..) sono implementati di **default** dal compilatore per venire utilizzati con variabili di tipo primitivo (int, double, string ecc..)

Per **overload degli operatori** si intende un'estensione degli utilizzi base degli operatori. Il sovraccaricamento serve per dare la possibilità al programmatore di utilizzare gli operatori con gli oggetti. Si può fare overload di un qualsiasi operatore ed tale procedura è possibile farla:

- **a membro**, ovvero ***all'interno della classe stessa***, quindi avendo accesso agli attributi della classe;
- **non a membro**, ovvero ***all'esterno della classe***. In questo caso, per avere accesso agli attributi privati e metodi della classe, si può utilizzare copiare la firma della funzione overloading all'interno della classe e dichiararla **friend**.

Esempi di overload di alcuni operatori

```
// ++ postfisso in classe figlia
//nel main quello che c'è sotto, no virtual, solo a membro
        B operator++(int a){
            B aux = *this;
```

```
            p++; //p è private di B
            return aux;
        }

    B* aux=nullptr;
    for(int i=0; i<DIM && aux==nullptr;i++){
        if(typeid(*vett[i]) == typeid(B)){
            cout << i << ")" << *vett[i] << endl;
            aux= ((B*)vett[i]);
            (*aux)++;
            cout << i << ")" << *vett[i] << endl;
            // cout << i << ")" <<*auxx;
        }
    }

// ++ prefisso classe figlia
// nel main quello che c'è sopra, no virtual,solo a membro
        C operator++() {
            ++y;
            return *this;
        }

// [] o membro o non a membro, lo stesso
char& A::operator[](int i) {
    return str[i % A::getLen()];
}

cout << "vett[5]=" << (*vett[0])[5] << endl;
(*vett[0])[5]= 1;
cout << "vett[5]=" << (*vett[0])[5] << endl;

//
string A::operator () (int i1, int i2) {
    string s = "";
    for(int i = i1; i < i2; i++) {
        s += A::get(i);
    }
    return s;
}
cout << "*vett[5]=" << *vett[5];
cout << "vett[5](1, 3)=" << (*vett[5])(1, 3) << endl << endl;

// +
short operator+(A& a1, A& a2) {
    return a1.getLen() + a2.getLen();
}
cout << (*vett[5]) + (*vett[6]) << endl;

cout <<(*vett[0])[5] << endl;
(*vett[0])[5]= 1;
cout << (*vett[0])[5];
```

EREDITARIETÀ

Per parlare di ereditarietà di classi, è necessario introdurre alcuni termini:

- classe **madre**, o **superclasse**, è quella classe che fa “capo” a tutte le altre. Essa può essere una **interfaccia** se ha *TUTTI* i metodi **virtual puri**(cioè uguali a 0) o **una classe astratta** se ha *ALMENO* un metodo virtuale. Chiaramente, nel main, non possono essere istanziati oggetti di classi astratte;

- classe **figlia**, o **sottoclasse**, è quella classe che eredita attributi e metodi dalla superclasse di riferimento.

Una sottoclasse può ereditare in vari modi e ciò dipende dal modificatore di accesso:

1. **PUBLIC** → class B: **public** A {}; In questo modo vengono ereditati tutti i metodi e attributi della classe A e non importa se sono private, protected oppure public;
2. **PROTECTED** → class B: **protected** A{}; In questo modo tutto quello che è scritto sotto public (in A) diventa protected in B;
3. **PRIVATE** → class B: **private** A{}; In questo modo tutto quello che è scritto sotto public e protected (in A) diventa private in B.

Overriding

Con overriding si intende una “specializzazione” di un metodo definito **virtual** nella superclasse. Facendo overriding si fa automaticamente overloading nelle sottoclassi.

> *E’ come fare overloading ma fra classi di una “famiglia”.*

```
class A{
  public:
    virtual print()=0;
};

class B: public A{
  public:
    print(){
      //. . .
    }
};
```

Polimorfismo

Si parla di polimorfismo quando un oggetto può assumere forme, aspetti, modi di essere diversi secondo le varie circostanze. In questo caso si ha un’interfaccia comune in una gerarchia ereditaria:

- **gerarchia di classi** con metodi che hanno lo stesso nome, ovvero overriding;
- **differenti implementazioni** per lo stesso metodo all’interno della gerarchia.
- **puntatori** (o **references**)di un qualche tipo X che **puntano** (**sono alias di**) a oggetti di classi derivate da X.

A **tempo di compilazione** non è noto il tipo di oggetto referenziato da puntatori e reference.

Se i metodi dell’interfaccia sono dichiarati **virtual** allora si ha polimorfismo a run-time.

```
class X {
  virtual void foo();
};
class Y: public X{
  void foo();
};
class Z:public Y {
  void foo();
};
int main(){
  Y y,Z z;
  X* ptr_y=&y; //puntatore a oggetto di tipo Y
  X& ref_z=z; // puntatore a oggettodi tipo Z

    ptr_y -> foo();//invocazione di Y::foo()
    ref_z.foo(); //invocazione di Z::foo()
}
```

Quindi, in breve:

- Una classe polimorfa definisce un'interfaccia comune a differenti implementazioni;
- Le differenti implementazioni sono presenti in classe derivate (direttamente o indirettamente) della classe polimorfa(gerarchia ereditaria);
- La "catena" di derivazioni deve essere pubblica;
- Le funzioni dell'interfaccia devono essere di tipo virtual;
- Il polimorfismo a tempo di esecuzione avviene mediante puntatori e/o reference.

> *Approfondisci slide:*
> *https://www.dmi.unict.it/farinella/Prog1/Lezioni/30_polimorfismo202122.pdf*

RTTI: Run Time Type Identification

E' necessario se è possibile fare un cast a tempo di compilazione (run-time) quando si parla di ereditarietà. Per fare un cast si usa solitamente static_cast oppure il cast "implicito" usando le parentesi tonde prima della variabile.
Lo **static_cast** è un metodo **non sicuro** in quanto, se non è possibile effettuare il cast, *non darà nessun tipo di "ritorno interessante" per il programmatore*.

In questo caso si può utilizzare il **dynamic_cast** che restituisce un **nullptr** se il **cast non è possibile** eseguirlo. In tal caso è possibile "castare" **dentro un if** e risolvere eventuali problemi di cast impossibile.

Questo metodo serve anche per *"monitorare"* l'andamento del cast durante il programma.

FUNZIONI TEMPLATE

Servono per gestire i tipi dei parametri formali passati alle funzioni o metodi.

```
template <class T, class C> //sono due classi di tipo differente
void foo(T &a, C &b);
//la funzione prende in input dei tipi generici

int main(){
foo(4,10) // specializzazione foo (int, int)
foo(3.4f, 6.4f) //specializzazione foo(float, float)
}
```

La funzione è specializzata per essere eseguita qualsiasi essi siano i tipi parametri attuali passati ad essa e questo si fa per evitare di definire troppe funzioni in overload.

A differenza dell'overload, le specializzazioni sono generate solamente quando la funzione viene richiamata.

La *class T* di tipo generico può essere anche di tipo primitivo e non per forza una classe (oggetto).

Se si dichiara un template, il parametro generico T **deve figurare tra i parametri formali della funzione**.

Quando si hanno parametri formali di classe T, nella funzione chiamante non posso inserire dei parametri attuali di tipi diversi fra loro.

```
template <class T>
  void foo (T a, T b) {....}

int main(){
  foo(10.5f, 29); //errore di compilazione. tipi differenti anche se generici. Devono essere dello
                  //stesso tipo
}
```

È possibile definire una o più funzioni non generiche:

```
1  template <class T> void print(T a, T b); //(1)
2  template <class C, class T> void print (T a, C b); //(2)
3  void print (double  a, double b); //(3)
```

La funzione 3 sarà invocata qunado si avranno parametri attuali 2 di tipo di double. altrimenti verranno richiamate le funzioni generiche.

```
1  template <class T>
2  T max(T p1, T p2){
3    if( p1 < p2 )
4      return p2;
5    else return p1;
6  }
```

In questo caso si deve essere sicuri che il tipo passato alle specializzazioni possano essere confrontati. Altrimenti bisogna fare un overloading dell'operatore **MINORE**. Un classico esempio è quando si crea una classe e quest'ultima viene passata alla funzione.

CLASSI TEMPLATE

```
1  template<typename T>
2    class Stack{
3      //...
4      T *ptr;
5      //...
6      void push(T);
7    };
```

Con **classe template** si vuole dire che quella classe lavorerà con delle variabili generiche

Se un metodo viene definito fuori dal corpo di una classe, essa va dichiarata come template. All'interno delle classi non è necessario farlo.

```
1  template <class T>
2  class Stack{
3     //...
4  }
5  //...
6  Stack<int> Interi;
7  Stack<double> Reali;
```

RICERCA, ORDINAMENTO E RICORSIONE

La ricerca di elementi è possibile applicarla quando si parla di una array. Tale ricerca si può fermare alla prima occorrenza oppure si può scorrere tutto l'array e vedere quante occorrenze ci sono al suo interno (tutto dipendente dal tipo di problema).

Ricerca dicotomica

DICOTOMICA = divisione in due parti in un array ordinato.

Sia A un **array ordinato** (in modo crescente o decrescente).

Si confronta la chiave di ricerca con l'emento centrale (M) dello array :

- se sono uguali, l'elemento è stato trovato;
- se la chiave di ricerca è minore di M, la ricerca prosegue iterativamente nella **prima metà di A**;
- se la chiave di ricerca è maggiore di M, la ricerca prosegue nella **seconda metà di A**;

Ordinamento

Esistono diversi tipi di algoritmi di ordinamento di un array: Bubble Sort, Selection Sort e Insertion Sort.

Bubble Sort

In caso di ordinamento crescente si usa la tecnica che consiste nel **far scorrere gli elementi massimi verso destra** come se fossero delle bolle che man mano escono dall'array. Alla fine del primo ciclo verrà sistemato il primo massimo. Al secondo ciclo verrà sistemato il secondo massimo e così via fino alla fine. Finchè ci sarà uno swap l'algoritmo andrà avanti. Quando non si faranno più swap vorrà dire che l'array sarà ordinato.

```
bool swapped = true;
while(swapped){
  swapped = false;
  for(int i = 0; i < length-1; i++ )
    if ( array[i] > array[i+1] ){
      swap( array , i , i+1 );
      swapped = true;
    }
}
```

si effettua un certo numero di visite dell'intero array:

1. ad ogni visita si confrontano **coppie di elementi contigui**:
2. se il primo valore e maggiore del secondo, essi vengono scambiati;
3. Se durante una visita non avviene alcuno scambio, ciò significa che l'array e ordinato, dunque l'algoritmo può terminare.

Selection Sort

Questa tecnica consiste nell'andare a selezionare ogni volta il minimo e posizionarlo correttamente. Questo procedimento avviene secondo i seguenti passi:

1. ricerca il minimo dello intero array (A[0 . . . n-1])
2. scambia il minimo con l'elemento di posto zero;
3. ricerca il minimo nel sotto-array A[1. . . n-1];
4. scambia il minimo con l'elemento di posto 1;
5. ecc..

```
1   for (int index = 0; index < length-1; index++)
2     {
3           //selezione o ricerca del minimo
4           min = index;
5           for (int i = index+1; i < length; i++)
6               if (array[i] < array[min])
7                   min = i;
8
9           //scambia minimo ed elemento di indice index
10          swap(array, min, index);
11    }
```

Insertion Sort

L'Insertion Sort si basa sull'inserimento di ogni elemento in un sotto-array ordinato. Funziona secondo i seguenti passi:

1. si consideri il **primo elemento** dell'array. Esso rappresenta un **sotto-array di lunghezza 1**;
2. si **inserisca** il **secondo elemento** al **posto giusto** nel sotto-array ordinato di lunghezza 1;
3. se questo è il minore del primo e unico elemento, quest'ultimo si sposterà a destra;
4. si **inserisca** al posto giusto **il terzo elemento**, spostando spostando i valori dell'array affinchè **l'ordinamento venga mantenuto**.
5. ecc..

Esempio di output.

[8] 4 6 1 2 7 5 3

[4 8] 6 1 2 7 5 3

[4 6 8] 1 2 7 5 3

[1 4 6 8] 2 7 5 3

[1 2 4 6 8] 7 5 3

[1 2 4 6 7 8] 5 3

[1 2 4 5 6 7 8] 3

Ricorsione

Si ha una ricorsione quando una funzione richiama se stessa per essere computata. Classico esempio è quello del fattoriale. Quello che è ricorsivo può essere scritto anche in maniera non ricorsiva.

```
long fattoriale(int n){
  if(n == 0)
    return 1;
  return
    n * fattoriale(n-1); //chiamata ricorsiva
}
```

Vantaggi:

- (+) **intuitivamente più semplice** da concepire;
- (+) **minor** numero di **linee di codice**;

Svantaggi:

- (-) **consuma molta memoria** rispetto ad una soluzione iterativa.
- (-) **consuma molto tempo** rispetto ad una soluzione iterativa;