# Lezione 02: Tipi fondamentali

## Tipi di dato in C++

In C++ possiamo distinguere i tipi `built-in` ed i tipi `user-defined`.
Di seguito partiamo indicando i tipi built-in:

- Bool
- char
- int(long, long long, ecc...)
- float
- double(long)

Come possiamo notare dall'elenco superiore alcuni tipi built-in hanno varie versioni con lunghezza e precisione differente, per problemi "normali" sono i tipi più utilizzati.
Di seguito elenchiamo i definiti dall'utente;

- enum
- classes
- tipi introdotti dalle librerie standard

I tipi possono essere mescolati nelle espressioni e possiamo controllarne la dimensione con l'operatore `sizeof`. La dimensione può tuttavia dipendere dall'implementazione, questo è possibile perché il linguaggio mette a disposizione svariati strumenti che agevolano in programmatore ma va evitato se non strettamente necessario per garantire la portatilità,

## Inizializzazioni

In C++ esistono quattro tipi diversi di inizializzazioni, di seguito elencate:

1. graffe(list initialization): int a1 {15};
2. uguale + graffe: int a2 = {15};
3. uguale: int a3 = 15;

4. tonde: int a4 (15);

Il primo metodo è sicuramente quello da preferire poiché controlla che eventuali conversioni non perdano informazione. In mancanza di un'effettiva inizializzazione viene attribuito da compilatore un valore di default anche se risulta consigliabile dichiararlo esplicitamente per evitare errore. Ovviamente è possibile effettuare anche inizializzazioni multiple in caso ad esempio di array.

```
int a[] {2,3,4}
```

# Tipo void

Non possono esistere oggetti di tipo void in C++, esso si può utilizzare solamente in due casi;

1. per indicare che una funzione non restituisce nulla;
2. per indicare il tipo di un puntatore ad un oggetto di tipo sconosciuto.

# Costanti

In C++ esistono due diverse tipologie di costanti:

- **Const:** questo identificatore sta ad indicare che il valore non può essere modificato dopo l'inizializzazione, esso può però essere determinato a runtime.
  Utile per variabili locali, parametri, puntatori, ecc...
- **Constexpr:** questo identificatore sta ad indicare che il valore deve essere noto a compile-time. Esso è quindi utilizzato per definire costanti che devono essere valutate dal compilatore come array di dimensione fissa, template, ecc...

```
//esempio comparativo tra const e constexpr

const int a = std::rand();    // OK (runtime value)
constexpr int b = 42;         // OK (compile-time value)
constexpr int c = a + 1;      // Errore! 'a' non è constexpr
```

# Puntatori e riferimenti

Un `puntatore` è una variabile che contiene l'indirizzo di un'altra variabile

```
int x = 10;
int *ptr = &x; // ptr contiene l'indirizzo di x
```

Elenchiamo di seguito le principali caratteristiche di un puntatore:

- Possono essere **null** (nullptr).
- Possono essere **riassegnati** (puntare ad altro).
- Bisogna **dereferenziarli** con * per accedere al valore puntato.
- Possono creare **puntatori a puntatori(int **).**

```
int y = 20;
ptr = &y; // ora ptr punta a y
*ptr = 30; // cambia y in 30
```

Un `riferimento` è un **alias**, cioè un altro nome per una variabile già esistente:

```
int x = 10;
int& ref = x; //ref è un alias per x
```

Elenchiamo di seguito le principali caratteristiche di un riferimento:

- Non può essere **null.**
- Deve essere inizializzato subito.
- Non può essere cambiato per riferirsi a qualcos'altro dopo.
- Non serve dereferenziale: ref si una come una normale variabile.

# Aritmetica dei puntatori

L'**aritmetica dei puntatori** consente di eseguire operazioni aritmetiche su puntatori per navigare tra indirizzi di memoria, particolarmente utile per gestire

array e strutture dati contigue in memoria. Forniamo di seguito un elenco delle operazioni consentite:

- **p + n** : sposta il puntatore avanti di n elementi (non byte).
- **p - n** : sposta il puntatore indietro di n elementi.
- **p2 - p1** : restituisce la distanza in numero di elementi tra due puntatori dello stesso array.
- **p++ / p—** : avanza o arretra di un elemento.

```
#include <iostream>

int main() {
    int a[] = {10, 20, 30, 40, 50};
    int* p = a;          // Puntatore al primo elemento

    std::cout << "Valore iniziale: " << *p << "\n";       // 10

    // Somma (p + n)
    std::cout << "Terzo elemento (p + 2): " << *(p + 2) << "\n";  // 30

    // Incremento (p++)
    p++;  // ora p punta a 20
    std::cout << "Dopo p++: " << *p << "\n";             // 20

    // Decremento (p--)
    p--;  // torna a 10
    std::cout << "Dopo p--: " << *p << "\n";             // 10

    // Sottrazione tra puntatori (p2 - p1)
    int* p2 = a + 4;  // punta a 50
    int distanza = p2 - p;  // 4
    std::cout << "Distanza tra p2 e p: " << distanza << "\n";

    // Sottrazione (p - n)
    int* p3 = p2 - 2;  // dovrebbe puntare a 30
```

```
    std::cout << "Elemento due prima di p2: " << *p3 << "\n"; // 30

    return 0;
}
```

## Puntatori a void

Un `void*` è un puntatore generico: può puntare a qualsiasi tipo di dato, ma **non può essere dereferenziato direttamente** (perchè il compilatore non sa che tipo di dato contiene).
Esso è spesso utilizzato per funzioni generiche, interfacce C ed allocazione dinamica.

```
int x = 42;
void* p = &x;       // Ok: p punta a un int, ma in modo generico

// Per usare il valore:
std::cout << *(static_cast<int*>(p));  // Necessario il cast!
```

## Puntatori costanti

Esistono due tipologia diverse di puntatori costanti;
i
`puntatori a dato costante` ed i `puntatori costanti.`

Nel caso di puntatori a dato costante il dato modificato non può essere modificato tramite il puntatore:

```
const int* p = &x;
*p = 5;       // Errore
p = &y;       // Ok, può puntare altrove
```

In caso di puntatore costante il puntatore non può essere spostato ma può modificare il dato puntato:

```
int* const p = &x;
*p = 5;       // Ok, modifica x
p = &y;       // Errore
```

Possiamo infine avere il caso della doppia costanza dove né il dato né il puntatore possono cambiare:

```
const int* const p = &x;
*p = 5;       // ❌ Errore
p = &y;       // ❌ Errore
```

# Array

Un `array` è una **collezione di elementi dello stesso tipo,** memorizzati in **blocchi contigui di memoria.** Ogni elemento è accessibile tramite indice (a partire da zero).

Di seguito mostriamo in maniera pratica come funzionano dichiarazione ed accesso agli elementi di un array:

```
int numeri[5];              // array di 5 interi (non inizializzati)
int voti[3] = {90, 85, 88};  // inizializzato
int valori[] = {1, 2, 3};  // dimensione = 3, in questo caso il compilatore riesce a de

std::cout << voti[0]; //stampa 90
voti[1] = 95; //modifica secondo elemento
```

Elenchiamo di seguito le caratteristiche principali:

- La **dimensione è fissa(**se non usi std::vector).
- Gli **elementi sono continui** in memoria.
- Un array può essere passato a funzioni come **puntatore** al primo elemento.
- Nessun **controllo di limiti**: accedere fuori dai limiti è **comportamento indefinito.**

In C++ abbiamo però delle alternative moderne al classico array che tutti conosciamo, ossia:

- std::array<T, N>: array **a dimensione fissa**, con funzioni utili.
- std::vector<T>: **array dinamico**, molto più flessibile.

```
#include <array>
std::array<int, 3> arr = {1, 2, 3};

#include <vector>
std::vector<int> vec = {1, 2, 3};
vec.push_back(4);  // dinamico
```

## Stringhe

Come in C, anche in C++ le `stringhe` classiche non sono altro che array di caratteri terminati da **carattere nullo** (`'\0'`).

```
char saluto[] = "Ciao";    // automatico: {'C','i','a','o','\0'}
char* msg = "Hello";       // puntatore a stringa letterale (costante)
```

Elenchiamo di seguito le caratteristiche principali:

- Serve spazio per `'\0'`.
- Usano funzioni di `<cstring>` (es: strlen, strcpy, strcmp, ecc...).
- Più **manuali** e soggette a errori(buffer overflow, ecc...).

In C++ esistono poi nella libreria standard delle stringhe più avanzate, che sono molto più comode e versatili.

```
#include <string>
std::string nome = "Giulia";
```

Rispetto alle stringhe standard hanno i seguenti vantaggi:

- Gestione automatica della memoria.

- Overload di operatori(+, ==, [], ecc...).
- Funzioni pratiche: length(), substr(), find(), append(), ecc...

```
std::string s1 = "Ciao";
std::string s2 = " mondo";
std::string risultato = s1 + s2;  // "Ciao mondo"
```

## Lvalue ed Rvalue

Un **lvalue** (”left value”), è qualcosa che ha un nome e un indirizzo in memoria, inoltre vi è possibile assegnargli un valore, ad esempio:

```
int x = 10;
x = 20;       // x è un lvalue
int* p = &x;  // puoi ottenere l’indirizzo
```

Un **rvalue** (”right value”) è un valore temporaneo, senza nome, che non ha un indirizzo stabile.
È un valore che non può stare a sinistra di un
=.

```
int y = 5 + 3;     // 5 + 3 è un rvalue
int z = y * 2;     // y * 2 è un rvalue
```