

# Tipi fondamentali, puntatori e riferimenti

Anna Corazza

aa 2023/24

# Dove studiare

- ▶ Str'13, capitoli 6 e 7
- ▶ Str'14, capitolo 8

**Str'13** Bjarne Stroustrup, The C++ Programming Language (4th edition), 2013

<https://www.stroustrup.com/4th.html>

**Str'14** Bjarne Stroustrup, Programming: Principles and Practice using C++ (2nd edition), 2014 <https://www.stroustrup.com/programming.html>

# Tipi di dato in C++

built-in      {    **bool**  
                  {    **char**  
                      {    **int**, long int, long long int, ...  
                          float, **double**, long double

user-defined    {                    enum  
                                      classes  
                      tipi introdotti dalle librerie standard

- ▶ Per i tipi char, integer e float esistono diverse versioni con lunghezze diverse.
- ▶ Per problemi “normali” possiamo usare quelli in bold e considerare il resto come variazioni per casi particolari.

# Tipi, variabili e aritmetica

- ▶ I tipi possono essere mescolati nelle espressioni: il C++ fa scelte ragionevoli.
- ▶ Possiamo controllare la dimensione con `sizeof`.
- ▶ Tuttavia la dimensione può dipendere dall'implementazione ed è **super importante** che il nostro programma sia portabile.
- ▶ Questo è vero **sempre**: non posso rischiare che non funzioni più con una nuova versione del mio compilatore!
- ▶ La ragione per prevedere tipi di dimensione diversa è quella di permettere allo sviluppatore di sfruttare l'hardware al meglio:
- ▶ ... ma questo nuoce alla portabilità e quindi va sfruttato solo quando **davvero** necessario
- ▶ Di sicuro non per quello che facciamo in questo corso, in cui invece dobbiamo dare priorità alla portabilità

# Inizializzazioni

- ▶ Quattro forme diverse:

1. graffe (list initialization): `int a1 {15} ;`
2. uguale + graffe: `int a2={15};`
3. uguale: `int a3=15;`
4. tonde: `int a4(15);`

- ▶ La prima è da preferire: controlla che eventuali conversioni di tipo non perdano informazione.
- ▶ In mancanza di inizializzazione, viene assegnato un valore di default: molto più leggibile e meno prone a errori farlo esplicitamente!
- ▶ Naturalmente non è detto che per inizializzare basti un solo valore: ad esempio, vettori:

---

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

---

# Il tipo booleano

- ▶ Astrattamente: può assumere solo due valori: `true` e `false`.
- ▶ In realtà, corrispondono a 0 e ogni cosa diversa da 0 (1 in stampa).
- ▶ Nelle espressioni, i booleani vengono convertiti in interi e i conti vengono eseguiti coi numeri interi.
- ▶ Se il risultato è 0, viene restituito `false`, per ogni altro valore `true`.
- ▶ Addirittura un puntatore può venir convertito implicitamente in booleano: se il puntatore non è nullo, corrisponde a `true`, altrimenti (`nullptr`) a `false`.

# I tipi carattere

- ▶ Sappiamo che esistono molti insiemi di caratteri (latini vs ideogrammi cinesi, tanto per fare un esempio) e codifiche.
- ▶ Di conseguenza non esiste solo `char`, ma diversi tipi che possono venir incontro a diverse esigenze: con o senza segno, abbastanza grandi da reggere unicode, etc.
- ▶ In condizioni standard, basta `char` (8 bit in quasi tutte le implementazioni).
- ▶ Se (ma non in questo corso) decidete di usare gli altri, attenzione perché possono creare errori subdoli.
- ▶ Letterali: singolo carattere tra apici semplici ('a','3',...)
- ▶ Vengono convertiti nel codice ASCII, ma è meglio (per leggibilità e per evitare errori) usare la notazione con gli apici.
- ▶ Alcuni caratteri speciali usano una notazione con il '\'; ad esempio: '\n', '\t', '\\', '\?', '\ ' , '\"'.

# I tipi intero

- ▶ Anche qui diversi tipi:
  - ▶ con o senza segno (`unsigned`)
  - ▶ diverse dimensioni (`short int`, `int`, `long int`, `long long int`)
- ▶ Usare `unsigned` per guadagnare un bit non ha senso.
- ▶ Gli `int` normali hanno sempre segno.
- ▶ I letterali sono ovviamente tutti i numeri interi, in notazione decimale, ottale (iniziano per 0: 02, 0123) o esadecimale (iniziano per 0x: 0x0, 0x2, 0x3f).
- ▶ Ottali e esadecimali di solito si utilizzano per esprimere pattern di bit.
- ▶ Il tipo di ogni letterale è deciso dal suo suffisso, se c'è (ad esempio: `u` o `U` per `unsigned`, `l` o `L` for `long`, `ll` o `LL` for `long long`) e dal tipo di lunghezza minima che ne contiene il valore.



# Tipi a virgola mobile (floating point)

- ▶ `double` è il default
- ▶ Sono letterali validi: `1.23`, `.23`, `-2.3`, `0.23`, `1.`, `1.0`, `1.2e10`, `1.23e-15` (niente spazi attorno alla `e`)
- ▶ Suffisso `f` se voglio che sia un `float` (`2.0f`)
- ▶ Suffisso `L` per `long double`.

# Tipo `void`

- ▶ Non possono esistere oggetti di tipo `void`.
- ▶ Si usa solo 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

`const` e `constexpr`

`const` : “Prometto di non cambiare questo valore” e il compilatore controlla che sia proprio così.

`constexpr` : “Da valutare a tempo di compilazione”.

- ▶ Possono essere `constexpr` anche funzioni, ma solo se molto semplici (solo un’istruzione di return)
- ▶ Distinzione sottile, che per il momento potete non considerare

# Puntatori e riferimenti

- ▶ Posso accedere ad una variabile
  - ▶ usando il nome
  - ▶ dal suo indirizzo di memoria, tenendo conto del tipo della variabile
- ▶ puntatori e riferimenti servono esattamente a questo, in due modi leggermente diversi

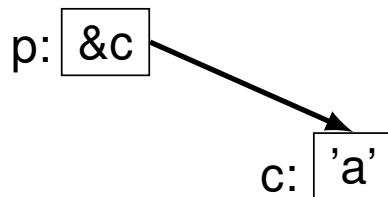
# Puntatori

- ▶ Supponiamo di avere una variabile `nomeVariabile` di tipo `T`
- ▶ Il tipo `T*` è un puntatore ad una variabile di tipo `T`: può memorizzare l'indirizzo di una variabile di tipo `T`
- ▶ Ad esempio:

---

```
char c = 'a';  
char* p = &c;
```

---



- ▶ `c` è una variabile di tipo `char` a cui viene assegnato il valore del letterale `'a'`
- ▶ `p` è una variabile di tipo puntatore a `char` a cui viene assegnato l'indirizzo di memoria in cui si trova `c`

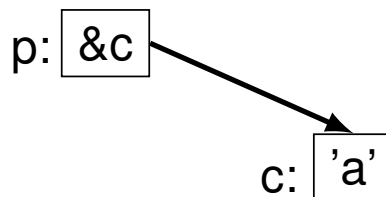
# Puntatori continua

- Continuiamo con l'esempio di prima:

---

```
char c = 'a';  
char* p = &c;  
char c2 = *p;
```

---



- `c2` assume il valore puntato da `p`, nel senso che `c2` assume il valore memorizzato all'indirizzo `p`
- quindi: la memoria viene letta sapendo il tipo della variabile, altrimenti sarebbe impossibile
- L'oggetto puntato da `p` è `c`, il cui valore è `'a'`
- in conclusione: `c2` assume il valore del letterale `'a'`

# Aritmetica dei puntatori

- ▶ Sui puntatori posso eseguire **operazioni aritmetiche**.
- ▶ La loro implementazione si basa proprio sui meccanismi di indirizzamento dell'hardware.
- ▶ La maggior parte dei dispositivi arriva a indirizzare il **byte**
- ▶ Quindi il più piccolo oggetto che può venir allocato e indirizzato indipendentemente è il `char`, che corrisponde ad un byte
- ▶ `bool` occupa almeno tanta memoria quanto un `char`
- ▶ Ottimo se ho bisogno di implementare applicazioni che scendano a quel livello di dettaglio (ad esempio, un sistema operativo)
- ▶ Se invece voglio fare una buona applicazione portabile, meglio non usare queste operazioni di basso livello.

# Altri esempi

- Abbiamo visto che per trasformare un qualsiasi tipo  $T$  nel corrispondente **puntatore a**  $T$  dobbiamo aggiungere al tipo il suffisso  $*$
- Esempi:

---

```
int* pi;          // pi: un puntatore ad  
                  intero  
char** ppc;       // ppc: un puntatore a  
                  puntato a carattere  
int* ap[15];      // ap: un vettore di 15  
                  puntatori a int
```

---



# Altri esempi

Complichiamo un po'

- ▶ Abbiamo visto che per trasformare un qualsiasi tipo  $T$  nel corrispondente **puntatore a**  $T$  dobbiamo aggiungere al tipo il suffisso  $*$
- ▶ Esempi con funzioni:

---

```
int (*fp) (char*); // fp: un puntatore a una
                    funzione
                    // che prende in ingresso un
                    // puntatore
                    // a carattere e restituisce un
                    // intero
int* f(char*) // f: un puntatore a funzione che
              // prende in ingresso
              // un puntatore a carattere e
              // restituisce un intero
```

---

# Puntatore a `void`

`void*`

- ▶ Se dichiariamo una variabile di tipo `void*`, le possiamo assegnare un puntatore ad un qualsiasi tipo di oggetto.
- ▶ Possiamo considerarlo un **puntatore ad un oggetto di tipo sconosciuto**
- ▶ **Però** non può essere né un puntatore a funzione né un puntatore a un membro di una classe (inclusi i membri funzione o metodi)
- ▶ Se ho due variabili di tipo `void*`, posso
  - ▶ assegnare il valore dell'una all'altra
  - ▶ confrontarle (sia eguaglianza che diseguaglianza)
  - ▶ posso convertirle **esplicitamente** ad un altro tipo
- ▶ Qualsiasi altra operazione può essere **pericolosa** e quindi va evitata: dà **errore** a tempo di compilazione
- ▶ Per usare un `void*` dobbiamo convertirlo esplicitamente ad un puntatore di un dato tipo

# Puntatore a void

## Esempi d'uso

---

```
void f(int* pi){
    void* pv = pi; // ok: converto implicitamente a
                    // puntatore a intero
    *pv; // errore: non posso accedere al valore
          puntato da void*
          // si dice <<dereferenziare>>
    ++pv; // errore: non posso incrementare il
           puntatore a void
           // non conosco la dimensione dell'oggetto
           puntato!
    int* pi2 = static_cast<int*>(pv);
              // questo lo posso fare: cast esplicito
    double* pd1 = pv; // errore
    double* pd2 = pi; // errore
    double* pd3 = static_cast<double*>(pv); //
              pericoloso
}
```

---

# Cast esplicito

`static_cast`

- ▶ In genere, usare un puntatore ad un tipo  $T1$  per puntare ad un oggetto di tipo  $T2$  è pericoloso
- ▶ Pensiamo ad esempio che la dimensione di un oggetto può dipendere dall'implementazione ...
- ▶ Cerchiamo quindi di usare questo operatore il meno possibile

# A cosa serve?

`void*`

- ▶ Il puntatore `void*` si usa per operazioni a basso livello sulla memoria
- ▶ In questi casi, gli usi più diffusi di `void*` sono:
  - ▶ voglio passare un parametro ad una funzione senza fare ipotesi sul tipo dell'oggetto puntato
  - ▶ la funzione deve tornare un puntatore ma senza fare ipotesi sul tipo dell'oggetto puntato
- ▶ Se l'operazione è a più alto livello, meglio utilizzare soluzioni basate su un progetto orientato agli oggetti

# nullptr

- ▶ Letterale che rappresenta il **puntatore nullo**, ovvero il puntatore che non punta a nessun oggetto
- ▶ Può venir assegnato a qualsiasi tipo di puntatore, ma non agli altri tipi built-in:

---

```
int* pi = nullptr;  
double* pd = nullptr;  
int i = nullptr; // errore: i non e' un  
                puntatore!
```

---

- ▶ Un unico `nullptr` per qualsiasi tipo di puntatore

# `nullptr` nell'antichità

- ▶ Nessun oggetto può venir allocato all'indirizzo 0 (ovvero il pattern con tutti i bit a zero), quindi una volta si usava l'intero 0 al posto del puntatore nullo
- ▶ Addirittura si definiva una macro `NULL` per rappresentare il puntatore nullo
- ▶ Tuttavia
  - ▶ la definizione di `NULL` dipende dall'implementazione (ad esempio, può essere 0 o 0L)
  - ▶ la definizione usata in C (`(void*) 0`) è illegale in C++
- ▶ Conclusione: usate `nullptr`!

# Array

- ▶ Dato un tipo `T`, `T[size]` indica un vettore di `size` elementi di tipo `T`
- ▶ L'indice va da `0` a `size-1`
- ▶ Esempi:

---

```
float v[3];  
char* a[32];
```

---

- ▶ Due modi di accedere agli elementi del vettore:
  - ▶ tramite `[]`
  - ▶ tramite puntatori (antico)



# Array: accesso agli elementi

- ▶ La struttura dati “vettore” non conserva la dimensione in memoria
- ▶ Quindi deve essere lo sviluppatore a mantenere un comportamento consistente

---

```
void f() {  
    int aa[10];  
  
    aa[6] = 9;  
    int x = aa[99]; // comport. indefinito  
}
```

---

- ▶ Quando si accede ad elementi del vettore fuori dai limiti, cosa succede è indefinito e quindi **pericoloso**
- ▶ Negli array built-in, la dimensione dell'array deve essere un'espressione costante: per usare dimensioni variabili, occorre usare **vector** della libreria standard
- ▶ Gli array multidimensionali sono rappresentati come array di array

# Limitazioni degli array built-in

- ▶ Si tratta di una struttura inerentemente di basso livello, da usarsi essenzialmente per costruire strutture di più alto livello, quali le strutture **vector** e **array** della libreria standard.
  - ▶ Non è possibile assegnare un array ad un altro array.
  - ▶ Come conseguenza, non è possibile passare un array ad una funzione per valore.
  - ▶ Il nome di un array viene implicitamente convertito ad un puntatore al suo primo elemento in molti contesti (poi vediamo meglio)
- ▶ Uno degli array più utilizzati è l'array di char terminato dal char 0: una stringa in stile C: conservata per compatibilità con librerie già esistenti

# Inizializzazione degli array

## Lista di valori

- Un array può venir inizializzato con una lista di valori:

---

```
int v1[] = {1, 2, 3, 4};  
char v2[] = {'a', 'b', 'c', 0}
```

---

- Non esiste un'operazione di copia tra array.
- Di conseguenza, **non** è possibile inizializzare un array usandone un altro:

---

```
int v3[4] = v1; // errore  
int v3 = v1; // errore
```

---

- Se un array viene inizializzato senza specificare la dimensione, ma con una lista di inizializzazione, la dimensione viene calcolata dal **numero di elementi** della lista.

# Inizializzazione degli array

Lunghezza della lista diversa dalla dimensione dell'array

- Se il numero di valori nella lista di inizializzazione è **minore** della dimensione dell'array, la lista viene completata con valori nulli.

---

```
int v4[8] = {1, 2, 3, 4};
```

---

è equivalente a:

---

```
int v4[] = {1, 2, 3, 4, 0, 0, 0, 0};
```

---

- Se tuttavia, oltre alla lista di inizializzazione viene specificata anche la dimensione, il numero di elementi della lista deve essere  $\leq$  della dimensione. Altrimenti si ottiene un **errore**.

---

```
char v5[2] = {'a', 'b', 0} // errore  
char v6[3] = {'a', 'b', 0} // OK
```

---

# Letterali stringa

- ▶ Un vettore di caratteri può venir inizializzato con un letterale stringa. Esempio: `"Sono una stringa"`.
- ▶ Un letterale stringa contiene un carattere in più di quelli che appartengono alla stringa: il carattere di terminazione `'\ 0'` corrispondente a 0.

---

```
sizeof("Bohr") == 5;
```

---

- ▶ Il tipo di un letterale stringa è:
  - ▶ array di caratteri costanti (`const char`)
  - ▶ quindi `"Bohr"` è un `const char[5]`

# Tipi di istruzione non più utilizzate

---

```
void f() {  
    char* p="Plato"; // errore da C++11 in poi  
    p[4]='e';        // errore: assegnamento ad una  
                     costante  
}
```

---

- ▶ I literali stringa sono costanti, quindi **immutabili**.
  - ▶ Se vogliamo usarla come inizializzazione e poi modificarla, dobbiamo usare l'array di caratteri (non costante):
- 

```
void f() {  
    char p[]="Plato"; // p diventa un array di 5  
                     caratteri  
    p[4]='e';        // che posso modificare  
}
```

---

# Stringhe restituite da funzioni

- I letterali stringa sono allocati staticamente, quindi possono essere restituiti da funzioni.

---

```
const char* error_message(int i) {  
    // ...  
    return "range error";  
}
```

---

# Ottimizzazioni

- ▶ Il codice viene ottimizzato, quindi in alcuni casi **può** succedere che due letterali stringa identici non vengano duplicati.
- ▶ Questo significa che non possiamo ipotizzare con certezza cosa succede.

---

```
const char* p="Heraclitus";  
const char* q="Heraclitus";  
  
void g() {  
    if(p==q) cout << "one!\n";  
}
```

---

- ▶ NB: L'espressione `p==q` confronta gli indirizzi, cioè i valori dei puntatori e non degli oggetti puntati.



# Stringa vuota e caratteri non grafici

- ▶ La **lista vuota** è "", ha tipo `const char[1]` e contiene il solo carattere `'\0'`.
- ▶ Per rappresentare i caratteri non grafici all'interno di una stringa possiamo utilizzare il `\`.

---

```
cout << "beep at end of message\a\n";
```

---

- ▶ Un problema possono essere le stringhe contenenti il carattere nullo
  - ▶ Sono perfettamente legali.
  - ▶ La maggior parte dei programmi considererà solo la parte prima del primo carattere nullo: `Jens\000Munk` verrà considerato come "Jens".

# Stringhe su più linee

- ▶ Il carattere di newline non può essere incluso in una stringa: una stringa non può stare su più righe. Devo invece usare `'\n'`
- ▶ Se una stringa è troppo lunga, posso tuttavia spezzarla:

---

```
char alpha[] = "abcdef"  
               "ghijkl";
```

---

è del tutto equivalente a

---

```
char alpha[] = "abcdefghijkl";
```

---

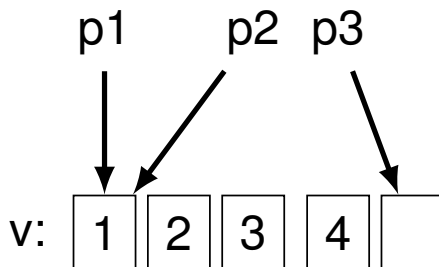
# Array e puntatori

- Legame stretto tra array e puntatori: il nome di un array può venir utilizzato come puntatore al suo primo elemento.

---

```
int v[] = {1,2,3,4};  
int* p1 = v; // conversione implicita:  
            // puntatore al primo elemento  
int* p2 = &v[0]; // stessa cosa, ma esplicita  
int* p3 = v+4; // puntatore all'elemento  
              // dopo l'ultimo
```

---



# Puntare fuori dell'array

- ▶ Puntare ad un elemento immediatamente successivo alla fine dell'array non dà errore, purché non si legga o scriva.
- ▶ Puntare a elementi prima o dopo questo dà risultato indefinito, e quindi non va fatto: è un **errore non segnalato**.
- ▶ In generale, è una cattiva idea **foriera di problemi** quella di convertire implicitamente un array in puntatore: tanto per cominciare si perde l'informazione sulla dimensione dell'array.

# Navigazione degli array

---

```
void fi(char v[]) {  
    for(int i=0; v[i]!=0; i++)  
        use(v[i]);  
}
```

```
void gi(char v[]) {  
    for(char* p=v; p!=0; p++)  
        use(*p);  
}
```

---

- ▶ ▶ La prima è la classica visita di un array in cui ho messo una sentinella se non conosco a priori la lunghezza dell'array.
- ▶ La seconda risulta molto meno leggibile.
- ▶ Non c'è nessuna ragione per cui una delle due dovrebbe essere più efficiente dell'altra.
- ▶ Un compilatore moderno genera normalmente lo stesso codice per entrambe.
- ▶ La prima è preferibile.

# Array multidimensionali

- ▶ Gli array multidimensionali sono rappresentati come **array di array**
- ▶ Esempio 3x5:

---

```
int ma[3][5];
```

```
void init_ma() {  
    for(int i=0; i<3; i++)  
        for(int j=0; j<5; j++)  
            ma[i][j] = 10*i+j;
```

```
void print_ma() {  
    for(int i=0; i<3; i++)  
        for(int j=0; j<5; j++)  
            cout << ma[i][j] << '\t';  
    cout << '\n';
```

- 
- ▶ NB: come per gli array ad una sola dimensione, le dimensioni non sono memorizzate in alcun modo nella struttura dati array.

# Array come argomento di funzioni

- Un array non può venir passato ad una funzione per valore, ma con un puntatore al suo primo elemento.

---

```
void comp(double arg[10]{
    for(int i=0; i < 10; i++)
        arg[i]+=99;
}

void f(){
    double a1[10];
    double a2[5];
    double a3[100];
    comp(a1);
    comp(a2); // disastro!
    comp(a3); // usa solo i primi 10
               elementi di a3
}
```

---

- Vengono modificati gli elementi del vettore e non di una sua copia.

# Riferimenti

- ▶ Un riferimento (reference) può essere visto come un nome alternativo per un oggetto, un alias.
- ▶ **Analogamente** ai puntatori:
  - ▶ è un alias per un oggetto
  - ▶ è implementato per conservare l'**indirizzo** di un oggetto
  - ▶ non richiede maggiori risorse computazionali
- ▶ **Diversamente** dai puntatori, il riferimento:
  - ▶ la sintassi è la stessa che avrei col nome dell'oggetto
  - ▶ si riferisce sempre all'oggetto con cui è stata inizializzata
  - ▶ non esiste un "riferimento null" da controllare: ogni riferimento si riferisce ad un oggetto



---

```
#include <iostream>
using namespace std;

int main(){
    int x = 10;

    // ref diventa un alias di x
    int& ref = x;

    // Il valore di x viene cambiato!
    ref = 20;
    cout << "x = " << x << '\n';

    // Se cambio il valore di x, cambia anche
    // quello di ref 30
    x = 30;
    cout << "ref = " << ref << '\n';

    return 0;
}
```

---

# Cosa ci si guadagna?

- ▶ Tra i vantaggi dei puntatori c'è quello di poter passare ad esempio ad una funzione una grande quantità di dati a basso costo.
- ▶ Però un puntatore ha una sintassi un po' pesante ...
- ▶ ... e può cambiare valore nel tempo.
- ▶ Inoltre bisogna sempre gestire la possibilità che il puntatore non stia puntando a nulla (`nullptr`).
- ▶ Coi riferimenti non ho questi problemi.