# Puntatori

**Mattia Curri - m.curri8@studenti.uniba.it**

**Informazioni sul tutorato (ufficiali)**:
https://www.uniba.it/it/ricerca/dipartimenti/informatica/orientamento-pcto-tutorato/tutorato/info-per-studenti/informazioni-per-utenza

**Materiale**: https://mattiacurri.github.io/tutorato

Un puntatore è una variabile che memorizza l'indirizzo di memoria di un'altra variabile.

Un puntatore è una variabile che memorizza l'indirizzo di memoria di un'altra variabile.

```c
#include <stdio.h>

int main() {
    int x = 5;

    printf("Valore di x: %d\n", x);
}
```

Un puntatore è una variabile che memorizza l'indirizzo di memoria di un'altra variabile.

```c
#include <stdio.h>

int main() {
    int x = 5;
    int *y;

    printf("Valore di x: %d\n", x);
    printf("Valore di y: %p\n", y);
}
```

```c
#include <stdio.h>

int main() {
    int x = 5;
    int *y = 0; // 0 e' un indirizzo speciale, e' l'indirizzo nullo

    printf("Valore di x: %d\n", x);
    printf("Valore di y: %p\n", y);
}
```

```c
#include <stdio.h>

int main() {
    int x = 5;
    int *y = 0; // 0 e' un indirizzo speciale, e' l'indirizzo nullo
    // Alternativa sintattica: int *y = NULL;

    printf("Valore di x: %d\n", x);
    printf("Valore di y: %p\n", y);
}
```

```c
#include <stdio.h>

int main() {
    int x = 5;
    int *y = 0; // 0 e' un indirizzo speciale, e' l'indirizzo nullo
    // Alternativa sintattica: int *y = NULL;

    printf("Valore di x: %d\n", x);

    y = &x; // L'operatore & restituisce l'indirizzo di una variabile
    printf("x e' memorizzato all'indirizzo: %p\n", y);
}
```

L'operatore & significa "non voglio il valore della variabile, voglio l'indirizzo in memoria in cui e' memorizzata".

Perchè devo specificare il tipo di dato del puntatore, visto che sono sempre indirizzi di memoria (e quindi numeri)?

Perchè devo specificare il tipo di dato del puntatore, visto che sono sempre indirizzi di memoria (e quindi numeri)?

```c
#include <stdio.h>

int main() {
    int x = 5;
    int *y = 0; // 0 e' un indirizzo speciale, e' l'indirizzo nullo
    // Alternativa sintattica: int *y = NULL;

    printf("Valore di x: %d\n", x);

    y = &x; // L'operatore & restituisce l'indirizzo di una variabile, "dammi l'indirizzo di x"
    printf("x e' memorizzato all'indirizzo: %p\n", y);

    *y = 10; // Con l'asterisco * dico "all'indirizzo di memoria contenuto in y, scrivi il valore 10"
    printf("Nuovo valore di x: %d\n", x);
}
```

Perchè devo specificare il tipo di dato del puntatore, visto che sono sempre indirizzi di memoria (e quindi numeri)?

```c
#include <stdio.h>

int main() {
    int x = 5;
    int *y = 0; // 0 e' un indirizzo speciale, e' l'indirizzo nullo
    // Alternativa sintattica: int *y = NULL;

    printf("Valore di x: %d\n", x);

    y = &x; // L'operatore & restituisce l'indirizzo di una variabile, "dammi l'indirizzo di x"
    printf("x e' memorizzato all'indirizzo: %p\n", y);

    *y = 10; // Con l'asterisco * dico "all'indirizzo di memoria contenuto in y, scrivi il valore 10"
    printf("Nuovo valore di x: %d\n", x);
}
```

Perchè il compilatore deve sapere quanti byte leggere/scrivere a quell'indirizzo di memoria.

## Una strana equivalenza tra puntatori e array in C

```c
#include <stdio.h>

int main() {
    int x = 5;
    int *y = 0; // 0 e' un indirizzo speciale, e' l'indirizzo nullo
    // Alternativa sintattica: int *y = NULL;

    printf("Valore di x: %d\n", x);

    y = &x; // L'operatore & restituisce l'indirizzo di una variabile, "dammi l'indirizzo di x"
    printf("x e' memorizzato all'indirizzo: %p\n", y);

    y[0] = 10; // equivale a *y
    printf("Nuovo valore di x: %d\n", x);
}
```

## Una strana equivalenza tra puntatori e array in C

```c
#include <stdio.h>

int main() {
    int x = 5;
    int *y = 0; // 0 e' un indirizzo speciale, e' l'indirizzo nullo
    // Alternativa sintattica: int *y = NULL;

    printf("Valore di x: %d\n", x);

    y = &x; // L'operatore & restituisce l'indirizzo di una variabile, "dammi l'indirizzo di x"
    printf("x e' memorizzato all'indirizzo: %p\n", y);

    y[1] = 10; // ora sto scrivendo nell'intero immediatamente dopo x, siamo fuori dai limiti!
    printf("Nuovo valore di x: %d\n", x);
}
```

Ma a cosa serve assegnare a `y` l'indirizzo di memoria di `x`, per poi modificare il valore di `x` passando da `y`?

Ma a cosa serve assegnare a `y` l'indirizzo di memoria di `x`, per poi modificare il valore di `x` passando da `y`?

Nel nostro caso potevamo farlo direttamente

Ma a cosa serve assegnare a `y` l'indirizzo di memoria di `x`, per poi modificare il valore di `x` passando da `y`?

Nel nostro caso potevamo farlo direttamente

Ma in altri casi no!

```c
#include <stdio.h>

void incrementa(int *p) {
    *p = *p + 1; // Incrementa il valore all'indirizzo di memoria puntato da p
}

int main() {
    int x = 5;
    int *y = NULL;
    printf("Valore di x prima: %d\n", x);

    y = &x;
    incrementa(y); // Passa l'indirizzo di x alla funzione
    incrementa(y); // Passa l'indirizzo di x alla funzione
    incrementa(y); // Passa l'indirizzo di x alla funzione

    printf("Valore di x dopo: %d\n", x);
}
```

```c
#include <stdio.h>

void incrementa(int *p) {
    *p = *p + 1; // Incrementa il valore all'indirizzo di memoria puntato da p
}

int main() {
    int x = 5;
    printf("Valore di x prima: %d\n", x);

    incrementa(&x); // Posso anche passare direttamente l'indirizzo di x!
    incrementa(&x); // Passa l'indirizzo di x alla funzione
    incrementa(&x); // Passa l'indirizzo di x alla funzione

    printf("Valore di x dopo: %d\n", x);
}
```

```c
#include <stdio.h>

void incrementa(int *p) {
    // Formulazione alternativa!
    p[0] = p[0] + 1; // Incrementa il valore all'indirizzo di memoria puntato da p
}

int main() {
    int x = 5;
    printf("Valore di x prima: %d\n", x);

    incrementa(&x); // Posso anche passare direttamente l'indirizzo di x senza passare da y!
    incrementa(&x); // Passa l'indirizzo di x alla funzione

    printf("Valore di x dopo: %d\n", x);
}
```

Ma perchè `*y` e' equivalente a `y[0]`?

Ma perchè `*y` e' equivalente a `y[0]`?

Questo sono due forme di **dereferenziazione**: accedere ad un valore attraverso il suo indirizzo di memoria.

Ma perchè `*y` è equivalente a `y[0]`?

Questo sono due forme di **dereferenziazione**: accedere ad un valore attraverso il suo indirizzo di memoria.

Funziona sia in lettura che in scrittura

Ma perchè `*y` è equivalente a `y[0]`?

Questo sono due forme di **dereferenziazione**: accedere ad un valore attraverso il suo indirizzo di memoria.

Funziona sia in lettura che in scrittura

```c
#include <stdio.h>

void incrementa(int *p) {
    printf("Before incr: %d\n", p[0]); // Lettura
    p[0] = p[0] + 1; // Scrittura
}

int main() {
    int x = 5;
    printf("Valore di x prima: %d\n", x);

    incrementa(&x); // Posso anche passare direttamente l'indirizzo di x senza passare da y!
    incrementa(&x); // Passa l'indirizzo di x alla funzione

    printf("Valore di x dopo: %d\n", x);
}
```

```c
#include <stdio.h>

int main() {
    int x = 5;
    printf("x e' registrata in %p e vale %d\n", &x, x);
}
```

```c
#include <stdio.h>

int main() {
    int x = 5;
    int *y = &x;
    printf("x e' registrata in %p e vale %d\n", y, x);
}
```

Complichiamoci la vita!

```c
#include <stdio.h>

int main() {
    int x = 5;
    int *y = &x;
    int **z = &y; // 😤😤😤😤😤
    printf("x e' registrata in %p e y e' registrata in %p\n", y, z);
}
```

```c
#include <stdio.h>

int main() {
    int x = 5;
    int *y = &x;
    int **z = &y; // 😨😨😨😨😨
    printf("x e' registrata in %p e y e' registrata in %p\n", y, z);
    // Tutti i puntatori sono grandi uguali
    printf("%d, %d, %d\n", (int)sizeof(x), (int)sizeof(y), (int)sizeof(z));
}
```

`**z` è un puntatore a un puntatore a un intero.

# Matematica dei puntatori

```c
#include <stdio.h>

int main() {
  char mystr[] = "Hello, world!";
  printf("Stringa: %s\n", mystr);
  printf("Indirizzo di mystr: %p\n", mystr);
}
```

Se volessi stampare solo il primo carattere di `mystr`?

```c
#include <stdio.h>

int main() {
  char mystr[] = "Hello, world!";
  printf("Stringa: %s\n", mystr);
  printf("Indirizzo di mystr: %p\n", mystr);
}
```

Se volessi stampare solo il primo carattere di `mystr`?

```c
#include <stdio.h>

int main() {
  char mystr[] = "Hello, world!";

  printf("%c\n", *mystr);

  // Oppure

  printf("%c\n", mystr[0]);
}
```

Se volessi stampare il primo *e il secondo* carattere di `mystr`?

```c
#include <stdio.h>

int main() {
  char mystr[] = "Hello, world!";

  printf("%c%c\n", *mystr, *(mystr + 1));

  // Oppure

  printf("%c%c\n", mystr[0], mystr[1]);
}
```

Ecco svelata perchè l'equivalenza tra dereferenziazione e accesso ad array!

```c
#include <stdio.h>

int main() {
  char mystr[] = "Hello!";

  char *p = mystr;

  printf("All'inizio p e' %p\n", p);

  while (*p) {
    putchar(*p); // Stampa il carattere puntato da p
    p++; // Spostati al carattere successivo
  }

  printf("\n");
  printf("Al termine p e' %p", p);
}
```

# Puntatori a Funzione

```c
#include <stdio.h>

int main() {
  printf("%p", main); // 😧😧😧😧
}
```

Il nome di una funzione in C è un puntatore all'inizio del codice della funzione stessa.

Un pò come un array è un puntatore al primo elemento.

```c
#include <stdio.h>

int main(void) {
  printf("%p", main); // 🤦🤦🤦🤦
}
```

Come creiamo un puntatore a funzione?

Come creiamo un puntatore a funzione?

1. Prendiamo il prototipo della funzione

   ```
   int main(void)
   ```

Come creiamo un puntatore a funzione?

1. Prendiamo il prototipo della funzione

   `int main(void)`

2. Mettiamo il nome della funzione tra parentesi

   `int (main)(void)`

Come creiamo un puntatore a funzione?

1. Prendiamo il prototipo della funzione
   `int main(void)`
2. Mettiamo il nome della funzione tra parentesi
   `int (main)(void)`
3. Prependiamo un asterisco `*` al *nome* della funzione
   `int (*main)(void)`

```c
#include <stdio.h>

int main(void) {
  int (*myf)(void);
  myf = main;

  printf("%p", myf);
  printf("%p", main);
}
```

```c
#include <stdio.h>

void hello(void) {
  printf("Hello!!\n");
}

void baubau(void) {
  printf("Baubau!!\n");
}

int main() {
  int (*x)(void);

  x = hello;
  x(); // Chiamo hello tramite il puntatore
  x = baubau;
  x(); // Chiamo baubau tramite il puntatore
}
```

E quindi? A che servono?

Esempio pratico: le **funzioni callback**

Le funzioni callback sono funzioni passate come argomenti ad altre funzioni, che poi le chiamano al momento opportuno.

```c
void esegui_operazione(int a, int b, int (*operazione)(int, int)) {
  int risultato = operazione(a, b);
  printf("Risultato: %d\n", risultato);
}

int somma(int x, int y) {
  return x + y;
}

int moltiplicazione(int x, int y) {
  return x * y;
}

int main() {
  esegui_operazione(5, 3, somma);           // Passa la funzione somma
  esegui_operazione(5, 3, moltiplicazione); // Passa la funzione moltiplicazione
}
```

Questa meccanica cosa ci permette di fare?

Questa meccanica cosa ci permette di fare?

- Permette di scrivere codice più generico e riutilizzabile
- Permette di implementare comportamenti personalizzati senza modificare il codice della funzione che esegue l'operazione

Vediamone un uso pratico in una libreria standard: la funzione `qsort` per ordinare array.

Vediamone un uso pratico in una libreria standard: la funzione `qsort` per ordinare array.

```c
#include <stdio.h>
#include <stdlib.h> // Qui c'e' qsort

int confronta_interi(const void *a, const void *b) {
    if (*(int *)a < *(int *)b) return -1;
    if (*(int *)a > *(int *)b) return 1;
    return 0;
}

int main() {
    int array[] = {5, 2, 9, 1, 5, 6};
    size_t n = sizeof(array) / sizeof(array[0]);

    // La firma di qsort è qsort(void *base, size_t nmemb, size_t size, int (*compar)(const void *, const void *));
    // qsort internamente userà la funzione confronta_interi per confrontare gli elementi dell'array
    // Come lo fa?
      // Se la funzione restituisce un valore negativo, il primo elemento è "minore" del secondo
      // Se restituisce un valore positivo, il primo elemento è "maggiore" del secondo
      // Se restituisce zero, sono uguali
    qsort(array, n, sizeof(int), confronta_interi);

    for (size_t i = 0; i < n; i++) {
        printf("%d ", array[i]);
    }
    printf("\n");
}
```

# References

https://www.youtube.com/watch?v=BBgZs-jd_QY&list=PLrEMgOSrS_3cFJpM2gdw8EGFyRBZOyAKY&index=11

https://www.youtube.com/watch?v=Ic7hL9Wt-ho&list=PLrEMgOSrS_3cFJpM2gdw8EGFyRBZOyAKY&index=11

https://www.youtube.com/watch?v=OIseV5lcx8w&list=PLrEMgOSrS_3cFJpM2gdw8EGFyRBZOyAKY&index=26