# Uso avanzato dei puntatori

Matteo Spanio Giulio Pitteri

3 aprile 2025

In questa lezione si presenta il modo in C di definire strutture dati complesse e di manipolarle attraverso i puntatori. Nel dettaglio vengono presentate strutture, unioni e enumerazioni. Si introduce l'utilizzo di queste strutture negli array e altre strutture dati complesse. Si introduce inoltre in maniera non formale l'algoritmo di ordinamento bubble sort.

## Strutture, Unioni ed Enumerazioni

Il C non permette di definire Classi e oggetti, si possono comunque definire dei tipi aggiuntivi:

- Strutture: insieme di dati
- Unioni: alternative tra tipi diversi
- Enumerazioni: insieme di costanti

### `struct`

Una struttura è un insieme di dati, si differenzia dagli array perché:

- gli elementi possono essere di tipo diverso;
- gli elementi sono identificati da un nome;

Di seguito un esempio di dichiarazione di una struttura:

```c
struct studente {
    char nome[20];
    char cognome[20];
    int matricola;
};
```

Come si può vedere dall'esempio, la dichiarazione di una struttura è introdotta dalla parola chiave `struct`, seguita dal nome della struttura e da una lista di dichiarazioni di variabili. Ricorda fortemente la dichiarazione di una classe in Java, in questo caso però non ci sono metodi, e i dati non si chiamano campi o attributi, ma membri. I membri hanno un nome e un tipo, possono essere di qualsiasi tipo, anche un'altra struttura. La grossa differenza con Java è che i membri sono sempre pubblici e non c'è modo di dichiararli privati.

### Inizializzazione

Si possono inizializzare le strutture quando vengono dichiarate:

```
struct studente s = {"Mario", "Rossi", 12345};
```

Si possono inizializzare meno elementi rispetto a quelli dichiarati, in questo caso gli elementi non inizializzati vengono impostati a 0 (nel caso delle stringhe l'inizializzazione a zero equivale a dire stringa vuota `""`).

> **Avviso**
>
> Il tipo di una variabile `struct` è `struct nome_struct`, omettere la parola struct è un errore.

Un modo alternativo di inizializzare una struttura è quello di specificare il nome dei membri:

```
struct studente s = {
    .nome = "Mario",
    .cognome = "Rossi",
    .matricola = 12345,
};
```

In questa modalità l'ordine degli elementi non è importante, ma è necessario specificare il nome di tutti i membri. Questo metodo è molto utile quando si hanno strutture con molti membri e si vuole inizializzare solo alcuni di essi.

### Accedere agli elementi

Per accedere ai singoli membri di una struttura si usa l'operatore `.` (punto) dopo il nome della variabile:

```
struct studente s = {"Mario", "Rossi", 12345};

printf("Nome: %s\n", s.nome);
printf("Cognome: %s\n", s.cognome);
s.matricola = 54321;
```

### Copie di strutture

Si può copiare interamente una struttura usando una variabile dello stesso tipo:

```
struct studente a, b = {"Matteo", "Spanio", 56789};
a = b;
```

In questo caso si può notare una delle principali differenze con gli array:

```
int a[] = {1, 2, 3, 4, 5};  ①
int *b;                     ②
b = a;                      ③
```

① `a` è un array di interi;
② `b` è un puntatore a intero;
③ `b = a` non è una copia dell'array `a` ma assegna a `b` l'indirizzo di `a`, pertanto qualsiasi modifica dei dati di `a` si riflette anche in `b` (e viceversa). Cioè `a` e `b` puntano alla stessa area di memoria.

Utilizzando le strutture invece, la copia è effettiva:

```
struct array {
    int dati[5];
};
struct array a = {{1, 2, 3, 4, 5}};  ①
struct array b;                      ②
b = a;                               ③
```

① `a` inizializzo `a` con un array di interi;
② `b` dichiaro `b` come struttura;
③ `b = a` copio `a` in `b`, vengono effettivamente copiati i dati dell'array `a.dati` in `b.dati`. Cioè `a` e `b` si trovano in aree di memoria diverse.

> **!** Importante
>
> Solo l'operatore `=` è valido tra 2 struct, gli operatori `==` e `!=` NON si possono usare per vedere se due strutture sono uguali.

### Dare i nomi alle strutture

Le stutture possono avere un nome, gli si può associare un tipo, oppure possono essere anonime.

Le strutture col nome, detto *structure tag*, si dichiarano come abbiamo già visto:

```
struct nome {
    type member_name;
};
```

### Strutture anonime

Nel caso in cui un si voglia usare una struct solo in un punto specifico del codice non è necessario associarvi un nome, si può dichiarare e associare direttamente:

```
struct { int x; int y; } punto;
punto.x = 12;
punto.y = 18;
```

Così facendo la struttura non ha un nome e non può essere riutilizzata facilmente, ogni volta che si vuole dichiarare una variabile di quel tipo bisogna riscrivere la struttura.

### Strutture con typedef

`typedef` è un operatore che permette di definire alias per i tipi:

```
typedef int Bool;
typedef float Euro;
typedef char* String;
```

Queste dichiarazioni permettono scrivere codice più chiaro.

Lo stesso effetto si può ottenere dichiarando delle macro:

```
#define BOOL int
#define EURO float
#define STRING char*
```

In generale si preferisce usare `typedef` perché è più chiaro e il compilatore può fare dei controlli sui tipi.

Dal momento che le strutture sono usate moltissimo e i programmatori sono pigri, solitamente, si preferisce omettere la parola chiave `struct` per riferirsi al tipo delle strutture grazie a typedef:

```c
typedef struct {
    int x;
    int y;
} Point;
Point punto = {
    .x = 1,
    .y = 2,
};
```

In questo modo si può dichiarare una variabile di tipo `Point` senza usare la parola chiave `struct`.

### Cast a strutture

Una volta definito il tipo con typedef, si può popolare una struttura con la stessa sintassi dell'inizializzazione, occorre fare un cast esplicito però:

```c
typedef struct {
    int x;
    int y;
} Point;

Point s;
s = (Point) { .x = 1, .y = 2 };
```

A prescindere da quale metodo di definizione si scelga, le strutture possono essere argomenti di funzioni e restituite da funzioni:

```c
#include <math.h>
typedef struct point {
    int x;
    int y;
} Point;

Point somma(Point a, struct point b) {
    return (Point) { .x = a.x + b.x, .y = a.y + b.y };
}

float distanza(Point a, Point b) {
    return sqrt(pow(a.x - b.x, 2) + pow(a.y - b.y, 2));
}
```

### `struct` e puntatori

Si possono creare puntatori a strutture (come per qualsiasi altro tipo di dato):

```
struct point *p;
Point *p1;

void foo(Point* p);
```

## Array di struct

Si possono creare array di strutture:

```
Point punti[10];
punti[0].x = 1;
```

Si può inserire un array in una struttura:

```
struct libretto {
    int voti[10];
    int num_voti;
};
```

Si può inserire una struttura in un'altra struttura:

```
struct data {
    int giorno;
    int mese;
    int anno;
};

struct persona {
    char nome[20];
    char cognome[20];
    struct data data_nascita;
};
```

si possono anche innestare struct anonime:

```
struct {
    char nome[20];
    char cognome[20];
    struct {
```

```
        int giorno;
        int mese;
        int anno;
    } data_nascita;
} persona;
```

Si possono inserire unioni[1] in una struttura:

```
struct {
    char nome[20];
    char cognome[20];
    union {
        int matricola;
        char codice_fiscale[16];
    } id;
} persona;
```

Si può inserire strutture in unioni:

```
union {
    struct {
        int giorno;
        int mese;
        int anno;
    } data;
    int intero;
} u;
```

### Esempio: Mondiali 1982

Vediamo ora un esempio di programma basato sulle strutture.

Si vuole creare un sistema informatico per la gestione squadra della nazionale Italiana di calcio che giocò ai mondiali del 1982:

1. implementare una struttura dati per memorizzare i dati di un giocatore di calcio;
2. memorizzare l'elenco dei giocatori della squadra;
3. ordinare i giocatori in base numero di maglia.

Un giocatore è identificabile da nome, cognome e numero di maglia (chiaramente si potrebbero aggiungere molti altri dati, come la data di nascita, il ruolo, ecc.). Si può quindi definire una struttura `Giocatore` che contenga questi dati:

[1] Si veda la sezione seguente per informazioni dettagliate sulle unioni.

```c
#define BUFFER_SIZE 128 // dimensione massima di un buffer
typedef struct giocatore
{
    char nome[BUFFER_SIZE];
    char cognome[BUFFER_SIZE];
    unsigned int numero_maglia; // non esistono numeri di maglia negativi
} Giocatore;
```

Il codice sopra riportato definisce la struttura `struct giocatore` e vi associa il nuovo tipo `Giocatore`. La struttura contiene tre membri: `nome`, `cognome` e `numero_maglia`, in quanto stringhe, il nome e cognome sono di tipo `char[]`, mentre il numero di maglia è un intero senza segno. Si noti che gli array vengono dichiarati con una dimensione massima `BUFFER_SIZE`, ciò vuol dire che il nome e il cognome possono avere una lunghezza massima di `BUFFER_SIZE - 1` caratteri, purtroppo C non controlla che questa condizione sia rispettata. In altre parole, all'interno del programma posso inizializzare una struttura `Giocatore` con un nome o cognome più lungo di `BUFFER_SIZE - 1` caratteri, ma il comportamento del programma in questo caso è indefinito. Questo è uno dei più grandi problemi di C.

### Memorizzare la lista dei giocatori

La struct `Giocatore` è di fatto un tipo di dato, possiamo quindi creare un array di giocatori dove memorizzare la formazione:

```c
#define NUM 22
Giocatore squadra[NUM];
```

La squadra è composta da 22 giocatori (11 titolari, 5 in panchina e 6 riserve). Questa dichiarazione crea un array di 22 elementi di tipo `Giocatore` e lo assegna alla variabile `squadra`. Quando si dichiara un array di strutture, si può accedere ai membri di ciascuna struttura usando la notazione `.` (punto):

```c
squadra[0].numero_maglia = 1;
```

Questo perchè `squadra[0]` è una struttura di tipo `Giocatore` e `numero_maglia` è un membro di questa struttura, quando invece si ha un puntatore a una struttura, si usa l'operatore `->` (freccia):

```c
Giocatore *p = &squadra[0];
p->numero_maglia = 1;
```

L'operatore `->` è una comodità sintattica, infatti `p->numero_maglia` è equivalente a `(*p).numero_maglia`.

### Ordinare i giocatori

Per ordinare i giocatori in base al numero di maglia, bisogna introdurre un algoritmo di ordinamento:

Il più intuitivo (ma non il più efficiente) è il Bubble Sort, di seguito una descrizione dell'algoritmo in pseudocodice:

```
Repeat n-1 times
    For i from 0 to n-2
        If numbers[i] and numbers[i+1] out of order
            Swap them
    If no swaps
        Quit
```

L'idea è quella di scorrere l'array di numeri e scambiare i numeri adiacenti se non sono in ordine. Questo processo viene ripetuto finché non si è completato un passaggio senza scambi. Questo algoritmo è molto inefficiente, ma è molto semplice da implementare. La complessità computazionale del Bubble Sort è $O(n^2)$.

L'implementazione dello pseudocodice è molto semplice, infatti si tratta di due cicli annidati e uno scambio di due elementi:

1. Si scorre l'array `squadra` `NUM - 1` volte;
2. Si scorre l'array `squadra` da `0` a `NUM - 2`;
3. Si controlla se `squadra[i]` e `squadra[i + 1]` sono fuori ordine, in tal caso **si scambiano**. La funzione swap l'abbiamo già vista nella lezione sui puntatori.

```c
void bubble_sort(Giocatore *squadra, size_t n)
{
    for (size_t i = 0; i < n - 1; i++)
    {
        for (size_t j = 0; j < n - 2; j++)
        {
            if (squadra[j].numero_maglia > squadra[j + 1].numero_maglia)
            {
                swap(&squadra[j], &squadra[j + 1], sizeof(Giocatore));
            }
        }
    }
}
```

Per ottimizzare il codice, si può usare un flag per controllare se durante un passaggio non si è effettuato alcuno scambio, in tal caso si può uscire dal ciclo:

```c
void bubble_sort(Giocatore *squadra, size_t n)
{
    for (size_t i = 0; i < n - 1; i++)
    {
        int scambiato = 0;
        for (size_t j = 0; j < n - 2; j++)
        {
            if (squadra[j].numero_maglia > squadra[j + 1].numero_maglia)
            {
                swap(&squadra[j], &squadra[j + 1], sizeof(Giocatore));
                scambiato = 1;
            }
        }
        if (!scambiato)
        {
            break;
        }
    }
}
```

Se stai consultando questa guida in formato HTML, puoi scaricare una versione completa del codice sorgente cliccando sul pulsante qui sotto, altrimenti puoi scaricarlo collegandoti a questo link.

### Unioni

Le unioni sono simili alle strutture, nel senso che permettono di usare membri di tipo diverso, la differenza sta nel fatto che una struttura tiene tutti i membri in memoria, mentre una union tiene solo un membro alla volta:

```c
union {
    int i;
    double d;
} u;

struct {
    int i;
    double d;
} s;
```

In questo caso `u` occupa in memoria lo spazio necessario per la variabile più grande, in questo caso `double` (8 byte), mentre `s` occupa lo spazio necessario per entrambi i membri (4 + 8 byte).

Si può inizializzare solo un elemento di una union:

```
union {
    int i;
    double d;
} u = { .i = 3 };
```

Si presti molta attenzione a non leggere mai un membro di una union che non è stato inizializzato, il comportamento del programma in questo caso è indefinito.

Come per le strutture, posso dare nomi alle unioni tramite tag oppure con typedef.

### Enumerazioni

In molti programmi si usano interi come "*codici*" per indicare varie cose. Un caso comune sono per esempio i codici di errore.

Spesso si usano macro o costanti, ma è più chiaro usare enumerazioni:

```
enum {
    CUORI,
    QUADRI,
    FIORI,
    PICCHE
} s1, s2;`
```

Ora le variabili `s1` e `s2` possono assumere solo uno dei quattro valori definiti.

Un esempio molto comune è la definizione del tipo booleano:

```
typedef enum {
    FALSE,
    TRUE
} Bool;
```

In questo modo si può usare `Bool` come tipo di variabile, e assegnare solo i valori `FALSE` e `TRUE`. Solitamente gli elementi di un'enumerazione sono scritti in maiuscolo, per distinguerli dalle variabili. In questo caso `FALSE` e `TRUE` sono costanti, FALSE ha valore 0 e TRUE ha valore 1.

Di default C assegna dei valori interi crescenti agli elementi dell'enumerazione, ma si possono assegnare valori specifici:

```c
enum {
    LUN = 1,
    MAR = 2,
    MER = 3,
    GIO = 4,
    VEN = 5,
    SAB = 6,
    DOM = 7
} giorno;
```

Se non si assegna un valore, C assegna il valore dell'elemento precedente più uno.

## Strutture dati

Le strutture sono molto utili per definire strutture dati. Per esempio si possono definire array dinamici, liste concatenate, alberi, pile, code…

### Array con strutture

Gli array in C sono molto limitati, infatti dobbiamo gestire la loro lunghezza *a mano*, per questo è una pratica comune creare una struttura che contenga un array e la sua lunghezza:

```c
#define MAX 1024

typedef struct intarray {
    int array[MAX];
    int length;
} IntArray;
```

Questa struttura dati si comporta come segue: `array` è un array con una dimensione massima di `MAX` elementi, `length` è il numero di elementi effettivamente presenti nell'array, ciò vuol dire che `length` è sempre minore o uguale a `MAX`. Di seguito un esempio di utilizzo:

```c
IntArray a = {.length=0};                          ①
a.array[a.length++] = 1;                           ②
printf("%d, %d\n", a.array[0], a.length);          ③
```

① Inizializzo la struttura `a` con `length=0`, non c'è bisogno di inizializzare l'array perchè si presuppone che prima di accedere all'array si verifichi sempre fino a che indice è valido l'array tramite `length`;

② Inizializzo il primo elemento dell'array con il valore 1 e incremento `length`, si noti che l'espressione `a.length++` restituisce il valore di `a.length` prima dell'incremento;

③ Stampo il primo elemento dell'array e la lunghezza dell'array.

Adesso si può passare la struttura come argomento di una funzione:

```
void stampa_array(IntArray a) {                                    ①
    for (int i = 0; i < a.length; i++) {                           ②
        printf("%d ", a.array[i]);
    }
    printf("\n");
}
```

① Si nota subito che non abbiamo più bisogno di passare la lunghezza dell'array come argomento.

② Si può usare `a.length` per scorrere l'array, pertanto se la dimensione è 0 il ciclo non viene eseguito, ecco perchè non c'è bisogno di inizializzare i valori dell'array se `length` è 0.

Se stai consultando questa guida in formato HTML, puoi scaricare un esempio di codice sorgente in cui si utilizza questa struttura dati cliccando sul pulsante qui sotto, altrimenti puoi scaricarlo collegandoti a questo link.