



Capitolo 6

Vettori e Strutture

pag. 161-191

Presenta: Prof. Misael Mongiovì



Vettori e strutture

- un *array* (o *vettore*) è una sequenza di oggetti dello stesso tipo
- gli oggetti si chiamano *elementi* dell'array e si numerano consecutivamente 0, 1, 2, 3.. ; questi numeri si dicono *indici* dell'array, ed il loro ruolo è quello di localizzare la posizione di ogni elemento dentro l'array, fornendo *accesso diretto* ad esso
- il tipo di elementi immagazzinati nell'array può essere qualsiasi tipo di dato predefinito del C++, ma anche tipi di dato definiti dall'utente
- se il nome del vettore è a , allora $a[0]$ è il nome del primo elemento, $a[1]$ è il nome del secondo elemento, ecc; l'elemento i -esimo si trova quindi nella posizione $i-1$, e se l'array ha n elementi, i loro nomi sono $a[0]$, $a[1]$, \dots , $a[n-1]$

| | | | | | | |
|---|------|------|------|------|------|------|
| a | 25.1 | 34.2 | 5.25 | 7.45 | 6.09 | 7.54 |
| | 0 | 1 | 2 | 3 | 4 | 5 |

← indice

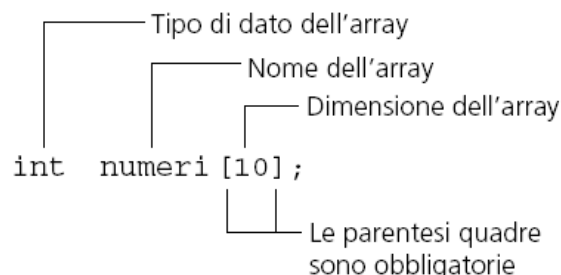


definizione di vettore

• *tipo_elementi* *nome_array*[*numero_elementi*];
numero_elementi può essere:

- un valore intero
- un'espressione costante
- una variabile istanziata prima della definizione dell'array

```
int numeri[10]; //Crea un array di 10 elementi int
```



```
int br[4] = {1, 5, 2, 4}; // crea array e lo inizializza  
char cr[4] = {'a', '7', 'B', '!'};
```



accesso agli elementi di un vettore

- si può accedere ad un elemento dell'array mediante il suo nome ed un *indice* che ne rappresenta la posizione
`nome_array[n]` ;
- C++ **non** verifica che gli indici dell'array stiano dentro la dimensione definita; ad esempio, se si accede a `numeri[12]` il compilatore non segnala alcun errore (*buffer overflow*)

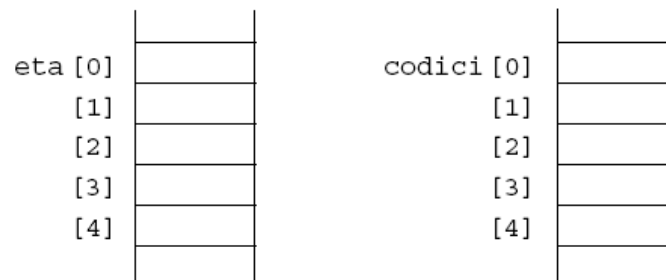
```
int eta[5]; // contiene 5 elementi: da eta[0] a eta[4]
int vendite[100];
float salario[25]; // un array di 25 elementi float
eta[4] = 35; // mette 35 nel quinto elemento del vettore eta
vendite[totale + 5] = 35; // mette 35 nell'elemento di indice
                        // pari al valore della variabile o
                        // costante totale aumentato di 5
salario[mese[i] * 5] // accede all'elemento del vettore salario
                        // il cui indice è dato dal valore dello
                        // i-esimo elemento del vettore mese
                        // moltiplicato per cinque
```

allocazione in memoria

- gli elementi degli arrays si immagazzinano in blocchi contigui

```
int eta[5];
```

```
char codici[5];
```



- si può utilizzare l'operatore `sizeof` per conoscere il numero di bytes occupati dall'array; ad esempio, supponiamo che si definisca un array di 100 numeri interi denominato `eta`:

```
n = sizeof(eta);
```


assegna 400 ad `n`
- se si vuole conoscere la dimensione di un elemento individuale dell'array si può scrivere:

```
n = sizeof(eta[0]);
```

inizializzazione di un vettore

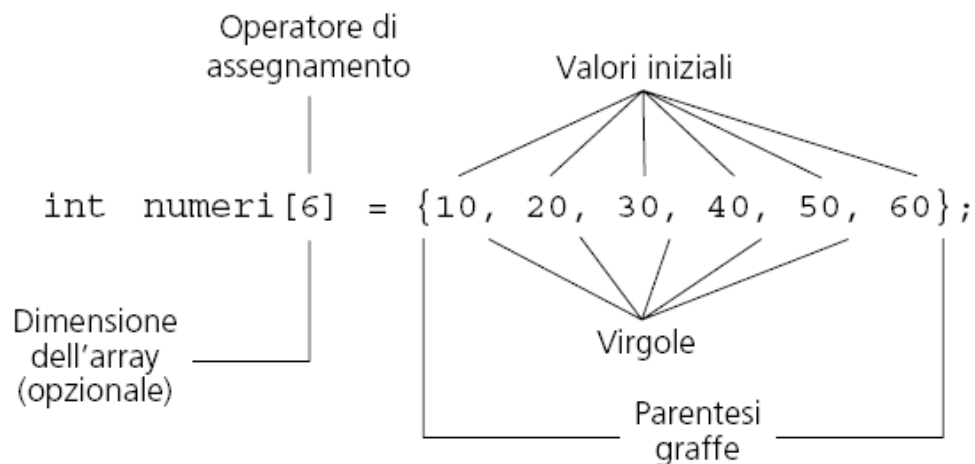
- per assegnare un valore ad un elemento di un array si può usare ovviamente l'operatore di assegnamento:

```
prezzi[0] = 10;
```

ma assegnare un intero array si può fare solo nella sua definizione:

```
int numeri[6] = {10, 20, 30, 40, 50, 60};
```

```
int numeri[] = {10, 20, 30, 40, 50, 60};
```



limiti dei vettori in C++

- no operazioni di confronto!
- no assegnamento sull'intero array!
- no operazioni aritmetiche!
- no restituzione da funzioni!



vettori di caratteri e stringhe

- le stringhe di testo sono arrays di caratteri terminate con il carattere nullo `\0` (b); senza di esso la stringa non è tale ma è un semplice array di caratteri (a)

```
char Stringa1[9] = "Mortimer"  
char Stringa2[] = {'M','o','r','t','i','m','e','r'}  
char Stringa2[] = {'M','o','r','t','i','m','e','r','\0'}
```

| | |
|-------------|---|
| Stringa1[0] | M |
| [1] | o |
| [2] | r |
| [3] | t |
| [4] | i |
| [5] | m |
| [6] | e |
| [7] | r |
| [8] | |

(a)

| | |
|-------------|----|
| Stringa2[0] | M |
| [1] | o |
| [2] | r |
| [3] | t |
| [4] | i |
| [5] | m |
| [6] | e |
| [7] | r |
| [8] | \0 |

(b)

carattere nullo



stringhe ed I/O

- le stringhe vengono gestite intelligentemente dall'*estrattore* e dall'*inseritore*, ma !!!!

```
#include <iostream>
using namespace std;
char stringa[8];
int main() {
    cout << "Scrivi qualcosa \n";
    cin >> stringa; // estrae l'estraibile e mette lo /0
    cout << "hai scritto \n";
    cout << stringa; // inserisce l'inseribile fino allo /0
    return 0;
}
```

buffer overflow

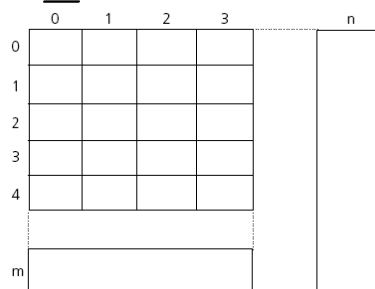
Scrivi qualcosa
Aldo
hai scritto
Aldo

Scrivi qualcosa
AldoFranco
hai scritto
AldoFranco

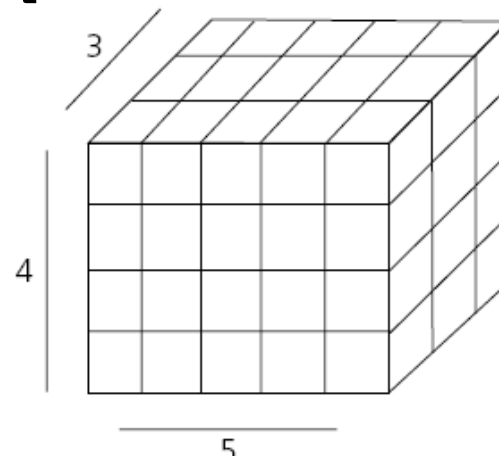
Scrivi qualcosa
Aldo Franco
hai scritto
Aldo

vettori multidimensionali

- gli arrays di arrays si dicono *bidimensionali*; hanno due indici e sono noti anche con il nome di *tabelle* o *matrici*
`tipo_elemento nome_array [NumRighe] [NumColonne]`



- gli arrays di arrays di arrays sono tridimensionali ...
`tipo_elemento nome_array [n] [m] [r]`



- e così via



vettori multidimensionali

- gli arrays multidimensionali si inizializzano normalmente

```
int tabella[2][3] = {{51, 52, 53}, {54, 55, 56}};
```

oppure:

```
int tabella[2][3] = {51, 52, 53, 54, 55, 56};
```

- anche gli assegnamenti sono intuitivi:

```
int x = tabella[1][0] // assegna ad x 54
```

```
tabella[1][2] = 58;    // sostituisce 56 a 58
```

vettori come argomenti di funzione

- gli arrays *si passano per riferimento*: quando s'invoca una funzione e le si passa un array come parametro, C++ tratta la chiamata come se vi fosse l'operatore di indirizzo & davanti al nome del vettore

```
main()
{
    char parola[4] = "ABC"
    cambiare(parola);
    cout << parola << endl;
    return;
}
```

parola

```
cambiare(char c[4])
{
    cout << c << endl;
    strcpy(c, "AMA");
    return;
}
```

- il 4 nel `char[4]` della figura precedente può essere omesso, ma se si passano array multidimensionali gli altri indici devono essere specificati; ad esempio, se il vettore `c[]` fosse stato una matrice l'intestazione della funzione sarebbe stata:
`cambiare(char c[][4])`



Capitolo 6. Vettori e strutture



```
emacs: *shell*
File Edit Mule Apps Options Buffers Tools Complete In/Out Signa
Open Dired Save Print Cut Copy Paste Undo Spell Replace Mail Info
#include <iostream.h>
int somma(int formale[], int n)
{
    int parziale = 0;
    for (int i = 0; i < n; ++i)
        parziale += formale[i];
    return parziale;
}
int main()
{
    int attuale1[5]={0,2,4,6,8};
    int attuale2[3]={1,3,5};
    cout << somma(attuale1, 4) << '\n';
    cout << somma(attuale2, 4) << '\n';
    cout << attuale2[3] << '\n';
    return 0;
}
ISO8-----XEmacs: vettoril.cpp (C+
[root@localhost FondInfo]# ./a.out
12
1074083143
1074083134
[root@localhost FondInfo]#
ISO8--*-XEmacs: *shell* (Shell:r
```

- funzioni che hanno argomenti formali di tipo array possono modificare gli array passati come argomenti attuali!

vettori multidimensionali come argomenti di funzione



?

```
emacs: vettori2.cpp
File Edit Mule Apps Options Buffers Tools C++ Help
Open Dired Save Print Cut Copy Paste Undo Spell Replace Mail Info Compile Debug News

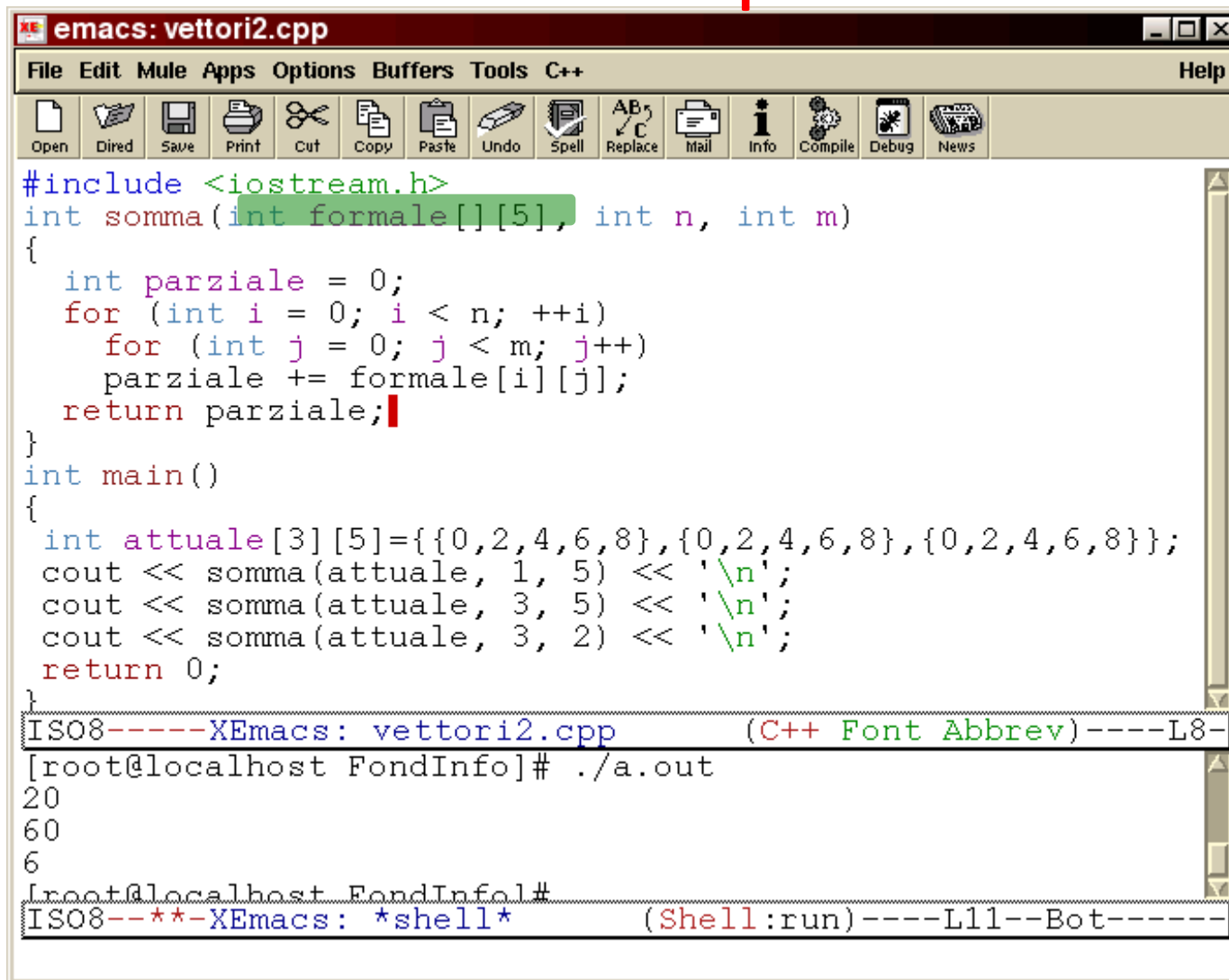
#include <iostream.h>
int somma(int formale[][], int n, int m)
{
    int parziale = 0;
    for (int i = 0; i < n; ++i)
        for (int j = 0; j < m; ++j)
            parziale += formale[i][j];
    return parziale;
}
int main()
{
    int attuale[3][5]={{0,2,4,6,8},{0,2,4,6,8},{0,2,4,6,8}};
    cout << somma(attuale, 3, 5) << '\n';
    return 0;
}

ISO8-----XEmacs: vettori2.cpp (C++ Font Abbrev)-----L12--All-----
cd /root/FondInfo/
g++ vettori2.cpp
vettori2.cpp:3: declaration of `formale' as multidimensional array must
have bounds for all dimensions except the first
vettori2.cpp: In function `int somma (int, int)':
vettori2.cpp:7: `formale' undeclared (first use this function)
vettori2.cpp:7: (Each undeclared identifier is reported only once for
each function it appears in.)
vettori2.cpp: In function `int main ()':
vettori2.cpp:13: cannot convert `int (*)[5]' to `int' for argument `1'
to `somma (int, int)'

Compilation exited abnormally with code 1 at Sun Nov 24 17:26:04

ISO8--*-XEmacs: *compilation* (Compilation Font:exit [exit-status 1]
```

vettori e puntatori



The screenshot shows an Emacs editor window titled "emacs: vettori2.cpp". The menu bar includes File, Edit, Mule, Apps, Options, Buffers, Tools, C++, and Help. The toolbar contains icons for Open, Diretd, Save, Print, Cut, Copy, Paste, Undo, Spell, Replace, Mail, Info, Compile, Debug, and News. The code in the editor is as follows:

```
#include <iostream.h>
int somma(int formale[][5], int n, int m)
{
    int parziale = 0;
    for (int i = 0; i < n; ++i)
        for (int j = 0; j < m; j++)
            parziale += formale[i][j];
    return parziale;
}

int main()
{
    int attuale[3][5]={{0,2,4,6,8},{0,2,4,6,8},{0,2,4,6,8}};
    cout << somma(attuale, 1, 5) << '\n';
    cout << somma(attuale, 3, 5) << '\n';
    cout << somma(attuale, 3, 2) << '\n';
    return 0;
}
```

Below the code, the output of the program is shown in a terminal window:

```
ISO8-----XEmacs: vettori2.cpp (C++ Font Abbrev)-----L8-
[root@localhost FondInfo]# ./a.out
20
60
6
[root@localhost FondInfo]#
ISO8--*-XEmacs: *shell* (Shell:run)-----L11--Bot-----
```

perché
adesso
funziona?





vettori e puntatori

- gli arrays sono implementati mediante puntatori
il nome di un vettore è un puntatore al suo primo elemento
- `int gradi[5] = {10, 20, 30, 40, 50};`
- `gradi[0] == *gradi; // 10`
- `gradi[1] == *(gradi + 1); // 20`
- `gradi[2] == *(gradi + 2); // 30`
- `gradi[3] == *(gradi + 3); // 40`
- `gradi[4] == *(gradi + 4); // 50`
- il nome di un array è però una *costante* puntatore, non una variabile puntatore; non si può cambiarne il valore



aritmetica dei puntatori

- se un'espressione p dà come valore l'indirizzo di un oggetto di tipo T , allora l'espressione $p+1$ dà come valore l'indirizzo di un oggetto di tipo T che si trova consecutivamente in memoria
- più in generale, data una variabile di tipo *puntatore al tipo T* , se le si aggiunge un certo numero intero n , il suo valore cambia in realtà di n moltiplicato per la dimensione del tipo T

```
int gradi[5] = {10, 20, 30, 40, 50};  
p = gradi; // p punta a 10  
p++; // adesso p punta a 20; infatti, poiché  
      // ogni elemento di gradi occupa 4 bytes,  
      // il valore di p è stato incrementato di 4  
      // (e non di 1)  
p--; // adesso p punta di nuovo 10  
*(p + 3) // restituisce 40
```

aritmetica dei puntatori

- non si possono *sommare* due puntatori
- non si possono *sottrarre* due puntatori
- non si possono *moltiplicare* due puntatori
- non si possono *dividere* due puntatori



aritmetica dei puntatori: esempi



```
emacs: vettori3.cpp
File Edit Mule Apps Options Buffers Tools C++ Help
[Icons]
#include <iostream.h>
int main()
{int vetInt[2] = {1,2}; char vetCar[3] = "ab";
cout << vetInt << " : vetInt \n";
cout << &vetInt[0] << " : &vetInt[0] \n";
cout << vetInt[0] << " : vetInt[0] \n";
cout << &vetInt[1] << " : &vetInt[1] \n";
cout << vetInt[1] << " : vetInt[1] \n";
cout << static_cast<void*>(vetCar) << " : vetCar \n";
cout << static_cast<void*>(&vetCar[0]) << " : &vetCar[0] \n";
cout << vetCar[0] << " : vetCar[0] \n";
cout << static_cast<void*>(&vetCar[1]) << " : &vetCar[1] \n";
cout << vetCar[1] << " : vetCar[1] \n";
cout << *vetInt << " : *vetInt \n";
cout << vetInt[0] << " : vetInt[0] \n";
cout << *(vetInt + 1) << " : *(vetInt + 1) \n";
cout << vetInt[1] << " : vetInt[1] \n";
int* pi = vetInt; char* pc = vetCar;
cout << pi << " : pi \n"; cout << &pi << " : &pi \n";
cout << pc << " : pc \n"; cout << &pc << " : &pc \n";
cout << static_cast<void*>(pc) << " : pc convertito \n";
cout << *pi << " : *pi \n"; cout << *(pi + 1) << " : *(pi + 1) \n";
cout << *pc << " : *pc \n"; cout << *(pc + 1) << " : *(pc + 1) \n";
cout << static_cast<int>(*(pc + 2)) << " : static_cast<int>(*(pc + 2)) \n";
return 0;
}
```

```
emacs: *shell*
File Edit Mule Apps Options Buffers Tools Complete In/Out Si
[Icons]
[root@localhost C++]# ./a.out
0xbffffaf8 : vetInt
0xbffffaf8 : &vetInt[0]
1 : vetInt[0]
0xbffffafc : &vetInt[1]
2 : vetInt[1]
0xbffffae0 : vetCar
0xbffffae0 : &vetCar[0]
a : vetCar[0]
0xbffffaef : &vetCar[1]
b : vetCar[1]
1 : *vetInt
1 : vetInt[0]
2 : *(vetInt + 1)
2 : vetInt[1]
0xbffffaf8 : pi
0xbffffadc : &pi
ab : pc
0xbffffad8 : &pc
0xbffffaef : pc convertito
1 : *pi
2 : *(pi + 1)
a : *pc
b : *(pc + 1)
0 : static_cast<int>(*(pc + 2))
[root@localhost C++]#
```



strutture (struct)

- una **struttura** (o *record*) è una collezione di elementi denominati *campi*, ognuno dei quali può contenere un dato di tipo diverso ad esempio, i campi della struttura CD potrebbero essere:

- titolo
- artista
- numero canzoni
- prezzo
- data di acquisto

- dopodiché bisogna decidere di che tipo sarà ciascun campo:

| • nome campo | • tipo di dato |
|--------------------|---------------------------------------|
| • titolo | • array di caratteri di dimensione 30 |
| • artista | • array di caratteri di dimensione 25 |
| • numero canzoni | • intero |
| • prezzo | • virgola mobile |
| • data di acquisto | • array di caratteri di dimensione 8 |

dichiarazione di un tipo struct

```
struct <nome della struttura>
{
    <tipo_dato_campo1> <nome_campo_1>;
    <tipo_dato_campo2> <nome_campo_2>;
    ...
    <tipo_dato_campon> <nome_campo_n>;
};
```

ad esempio, la dichiarazione della struttura CD precedente sarebbe:

```
struct CD
{
    char titolo[30];
    char artista[25];
    int num_canzoni;
    float prezzo;
    char data_acquisto[8];
}
```



definizione variabili di tipo struct

- si possono definire di due modi
 - elencandole dopo la parentesi graffa di chiusura della dichiarazione della struttura

```
struct CD
{
    char titolo[30];
    char artista[25];
    int num_canzoni;
    float prezzo;
    char data_acquisto[8];
} cd1, cd2, cd3;
```

- scrivendo il nome della struttura seguita dalle variabili di quel tipo
CD cd1, cd2, cd3;





inizializzazione variabili di tipo struct

- si possono inizializzare in qualunque punto del programma, anche nella definizione del tipo struct; per esempio:

```
struct data
{
    int mese;
    int giorno;
    int anno;
} data_di_nascita = {1,6,1982};
struct libro cd1 = {
    "Hebron Gate",
    "Groundation",
    9,
    11.99,
    "08/10/07"
};
```

- a differenza dei vettori, le variabili di tipo struct si possono assegnare

```
struct libro cd2;
cd2 = cd1;
```

accesso ai campi del tipo struct

- per accedere in lettura o in scrittura ad un singolo campo della struttura si utilizza l'operatore punto (.)

- `<variabile_struct>.<nome_campo> = valore;`

- esempi:

```
cd1.prezzo = 30.75;  
cd1.num_canzoni = 7;  
strcpy (cd1.titolo, "Io non so parlar d'amore");  
cout << "Qual'è la potenza del motore? ";  
cin >> miaMacchina.PotenzaCV;  
cout << "La potenza è di "  
    << miaMacchina.PotenzaCV  
    << " CV.";
```





strutture annidate

- un campo di una struttura può a sua volta essere di tipo struttura

```
• struct info
• {
•     char nome[30];
•     char indirizzo[25];
•     char citta[20];
•     char provincia[20];
•     long int cod_postale;
• };
• può diventare un campo di un'altra
• struct impiegato
• {
•     struct info anagrafica;
•     double salario;
• } capoufficio;
```

i campi della sottostruttura si raggiungono mediante doppio uso dell'operatore punto

```
• cout << "Introduca il nome del capoufficio: ";
• cin >> x.anagrafica.nome;
```



vettori di strutture

- *tipo_struttura* *nome_array* [*dimensione*]
- gli arrays di strutture sono utili per rappresentare basi di dati, ad esempio:

```
libro libri[100];
```

riserva spazio in memoria per ospitare un array di 100 elementi di tipo `libro`; per accedere ai campi di ognuno degli elementi struttura si utilizzano l'indice dell'array e l'operatore punto

```
libri[0].anno = 1994;  
strcpy(libri[0].titolo, "C++");  
cin >> libri[0].autore;  
cout << libri[0].editore;
```



unioni

- somigliano alle strutture, ma invece di allocare i campi in maniera contigua li sovrappongono nella stessa posizione di memoria; la sintassi è:
 - **union** *nome* {
 - *tipo_1 campo_1;*
 - ...
 - *tipo_n campo_n;*
 - };
- la quantità di memoria riservata per una unione è data dal campo più grande, perché i campi condividono la stessa zona di memoria
- servono per risparmiare memoria