



UNIVERSITÀ  
degli STUDI  
di CATANIA

# Strutture, union, enum, operatori bitwise

Corso di programmazione I AA 2021/22

Corso di Laurea Triennale in Informatica

---

Prof. Giovanni Maria Farinella

Web: <http://www.dmi.unict.it/farinella>

Email: [gfarinella@dmf.unict.it](mailto:gfarinella@dmf.unict.it)

Dipartimento di Matematica e Informatica

Le strutture sono molto usate nel linguaggio C.

Definite mediante la keyword **struct**.

Una struttura è un tipo **composto di un insieme di dati** (comunemente detto record).

La struttura C è l'antenato della classe C++. A differenza di quest'ultima:

- **contiene solo attributi** (NO funzioni/metodi)
- non ci sono specificatori di accesso (qualunque membro è public)

# Strutture in C

**typedef** si può usare in C/C++ per definire alias di tipi esistenti.

```
1  /* Codice C */
2  struct struttura{
3      int a;
4      float x;
5  };
6
7  struct struttura str;
8  str.x = 0.8; // operatore punto
9  str.a = 90;
10
11 typedef struct struttura mystruct; //alias
12 mystruct str1;
13 mystruct* str_ptr = &str1;
14 str->a = 17; //operatore narrow
```

# Strutture in C

In C, l'unico modo di “associare” funzioni alle strutture, è quello di dichiarare all'interno di esse dei **puntatori a funzioni.**, e poi assegnare al puntatore una funzione.

```
1  struct struttura{
2      float x, y;
3      float (*sum) (struct struttura *ptr);    };
4
5  float sum_func(struct struttura *ptr){
6      return str->x + str->y;    }
7
8  struct struttura str;
9  str.x = str.y = 0.9;
10 str.sum = sum_func; // associa la funzione al puntatore
11 printf("sum=%f\n", str.sum(&str));
```

Puntatore a funzione vs funzione.

```
1  int sum(int a, int b);  
2  
3  int (*sum_ptr)(int a, int b);
```

Linea 1: **funzione di nome sum** con due paramatri formali int che restituisce un intero.

Linea 3: **puntatore ad una funzione** che prende due paramatri formali int e che restituisce un intero. Il nome del puntatore è `sum_ptr`.

Esempi svolti

33\_01.c

La parola chiave `struct` in C++ è stata mantenuta per ragioni di **compatibilità con il C**.

Con essa si possono definire le classi, come con la parola chiave `class`.

`class` vs `struct`:

- Se usata la parola **class**, i membri con visibilità *unqualified* sono **private** per default;
- viceversa, se usata la parola **struct**, i membri con visibilità *unqualified* sono **public** per default;

# Strutture in C++

## class vs struct

```
1  struct A{
2      int x, y, z; //public!
3
4      A(){}; //public!
5  };
6
7  class B{
8      int x, y, z; //private
9
10     void doSomething() { /* ... */ } // private
11
12     public:
13         B(){};
14     };
```



Esempi svolti

33\_01.cpp

## typedef vs using

typedef ancora consentito in C++, ma si può usare anche la parola using.

```
1
2  class Test {};
```

3

```
4  typedef int myint; // myint alias di int
5  using anint = int; // anint alias di int
6
7  typedef Test mytest; // mytest alias di Test
8  using atest = Test; // atest alias di Test
```

Esempi svolti

33\_02.cpp

Una enumerazione è un tipo di dato che può assumere un valore tra un insieme di **nomi** definiti all'interno di essa.

```
1  enum Mese{  
2      Gennaio ,  
3      Febbraio ,  
4      //...  
5      Dicembre  
6  };  
7  //..  
8  Mese m = Mese::Gennaio;
```

I **nomi** definiti in una enum sono “esportati” nello scope in cui la enumerazione è stata definita;

```
1  enum Mese{  
2      Gennaio , Febbraio , /* ... */ , Dicembre  
3  };  
4  //..  
5  Mese m = Mese::Gennaio;    // OK  
6  m = Dicembre; //OK
```

Istruzioni alle linee **5 e 6 equivalenti**.

# Enumerazioni

I **nomi** di una enumerazione sono rappresentati in memoria come numeri interi, i cui valori **partono da zero**.

Conversioni `int`→`enum` debbono essere fatte esplicitamente.

Conversioni `enum`→`int` possono essere implicite.

```
1  enum Mese{
2      Gennaio , Febbraio , /* ... */ Dicembre // 0,1,2,..
3  };
4  //..
5  Mese m = Gennaio; // OK
6  Mese m2 = static_cast<Mese>(10); // OK, m2 = Novembre
7  m2 = 10; // Compile-time error
8  int mese = m2; // enum→int implicita , OK
```

# Enumerazioni

Le enumerazioni si possono usare nei costrutti switch

```
1  string traduci_mese(Mese m){
2  switch(m){
3      case Gennaio:
4          return "Gennaio";
5          break;
6      case Febbraio:
7          return "Febbraio";
8          break;
9      //...
10     case Dicembre:
11         return "Dicembre";
12         break;
13     }
14 }
```

Scope vs name clash.

```
1  enum Mesi{  
2      Gennaio ,  
3      Febbraio ,  
4      /* ... */,  
5      Dicembre  
6  };  
7  
8  //Compile-time error! name clash  
9  enum MeseEstivo{  
10     Giugno , Luglio , Agosto  
11  };
```



## Scope vs rappresentazione delle enumerazioni.

```
1  enum MeseInvernale{ //0,1,2
2      Dicembre , Gennaio , Febbraio
3  };
4
5  enum MeseEstivo{ //0,1,2
6      Giugno , Luglio , Agosto
7  };
8  //..
9  MeseInvernale m = Dicembre; //Dicembre=0, Giugno=0
10 if (m==Giugno) // Warning a compile-time!
11     //...
```

Come risolvere i problemi precedenti

- name clash (pag. 15)
- confronto tra enumerazioni differenti (pag. 16)

Dichiarare **enum class**:

- i nomi non sono esportati nello scope di definizione delle enum (OK name clash e confronti!)
- inoltre, sono totalmente **tipizzati**: anche conversioni `enum→int` necessitano di type-cast;

## enum class

```
1  enum class MeseInvernale{//0,1,2
2      Dicembre, Gennaio, Febbraio
3  };
4  enum class MeseEstivo{//0,1,2
5      Giugno, Luglio, Agosto
6  };
7  //..
8  MeseInvernale m = MeseInvernale::Dicembre; //necessario!
9  if(m==MeseEstivo::Giugno) // Compile-time error!
10     //...
11  int mese = static_cast<int>(m); // type-cast necessario!
```

## Esempi svolti

33\_03.cpp

33\_04.cpp

33\_05.cpp

# Union

Una **union** è una struttura i cui membri sono allocati tutti a partire dallo stesso indirizzo di memoria.

Lo spazio realmente occupato in memoria sarà uguale alla dimensione del campo più grande.

Di conseguenza tale struttura potrà contenere non più di un dato alla volta.

```
1  union Value {char c; int num;};  
2  //...  
3  Value v;  
4  v.c = 'z';  
5  //...  
6  v.num = 10;
```

# Union

Il record può essere un carattere oppure un numero.

La struttura di tipo union consente di risparmiare spazio in memoria.

```
1  enum Type { number, character };
2
3  union Value{ int num; char c; };
4
5  struct Record { Type t; Value v };
```

Esempi svolti

33\_06.cpp

# Operatori “bitwise” (bit a bit)

Gli operatori bitwise consentono di **operare sui singoli bit contenuti all'interno di una cella di memoria.**

**Operatori di spostamento** (shift) a destra e sinistra.

Si applicano ai tipi interi con o senza segno.

Sia  $a$  una **variabile intera senza segno** (unsigned {short,int,long}):

Operatore	Tipo	Operazione
$\ll$ (shift a sinistra)	binario	$a \ll b == a * 2^b$
$\gg$ (shift a destra)	binario	$a \gg b == \lfloor a * 2^{-b} \rfloor$



# Operatori “bitwise” (bit a bit)

Nello shift a sinistra di  $k$  bit:

- i  $k$  bit meno significativi diventano zero;
- i  $k$  bit più significativi si perdono, lasciando posto a quelli meno significativi;

Nello shift a destra di  $k$  bit:

- i  $k$  bit più significativi diventano zero;
- i  $k$  bit meno significativi si perdono, lasciando posto a quelli più significativi;

# Operatori “bitwise” (bit a bit)

Esempio:

Sia  $a$  una variabile intero corto (short), senza segno (unsigned).

Se  $a = 15$  (dec) = **00000000 000001111**b.

1. Shift  $a$  SINISTRA (bit a bit):  $a \ll 1$

( 00000000000001111  $\ll 1$  ) = 00000000000011110 = 30 (dec) =  $15 * 2^1$

2. Shift  $a$  DESTRA (bit a bit):  $a \gg 1$

( 00000000000001111  $\gg 1$  ) = 0000000000000111 = 7 (dec) =  $\lfloor 15 / (2^1) \rfloor$

# Operatori “bitwise” (bit a bit)

## Operatori logici bitwise:

AND	&
OR	
NOT (complemento a uno)	~
XOR	^

La semantica è la stessa di quella degli operatori logici per le variabili booleane, ma gli operatori logici bitwise **operano sui singoli bit**.

### Bitwise **AND** (binary): &

1111000000000000 &

0000000000001111

---

0000000000000000

**Bitwise OR** (binary): |

$$\begin{array}{r} 1111000000000000 \mid \\ 0000000000001111 = \\ \hline 1111000000001111 \end{array}$$

**Bitwise NOT** (unary) :  $\sim$

$( \sim 1111000000000000 ) = 0000111111111111$

**Bitwise XOR** (binary):  $\wedge$

1111000000000000  $\wedge$

0000000000001111

---

1111000000001111

## Operatori “bitwise” (bit a bit)

Esempi svolti

33\_07.cpp



FINE