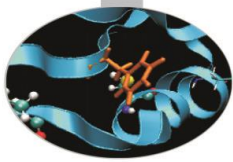
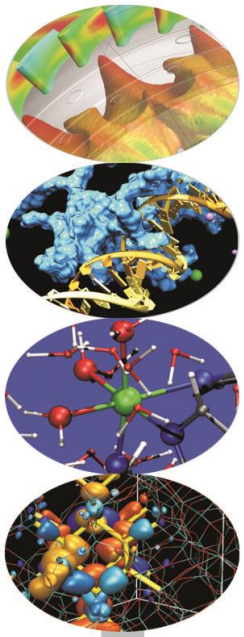
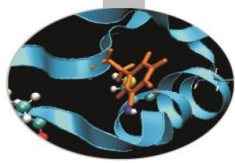


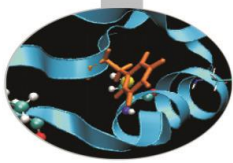
Dati Strutturati





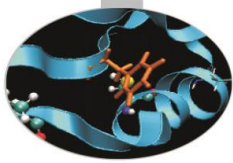
Indice

- **L'istruzione enum**
- **L'istruzione typedef**
- **Le struct in C**
- **Le struct in C++**
- **Le union**



Tipi di dato derivato

- A partire dai tipi di dato base, vengono creati i cosiddetti tipi derivati. Un array è un esempio di tipo di dato derivato in cui tutti gli elementi sono dello stesso tipo. Non sempre questo tipo di dato è sufficiente a rappresentare correttamente la variabile del nostro problema.
- Gli altri tipi derivati sono:
  - Enumerazioni
  - Istruzione typedef
  - Struct
  - Union e campi di bit



Enumerazioni

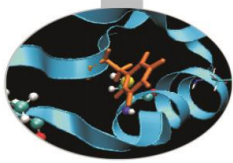
Il tipo di dato enumerazione (enum) viene utilizzato per gestire in maniera raffinata un insieme di valori interi specificati dall'utente.

```
enum nome_enumerazione{lista_elementi};
```

```
enum{bianco, rosso, verde,
   blue=10,nero,arancio=blue+8};
/* è del tutto equivalente ad elencare, "enumerare"
   i singoli valori*/
const int bianco=0,rosso=1,verde=2,blue=10,nero=11,
            arancio=18;
```

Per default il primo elemento è indicizzato da zero e tutti gli elementi che lo seguono hanno valori incrementali rispetto a questo

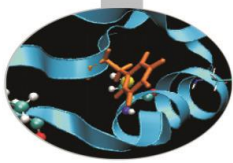
Una volta che l'enumerazione è stata creata con un nome questa costituisce un nuovo tipo di dato



Enumerazioni

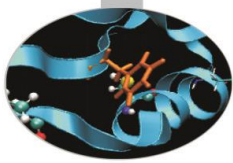
La conversione da enum ad intero è implicita, il viceversa no

```
enum condizione {alto,medio,basso};

condizione x, y;

X = alto;    //ok, variabile di tipo condizione
             //  inizializzata

Y = 1;       //errore non esiste una conversione da
             //  intero a condizione

Y = condizione(1); //ok, conversione esplicita di tipo,
                   // da int a condizione

int i = basso;     //ok, accettata la conversione da enum
                   a int
```



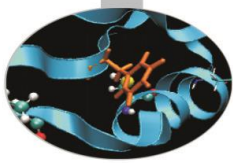
Enumerazioni

```
/*l'uso tipo del costrutto enum è in congiunzione con il
   costrutto switch*/
enum cond_cont {Dirichlet=1, Neumann, Robin};
cond_cont a;
a = Robin; //equivalente a scrivere a=cond_cont(3)

switch(a){
case Dirichlet:
   cout<<"Trattamento condizioni al contorno Dirichlet"<<endl;
   break;
case Neumann:
   cout<<"Trattamento condizioni al contorno
        Neumann"<<endl;
   break;
case Robin:
   cout<<"Trattamento condizioni al contorno Robin"<<endl; break;
Default:
   cout<< "condizione al contorno non definita"<<endl;
}
```



Istruzione typedef

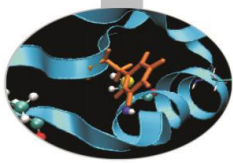
- Il C è un linguaggio fortemente tipizzato (e questa caratteristica è stata accentuata nel C++) per cui ogni variabile è associata ad un tipo di dato.
- Tramite l'istruzione *typedef* è però possibile definire un nuovo nome da associare ad un tipo di dato definito dal programmatore.
- L'effetto che si ottiene è di un codice più facilmente leggibile.
- Di fatto non si sta però definendo nessun nuovo tipo di dato.
- Solo con l'uso delle classi in C++ si può ottenere come risultato di definire un nuovo tipo di dato che abbia gli stessi privilegi sintattici rispetto a quelli forniti dal linguaggio.



Istruzione typedef

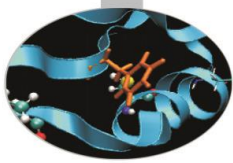
```
/*sintassi utilizzo*/
int fagioli;                          typedef int Legumi;
fagioli=18;                           Legumi fagioli=18;
/*queste due versioni sono equivalenti per il
   compilatore*/
typedef enum{FALSE=0,TRUE} Boleani;
Boleani flag=TRUE;
/*Si noti che in C++ il tipo bool esiste già come
   predefinito*/
/*il tipico esempio di utilizzo di typedef è per nominare
   dati strutturati*/
typedef struct{int data_di_nascita; char *nome; char
   *cognome;}Impiegato;
Impiegato azienda[100];
```

C-struct

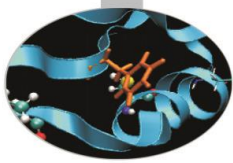


- Il tipo di dato strutturato eterogeneo per eccellenza in C/C++ è la *struct*
- La grande differenza tra la sintassi delle *struct* nei due linguaggi risiede nel fatto di poter dichiarare (C++) o meno (C) funzioni all'interno di *struct.*
- Per accedere ai dati viene usato l'operatore (.).
- Di seguito verranno illustrate queste differenze tramite esempi.

C-struct



```
/*definizione di una struttura elementare mystruct*/
struct mystruct{     //nome della struct
   int mydata1;      //membro1 della struttura
   double mydata2; //membro2 della struttura
   char mydata3;     //membro3 della struttura
}; /*chiusura blocco di istruzioni struct */

/*uso di 2 variabili strutturate */
  struct mystruct myvar1, myvar2; // In C++ struct
                                  // si può omettere

/*posso definire le variabili anche in linea con la
   definizione della struct stessa*/
struct mystruct{
   int mydata1;
   double mydata2;
   char mydata3;
} myvar1, myvar2;
```

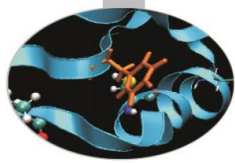


C-struct

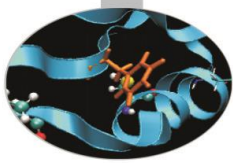
- L'uso che si fa delle *struct* in C essenzialmente è quello di gestire gruppi di variabili logicamente interconnesse tra loro.
- Raggruppandole in una *struct* risulta poi più agevole modificare/gestire il codice che le contengono.

```
/*inizializzazione, accesso ai membri e assegnamento*/
struct mystruct{
        int mydata1;
        double mydata2;
        char mydata3;
};
struct mystruct myvar1={3,88.6,'p'}; /*inizializzazione*/
struct mystruct myvar2={5,55.3,'r'};
myvar1.mydata1=1;/*accesso ai membri*/
myvar1.mydata2=90.0;
myvar1.mydata3= 'X';
myvar2=myvar1;    /*assegnamento, ora tutti i membri di
                  myvar2 hanno gli stessi valori dei
                  menbri di myvar1*/
```

C-struct

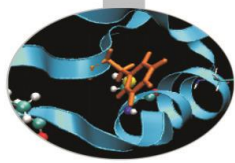


```
/*strutture contenti membri  array*/
struct mystruct{
   int mydata1[2];
   double mydata2[5];
   char mydata3;
};
mystruct myvar1={{3,5},{88.6,43.7,77.9},'p'};

myvar1.mydata1[0]=1;
myvar1.mydata1[1]=2;
myvar1.mydata2[3]=44.70;
myvar1.mydata2[4]=90.0;
myvar1.mydata3= 'X';

/* per l'accesso ai membri di fatto si usa l'operatore (.) in
   congiunzione con l'operatore([])*/
```



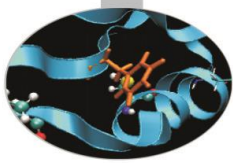
C-struct

```
/* array di struct; quando le variabili di tipo struct
   crescono in numero può essere utile raggrupparle in un
   array */
 struct mystruct{
   int mydata1[2];
   double mydata2[5];
   char mydata3;
   } many[3];
 many[0].mydata1[1]=33; /*nella prima variabile dell'array
                          many modifico il secondo valore
                          del membro mydata1*/
 many[1].mydata2[3]=77,5; /*nella seconda variabile
                          dell'array many modifico il
                                quarto valore del membro
   mydata2*/
 many[2].mydata3='g'; /*nella terza variabile dell'array
                          many modifico il valore
                          del membro mydata3*/
```



C-struct

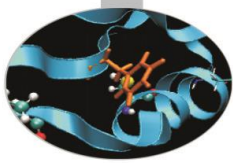
```
/*struct con membri struct*/
strcut inside{
   double mydatainside1;
   char mydatainside2;
};
struct mystruct{
   int mydata1;
   struct inside mydata2;
} myvar1;

myvar1.mydata1=3;
myvar1.mydata2.mydatainside1=65.8;
myvar1.mydata2.mydatainside2='u';
```



C++-struct

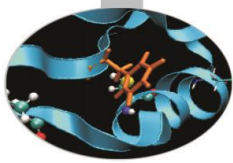
```
/*in C++ vale tutto quanto detto per il C, con
  l'aggiunta di poter definire funzioni come membri
  della struct*/


struct date {
  void init_date ( date& d, int, int, int);
  void add_year (date& d, int n);
  void add_month (date& d, int n);
  void add_day (date& d, int n);
  int d, m, y;
};
/*per utilizzare le funzioni membro si usa sempre
  l'operatore (.)*/
```

C++-struct

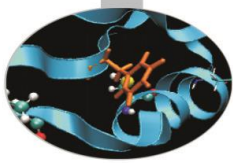


```
#include <iostream>
using namespace std;
struct confunc {
        int A, B;
        int getA() { return A;} // ritorna il valore di A
        int getB() { return B;} // ritorna il valore di B
        void setA(int n) { A = n;} // assegna un nuovo valore a A
        void setB(int n) { B = n;} // assegna un nuovo valore a B
        } myvar;

int main() {
        myvar.setA(15);
        myvar.setB(63);
        cout << myvar.getA() << endl;
        cout << myvar.getB() << endl;
        return 0;
         }
```

Esempio

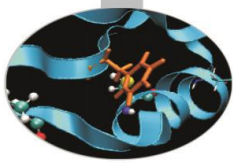


```
/*esempio C/C++ struct con typedef*/
typedef struct {
        int data1;
        int data2;
        }
         MY_S;

/* in questo modo abbiamo definito un typedef per MY_S ma
attenzione che questa non è una variabile ma un nome alternativo
per chiamare una variabile strutturata */

int  main() {
        MY_S my_var_s; // ok, usiamo il typedef
        my_var_s.data1=6; //ok usiamo la variabile
        my_var_s.data2=52;
        MY_S.data1 = 5; /* errore MY_S è il nome del typedef
                                struct non di una variabile */
        return 0; }
```



Commenti

- In C++ è inoltre possibile, tramite l'uso di parole chiave *(public, private*), non presenti in C, ottenere l'incapsulamento dei dati e/o di funzioni definite in una *struct*
- In questo modo tramite una *struct* è possibile ottenere tipi di dato "simili" a quelli che si otterrebbero tramite il costrutto di classe.
- Essendo questo un corso introduttivo ed esulando dal programma, non verrà affrontato.

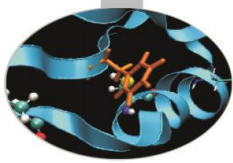


Union

- Le *union* sono particolari *struct* atte a salvare spazio in memoria.
- Solo un membro di una union può esistere in un determinato istante di tempo.
- Lo spazio massimo occupato in memoria da una union è pari allo spazio massimo occupato dal suo membro più grande.
- Generalmente vengono usate all'interno di *struct* per definire dati che possono esistere solo alternativamente (vedi esempio).

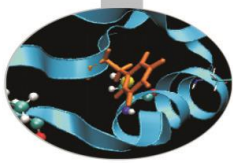
```
union misticanza{              // union con tre membri
                int j;
                char a;
                double b;      // massimo spazio occupato
                };
misticanza mix;                // variabile di tipo misticanza
```

Esempio



```
/*esempio union*/
#include<iostream>
union misticanza{
        int j;
        char a;
        double b;
};
int main()
{
misticanza mix;        //allocazione di memoria sizeof(double)
cout<<"solo il membro j viene utilizzato"<<endl;
mix.j=9;
cout<<"membro j: "<<mix.j<<endl<<"membro a: "<<mix.a<<endl<<"membro b:
   "<<mix.b<<endl;


cout<<"solo il membro a viene utilizzato"<<endl;
mix.a='P';


cout<<"membro j: "<<mix.j<<endl<<"membro a: "<<mix.a<<endl<<"membro b:
   "<<mix.b<<endl;
```

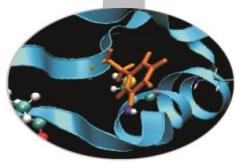
Esempio



```
cout<<"solo il membro b viene utilizzato"<<endl;
mix.b=56.9;
cout<<"membro j: "<<mix.j<<endl<<"membro a:
   "<<mix.a<<endl<<"membro b: "<<mix.b<<endl;
return 0;}
```

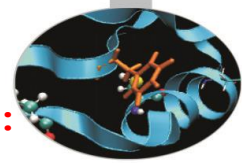
```
[user@node001 user]$ ./a.out
solo il membro j viene utilizzato
membro j: 9
membro a:
membro b: 4.85998e-270
solo il membro a viene utilizzato
membro j: 80
membro a: P
membro b: 4.85998e-270
solo il membro b viene utilizzato
membro j: 858993459
membro a: 3
membro b: 56.9
```

Commenti

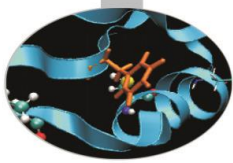


- L’esempio mostra come essendo definibili solo una dei tre dati alla volta il valore degli altri due è impredicibile.
- Il compilatore non pone alcun vincolo su questo.
- E’ compito del programmatore utilizzare correttamente i dati che sono definiti in quel momento.

Esempio union in struct



```
/* uso (salva memoria) possibile di union all'interno di struct:
   si supponga di avere una struttura figura2d (quadrato o
   triangolo) ma che ovviamente non può essere contemporaneamente
   un triangolo e un quadrato */
struct figura2d {
   char[20] nome;
   bool tipo; /* 0 se triangolo, 1 se quadrato (etichetta di
                 tipo)*/
   union {
        triangolo tria;        //triangolo è a sua volta una
   struct
        quadrato quad; //quadrato è a sua volta una struct
   };
};
int main(){
figura2d fig1;
fig1.nome="figura1";
fig1.tipo=0;
fig1.tria.base= 12.9;
fig1.tria.altezza=5.5;
}
```



Commenti

- Dal momento che una *figura 2d* non può essere contemporaneamente un triangolo o un rettangolo tramite l'uso di union si evidenzia questo aspetto
- Si noti comunque come tutto il peso dell'implementazione corretta sia a carico del programmatore