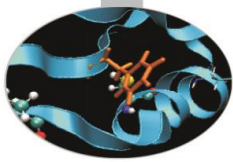
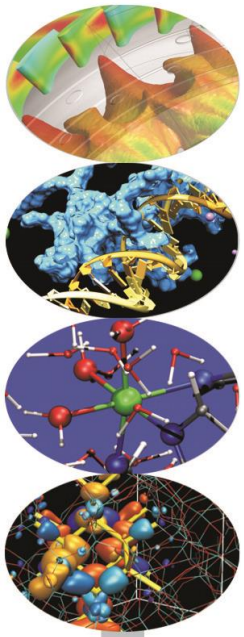
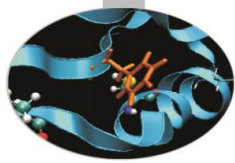


Dati Strutturati

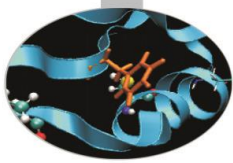


Indice



- **L'istruzione enum**
- **L'istruzione typedef**
- **Le struct**
- **Le union**

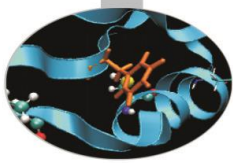
Data structure



- Perché introdurre i dati strutturati:
  - Necessità di gestire facilmente e manipolare grandi moli di dati
  - Possibilità di introdurre ‘concetti’ all’interno del codice
  - Rendere il codice più leggibile

Il C mette a disposizione:

- Scalari
- Array
- Dati Strutturati



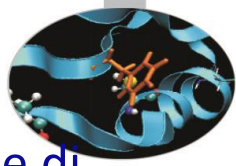
Tipi di dato derivato

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

Enumerazioni



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

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

```
Enum Colore{bianco, rosso, blue=10, nero=18};
```

È del tutto equivalente a:

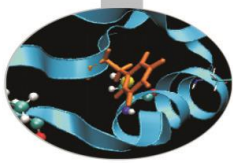
```
const int bianco=0, rosso=1, blue=10, nero=18;
```

Oppure a

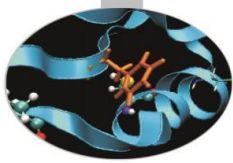
```
#DEFINE bianco = 0
#DEFINE rosso = 1
#DEFINE blue = 10
#DEFINE nero = 18
```

L'enum definisce un set di valori interi costanti definiti da un identificatore.

Enumerazioni



- Per default il primo elemento è indicizzato da zero e tutti gli elementi che lo seguono hanno valori incrementali rispetto a questo.
- I valori possono essere ripetuti
- Gli identificatori devono essere univoci
- Una volta che l'enumerazione è stata creata con un nome questa costituisce un nuovo tipo di dato e può essere utilizzata per istanziare nuovi dati.

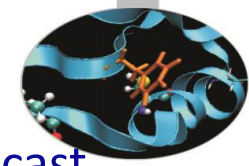


Enumerazioni

- Esempio:

```
enum Days {Mon, Tue, Wed,Thur,Fri, Sat, San};
enum Days Today = Mon;
enum Days2{Mon=1, Tue, Wed,Thur,Fri, Sat, San};
enum Days3{Mon=1, Tue, Wed=7,Thur,Fri, Sat,
San=7};
```

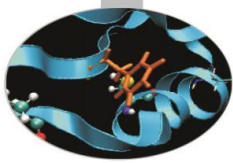
Enumerazioni



La conversione da enum ad intero è implicita, il viceversa no e necessita un cast esplicito.

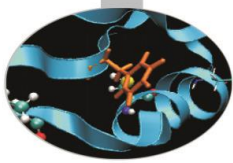
```
enum Mesi{Gennaio=1, Febbraio, Marzo, Aprile,
   Maggio,Giugno,Luglio, Agosto,Settembre, Ottobre
   Novembre, Dicembre};

enum Mesi x, y;

x = Febbraio;      // ok, variabile di tipo condizione
                   //  inizializzata

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

y = Mesi(1);       //ok, conversione esplicita di tipo,

                   // da int a condizione

int mese = Febbraio;     //ok, accettata la conversione da
                         //enum a int

int mese2 = Mesi(30)     // Attenzione non c'è controllo
                         //sul valore
```

Enumerazioni



```
/*l'uso tipico del costrutto enum è insieme al costrutto switch*/
enum Condition {Dirichlet=1, Neumann, Robin};
enum Condition a;
a = Robin;

switch(a){
case Dirichlet:
        /* Condizioni al contorno Dirichlet */
        break;
case Neumann:
        /* Condizioni al contorno Neumann */
        break;
case Robin:
        /* Condizioni al contorno Robin */
       break;
Default:
        fprintf(stderr, "Condizione al contorno non definita\n");
}
```

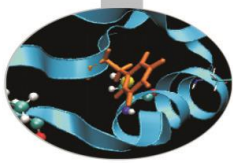
Garantisce maggiore leggibilità al codice



Enumerazioni

- Come parametri a funzioni. La funzione AlignObject allinea una stringa a destra, a sinistra, o al centro

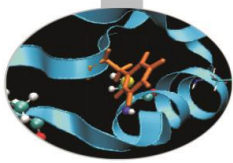
```
enum  Align {Right, Center, Left};
void AlignObject(char * var, enum Align type)
{
      if(type == Right)
            //do something
      else
            //do something
}
AlignObject(``My test'', Left);
```



Istruzione typedef

- Il C è un linguaggio fortemente tipizzato (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 e non solo.
- L'effetto che si ottiene è un codice più facilmente leggibile.
- Di fatto non si sta però definendo nessun nuovo tipo di dato
- Si introducono *alias* a tipi di dato

Istruzione typedef

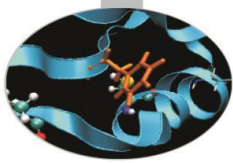


```
/*esempi di utilizzo*/
```

```
int number;                    typedef int Integer;
number = 18;                   Integer number=18;
```

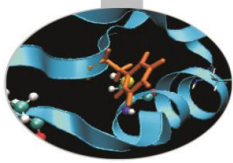
```
typedef int INTVECTOR[3];
typedef int *INTPTR;
```

```
int a=10;
INTPTR pt= &a;
INTVECTOR v={1,3,4};
printf(``value of integer %d\n'',*pt);
```



Istruzione typedef

```
/*queste due versioni sono equivalenti per
  il compilatore*/
typedef enum{FALSE=0,TRUE} Booleani;
Booleani flag=TRUE;

/*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];
```



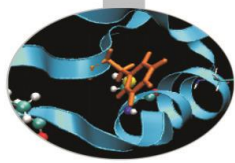
Istruzione typedef

- *typedef* è ampiamente utilizzato nella C Standard Library
- Per esempio il tipo *size_t* definito nella libreria *stddef.h*, utilizzato come return value nella funzione *sizeof*().
- size_t è un alias definito come:

```
#define __SIZE_TYPE__  long unsigned int
typedef __SIZE_TYPE__ size_t;
```

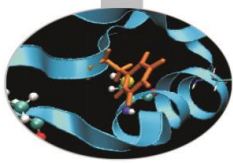
- Per esempio *clock_t* definito nella *time.h* usato come return value nella funzione *clock()* è a sua volta un alias.

C-struct



- Il tipo di dato strutturato eterogeneo per eccellenza in 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.*
- Una struct è una collezione etorogenea di dati legati tra di loro.
- Una struct è un tipo di dato user-defined
- Una struct è composta da campi eterogenei tra di loro

C-struct



La sintassi per definire una struct è la seguente:

struct [<tag_identifier>]

{

<type_specifier> <identifier>;

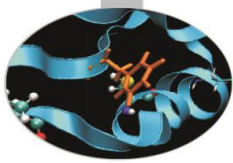
<type_specifier> <identifier>;

…

};

```
/*definizione di una struttura elementare mystruct*/
struct Mystruct{     //nome della struct
   int Integer;      //membro1 della struttura
   double Double; //membro2 della struttura
   char Char;        //membro3 della struttura
};
```

Ciascuna struct occupa la memoria necessaria per gestire i suoi attributi.



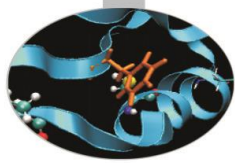
C-struct

Utilizzo di dati struct

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

/* posso definire le variabili anche contestualmente alla
definizione della struct */
struct mystruct{
       int Integer;
       double Double;
       char char;
} myvar1, myvar2;
```

Il compilatore riserva memoria sono nel momento in cui vengono istanziati i dati.
La dichiarazione del template della struct senza istanze non comporta allocazione di memoria.

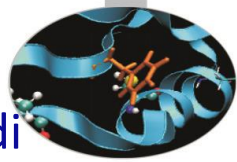
C-struct



- E' possibile avere struct annidate.
- Un struct senza tag è detta struttura anonima.

```
struct BigStruct
{
        int Integer;
        struct
        {
                int c;
                int b;
        } MyStruct;          //struct anonima
};
struct BigStruct big;    // struct di struct
```

C-struct



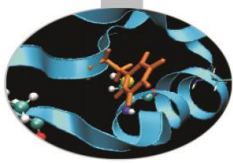
L'uso che si fa delle *struct* in C essenzialmente è quello di gestire gruppi di variabili logicamente interconnesse tra loro.

Raggruppandole in una *struct* risulta poi più agevole modificare/gestire il codice che le contengono.

L'accesso ai dati di una struct avviene tramite l'operatore di . O tramite l'operatore -> se la struct è acceduta tramite puntatore.

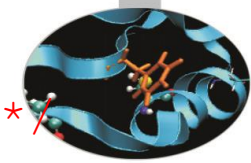
```
struct Mystruct
{
        double d;
        int i;
        char c;
};
```

C-struct



```
/* Inizializzazione: */
struct Mystruct myvar1={3,88.6,'p'};
struct Mystruct myvar2={5,55.3,'r'};
struct Mystruct *ptr = & myvar2;
struct Mystruct * ptr2 = (Mystruct*)malloc(sizeof(Mystruct)); /*
allocazione dinamica*/
ptr2->i=2;
ptr2- >d=2.0;
ptr2- >c='f';
myvar1.i=1;     /* accesso ai singoli membri */
myvar1.d=90.0;
myvar1.c= 'X';
myvar2=myvar1; /* copia del contenuto */
ptr->mydata1=2;    /* accesso tramite puntatore */
free(ptr2); /* deallocazione puntatore */
```

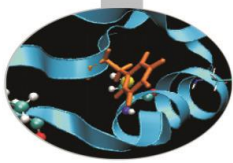
C-struct



```
struct mystruct{   /*strutture contenti membri  array*/
   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';
```

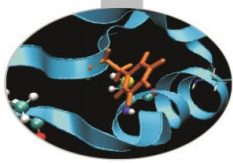
Per l'accesso ai membri di fatto si usa l'operatore (.) in congiunzione con l'operatore ([])

C-struct



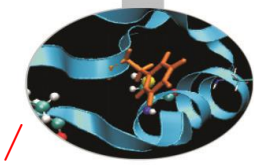
```
struct mystruct{      /* array di struct */
   int mydata1[2];
   double mydata2[5];
   char mydata3;
   } many[3];

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

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

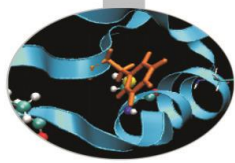
Esempio



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

In questo modo si è definito un typedef per MY_S ma attenzione che questa non è una variabile, ma un nome alternativo per chiamare una variabile strutturata

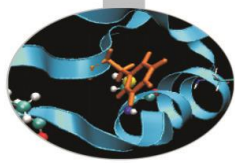
```
int  main() {
    MY_S my_var_s; // ok, usiamo il typedef
    my_var_s.data1=6; //ok usiamo la variabile
    my_var_s.data2=52;
    MY_S.data1 = 5; /* errore MY_S è il nome del typedef
                              struct non di una variabile */
```



Esempi: lista concatenata

- La lista concatenata è una struttura che organizza i dati in maniera sequenziale.
- Mentre gli elementi di un array sono accessibili direttamente, gli elementi di una lista devono essere acceduti sequenzialmente: se voglio accedere all'iesimo elemento devo scandire la lista dal primo all'(i − 1)esimo elemento.
- Mentre le dimensioni di un array sono rigide, le dimensioni di una lista variano dinamicamente, tramite le operazioni di inserzione e cancellazione di elementi.
- Elementi logicamente adiacenti in una lista possono essere allocati in posizioni di memoria non adiacenti.

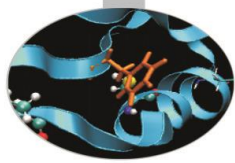
Esempio: lista concatenata



Le struct permettono di realizzare strutture dati flessibili quali code, pile, liste. La base per realizzare queste costruzioni è definire una struct contenente un puntatore a sé stessa:

```
struct Nodo {
    int Val;
    struct Nodo *Prossimo;
};
```

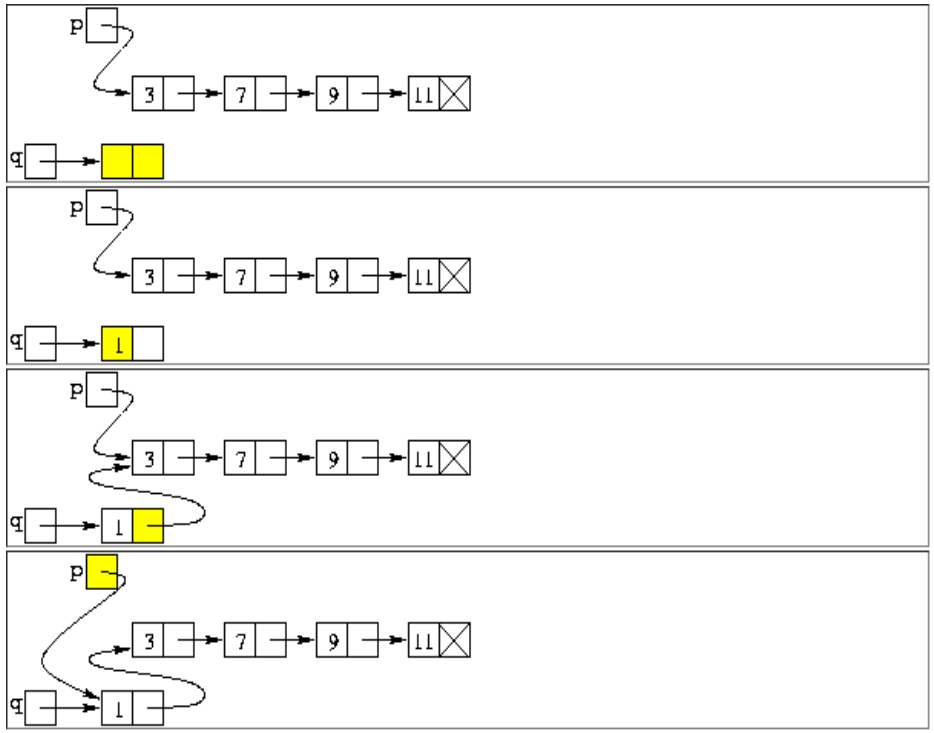
Il C++ ha strumenti appositi per realizzare e manipolare strutture dati di questo tipo.

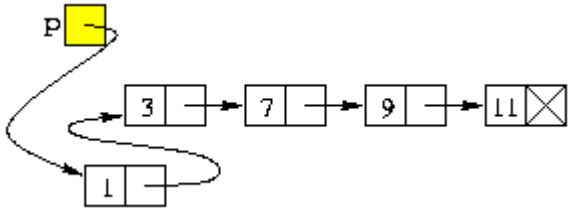


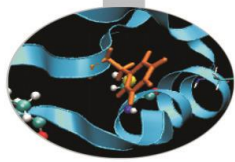
Esempio

Inserimento di un nodo all'inizio della lista. La lista di nodi è rappresentata da un puntatore

```
Nodo* p=NULL;

Nodo *q = malloc(sizeof(Nodo));

q->val=1;
q->Prossimo=p;

p=q;
```



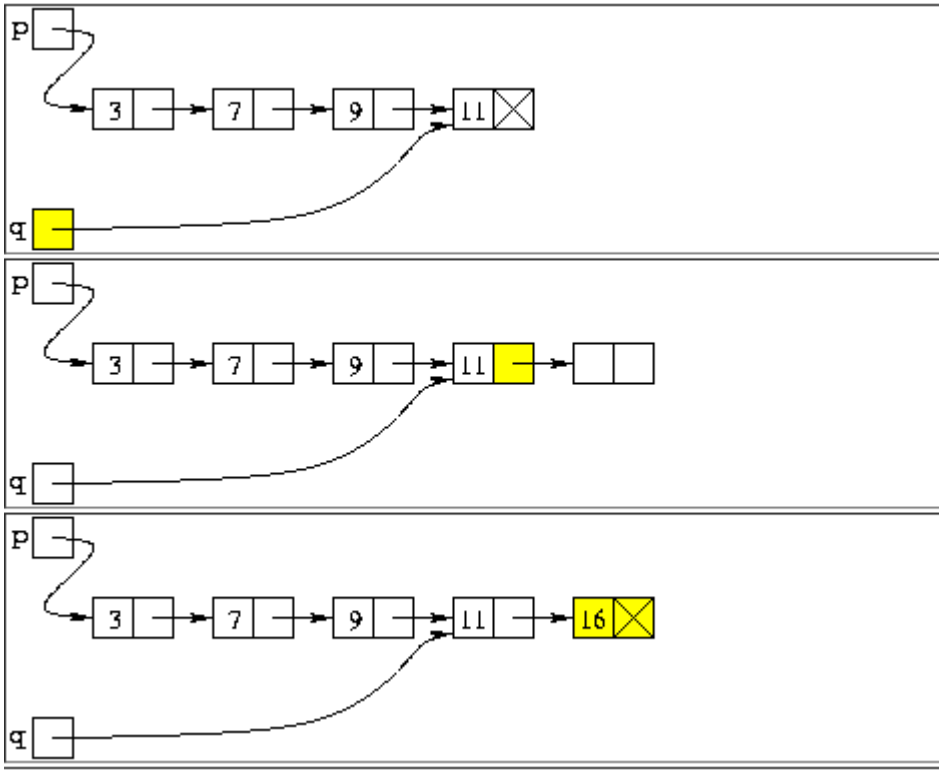


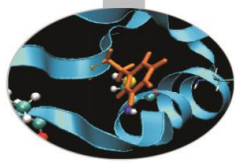


Esempio

- Inserimento di un nodo in fondo alla lista

```
if ( p == NULL )
{
     p = new Nodo;
     p->Val = 16;
     p->prossimo = NULL;
}
else
{
     Nodo*q;
     for(q=p; q->Prossimo !=NULL;
     q= q->Prossimo){}
     q->Prossimo = new Nodo;
     q->Val=16;
     q->Prossimo=NULL;
}
```

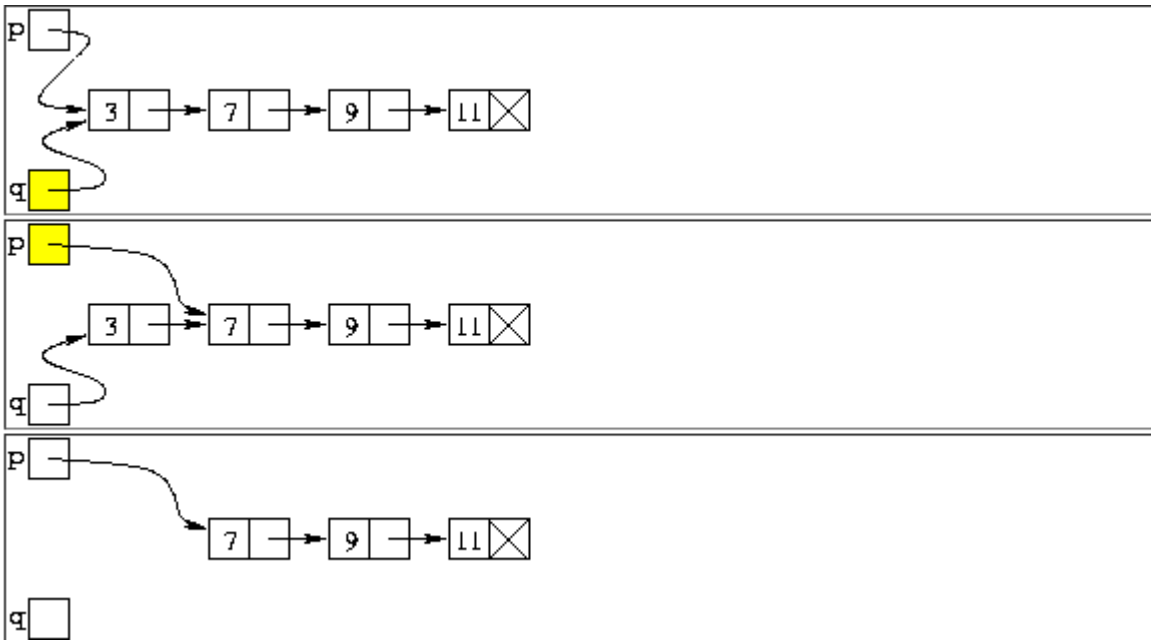


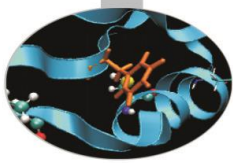


Esempio

- Eliminazione del primo elemento della lista

```
Nodo * q= p;
p=p->prossimo
delete q;
```

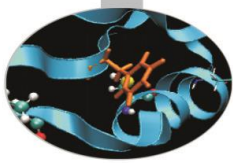




Esempio

Per stampare il contenuto della lista

```
Nodo *q; // Utilizza un nodo ausiliario q
q= p; // Punta q al primo nodo della lista
while ( q != NULL )
{
      printf(''%d \n'', q->Val);
      q = q->prossimo;
}
```



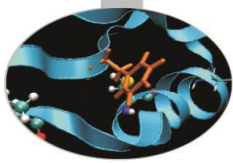
Esempio

- Per effettuare una ricerca

```
int n;
Nodo *q = p->prossimo;
while ( q != NULL && q->Val != n )
   q = q->prossimo;
```

Tramite struct possiamo implementare container di tipo Stack (LIFO) o Queue (FIFO).

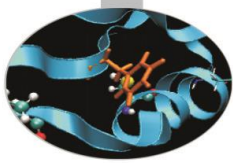
La STL C++ definisce strutture dati generiche, algoritmi e iteratori.



C-struct vs C++

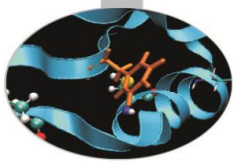
- Le struct definite in C consentono solo la definizione di dati aggregati.
- All'interno della definizione di una struct non è possibile definire funzioni. In C++ si!
- Possiamo solo definire puntatori a funzioni.

```
typedef int(*Operation)(int,int);   //Puntatore a
                                     funzione

typedef struct Data
{
    int a;
    Operation opt;
}
```



C-struct vs C++

```
/* * Add two numbers a and b */
int Add ( int a , int b )
{ return a+b ; }
/* * Multiple two numbers a and b *
/ int Multi ( int a , int b )
{ return a*b ; }
int main (int argc , char **argv)
{
    Data obj;
    obj.opt = Add;    //Add function
    obj.result = obj.opt(5,3);
    printf (" the result is %d\n", obj.result );
    obj.opt= Multi; //Multi function
    obj.result = obj.opt(5,3);
    printf (" the result is %d\n", obj.result );
    return 0 ;
}
```



Union

Le *union* sono particolari *struct* atte a salvare spazio in memoria.

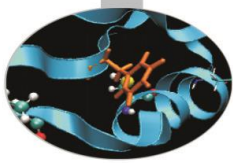
Solo un membro di una union può esistere in un determinato istante di tempo.

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).

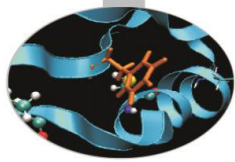
```
union poli_tipo{              // union con tre membri
                int j;
                char a;
                double b;     // massimo spazio occupato
                };
union poli_tipo mix;// mix è una variabile poli_tipo
```

Esempio



```
#include <stdio.h>  /* file e-poli_tipo.c */
typedef union {
   int j;  char a; double b;
} poli_tipo ;

int main() {  /* memoria allocata: sizeof(double) */
  poli_tipo mix;
  printf("solo il membro j viene utilizzato");
  mix.j = 9;
  printf(" membro j = %d;\t membro a = %c;\t membro b =
   %lf\n",mix.j,mix.a,mix.b);

  printf("solo il membro a viene utilizzato");
   mix.a = 'P';
  printf(" membro j = %d;\t membro a = %c;\t membro b =
   %lf\n",mix.j,mix.a,mix.b);
```

Esempio



```
  printf("solo il membro b viene utilizzato");
  mix.b = 56.9;
  printf(" membro j = %d;\t membro a = %c;\t membro b =
       %lf\n",mix.j,mix.a,mix.b);

return 0;}
```

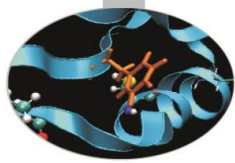
Output:

solo il membro j viene utilizzato membro **j = 9;** membro a = ; membro b = 0.000000

solo il membro a viene utilizzato membro j = 80; **membro a = P;** membro b = 0.000000

solo il membro b viene utilizzato membro j = 858993459; membro a = 3; **membro b = 56.900000**

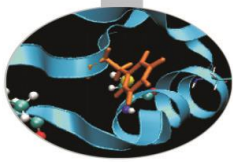
Commenti



L’esempio mostra come essendo definibili ogni volta solo uno dei tre dati, 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 ogni momento.

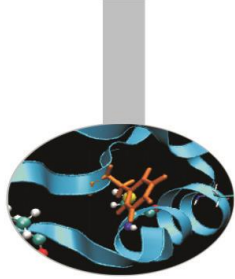


Esempio: struct vs union

```
union Union
{
        int i;
        double d;
};
```

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

```
int main()
{
        union Union u;
        struct Struct s;
        printf("Mem of union %d\n",sizeof(u));
        printf("Mem of struct %d\n", sizeof(s));
        return 0;
}
```

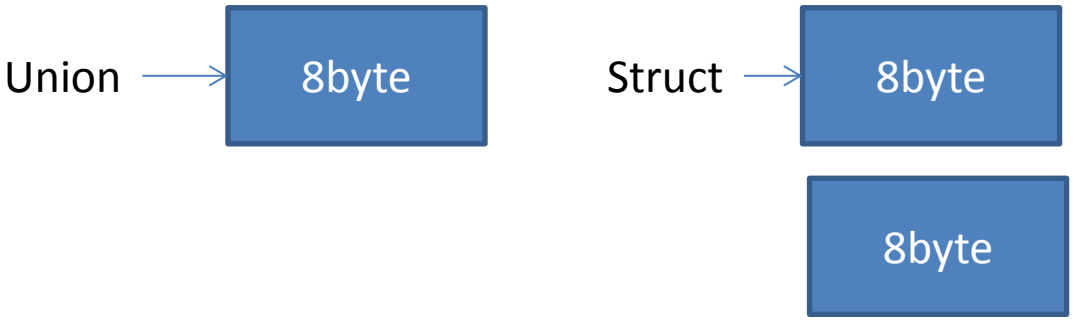


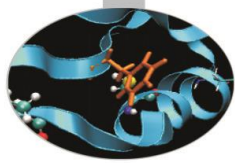
Esempio: struct vs union

- Output:

  Mem of union 8

  Mem of struct 16



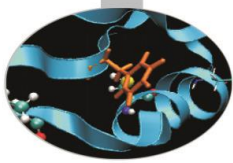


Esempio

La dimensione della union coincide con la dimensione del dato più grande.

```
typedef union
{
    int a;
    int b;
    double c;
}u1;

typedef union
{
    int a;
    char c;
}u2;
```

Esempio



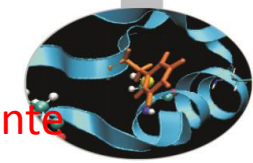
```
int main()
{
        u1 one;
        u2 two;
        printf("Dimension one: %d \n", sizeof(one));
        printf("Dimension two: %d\n" ,sizeof(two));
        return 0;
}
```

OUTPUT:

Dimension one: 8

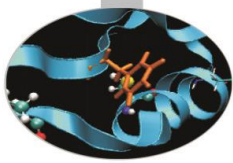
Dimension two: 4

Esempio union in struct



Si supponga di avere una struttura figura2d (quadrato o triangolo) ma che ovviamente non può essere contemporaneamente un triangolo e un quadrato

```
typedef enum Tipo {TRIANGOLO, QUADRATO}
struct figura2d {
   char[20] nome;
   Tip tipo;   /* 0 se triangolo, 1 se quadrato*/
   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=TRIANGOLO;
    fig1.tria.base= 12.9;
    fig1.tria.altezza=5.5;
}
```



Esempio union in struct

```
figura2d fig2;
fig2.nome="figura2";
fig2.tipo=QUADRATO;
fig2.quad.base= 12.9;
fig2. quad.altezza=5.5;
```

Dal momento che una *figura 2d* non può essere contemporaneamente un triangolo o un rettangolo, con l'uso di *union* si evidenzia questo aspetto

Si noti comunque come tutto il peso dell'implementazione corretta sia a carico del programmatore