# Strutture e ricorsione

- Strutture
- Programmazione ricorsiva

# `struct`: tipo di dato user-defined(I)

Oltre ai tipi di dati semplici, quali ***`int`***, ***`char`***, ***`float`***..., in C e' possibile creare nuovi tipi aggregando tipi di dati eterogenei sotto un unico nome. Questo nuovo tipo di dato va sotto il nome di *struttura* e si definisce con la parola chiave ***`struct`***:

```
struct [nome-struttura] {
    tipo_1 var_1;
    tipo_2 var_2;
        .
        .
    tipo_N var_N
} [identificativo];
```

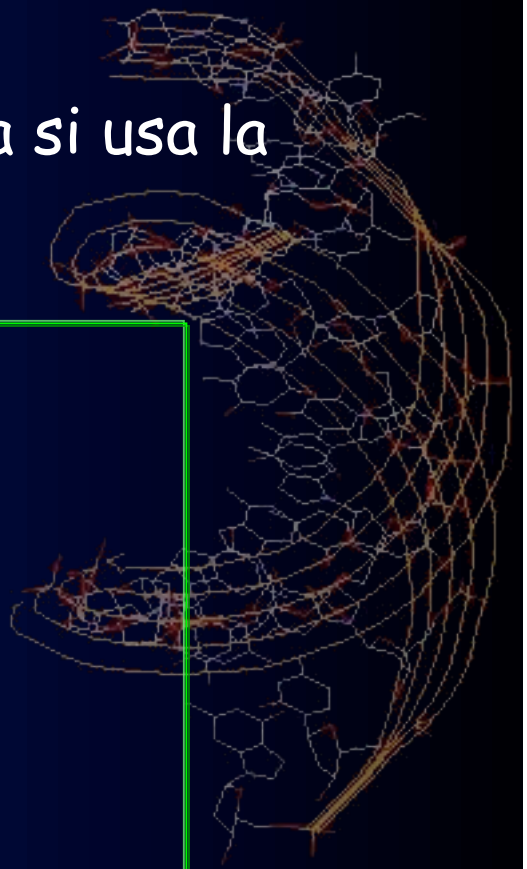
nome della struttura (tag)

membri o campi della struttura

# `struct`: tipo di dato user-defined(II)

Per accedere ai singoli campi della struttura si usa la notazione:

```
identificativo.var_1 = …;
identificativo.var_2 = …;
.
.
.
identificativo.var_n = …;
```



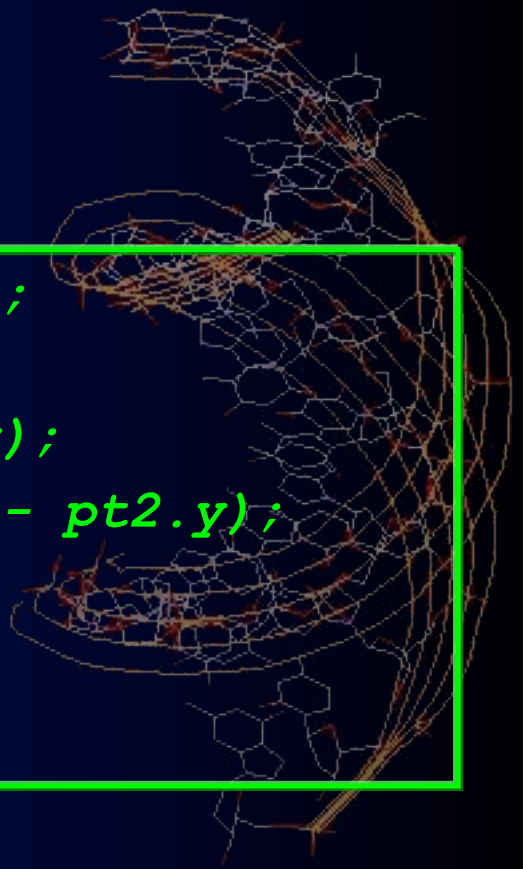
# Es: distanza tra due punti

```
#include <stdio.h>
#include <math.h>
struct point {
   double x;
   double y;
};
main()
{
  int i = -1, n;
  float dist;
  struct point pt1, pt2;

  printf("\n Enter x and y of point 1: ");
  scanf("%lf %lf", &pt1.x, &pt1.y);
. . .
```

*definizione della struttura.*

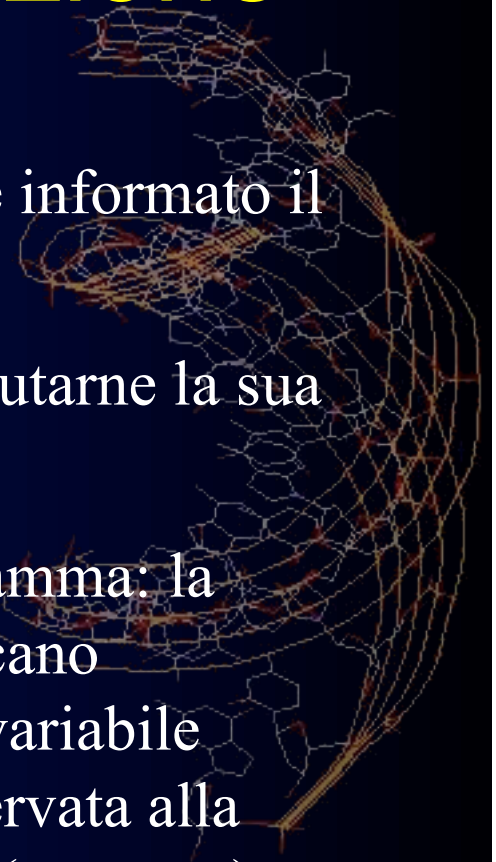
*dichiarazioni di variabili di tipo point.*

# . . . . continua



```
  printf("\n Enter x and y of point 2: ");
  scanf("%lf %lf", &pt2.x, &pt2.y);
  dist = (pt1.x - pt2.x) * (pt1.x - pt2.x);
  dist = dist + (pt1.y - pt2.y) * (pt1.y - pt2.y);
  dist = (dist > 0 ) ? sqrt(dist): 0.;
  printf("Distance = %lf", dist);
}
```

# Definizione vs. Dichiarazione



Nella definizione di una variabile (struttura) viene informato il compilatore sulla composizione della stessa.

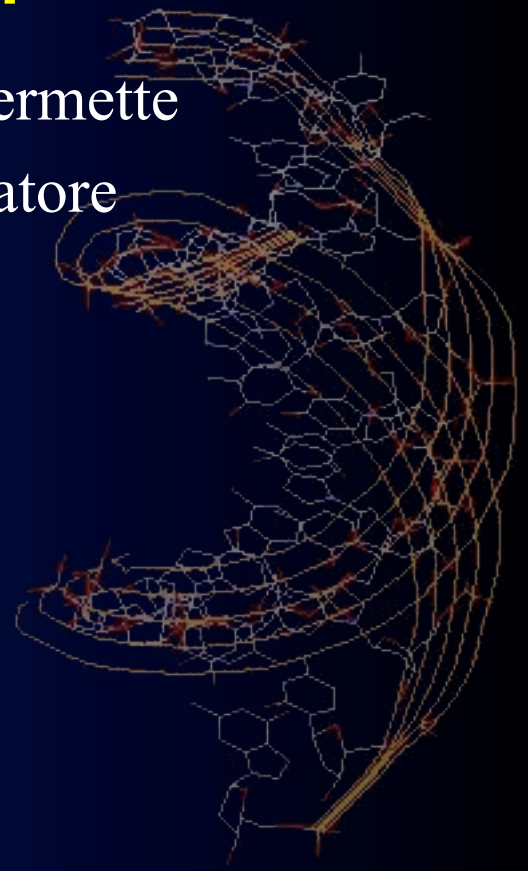
Noto il tipo di variabile (struttura) e' possibile valutarne la sua dimensione in termini di bit.

La dichiarazione riguarda l' esecuzione del programma: la dichiarazione e' trasformata in istruzioni che allocano fisicamente la memoria necessaria alla vita della variabile (struttura). Una zona di memoria viene quindi riservata alla variabile quando questa e' dichiarata: la variabile (struttura) esiste dal momento in cui viene dichiarata.

# Utilizzo di typedef

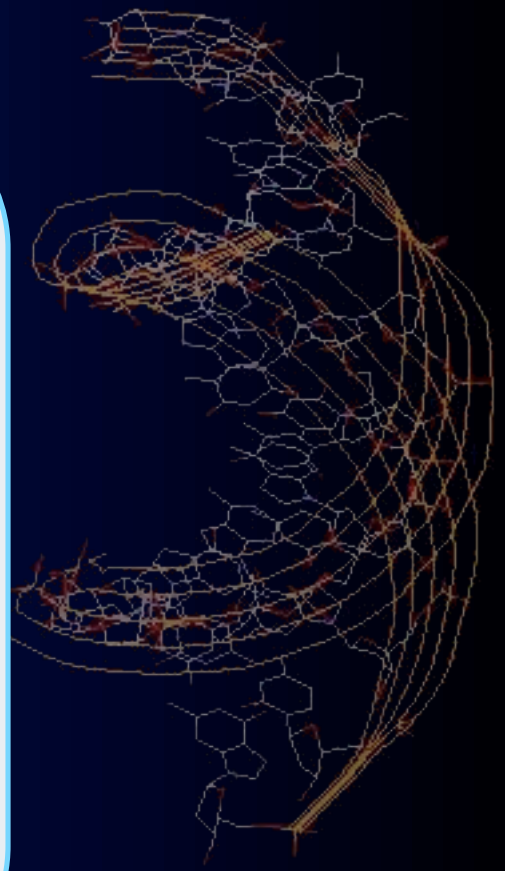
Il linguaggio C fornisce il meccanismo *typedef* che permette al programmatore di associare un tipo ad un identificatore

```
typedef char uppercase;
typedef int FEET;
typedef unsigned int site_t;  /* stddef.h */
…..
uppercase u;
FEET length, width;
```



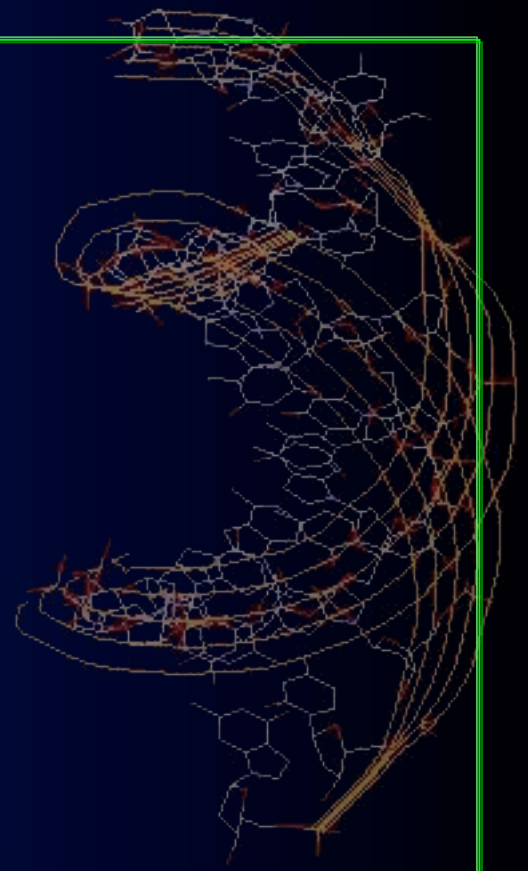
# Esempio

```
struct complex {
        double re;
        double im;
};
typedef struct complex complex;
void add(complex *a, complex *b, complex *c){
*a.re = *b.re + *c.re;
*a.im = *b.im + *c.im;
}
```

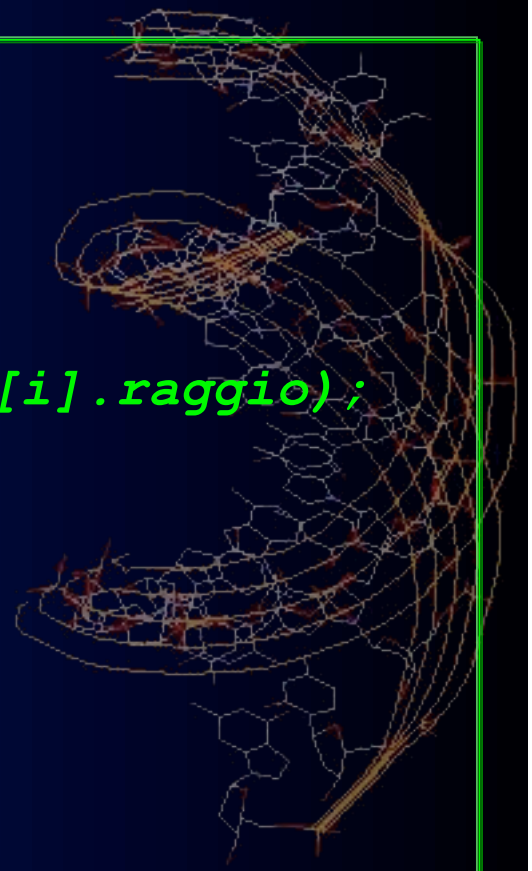


# Array di strutture

```
#include<stdio.h>
struct Circle {
    float raggio;
    float area;
};
float CircleArea(float Radius);

main()
{
   int i = -1, n;
   struct Circle circle[100];
   printf("\n Enter 0 to QUIT.");
. . . .
```

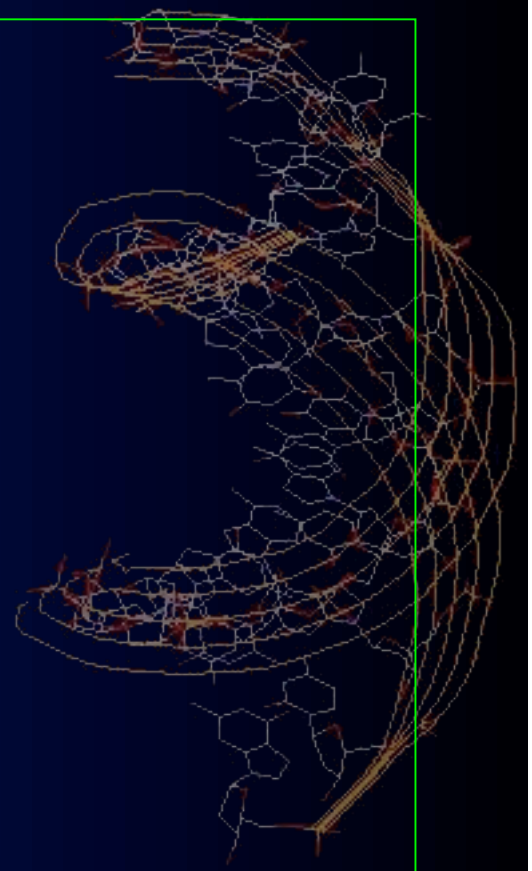


```
  do{
      i++;
      printf("\n Radius ? ");
      scanf("%f", &circle[i].raggio);
      circle[i].area = CircleArea(circle[i].raggio);
  } while(circle[i].raggio != 0)
  n = --i;
  for(i = 0; i < n; i++){
      printf("\n Radius =%f, Area = %f",
                          circle[i].raggio,
                          circle[i].area);
  }
}
```



# Puntatori & Strutture (I)

```
#include <stdio.h>
struct point {
   double x;
   double y;
};
main()
{
  struct point A1, A2, *pA;
  A1.x = 10.3;
  A1.y = 10.6;
  A2.x = 8.3;
  A2.y = 3.6;
  pA = &A1;
  printf("\n(x,y) = (%lf, %lf)", (*pA).x,(*pA).y);
}
```



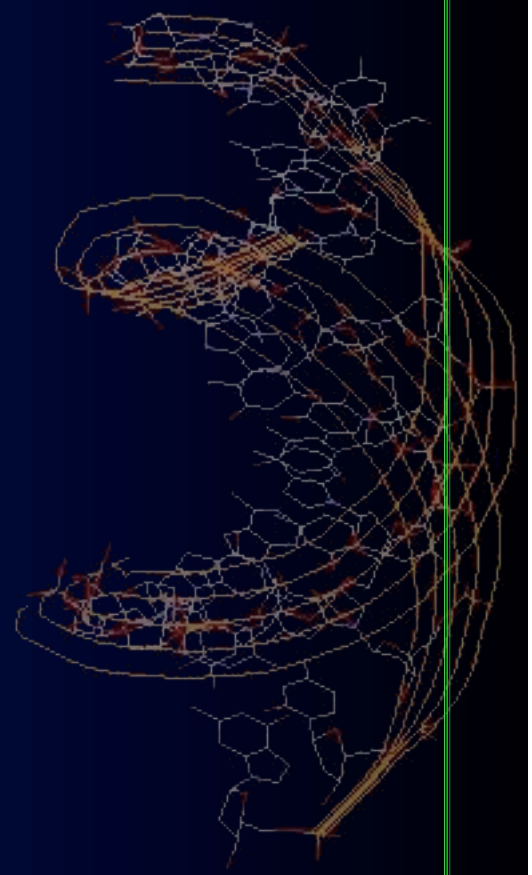
# Puntatori & Strutture (II)

```
#include <stdio.h>
struct point {
   double x;
   double y;
};
main()
{
  struct point A1, A2, *pA;
  A1.x = 10.3;
  A1.y = 10.6;
  A2.x = 8.3;
  A2.y = 3.6;
  pA = &A1;
  printf("\n(x,y) = (%lf, %lf)", pA->x, pA->y);
}
```

*operatore "freccia"*

# Puntatori & Strutture (III)

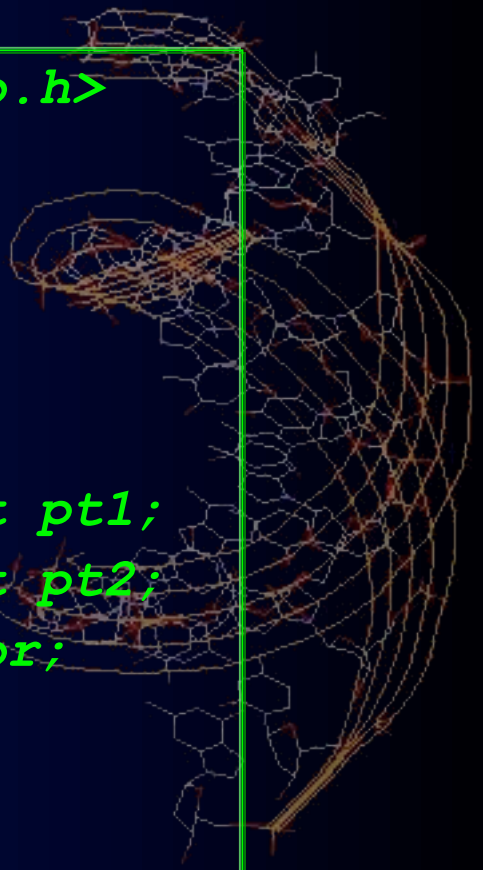
```
#include <stdio.h>
struct point {
   double x;
   double y;
};
main()
{
  struct point A1, A2, *pA;
  pA = &A1;
  pA->x = 10.3;
  pA->y = 10.6;
  pA = &A2;
  pA->x = 8.3;
  pA->y = 3.6;
  printf("\n(x,y) = (%lf, %lf)", pA->x, pA->y);
}
```



# Puntatori & Strutture (IV)

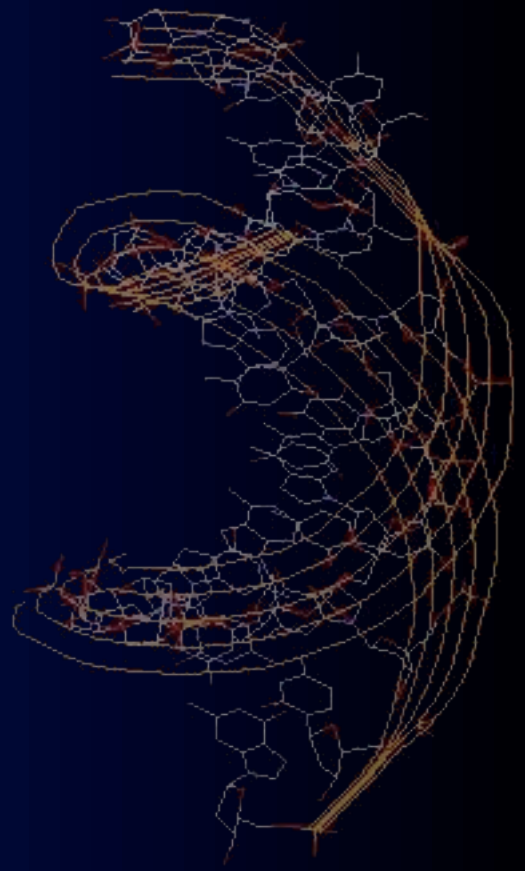
Annidiamo due strutture: definiamo la struttura "**point**" e la struttura "**rect**".

```
#include <stdio.h>
struct point {
    double x;
    double y;
};
struct rect {
    struct point pt1;
    struct point pt2;
    int FillColor;
};
main()
{
. . . . . .
```



# Puntatori & Strutture (V)

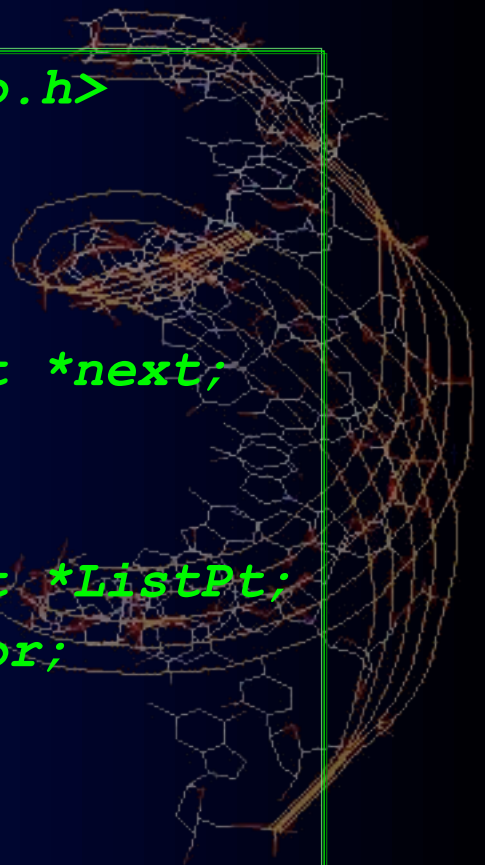
```
main()
{
   struct rect R, *pR;
   pR = &R;
   R.pt1.x = 10.3;
   R.pt1.y = 10.3;

   pR->pt2.x = 20.3;
   pR->pt2.y = 30.5;

. . .
```



# Puntatori & Strutture (VI)

Una struttura può contare tra i suoi campi puntatori a qualsiasi tipo di dato. Sono perciò ammessi anche puntatori a struttura, persino *puntatori a struttura identificata dal medesimo tag*.

```
#include <stdio.h>
struct point {
    double x;
    double y;
    struct point *next;
};
struct rect {
    struct point *ListPt;
    int FillColor;
};
main()
{
. . . . . .
```



# Programmazione ricorsiva

Il C permette di scrivere funzioni che operino in forma ricorsiva, ovvero che chiamino sé stesse. In generale una funzione di tipo ricorsivo avrà un **caso base,** di semplice valutazione, a cui chiamata dopo chiamata si ricondurrà la parte ricorsiva propriamente detta.

```
int fattoriale(int n){
 if(n==0 || n==1)
      return 1;
 else
      return(n*fattoriale(n-1));
}
```

Caso base

Ricorsione

# Ricorsività: pro e contro

I vantaggi della programmazione ricorsiva risiedono principalmente nell'eleganza e leggibilità dell'algoritmo, nonché nella similitudine col modo di pensare logico-matematico per induzione.

Va però ricordato che siccome ogni chiamata (attivazione) della funzione necessita di una copia di tutto lo spazio di memoria della funzione stessa, la ricorsione è in genere poco efficiente nell'uso della risorsa spazio. Inoltre una soluzione di tipo ricorsivo può essere ricondotta ad una di tipo iterativo.