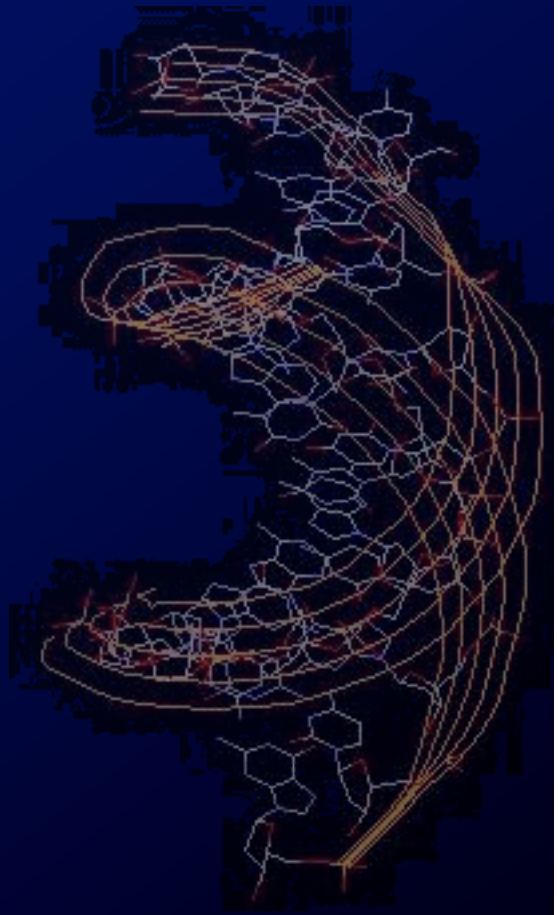

Utilizzo dei puntatori

Perche' i Puntatori ?

I puntatori hanno tre importanti applicazioni:

1. Passaggio di argomenti a funzioni;
2. Allocazione dinamica della memoria;
2. Organizzazione delle Strutture dati (liste, alberi, etc..);

Functions



Una funzione e' un segmento di programma che assolve un compito elaborativo circoscritto e piu' facilmente identificabile.

Le funzioni sono utilizzate per dividere un programma in un insieme di sottoprogrammi (modularizzazione).

Nello C l'uso delle funzioni è semplice ed efficiente.
Infatti i programmi in C sono tipicamente articolati in molte funzioni elementari piuttosto che da poche funzioni molto articolate.

Una funzione è un blocco di codice a sé stante ed isolato dal resto del programma.
Riceve dati e fornisce un risultato: ciò che avviene al suo interno è sconosciuto alla rimanente parte del programma, con la quale l'unica interazione e' il passaggio di dati di ingresso e uscita.

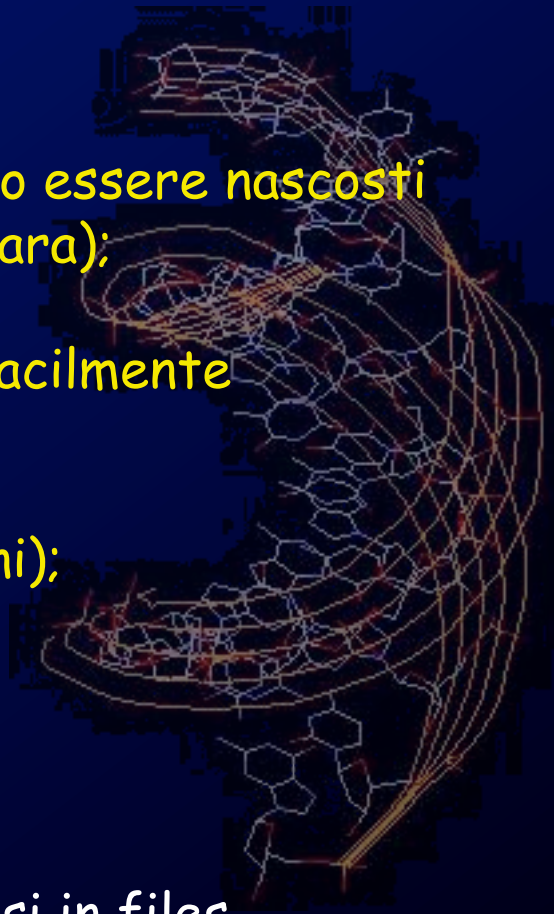
La modularizzazione favorisce:

1. la comprensibilita' del codice (i dettagli possono essere nascosti e l'organizzazione logica quindi ne risulta piu' chiara);

2. le sue possibili modifiche (circoscritte e piu' facilmente localizzabili);

3. la riutilizzazione del codice (librerie di funzioni);

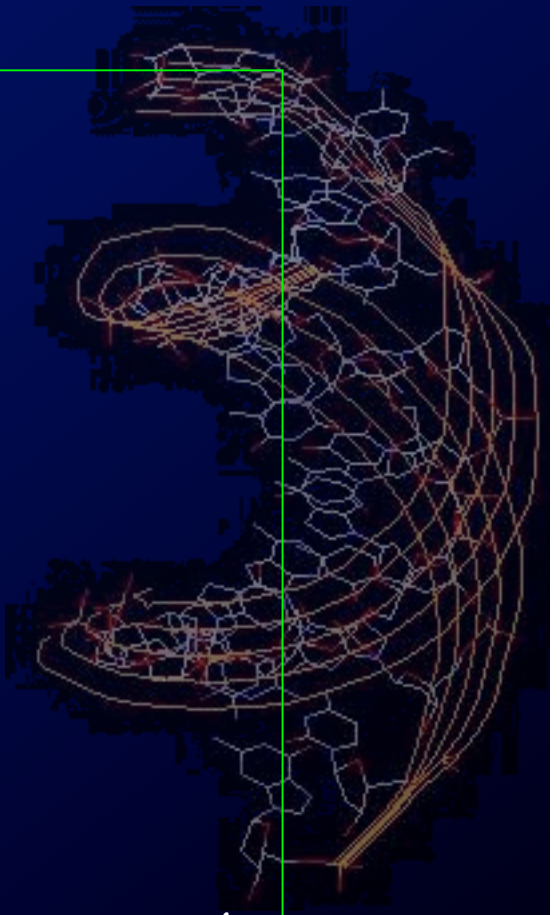
4. la localizzazione degli errori (debugging)



Inoltre il programma e le function possono trovarsi in files diversi, compilabili separatamente.

Esempio

```
#include <stdio.h>
/* prototipi delle funzioni*/
int Somma(int a, int b);

main()
{
/*Dichiarazione variabili e costanti*/
int a,b,somma;

printf("\n Inserire due numeri interi:");
scanf("%d %d", &a, &b);
somma = Somma(a,b);
printf("La somma e' %d",somma);
/*Fine blocco principale e uscita dal programma*/
}
```



. . .definizione della funzione

intestazione (header)

argomenti o parametri <u>formali</u>: NON devono essere ridefiniti nel corpo della funzione

```
int Somma(int a, int b)
{
    int somma; //variabili locali
    somma=a+b;
    return somma;
}
```

corpo

<u>IMPORTANTE</u>: Tutte le variabili create nel corpo della funzione sono locali, nel senso che sono create al momento della dichiarazione e distrutte quando la funzione termina: non sono riconosciute all'esterno della funzione stessa.

Una funzione esegue il proprio compito ogni qualvolta viene indirizzata ("chiamata" o "invocata") da qualche altra porzione del programma.

Dopo il completamento del compito la funzione ed il suo stack di dati viene cancellato dalla memoria ed il flusso del programma continua dall'istruzione successiva a quella della chiamata.

Parametri Formali e Reali

```
#include <stdio.h>
/* prototipi delle funzioni*/
int Somma(int a, int b);

main()
{
/*Dichiarazione variabili e costanti*/
int a,b,somma;

printf("\n Inserire due numeri interi:");
scanf("%d %d", &a, &b);
somma = Somma(a,b);
printf("La somma e' %d",somma);
/*Fine blocco principale e uscita dal programma*/
}
```

dichiarazione della funzione: prototipo. Notare che basta indicare il tipo delle variabili

parametri reali: "actual parameters"

Formalmente:

```
Tipo_ritornato Nome_Funzione(Lista_Parametri_Formali)
{  // inizio Corpo_Funzione
   Blocco di istruzioni;
   return [espressione];
}
```

Il tipo di dato dell'espressione ritornata al programma chiamante deve essere dello stesso tipo di dato definito nell'header.

Siccome solo una espressione puo' essere inclusa nella istruzione **`return`**, una funzione puo' ritornare solo un valore. Questa limitazione e' superata dall'uso dei puntatori.

Passaggi dei parametri

Passaggio di parametri

Perche' questa distinzione fra parametri formali e reali?

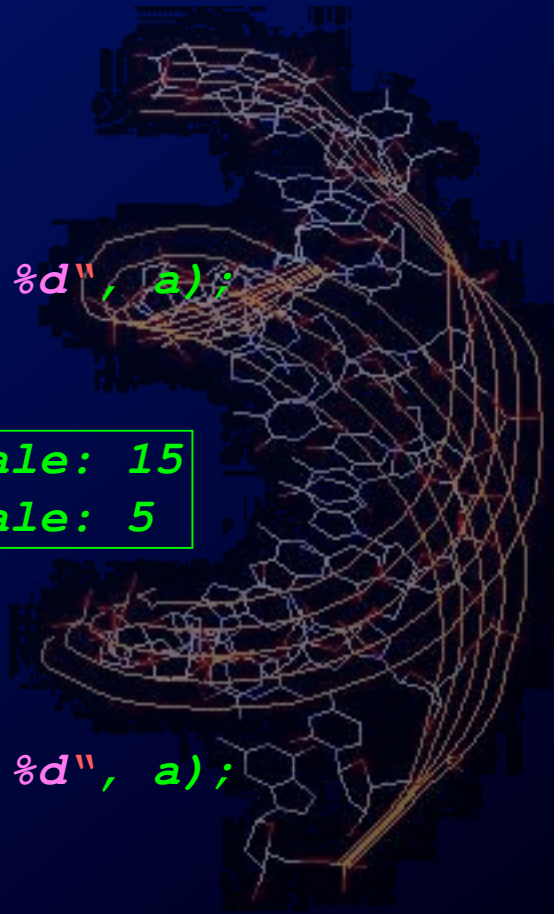
Quando un singolo valore e' passato ad un funzione attraverso un argomento reale, il valore dell'argomento reale e' *copiato* nell'argomento formale della funzione: solo il valore di un argomento formale puo' quindi essere alterato all'interno della funzione.

Questo meccanismo per il passaggio di parametri e' detto passaggio per valore (passing by value).

Passing by value: esempio

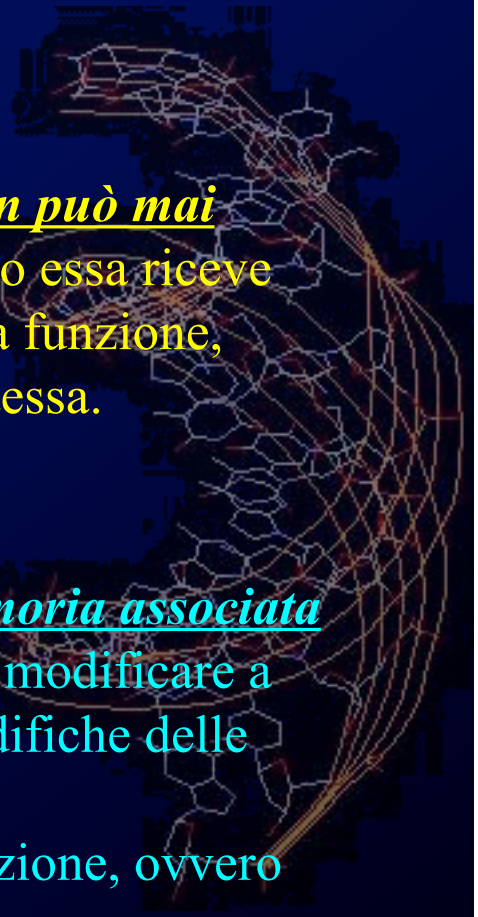
```
#include <stdio.h>
void Modify(int a)
{
  a += 10;
  printf("\n Valore del Parametro Formale: %d", a);
}

main()
{
  int a = 5;

  Modify(a);
  printf("\n Valore del Parametro attuale: %d", a);
}
```

```
Valore del Parametro Formale: 15
Valore del Parametro attuale: 5
```



Notate che la funzione non ritorna un valore (void) ed il return non e' specificato. Manca inoltre il prototipo poiche' il corpo della funzione e' inserito prima dell'utilizzo della funzione stessa.

Passing by value



E' estremamente importante ricordare che una ***funzione non può mai modificare i parametri attuali*** che le sono passati, in quanto essa riceve una copia degli stessi. Quando si passa una variabile ad una funzione, viene passato il suo valore, cioè una copia della variabile stessa.

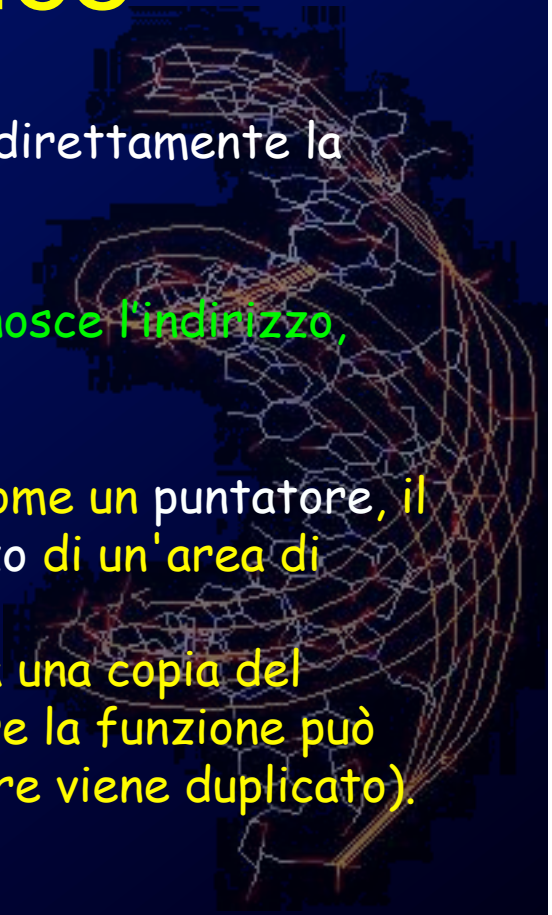
La funzione chiamata, perciò, ***non accede all'area di memoria associata alla variabile***, ma alla sua copia privata: essa può dunque modificare a piacere i parametri ricevuti senza il rischio di causare modifiche delle variabili della funzione chiamante.
Le copie dei parametri attuali sono, inoltre, locali alla funzione, ovvero sono create e distrutte al termine della funzione.

Passing by reference

Ma esiste un meccanismo per far accedere all funzione direttamente la variabile senza farne una copia??

Una funzione puo' accedere una variabile reale se ne conosce l'indirizzo, ovvero il **puntatore** !

Se allora definisco parametro formale di una funzione come un puntatore, il parametro attuale corrispondente rappresenta l'indirizzo di un'area di memoria:
Dunque coerentemente, alla funzione chiamata è passata una copia del puntatore, ma tramite l'indirizzo contenuto nel puntatore la funzione può accedere all'area di memoria "originale", (solo il puntatore viene duplicato).

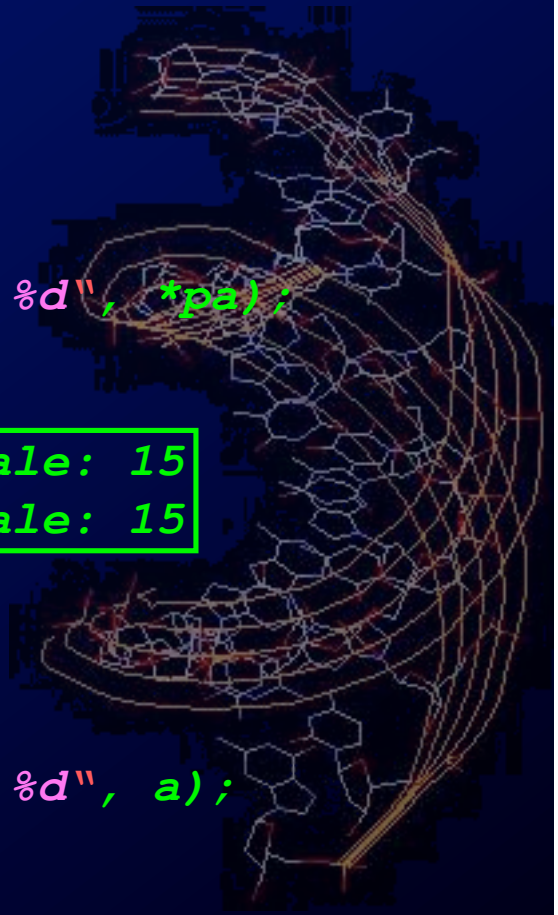


Questo meccanismo per il passaggio di parametri e' detto passaggio per riferimento (passing by reference).

Passing by reference: esempio

```
#include <stdio.h>
void Modify(int *pa)
{
  *pa += 10;
  printf("\n Valore del Parametro Formale: %d", *pa);
}

main()
{
  int a = 5;

  Modify(&a);
  printf("\n Valore del Parametro attuale: %d", a);
}
```

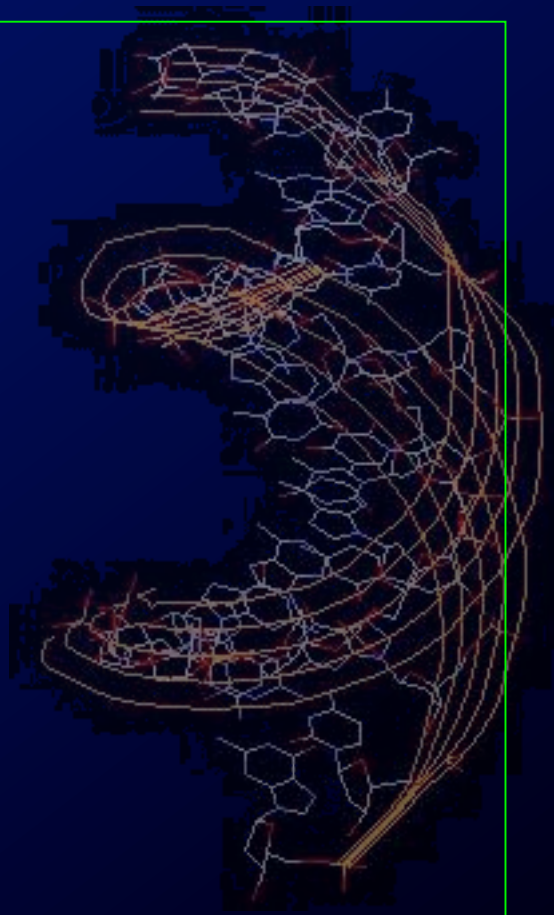
```
Valore del Parametro Formale: 15
Valore del Parametro attuale: 15
```



Notate la diversa definizione della funzione ed il passaggio come argomento dell'indirizzo di a utilizzando l'operatore &.

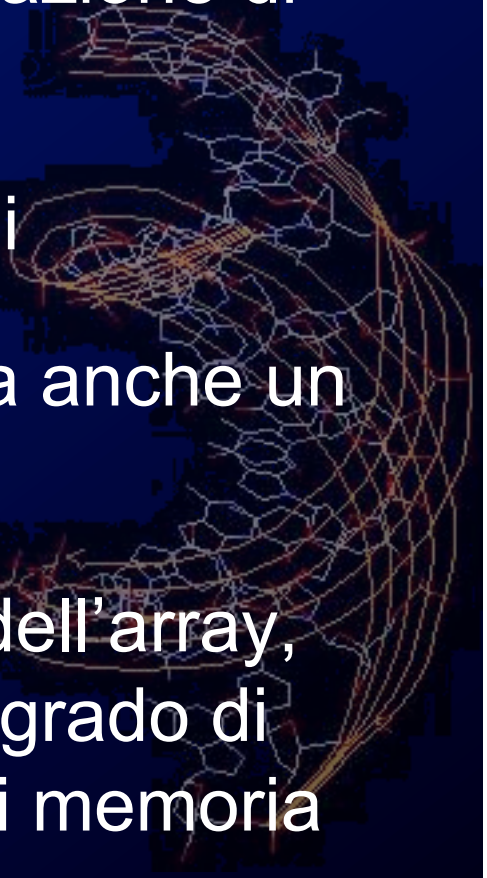
swapping: scambio

```
#include <stdio.h>
void swap(int *pa, int *pb)
{
   int temp;

   temp = *pa;
   *pa = *pb;
   *pb = temp;
}

main()
{
   int a = 5, b = 8;
   printf("\n Prima:    a = %d b = %d", a, b);

   swap(&a, &b);
   printf("\n Dopo:     a = %d b = %d", a, b);
}
```



Puntatori ed Array

- Un puntatore è un indirizzo ad una locazione di memoria o ad una serie di locazioni
- Un array è proprio un serie di locazioni
- ma allora posso passare per referenza anche un array.....
- Se infatti conosco la prima locazione dell'array, conosco il tipo e la lunghezza sono in grado di scorrerlo senza sforare in altre zone di memoria

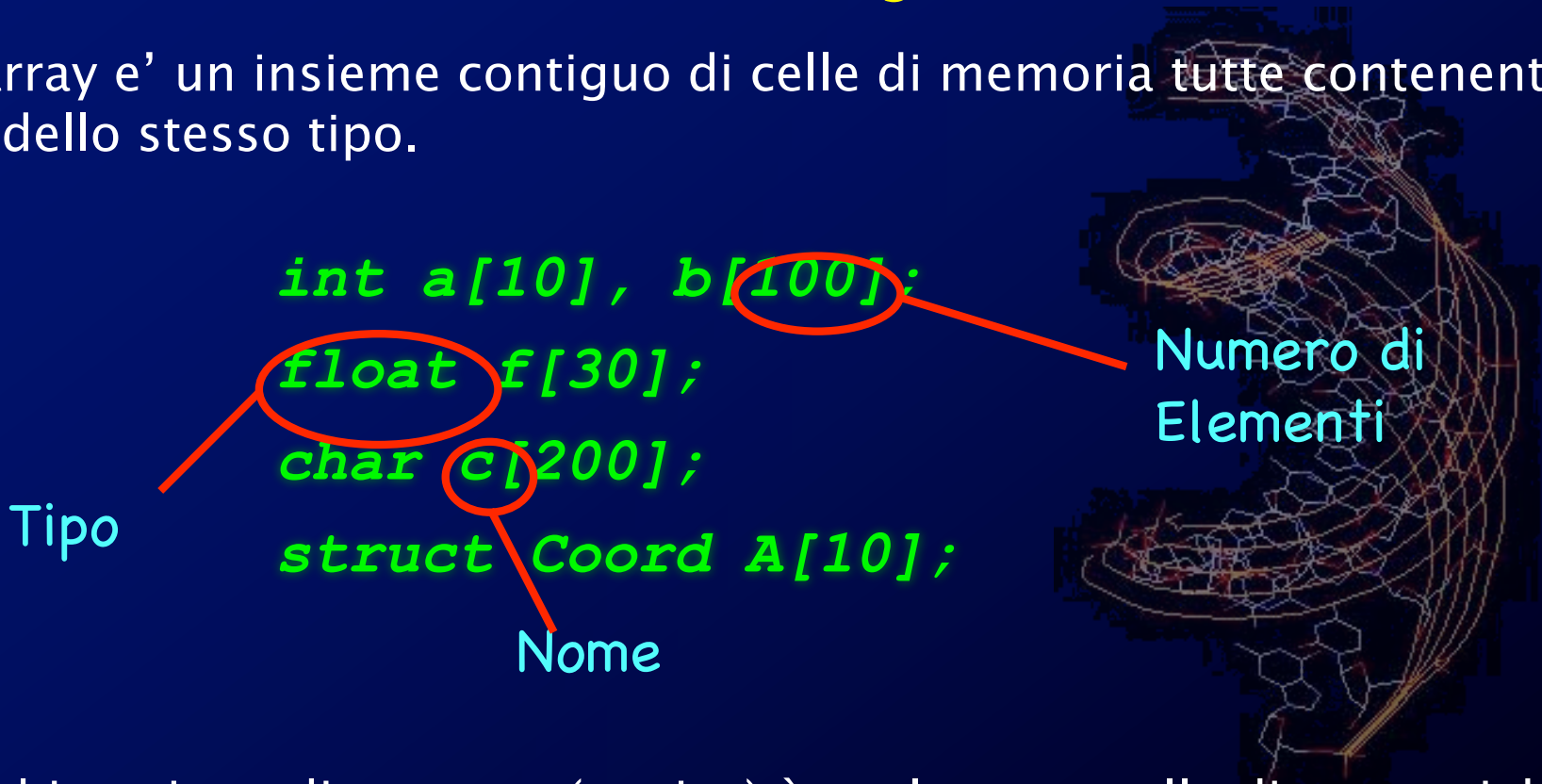


Array

- Finora abbiamo visto solo variabili semplici
- Spesso però abbiamo bisogno di un insieme di variabili simili.
- Ad esempio un vettore a 3 componenti
- Si parla in questo caso di ARRAY di variabili
- Questo è in pratica un gruppo di variabili tutte con lo stesso nome ma accessibili attraverso un indice.

Array

Un array e' un insieme contiguo di celle di memoria tutte contenenti dati dello stesso tipo.



La dichiarazione di un array (statico) è analoga a quella di una variabile ad eccezione del fatto che il nome dell'array è seguito dal numero di elementi che lo compongono, racchiuso tra parentesi quadre.

In C sono possibili due categorie di array:

Array Statici: sono quelli la cui dimensione viene definita all'interno del programma nella dichiarazione.

```
int a[10], mat[10][20];
float f[30];
struct Coord A[10];
```

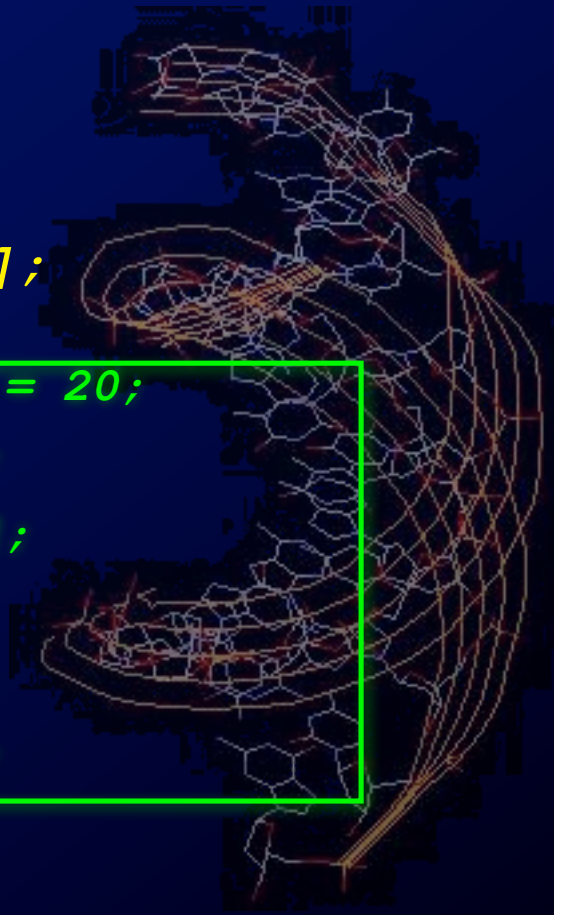
Array Dinamici: sono quelli la cui dimensione viene determinata, ed eventualmente anche modificata, solo durante l'esecuzione del programma (in run-time).

Dichiarazione di array statici:

```
const int Dim, Dim1,. . DimN;
TipoDato nome_array[Dim][Dim1]..[DimN];
```

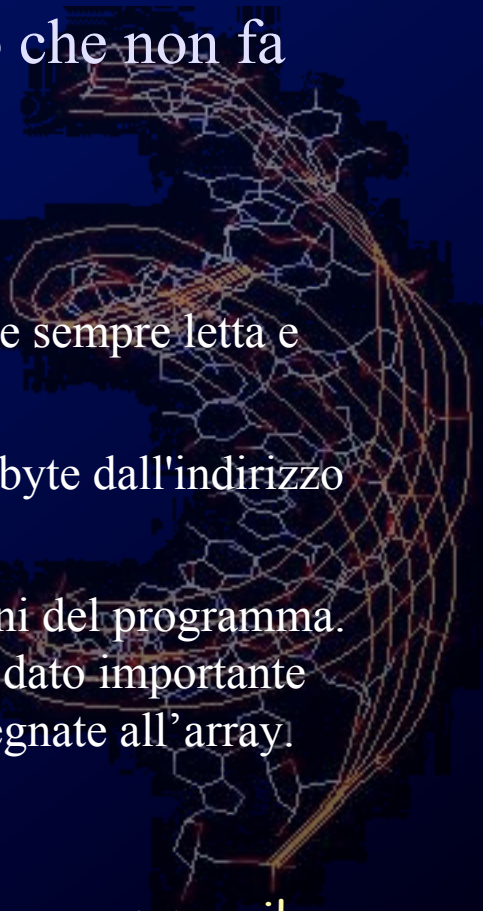
Esempi di dichiarazioni →

```
const int Dim1 = 5, Dim2 = 20;
int a[Dim1], mat[10][20];
unsigned int Coeff[3][10];
float f[Dim2];
struct Coord A[10];
int B[] = {10, 20, 2, 8};
```



Gli array unidimensionali sono anche detti vettori;
gli array bidimensionali sono detti matrici.

Cosa accade se si tenta di referenziare un elemento che non fa parte dell'array, ad esempio `a[10]` ?



Il compilatore non dà errore: la memoria relativa ad a[10] può essere sempre letta e scritta, ma non fa parte dell'array!!

Per il compilatore, a[10] è semplicemente la word che si trova a 40 byte dall'indirizzo al quale l'array è memorizzato.

In caso di lettura, il valore letto non ha alcun significato logico ai fini del programma. In caso di scrittura, invece, si rischia di sovrascrivere qualche altro dato importante che risiede nella locazione di memoria subito dopo le locazioni assegnate all'array.

Se la locazione di memoria referenziata non appartiene al programma, il programma stesso termina con un errore (segmentation fault).

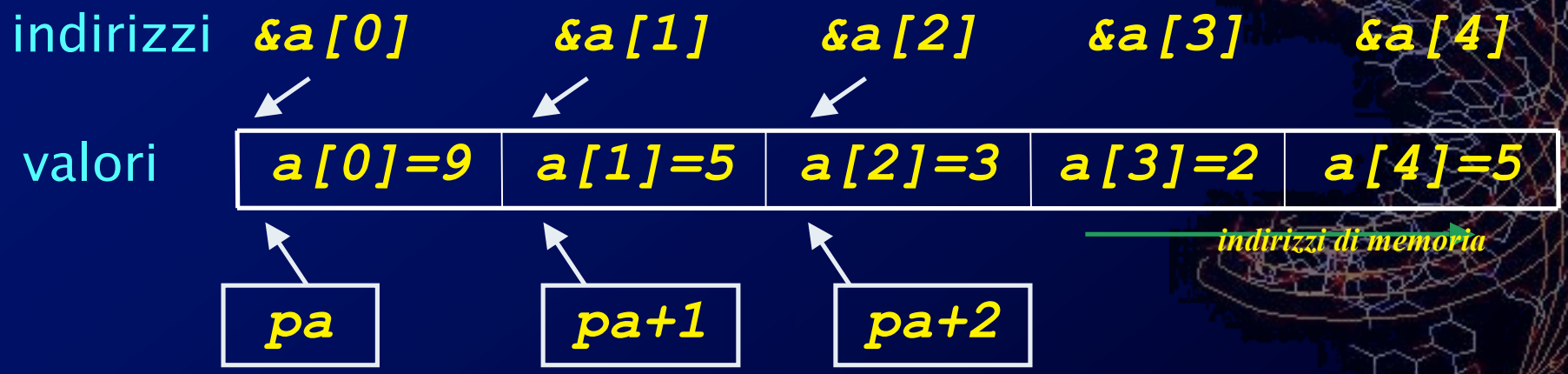
Tutti gli elementi di un array, come qualsiasi altro oggetto in memoria, hanno un indirizzo scelto dal compilatore.

Il programmatore non può modificarlo, ma può conoscerlo ?

In C, il nome di un array e' una variabile che contiene l'indirizzo del primo elemento dell'array stesso, ovvero e' un puntatore alla prima locazione di memoria assegnata all'array.

```
int *pa, a[5];
. . . . .
pa = a; /* corretto */
pa = &a; /* equivalente */
```

```
int *pa, a[]= {9, 5, 3, 2, 5};
pa = a;  /* corretto */
pa = &a; /* equivalente */
```

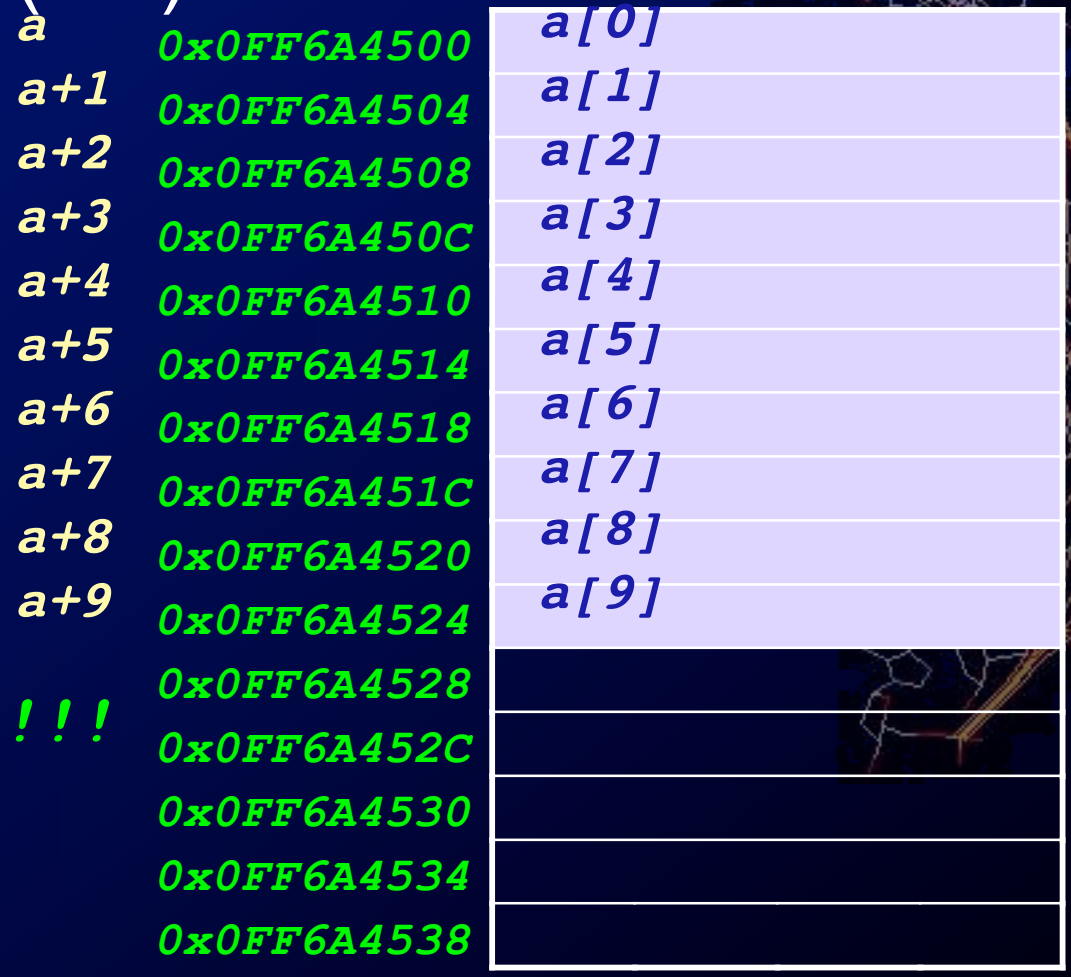


Se ***pa*** punta al primo elemento dell'array, ***pa+1*** punta a quello successivo: ***pa+n*** punta all'n-esimo elemento dopo il primo, indipendentemente dal tipo o dimensione delle variabili nell'array.

Artimetica dei puntatori

- Cosa significa *(a+i)?

```
int a[10];

a==0x0FF6A4500

a+1==0x0FF6A4504

a+1 != 0x0FF6A4501 !!!
```



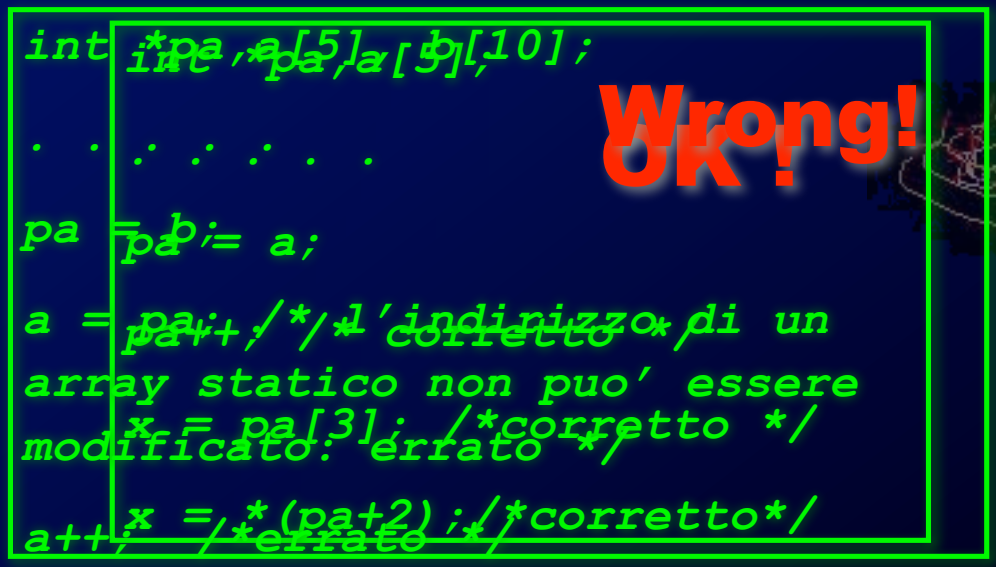
Dereferenziando pa:

`*pa`	*ha valore a[0]*	⇔	`*a`	*ha valore a[0]*
`*(pa+1)`	*ha valore a[1]*		`*(a+1)`	*ha valore a[1]*
`.....`					
`*(pa+n)`	*ha valore a[n]*		`*(a+n)`	*ha valore a[n]*

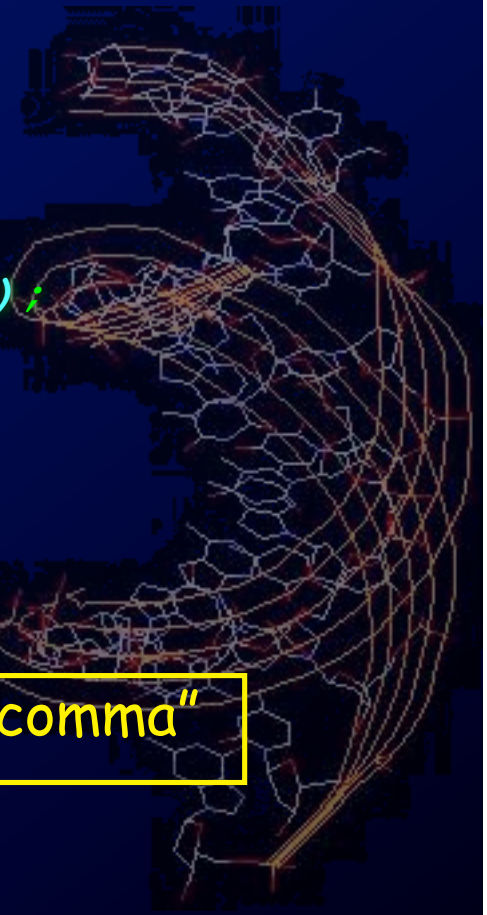
L'indice di un elemento di un array ne esprime l'*offset* dal primo elemento dell'array stesso:
il primo elemento di un array ha offset 0 rispetto a se stesso;
il secondo ha offset 1 rispetto al primo;
il terzo ha offset 2, cioè dista 2 elementi dal primo...
Il compilatore "ragiona" sugli arrays in termini di elementi, e non di byte.

Attenzione: rimane comunque una differenza sostanziale tra nome dell'array e puntatore:

il valore di un puntatore puo' variare, il valore del puntatore dichiarato come array non puo' variare.

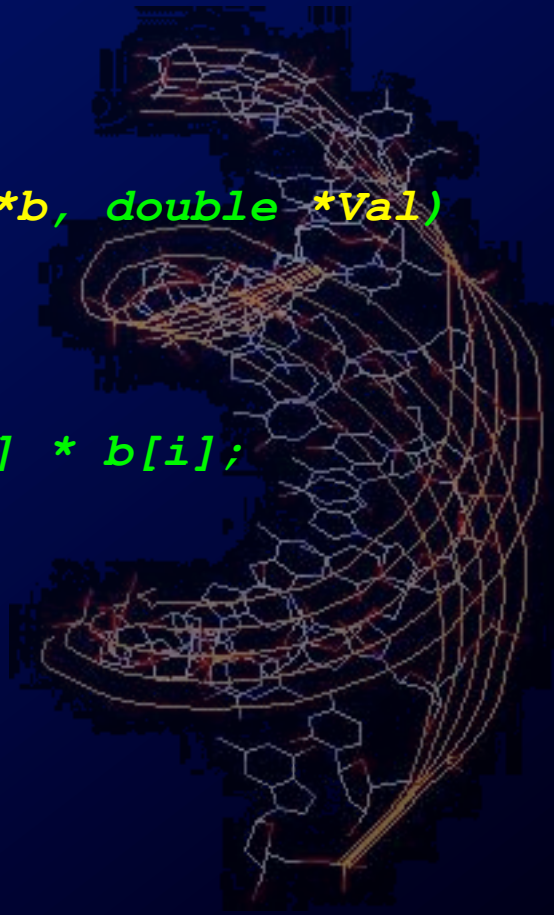


```
main(){
    int i, *pa, a[]= {8,12,56,3,9};

    printf("indirizzo di a: 0x%X\n", a);
    printf("pa punta all'indirizzo: 0x%X\n", pa);
    for(i = 0, pa = a; i < 5; i++, pa++){
        printf("a[%d]= %d\n", i, *pa);
    }
}
```



Osservazione: l'accesso agli elementi di un array mediante puntatori produce un eseguibile che e' in generale piu' veloce.

Passare un puntatore e' passare un . . . .array



```
#include <stdio.h>
void ProdScalare(int Dim, double *a, double *b, double *Val)
{
  int i;
  *Val = 0.;
  for(i = 0; i < Dim; i++) *Val = *Val + a[i] * b[i];
}

main()
{
  double pippo[] = {3.5, 5.4, 8.4};
  double pluto[] = {6.4, 1.2, 4.5}, Val;

  ProdScalare(3, pippo, pluto, &Val);
  printf("\n Il Valore del Prodotto Scalare e': %lf", Val);
}
```

ed ancora . . . .

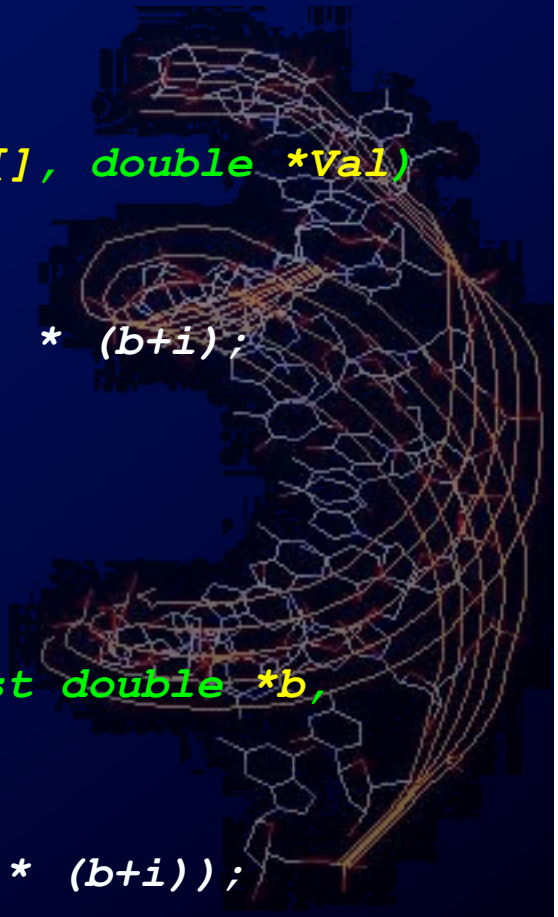
```
#include <stdio.h>
void ProdScalare(int Dim, double a[], double b[], double *Val)
{
  *Val = 0.;
  for(i = 0; i < Dim; i++) *Val = *Val + a[i] * b[i];
}
```

```
#include <stdio.h>
void ProdScalare(int Dim, const double *a, const double *b,
                 double *Val)
{
  *Val = 0.;
  for(i = 0; i < Dim; i++) *Val = *Val + a[i] * b[i];
}
```

ed ancora . . . .

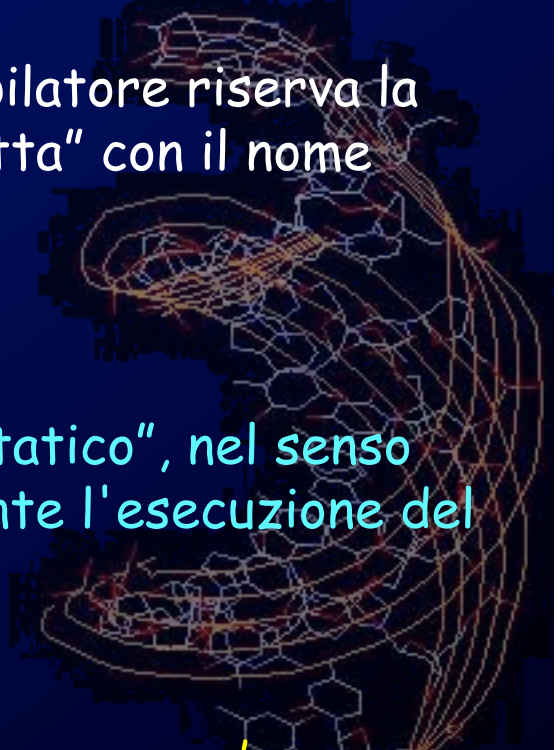
```
#include <stdio.h>
void ProdScalare(int Dim, double a[], double b[], double *Val)
{
  *Val = 0.;
  for(i = 0; i < Dim; i++) *Val = *Val + *(a+i) * (b+i);
}
```

```
#include <stdio.h>
void ProdScalare(int Dim, const double *a, const double *b,
                 double *Val)
{
  for(i = 0,*Val = 0.; i < Dim; *Val += *(a+i) * (b+i));
}
```



Allocazione dinamica della memoria

Quando una qualunque variabile è dichiarata, il compilatore riserva la quantità di memoria ad essa necessaria e le "etichetta" con il nome scelto dal programmatore:



in questi casi la memoria e' allocata in modo "statico", nel senso che quanto dichiarato non è modificabile durante l'esecuzione del programma stesso;

Ad esempio, il numero di elementi di un array statico non puo' essere modificato dopo la sua dichiarazione. Cio' rende l'uso della memoria inefficiente nei casi in cui il numero degli elementi dell'array non puo' essere noto a priori.

Allocazione dinamica della memoria

Questa rigidita' nell'uso della memoria e' superata in C mediante la funzione malloc():

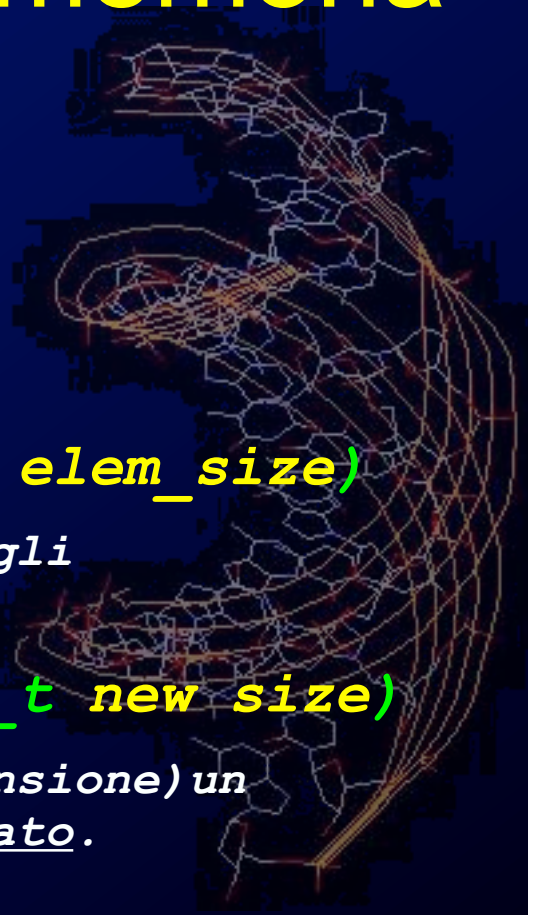
```
void *malloc(size_t numero_di_bytes)
```

La funzione **`malloc()`** permette l'allocazione dinamica della memoria: ritorna il puntatore al primo byte dell'area di memoria richiesta e riservata dal sistema operativo al processo richiedente.

Quando questa memoria non e' piu' utile al processo, la si deve restituire al sistema operativo ("liberare") mediante la chiamata all funzione:

```
void free(void *address)
```

Le funzioni per gestire la memoria



```
#include <malloc.h>
```

```
void *malloc(size_t num_bytes)
```

Alloca un blocco di memoria.

```
void *calloc(size_t num_elems, size_t elem_size)
```

Alloca un vettore ed inizializza tutti gli elementi a zero.

```
void *realloc(void *mem_address, size_t new size)
```

Alloca nuovamente(modificandone la dimensione)un blocco di memoria precedentemente allocato.

```
void free(void *mem_address)
```

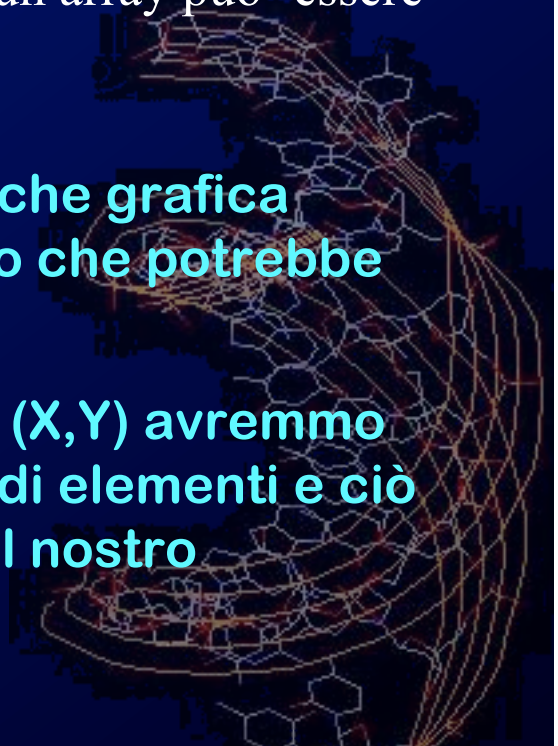
Libera un blocco di memoria.

Es: Uso della funzione *malloc*

```
#include <stdio.h>
#include <stdlib.h>

main()

{

  int *p;

/* Riserva una zona di memoria da contenere un intero.
L'indirizzo iniziale di questa zona viene restituito e
memorizzato nella variabile puntatore p.*/

  p = (int *)malloc(sizeof(int));

  if( p == NULL ){ //verifica

     printf ("Error: Memoria non disponibile \n");

     exit(1);

  }

  *p = 10;

  printf("L'indirizzo di p e':%x\n *p = %d\n", p, *p);

  free(p); //libera la memoria allocata
```

Array Dinamici

Nella maggior parte dei casi pratici, il numero di elementi di un array puo' essere determinato solo durante l'esecuzione del programma.

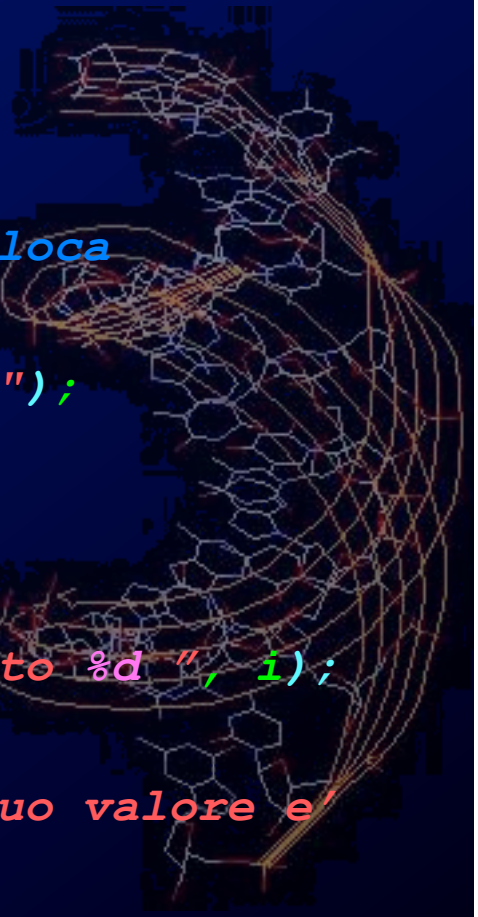


Es: Supponiamo di voler scrivere un programma che grafica automaticamente le coppie (X,Y) . Il primo numero che potrebbe essere dato in input è il numero di punti.

Usando gli array statici per conservare le coppie (X,Y) avremmo bisogno di decidere a priori un numero massimo di elementi e ciò implicherebbe dei grossi limiti all'applicabilità del nostro programma.

La funzione `malloc()` ci consente di decidere la dimensione dell'array durante l'esecuzione del programma (in run-time). Gli array le cui locazioni di memoria sono allocate in run-time sono detti pertanto dinamici.

```
main()
{
   int *p, dim,i;
   printf("\n Numero di elementi dell'array: ");
   scanf("%d", &dim);
   p = (int *) malloc( dim * sizeof(int)); // alloca
   if(p == NULL){ //verifica
      printf ("Error: Memoria non disponibile \n");
      exit(1);
   }
   for(i = 0;i < dim; i++){
       printf("\n Inserire il valore dell'elemento %d ", i);
       scanf("%d", &p[i]);
       printf("L'indirizzo di p[%d] e':%x\n il suo valore e'
                       *p = %d\n", i, p+i, p[i]);
    }
    free(p); //libera la memoria allocata
}
```



Array Statici vs. Array Dinamici

Dal punto di vista dell'immagazzinamento di un insieme di dati

```
double a[10];
```

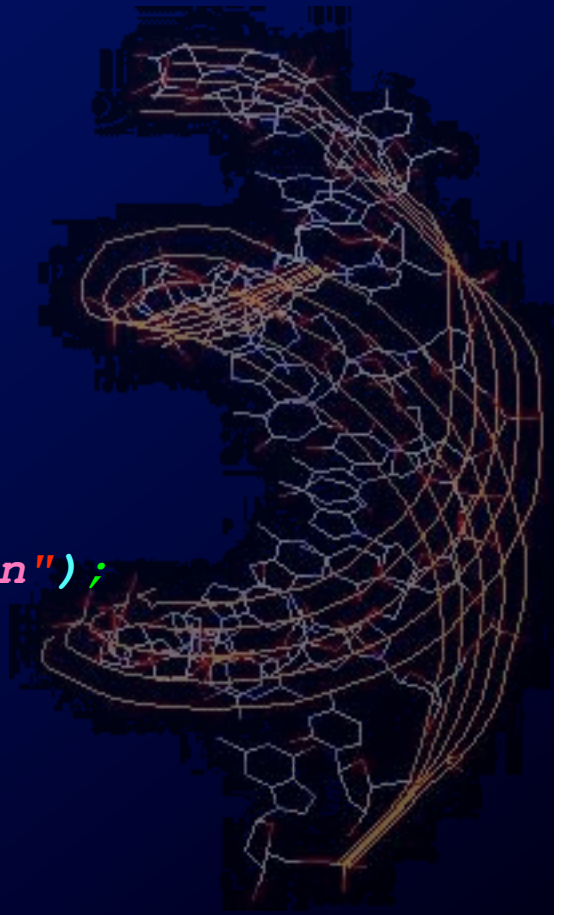
e' equivalente a

```
double *a;
a = (double *) malloc(10 * sizeof(double))
```

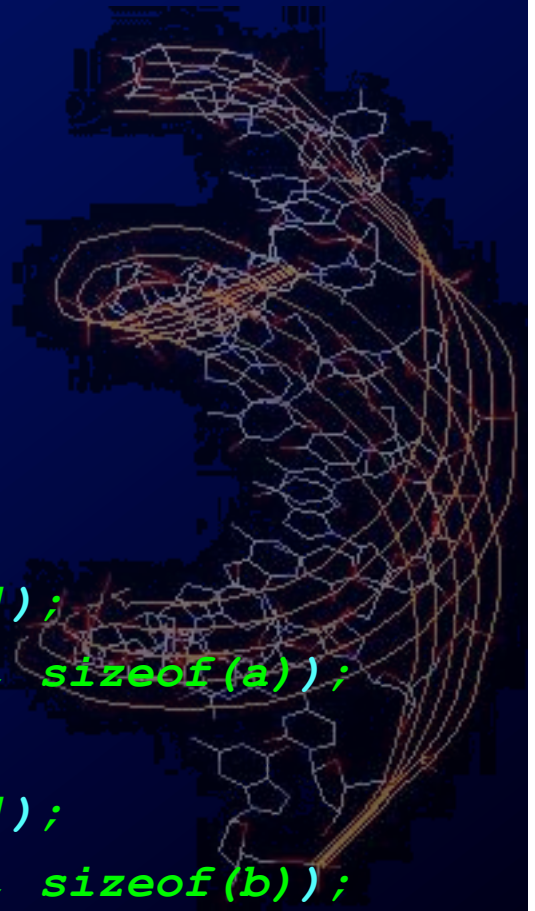
In entrambi i casi abbiamo definito un vettore di 10 elementi che possiamo individuare singolarmente tramite la notazione `a[0]` ecc. o `a[1]` ecc.

Esiste però una differenza... verificabile con il seguente esempio

```
void *MyMalloc(size_t nbytes)
{
    void *p = 0;
    p = malloc(nbytes);
    if(p == NULL){ //verifica
      printf("Error: Not enough memory \n");
      exit(1);
   }
   return p;
}
. . . . .
```



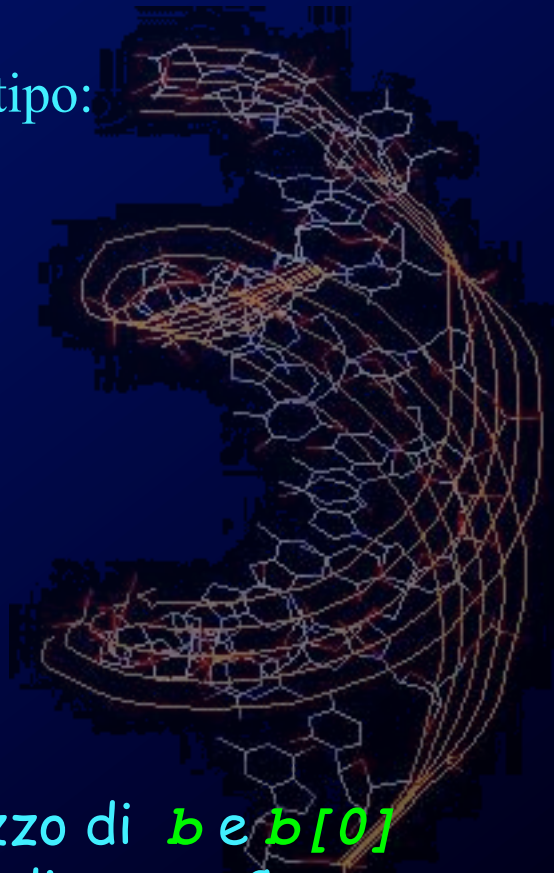
```
. . . .
main()
{
   const int dim = 10;
   int i, a[dim], *b;
   b = (int *)MyMalloc(dim*sizeof(int));
   printf("\n Indirizzo di a    : %x", &a);
   printf("\n Indirizzo di a[0]: %x", &a[0]);
   printf("\n Dimensione di a   : %d bytes", sizeof(a));
   printf("\n Indirizzo di b    : %x", &b);
   printf("\n Indirizzo di b[0]: %x", &b[0]);
   printf("\n Dimensione di b   : %d bytes", sizeof(b));
   free(b);//libera la memoria allocata
}
```



Esempio (III)

L'output del programma precedente potrebbe essere del tipo:

```
Indirizzo di a    : 0xBFFFF820
Indirizzo di a[0]: 0xBFFFF820
Dimensione di a   : 40 byte
Indirizzo di b    : 0xBFFFF864
Indirizzo di b[0]: 0x500120
Dimensione di b   : 4 byte
```



L'indirizzo di ***a*** e ***a[0]*** coincidono mentre l'indirizzo di ***b*** e ***b[0]*** sono diversi così come sono diverse le dimensioni di ***a*** e ***b***. Come mai ?