Corso di Informatica A.A. 2009- 2010

Lezione 12

Allocazione dinamica
della memoria

La funzione `calloc()`

La funzione `calloc()` alloca memoria dinamicamente.

```
void *calloc( size_t num, size_t size);
```

Numero di oggetti da allocare

Dimensione di ogni oggetto

Se l'allocazione ha successo la memoria allocata viene "pulita" e la funzione ritorna un puntatore al primo byte. In caso contrario la funzione ritorna `NULL`

La funzione calloc()

```
/* Demonstrates calloc(). */

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

int main()
{
    unsigned num;
    int *ptr;

     printf("Enter the number of type int to allocate: ");
     scanf("%d", &num);

     ptr = (int*)calloc(num, sizeof(int));

     if (ptr != NULL)
         puts("Memory allocation was successful.");
     else
         puts("Memory allocation failed.");
     return(0);
 }
```

La funzione `realloc()`

La funzione `realloc()` consente di modificare le dimensioni di un blocco di memoria precedentemente allocato.

```
void *realloc(void *ptr, size_t size);
```

Se esiste sufficiente spazio allora la memoria addizionale è allocata e la funzione ritorna `ptr`

Se non esiste sufficiente spazio per espandere il blocco di memoria nella sua posizione corrente, allora viene allocato un nuovo blocco di dimensione size e i dati sono copiati. Il vecchio blocco è liberato e la funzione ritorna un puntatore al nuovo blocco

La funzione `realloc()`

Se l'argomento `ptr` è NULL allora la funzione alloca un blocco di `size` bytes e ritorna un puntatore al blocco stesso (come `malloc()`)

Se l'argomento `size` vale 0 allora il blocco cui punta `ptr` viene liberato e la funzione ritorna NULL

Se la memoria è insufficiente per la riallocazione allora la funzione ritorna NULL e il blocco resta immutatao

La funzione `free()`

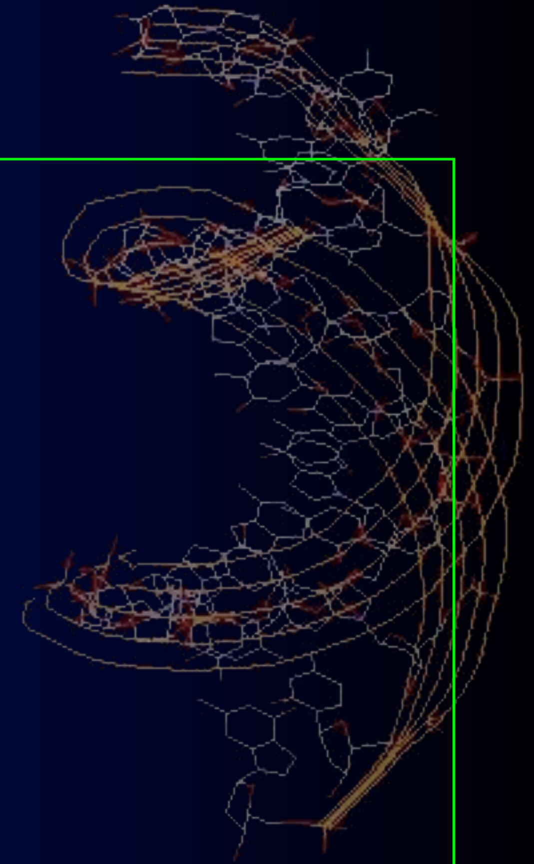
Quando il programma termina di utilizzare un blocco di memoria allocato dinamicamente è buona norma liberare tale blocco di memoria per renderlo disponibile per altri usi

```
void free(void *ptr);
```

La funzione `free()` libera il blocco di memoria cui punta ptr purché tale blocco sia stato allocato con `malloc()`, `calloc()` o `realloc()`

```c
/* Using free() to release allocated dynamic memory. */

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

#define BLOCKSIZE 30000

int main()
{
    void *ptr1, *ptr2;

    /* Allocate one block. */

    ptr1 = malloc(BLOCKSIZE);

    if (ptr1 != NULL)
        printf("\nFirst allocation of %d bytes successful.",BLOCKSIZE);
    else
    {
        printf("\nAttempt to allocate %d bytes failed.\n",BLOCKSIZE);
        exit;
    }

    /* Try to allocate another block. */

    ptr2 = malloc(BLOCKSIZE);

    if (ptr2 !=NULL)
    {
        /* If allocation successful, print message and exit. */

        printf("\nSecond allocation of %d bytes successful.\n",
               BLOCKSIZE);
        exit;
    }

    /* If not successful, free the first block and try again.*/

    printf("\nSecond attempt to allocate %d bytes failed.",BLOCKSIZE);
    free(ptr1);
    printf("\nFreeing first block.");

    ptr2 = malloc(BLOCKSIZE);

    if (ptr2 != NULL)
        printf("\nAfter free(), allocation of %d bytes successful.\n",
               BLOCKSIZE);
    return;
}
```

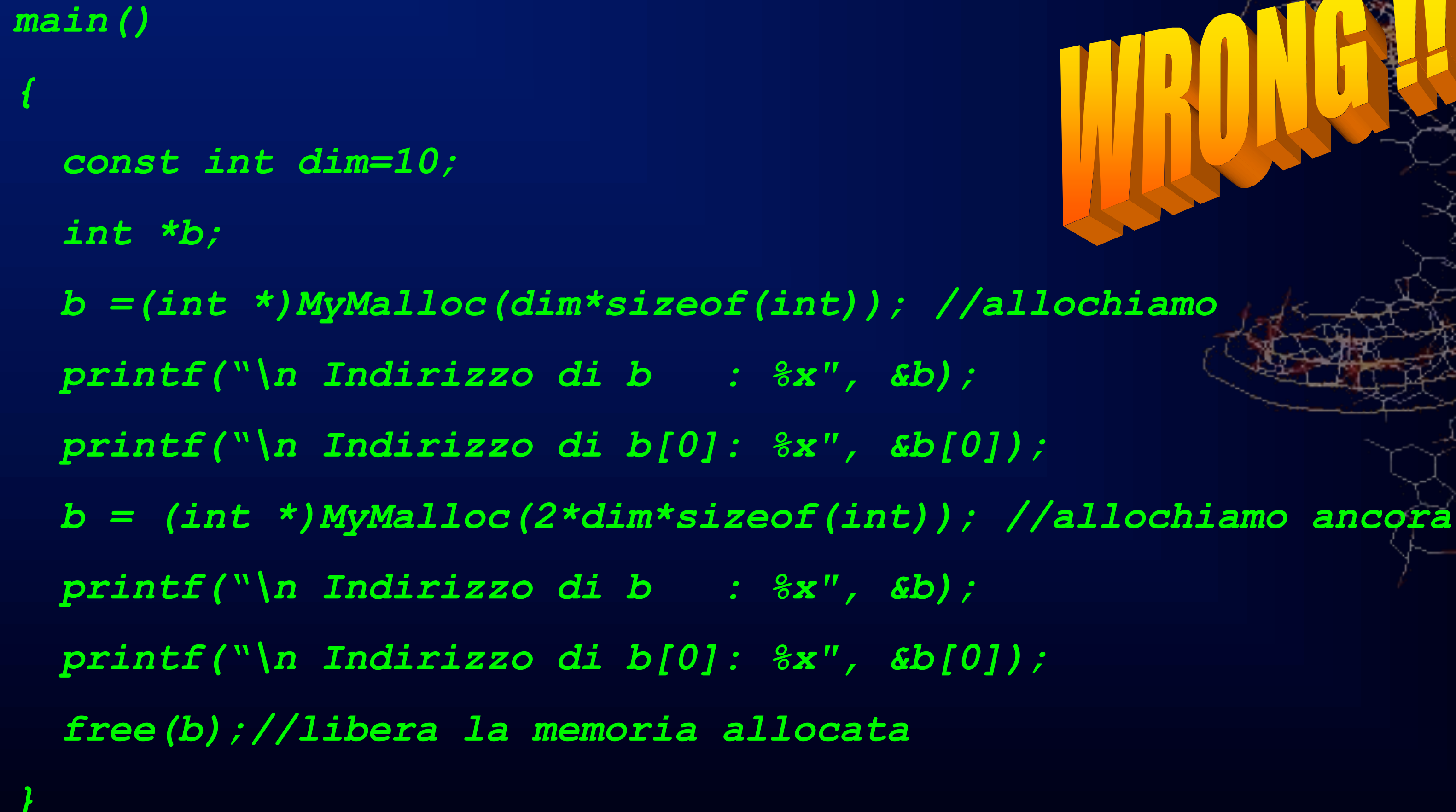
Esempio (I)



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

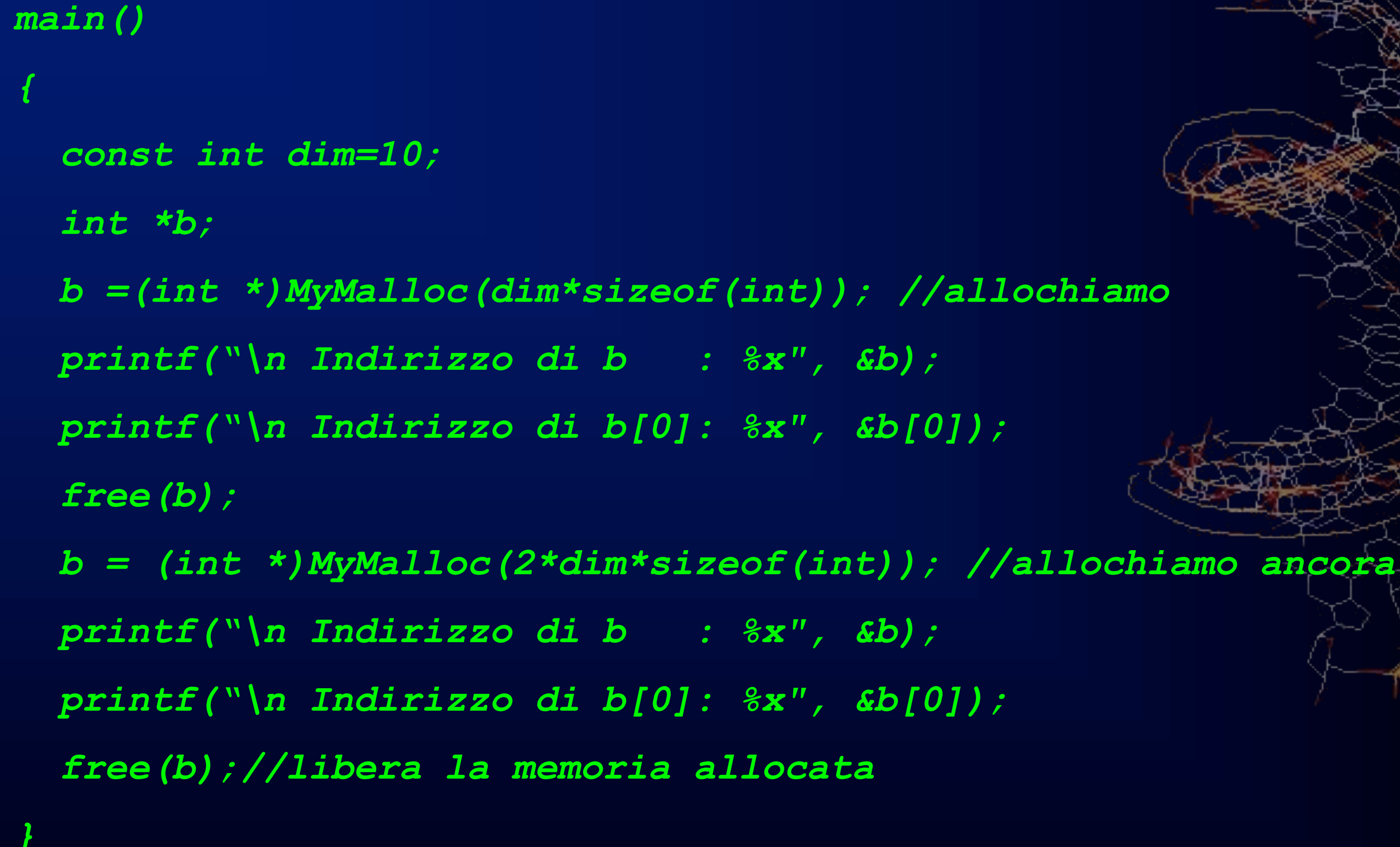
Memory Leakage

Se si “perde” il valore di un puntatore di una zona di memoria allocata, quella memoria non e’ piu’ utilizzabile dal processo e non e’ piu’ “liberabile”: rimane quindi proprieta’ del processo fino alla sua terminazione.




```
main()
{
  const int dim=10;
  int *b;
  b =(int *)MyMalloc(dim*sizeof(int)); //allochiamo
  printf("\n Indirizzo di b    : %x", &b);
  printf("\n Indirizzo di b[0]: %x", &b[0]);
  b = (int *)MyMalloc(2*dim*sizeof(int)); //allochiamo ancora
  printf("\n Indirizzo di b    : %x", &b);
  printf("\n Indirizzo di b[0]: %x", &b[0]);
  free(b);//libera la memoria allocata
}
```

Memory Leakage



```
main()
{
  const int dim=10;
  int *b;
  b =(int *)MyMalloc(dim*sizeof(int)); //allochiamo
  printf("\n Indirizzo di b   : %x", &b);
  printf("\n Indirizzo di b[0]: %x", &b[0]);
  free(b);
  b = (int *)MyMalloc(2*dim*sizeof(int)); //allochiamo ancora
  printf("\n Indirizzo di b   : %x", &b);
  printf("\n Indirizzo di b[0]: %x", &b[0]);
  free(b);//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];
```

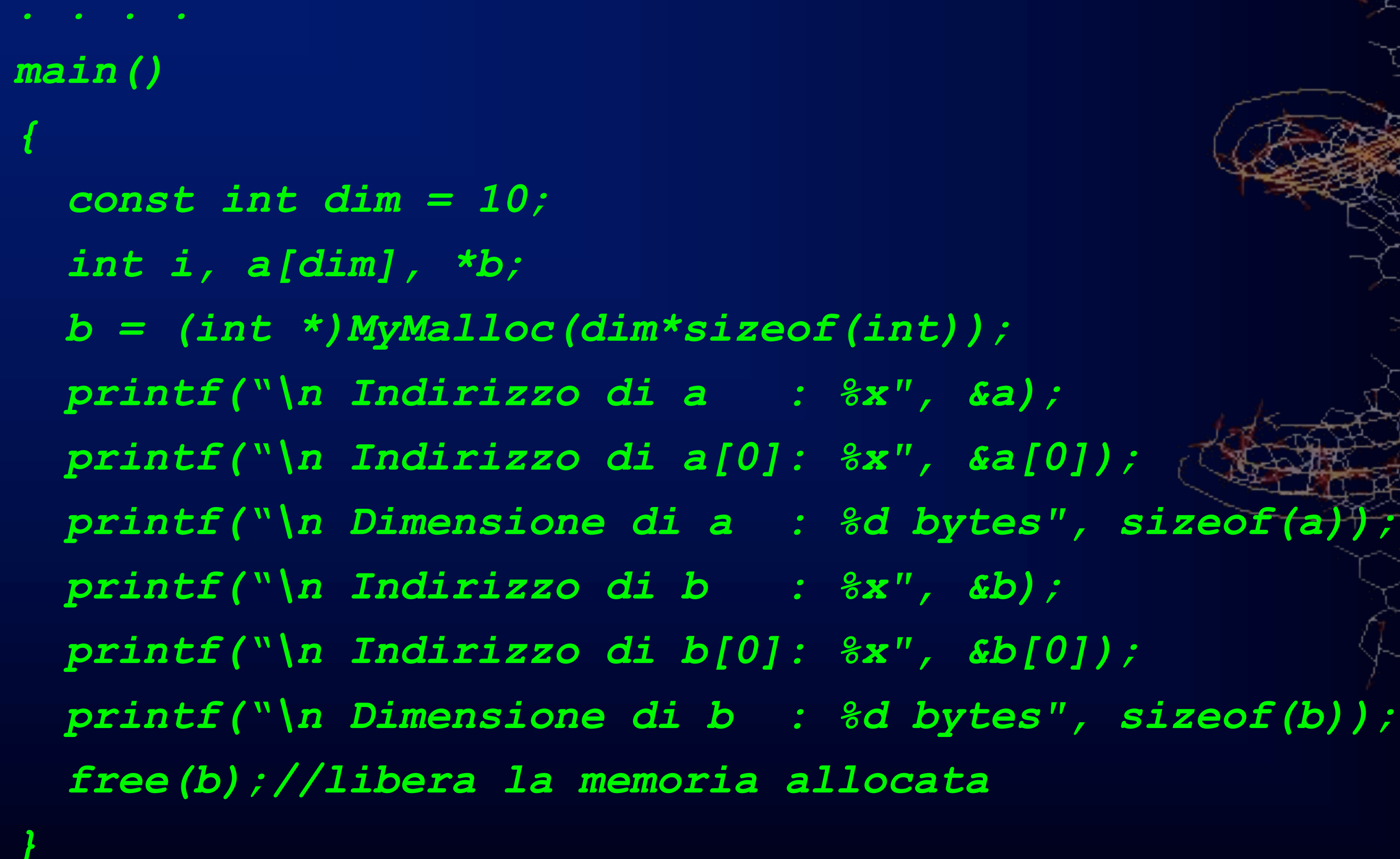
e' equivalente a

```
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

Esempio (II)

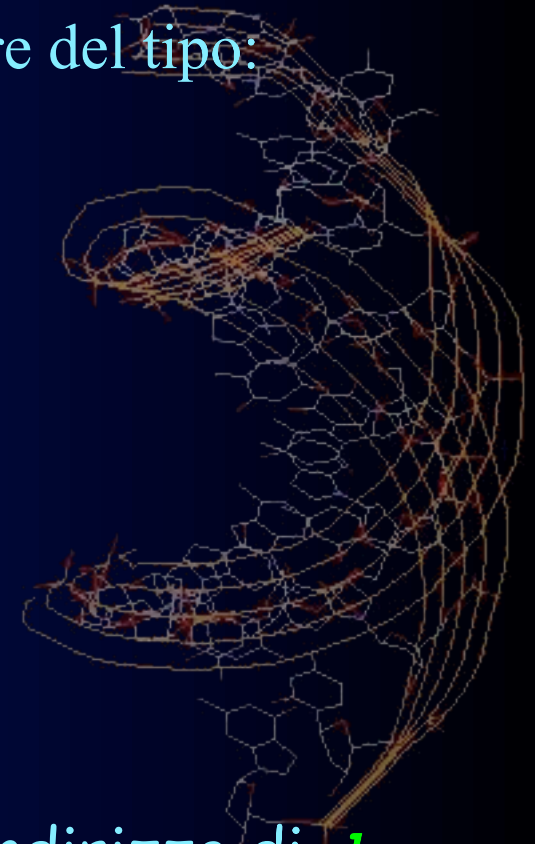


```
. . . .
main()
{
  const int dim = 10;
  int i, a[dim], *b;
  b = (int *)MyMalloc(dim*sizeof(int));
  printf("\n Indirizzo di a    : %x", &a);
  printf("\n Indirizzo di a[0]: %x", &a[0]);
  printf("\n Dimensione di a   : %d bytes", sizeof(a));
  printf("\n Indirizzo di b    : %x", &b);
  printf("\n 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    : 5FB9
Indirizzo di a[0]: 5FB9
Dimensione di a   : 40 byte
Indirizzo di b    : 9AC2
Indirizzo di b[0]: 2C84
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 ?

Array di Puntatori

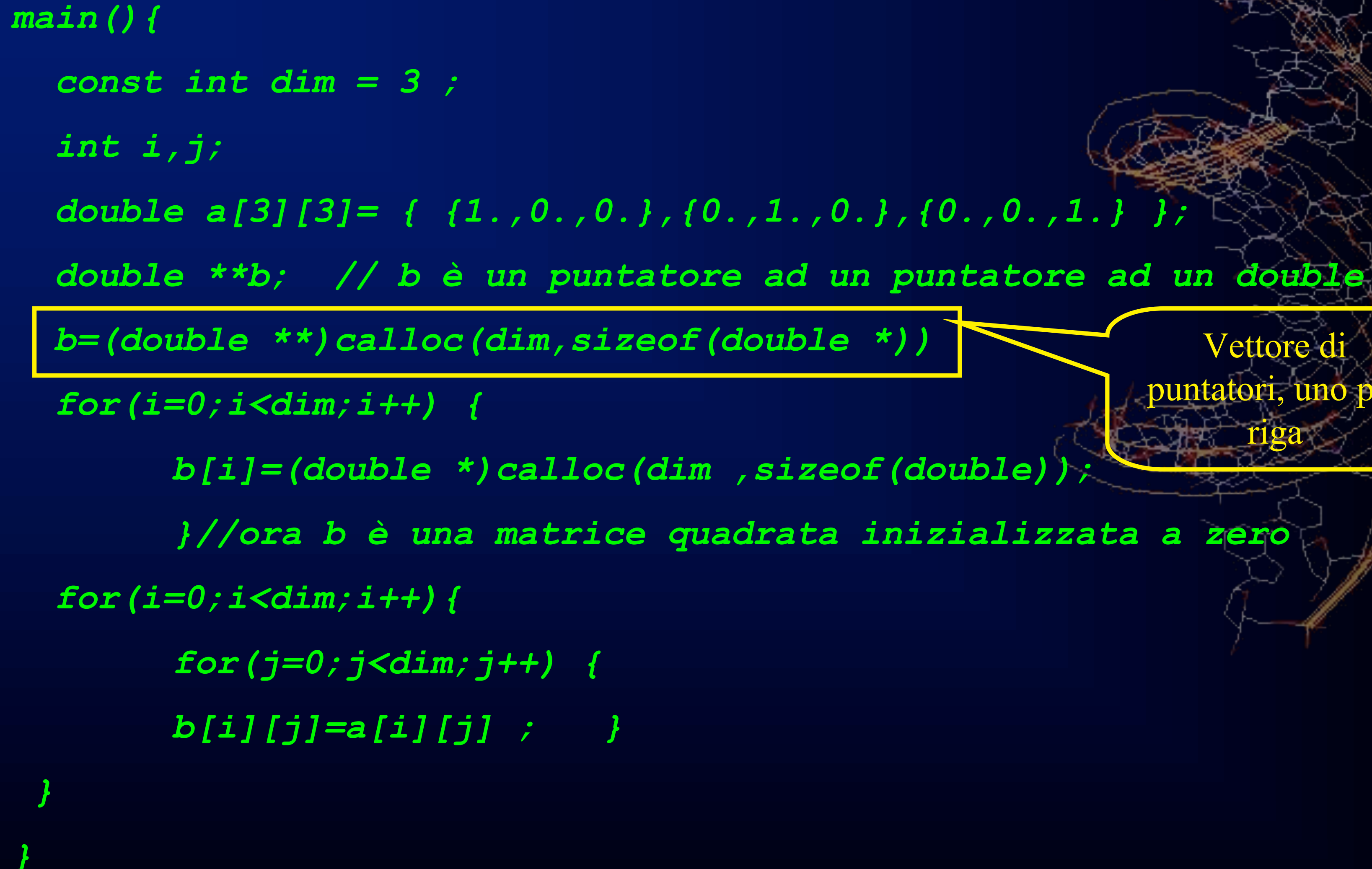
Una matrice bidimensionale può essere rappresentata come un array a due indici, o come un array di puntatori

```
main(){
  const int dim = 3 ;
  int i,j;
  double a[3][3]= { {1.,0.,0.},{0.,1.,0.},{0.,0.,1.} };
  double *b[dim];/* a è una matrice quadrata di double
        b è un array di puntatori */
  for(i=0;i<dim;i++) {
        b[i]=(double *)calloc(dim ,sizeof(double));
        }//ora b è una matrice quadrata inizializzata a zero
  for(i=0;i<dim;i++){
        for(j=0;j<dim;j++) {
        b[i][j]=a[i][j] ;   }
  }
}
```

Un vettore di double per ciascun puntatore

Puntatori a Puntatori

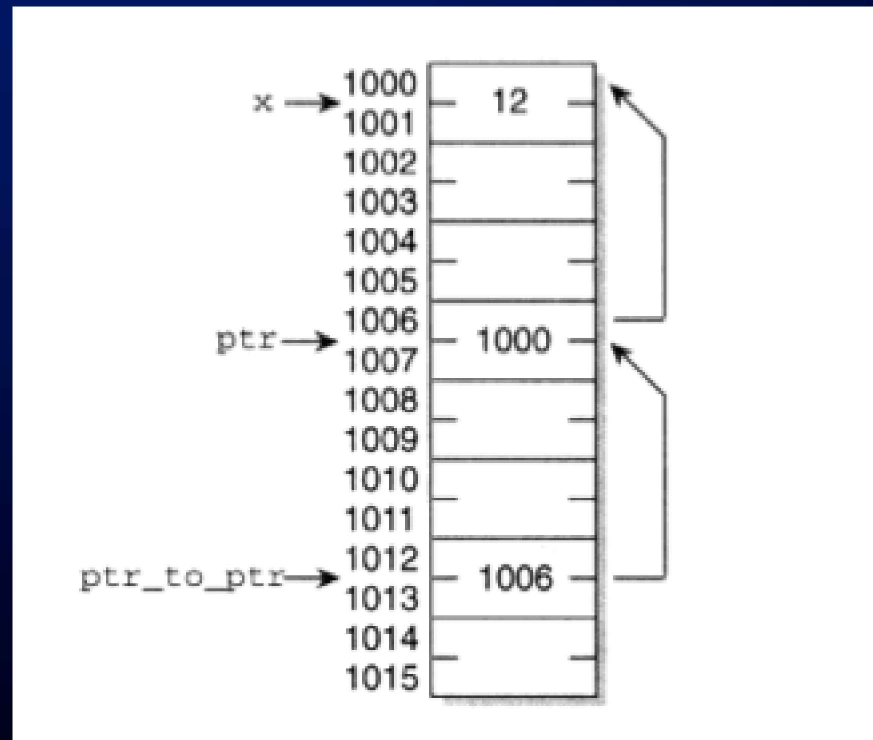
Un array di puntatori è…un puntatore a puntatori !



```
main(){
  const int dim = 3 ;
  int i,j;
  double a[3][3]= { {1.,0.,0.},{0.,1.,0.},{0.,0.,1.} };
  double **b;  // b è un puntatore ad un puntatore ad un double
  b=(double **)calloc(dim,sizeof(double *))
  for(i=0;i<dim;i++) {
        b[i]=(double *)calloc(dim ,sizeof(double));
        }//ora b è una matrice quadrata inizializzata a zero
  for(i=0;i<dim;i++){
        for(j=0;j<dim;j++) {
        b[i][j]=a[i][j] ;    }
  }
}
```

Puntatori a Puntatori

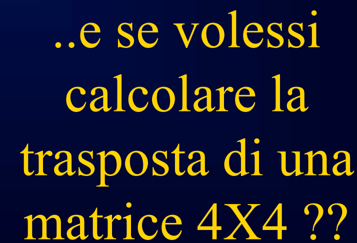
```
int x = 12;                    /* x is a type int variable. */
int *ptr = &x;                 /* ptr is a pointer to x. */
int **ptr_to_ptr = &ptr;       /* ptr_to_ptr is a pointer to a */
                               /* pointer to type int. */
```



Matrici come argomento di funzioni

I puntatori a puntatori sono spesso l'unico mezzo pratico di passare matrici come argomenti di funzioni. Ciò accade perché il compilatore ha bisogno che nel prototipo e nella definizione della funzione venga specificata ogni dimensione di un array successiva alla prima. Ad esempio volendo costruire una funzione per calcolare la trasposta si avrebbe:

```
void  trasponi3{double a[][3],double at[][3]) {
  const int dim = 3 ;
  int i,j;
  for(i=0;i<dim;i++){
        for(j=0;j<dim;j++) {
           at[i][j]=a[j][i]}
  }
}
```



Esempio

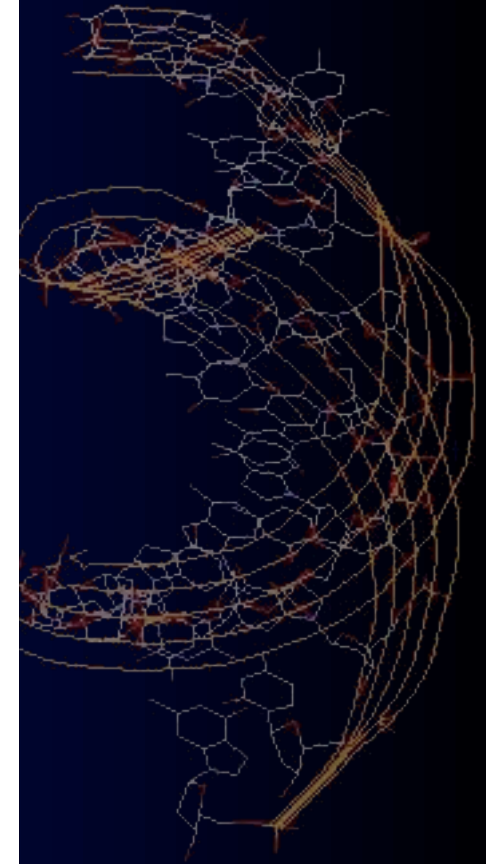
```
void  trasponi4(double a[][4],double at[][4]) {
  const int dim = 4 ;
  int i,j;
  for(i=0;i<dim;i++){
        for(j=0;j<dim;j++) {
           at[i][j]=a[j][i]}
  }
}
```

…oppure, per poter calcolare la trasposta di QUALSIASI matrice…

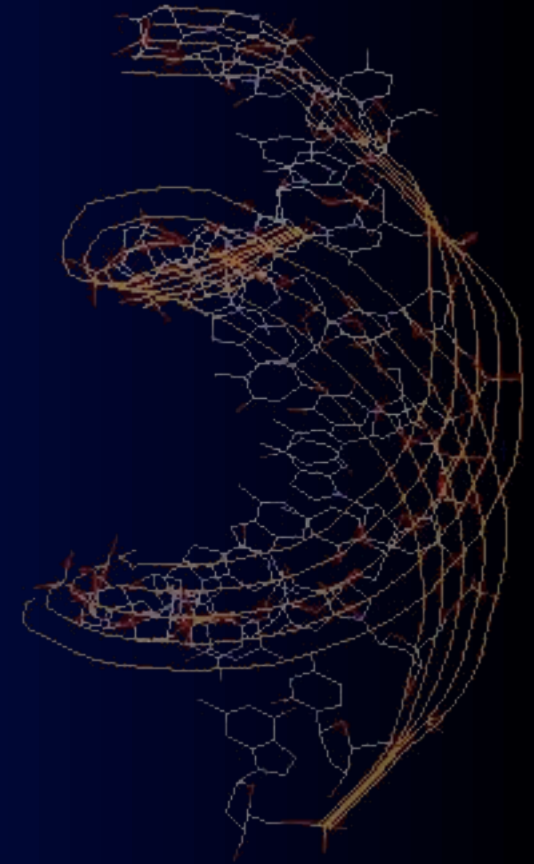
Esempio II

```
void  trasponi (double **a,double **at,int dim)
{
  /* a e at sono matrici >*dinamiche*< allocate altrove */
  int i,j;
  for(i=0;i<dim;i++){
        for(j=0;j<dim;j++) {
           at[i][j]=a[j][i]
        }
  }
}
```

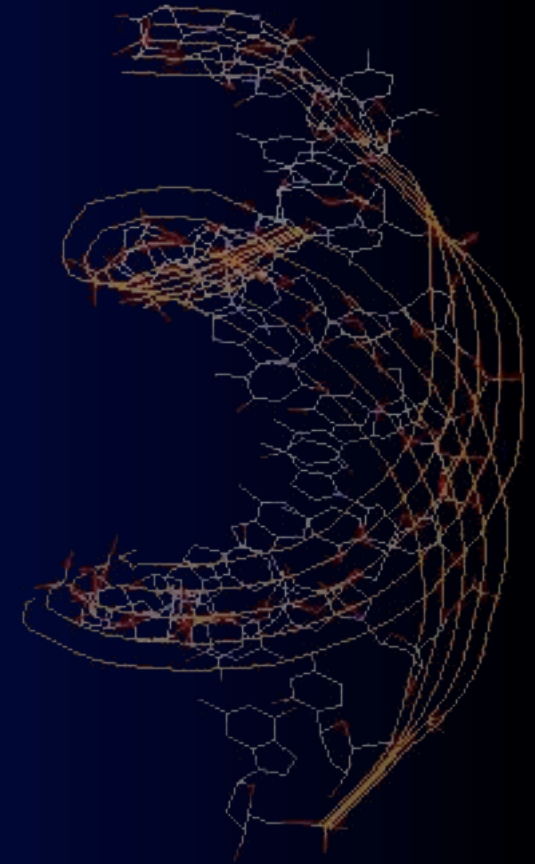
```
#include <stdio.h>

void printarray_1(int (*ptr)[4]);
void printarray_2(int (*ptr)[4], int n);

int main()
 {
     int multi[3][4] = { { 1, 2, 3, 4 },
                         { 5, 6, 7, 8 },
                         { 9, 10, 11, 12 } };

     /* ptr is a pointer to an array of 4 ints. */

     int (*ptr)[4], count;

     /* Set ptr to point to the first element of multi. */

     ptr = multi;

     /* With each loop, ptr is incremented to point to the next */
     /* element (that is, the next 4-
ment integer array) of multi. */

     for (count = 0; count < 3; count++)
         printarray_1(ptr++);

     puts("\n\nPress Enter . . . ");
     getchar();
     printarray_2(multi, 3);
     printf("\n");
     return(0);
 }
```



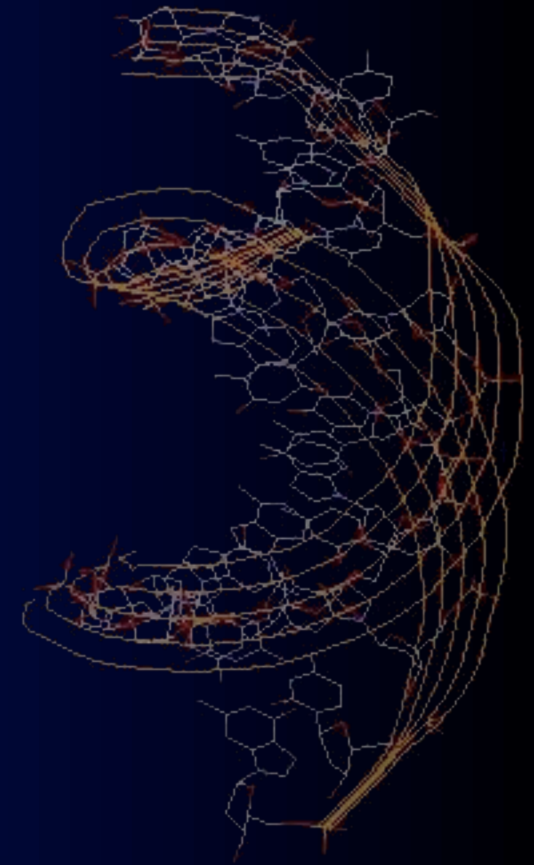
```
void printarray_1(int (*ptr)[4])
{
/* Prints the elements of a single four-element integer array. */
/* p is a pointer to type int. You must use a type cast */
/* to make p equal to the address in ptr. */

     int *p, count;
     p = (int *)ptr;

     for (count = 0; count < 4; count++)
         printf("\n%d", *p++);
}

void printarray_2(int (*ptr)[4], int n)
{
/* Prints the elements of an n by four-element integer array. */

     int *p, count;
     p = (int *)ptr;

     for (count = 0; count < (4 * n); count++)
         printf("\n%d", *p++);
}
```



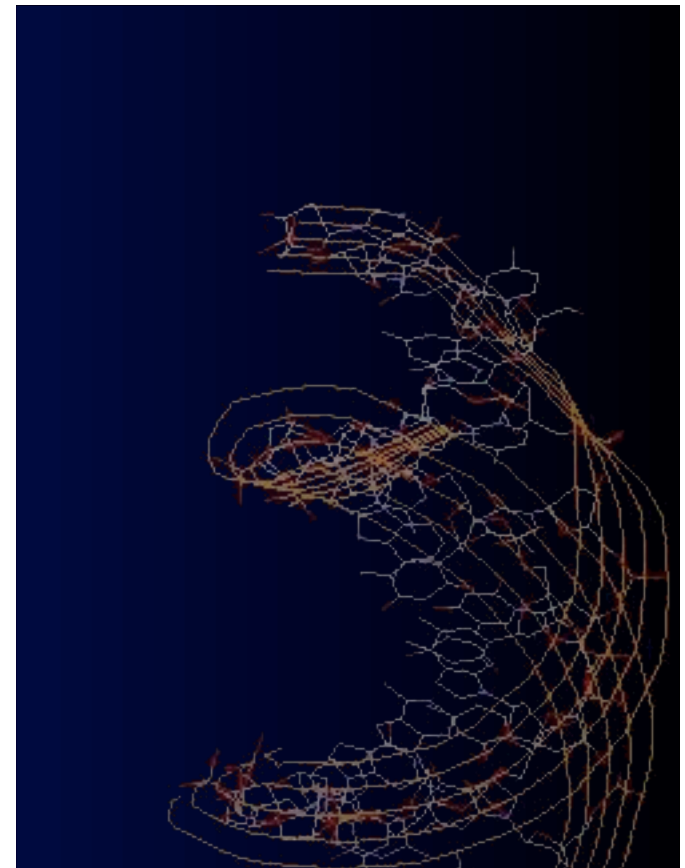
Esempi



```c
#include <stdio.h>

#define SIZE 10

/* function prototypes */
void addarrays( int [], int []);

main()
{
   int a[SIZE] = {1, 1, 1, 1, 1, 1, 1, 1, 1, 1};
   int b[SIZE] = {9, 8, 7, 6, 5, 4, 3, 2, 1, 0};

   addarrays(a, b);

   return 0;
}

void addarrays( int first[], int second[])
{
    int total[SIZE];
    int *ptr_total = &total[0];
    int ctr = 0;

    for (ctr = 0; ctr < SIZE; ctr ++ )
    {
       total[ctr] = first[ctr] + second[ctr];
         printf("%d + %d = %d\n", first[ctr], second[ctr], total[ctr]);
    }
 }
```



```c
#include <stdio.h>

#define SIZE 10

/* function prototypes */
void copyarrays( int [], int []);

main()
{
     int ctr=0;
     int a[SIZE] = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10};
     int b[SIZE];

     /* values before copy */
     for (ctr = 0; ctr <SIZE; ctr ++ )
     {
        printf( "a[%d] = %d, b[%d] = %d\n",
                ctr, a[ctr], ctr, b[ctr]);
     }

     copyarrays(a, b);

     /* values after copy */
     for (ctr = 0; ctr < SIZE; ctr ++ )
     {
        printf( "a[%d] = %d, b[%d] = %d\n",
                ctr, a[ctr], ctr, b[ctr]);
     }

     return 0;
}
```



```c
void copyarrays( int orig[], int newone[]
{
     int ctr = 0;

     for (ctr = 0; ctr < SIZE; ctr ++ )
     {
        newone[ctr] = orig[ctr];
     }
}
```