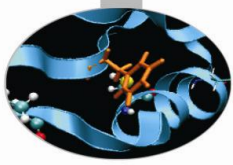
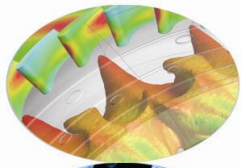
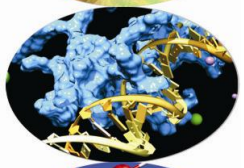
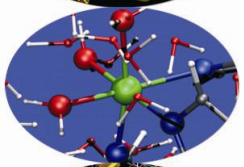
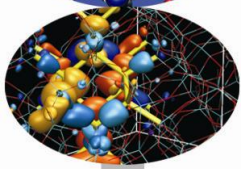
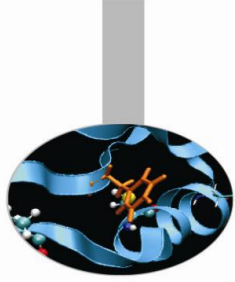


Funzioni II Parte

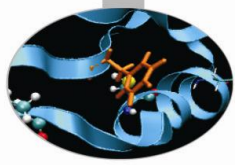







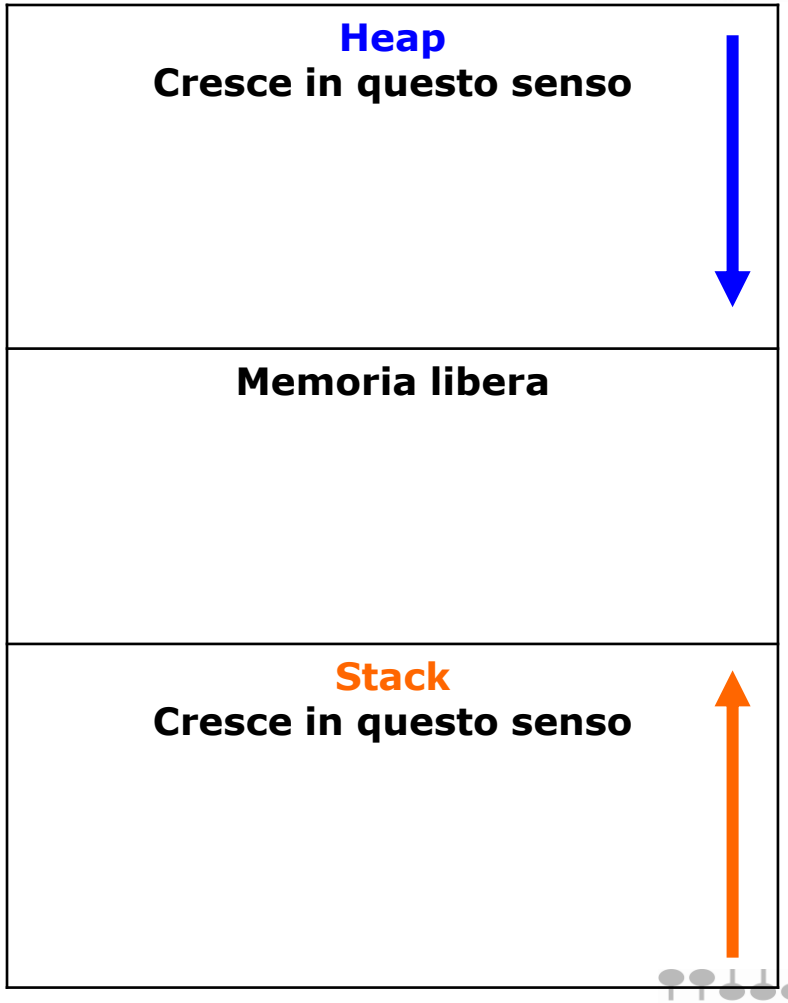
Indice

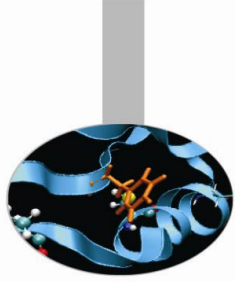
- L'uso della memoria
- L'allocazione dinamica della memoria in C
- Le funzioni malloc, calloc, realloc e free
- L'allocazione dinamica in C++
- Gli operatori new e delete
- I tipi restituiti
- La restituzione di reference
- Il passaggio di array a funzioni
- Allocazione dinamica di memoria per matrici

Uso della memoria



- **Global** tutte le variabili dichiarate come globali o statiche in un programma C/C++ ricadono in questa zona.
- **Heap** la memoria allocata tramite le funzioni che verranno discusse nel seguito di questo modulo fa uso di una zona particolare detta Heap.
- **Stack** tutte le variabili locali, i parametri attuali passati alle funzioni, gli indirizzi ritornati dalle funzioni sfruttano una zona della memoria che viene detta stack.
- In molti sistemi lo stack e l'heap sono allocati da lati opposti della memoria libera e il loro verso di crescita è pertanto opposto

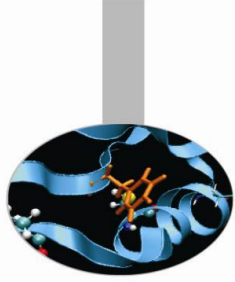




Allocazione dinamica della memoria in C

- L'allocazione dinamica permette, in generale, la gestione della memoria heap.
- Il C fornisce quattro funzioni preposte a questo scopo: malloc, calloc e realloc per l'allocazione; free per la deallocazione. Tutte quante sono contenute all'interno della libreria stdlib.h.

- L'uso della memoria dinamica richiede estrema accortezza da parte del programmatore al fine di ottimizzare l'utilizzo delle risorse di memoria senza commettere errori che possono rivelarsi anche gravi.



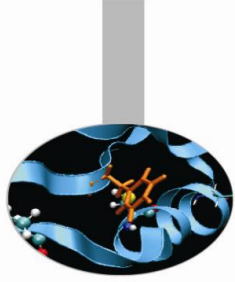
malloc

```
void *malloc(size_t number_of_bytes);
```

Questa funzione ritorna un puntatore a void che contiene l'indirizzo della locazione di memoria a partire dalla quale vengono allocati `number_of_bytes byte`.
L'unico argomento passato è di tipo `size_t` che è un sinonimo di unsigned long, definito all'interno dell'header file stddef.h .
Se non è disponibile la quantità di memoria richiesta, viene restituito il puntatore nullo.
Il puntatore a void restituito deve essere convertito, tramite casting, a puntatore al tipo di dati che verrà ospitato in quell'area di memoria.
E' buona norma utilizzare un espressione come `costante*sizeof(<nome_tipo>)` per passare come argomento a *malloc* l'ammontare corretto della memoria di cui si necessita.

Esempio:

```
char* ch_ptr;
ch_ptr = (char*) malloc(50*sizeof(char)); // blocco di 50 char
```



calloc e realloc

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

Ritorna un puntatore ad uno spazio di memoria allocato per un array di num elementi ciascuno di dimensione size .

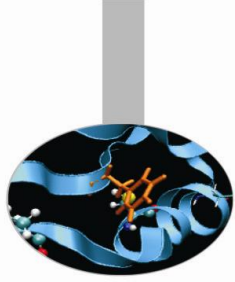
Esempio:

```
int* int_ptr;
int_ptr = (int*) calloc(20,sizeof(int)); // blocco di 20 int
```

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

Modifica la dimensione di un blocco di memoria allocato in precedenza e puntato da ptr. La nuova dimensione del blocco è pari a size e può essere più grande o più piccola di quella iniziale. Viene restituito un puntatore al nuovo spazio di memoria.

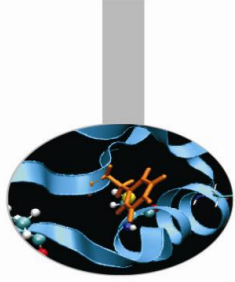
calloc e realloc



Esempio:

```
void *ptr;
ptr=int_ptr;
ptr=realloc(ptr,10*size(int)); // blocco di 10 int
```

Come per la malloc, nel caso in cui non sia possibile allocare memoria le funzioni calloc e realloc restituiscono il puntatore nullo.



free

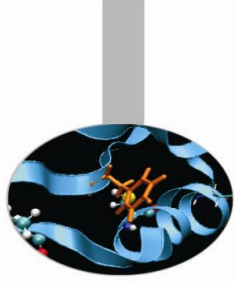
```
void *free(void* ptr);
```

La funzione free libera una porzione di memoria, allocata dinamicamente, a partire dalla locazione specificata da ptr.

Esempio:

```
char *strPtr;
strPtr = (char*) malloc(100);
free(strPtr);
```

Esempio



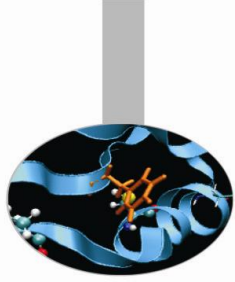
esempio: uso di malloc e free

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

int main(){
        int* pi;
        pi=(int*) malloc(sizeof(int));
        *pi=90;
        if(!pi){
                printf("Not enough memory \n");
                return 1;
        }
        printf("Integer: %d \n",*pi);
        free(pi);
        return 0;
}
```

•output:

```
Integer: 90
```

Esempio: uso di calloc e free

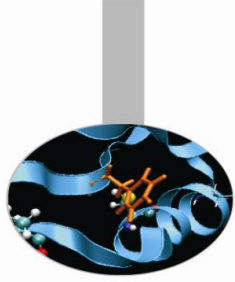


```
#include<stdio.h>
#include<stdlib.h>
int main(int argc, char* argv[]){
        double* pd;
        int i;
        int k=atoi(argv[1]); // conversione da char a int

   pd=(double*) calloc(k,sizeof(double));
   if(!pd){
        printf("Not enough memory \n");
        return 1;
   }
   for(i=0;i<k;i++)
        pd[i]=90.0+i;
   for(i=0;i<k;i++)
        printf("%f \n",pd[i]);
   free(pd);
   return 0;
}
```

- output:

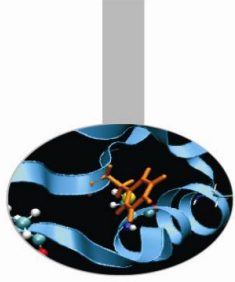
```
./v.x 5
90.000000
91.000000
92.000000
93.000000
94.000000
```

L'allocazione dinamica in C++



•Per l'allocazione dinamica della memoria in C++ è possibile utilizzare new, new[ ] e delete, delete[ ]. Tramite questi operatori è possibile gestire dinamicamente spazi di memoria che non sottostanno alle regole di scope. In caso di fallimento dell'allocazione il puntatore restituito punta a 0.
•Questa libertà però comporta delle responsabilità per il programmatore che deve tenere sempre traccia della memoria allocata e **deallocarla sempre** in maniera esplicita quando non sia più necessaria; il rischio che si corre è quello di un run time error di tipo overflow.
•L'utilizzo di questi operatori è rivolto anche alla gestione di quantità definite solo run-time, pratica impossibile con gli array la cui dimensione deve essere definita a compile-time.

new e delete

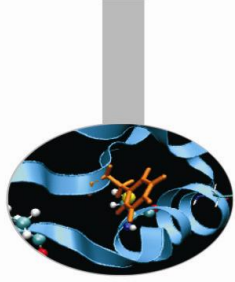


•Sono gli operatori del C++ per l'allocazione e la deallocazione della memoria dinamica occupata da puntatori a variabili ed a oggetti.
•Hanno bassa precedenza ed associatività da destra a sinistra.
Esempio:

```
int* ptr_int;
ptr_int = new int;
*ptr_int = 20;
delete ptr_int;
```

•Quando agiscono su array sono accompagnati dalle parentesi quadre: **new[ ], delete[ ]**
Esempio:

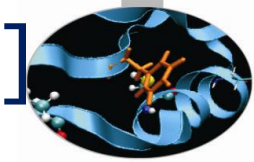
```
double* db_arr = new double [10];
delete[] db_arr;
```



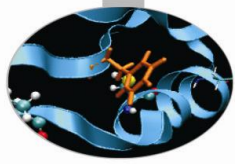
Esempio: new e delete

```
#include<iostream>
using namespace std;
int main (){
  int *pi;
  pi = new int (90) ;  /* allocazione dinamica con
                                  inizializzazione    */
  if (!pi) {
        cout << "Not enough memory" << endl;
        return 1;
  }
  cout << *pi;
  delete pi;           /* deallocazione esplicita
                                  della memoria                */
  return 0; }
```

Esempio: uso di new [] e delete []

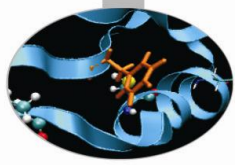


```
#include<iostream>
#include<stdlib.h>
using namespace std;
int main (int argc, char *argv[]){
  double *pd;
  int i, k = 0;

  if ( argc > 1 )
     k = atoi(argv[1]); /* In chiamata bisognerà specificare
                           argv[1] altrimenti k = 0.*/
  pd = new double [k]  ;

  if (!pd) {
    cout << "allocazione in memoria fallita!!" << endl;
    return 1;
  }
  for (i=0; i<k; i++) pd[i] = 90.0 + i;
  for (i=0; i<k; i++) cout << pd[i] << endl;
  delete[] pd;          //deallocazione esplicita della memoria
  return 0; }
```



Le funzioni: i tipi restituiti

- Una funzione può restituire qualsiasi tipo predefinito, costruito dall'utente o costruito sui tipi predefiniti. In particolare possiamo avere funzioni che restituiscono *puntatori* o *reference* ad un tipo predefinito.
- Anche in questo caso ciò può essere utile quando deve essere ritornata alla sezione chiamante un ampio numero di dati: la restituzione *per valore* di dati ne implica infatti, come il passaggio, la creazione di una *copia* in memoria. Restituendo un *reference* o un *puntatore*, invece, viene ritornato solo un *indirizzo*.
- Quando vengono restituiti reference o puntatori, questi **non** devono fare riferimento a variabili locali, i cui valori vengono distrutti all'uscita della funzione. Per ovviare a questo problema possiamo dichiarare come *static*, all'interno della funzione, le variabili da ritornare o allocarle dinamicamente tramite il comando *new*, nel caso di puntatori.
- Come visto anche negli esempi precedenti, è facile incontrare in C/C++ funzioni che non restituiscono nulla e sono dichiarate come:
- `void nome_funzione(lista argomenti);`



Esempio

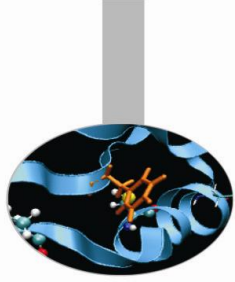
esempio1: scriviamo un semplice programma che svolga l'addizione di due interi facendo uso della funzione "somma" che restituisce al main un puntatore a int.

```
#include<iostream>
using namespace std;
int* somma(int, int);
int main(){
   int var_a = 2, var_b = 4, *i_ptr;
   cout <<"The sum is: ";
   i_ptr = somma(var_a,var_b);
   cout << *i_ptr <<endl;
   return 0;
}
int* somma(int a1, int a2){
                        // restituisce un punt
    int *sum = new int;          // allocazion
    *sum = a1+a2;
    return sum;
}
```

- **output:**

```
The sum is: 6
```

Esempio

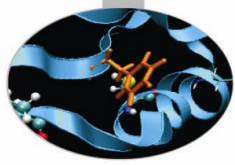


esempio2: modifichiamo la funzione “somma” in modo che restituisca al main un reference a int.

```
#include<iostream.h>
int& somma(int, int);
int main(){
   int var_a = 2, var_b = 4, sm;
   cout <<"The sum is: ";
   sm = somma(var_a,var_b);
   cout << sm << endl;
   return 0;
}
int& somma(int a1, int a2){
                        // restituisce un refe
   static int sum;      // sum è dichiarata st
   sum = a1 + a2;
   return sum;
}
```

- **output:**

```
The sum is: 6
```

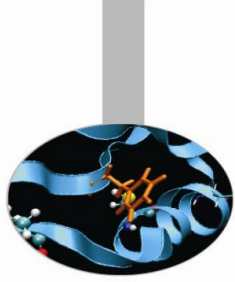


Restituzione di reference

- Quando una funzione restituisce un reference, essa stessa può essere utilizzata come *left value* in un'istruzione di *assegnamento*. Il valore assegnato alla funzione, infatti, sarà automaticamente assegnato anche alla variabile (o a qualsiasi altra entità) referenziata dalla funzione stessa.
- **esempio:** assegnamento di una costante ad una funzione che restituisce un reference

```
#include<iostream.h>
int value=20;              // variabile globale
int& fun_val();            // prototipo della funzione
int main() {
  cout << "The starting value is: " << value << endl;
  cout << "Calling fun_val" << endl;
  cout << "The value is: " << fun_val() << endl;
  fun_val()=32;            // assegnamento di una costante alla
                                  // funzione fun_val()
  cout << "After assigning a new value to fun_val:" << endl;
  cout << "funval() = " << fun_val() << endl;
  cout << "value = " << value << endl;
  return 0; }
```

Restituzione di reference



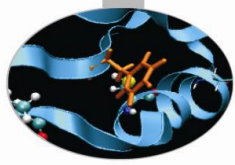
```
// definizione di fun_val
int& fun_val() { return value; }
```

Output:

```
The starting value is: 20
Calling fun_val
The value is: 20
After assigning a new value to fun_val:
funval() = 32
value = 32
```

- Come è evidente l'istruzione **fun_val()=32** cambia il valore di value da 20 a 32. Questo è dovuto al fatto che fun_val() restituisce un reference a value che dunque diventa, seppur implicitamente, il left value dell'istruzione di assegnamento ovverosia fun_val() si comporta come *alias* di value.
- Attenzione: il programma funziona correttamente perché value è stata definita come variabile globale.

Il passaggio di array a funzioni



•Sfruttando la corrispondenza tra array e puntatori, il C/C++ permette di passare array a funzioni *solo* per riferimento (modalità: passaggio per puntatore). Una funzione in C/C++ è, dunque, *sempre* in grado di agire sulle locazioni di memoria occupate dagli elementi dell'array. L'argomento richiesto dalla chiamata di una funzione è semplicemente il nome dell'array stesso che, come sappiamo, corrisponde all'indirizzo del suo elemento di posizione zero.

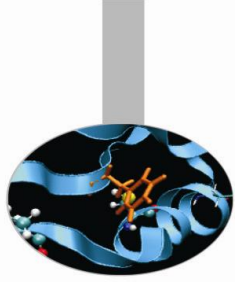
•Il prototipo di una funzione cui viene passato un array può apparire come:

```
 tipo_restituito nome_funzione(tipo_array[ ], int
dim_array);
```

oppure:

```
 tipo_restituito nome_funzione(tipo_array *, int
dim_array);
```

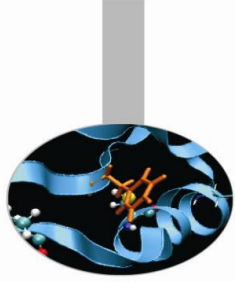
ove la dimensione dell'array è un parametro opzionale.



Esempio

- **esempio**: scrittura di una funzione che calcola il quadrato degli elementi di un array

```
#include<iostream.h>
void arr_sqr(int[ ], int);     // prototipi delle funzioni che
void print(int*, int);         // passano gli array

int main(){
  const int dim=5;
  int arr_int[dim]={1,2,3,4,5};
  cout << "The array components are: " << endl;
  print(arr_int, dim);
  arr_sqr(arr_int, dim);
  cout << "The squares of the array components are: " << endl;
  print(arr_int, dim);
  return 0; }
```

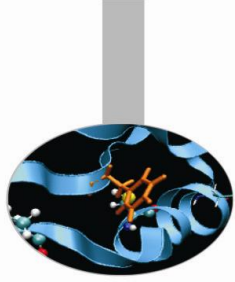


Esempio

```
// funzione per il calcolo dei quadrati delle componenti di un
array
void arr_sqr(int* array, int num){
        for(int i=0; i<num; i++)
        *(array+i)=array[i] * array[i];
                                  // dereferenziazione e prodotto
}
// funzione per la stampa delle componenti di un array
void print(int array[ ], int num){
  for(int i=0; i<num; i++)
  cout << array[i] << " ";
  cout << endl;
}
```

- **output:**

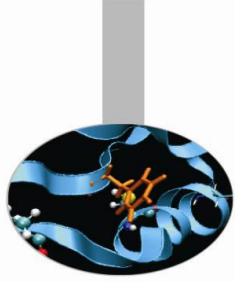
```
The array components are:
1 2 3 4 5
The squares of the array components are:
1 4 9 16 25
```



Array Multidimensionali

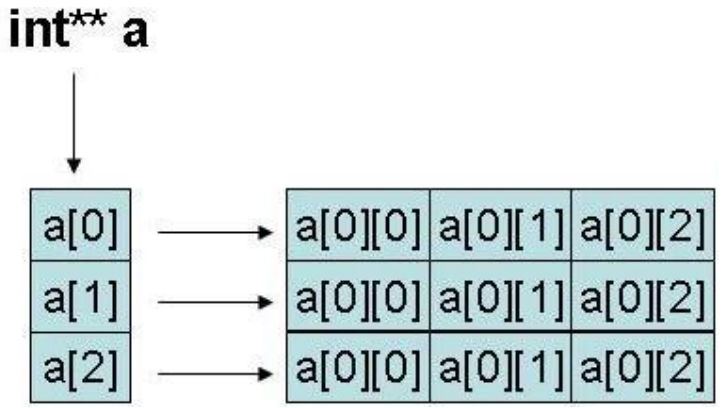
Nel caso di array bidimensionali per allocare/deallocare dinamicamente memoria si faccia riferimento alla seguente sintassi

```
int main(int argc, char* argv[]){
int i;
int nr=3, nc=4;
int **array;
printf('Allocazione della matrice\n')
array=new int*[nr];
 if(array==NULL) {
   printf('Impossibile allocare  memoria\n');
   exit(1);
 }
 printf("Allocazione array di %d puntatori\n",nr);
 for(i=0;i<nr;i++) {
 array[i]=new int[nc];
 if(array[i]==NULL){
 printf('Impossibile allocare memoria\n');
 exit(1);
 }
```

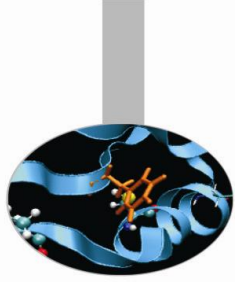


Esempio

```
else printf("Allocazione puntatore %d di %d elementi\n" ,i,nc);
}
for(i=0;i<nr;i++) {
   delete[] array[i];
   printf("Deallocazione puntatore  %d di %d elementi\n", i,nc);
}
delete[] array;
printf("Deallocazione completa\n");
return 0;
}
```



Esempio



OUTPUT

```
Allocazione della matrice
Allocazione array di 3 puntatori
Allocazione puntatore 0 di 4 elementi
Allocazione puntatore 1 di 4 elementi
Allocazione puntatore 2 di 4 elementi
Deallocazione puntatore  0 di 4 elementi
Deallocazione puntatore  1 di 4 elementi
Deallocazione puntatore  2 di 4 elementi
Deallocazione completa
```