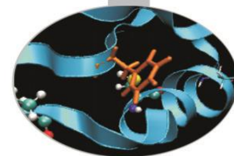
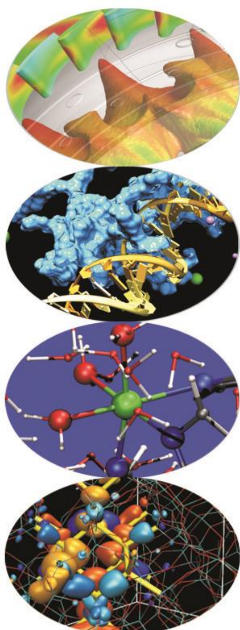
CINECA SCAI
SuperComputing Applications and Innovation

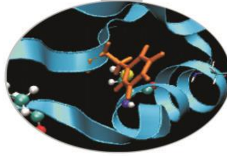


Funzioni II Parte



CINECA

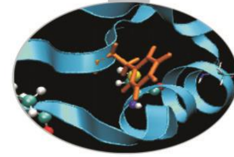
SCAI
SuperComputing Applications and Innovation



Indice

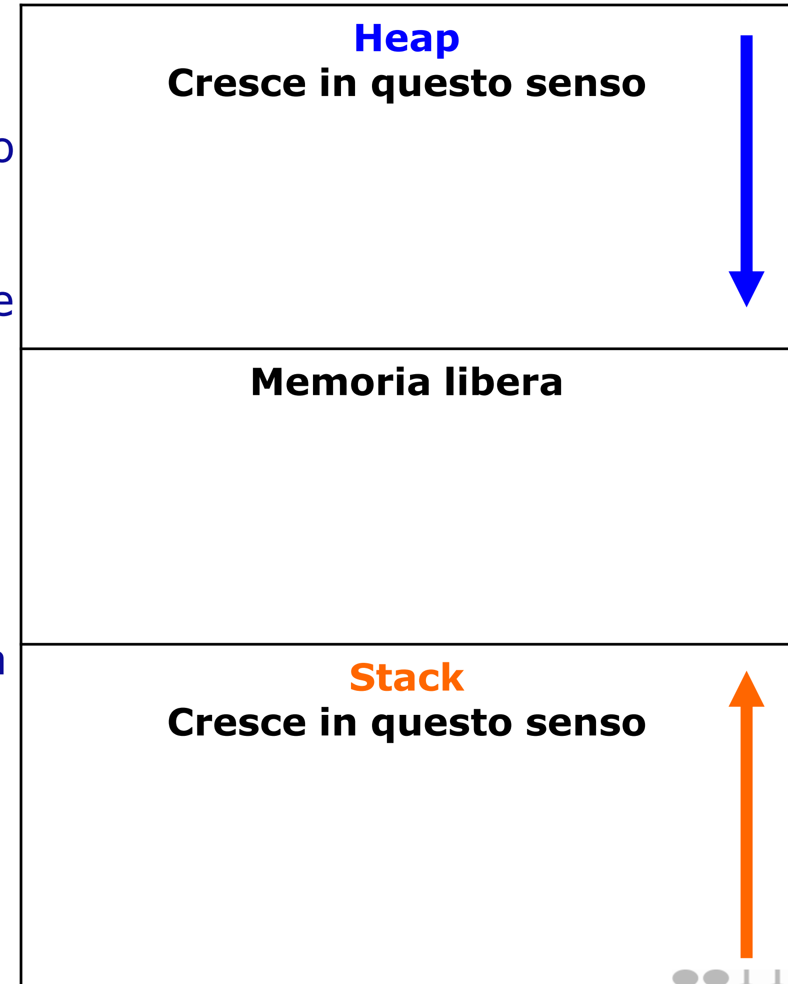
- L'uso della memoria
- L'allocazione dinamica della memoria in C
- Le funzioni malloc, calloc, realloc e free
- L'allocazione dinamica in C++
- Gli operatori new e delete
- I tipi restituiti
- La restituzione di reference
- Il passaggio di array a funzioni
- Allocazione dinamica di memoria per matrici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

CINECA



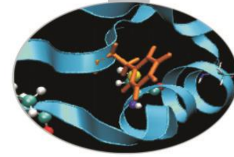
Uso della memoria

- **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



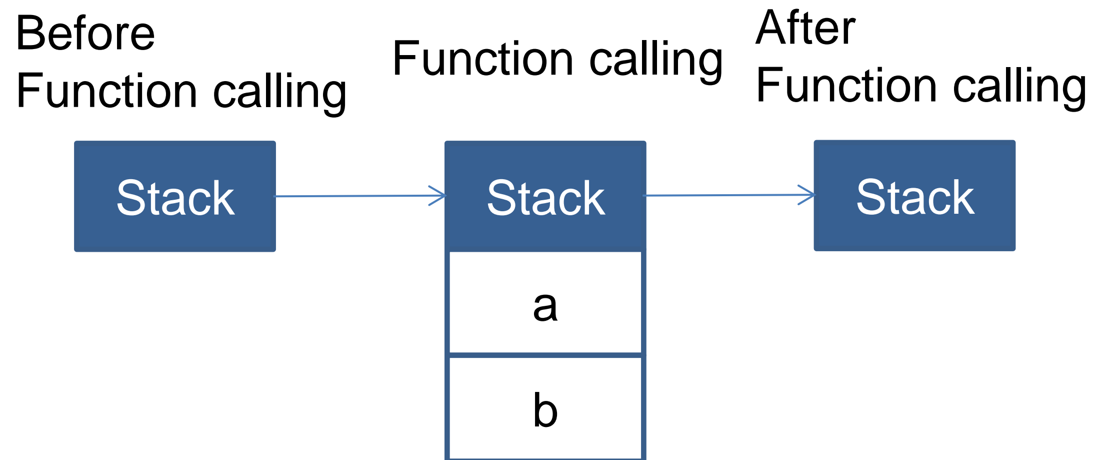
SuperComputing Applications and Innovation

Uso della memoria



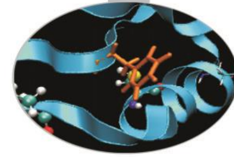
Varibili *local* hanno un ciclo di vita legato alle porzioni di codice in cui sono definite.

```
void x()
{
        int a;
        int b;
        return ;
}

int main()
{
        x();
        return
}
```



La memoria viene allocata e deallocata in maniera automatica sullo stack, favorendone il riutilizzo. Le varibili hanno un lifetime limitato alla chiamata della funzione

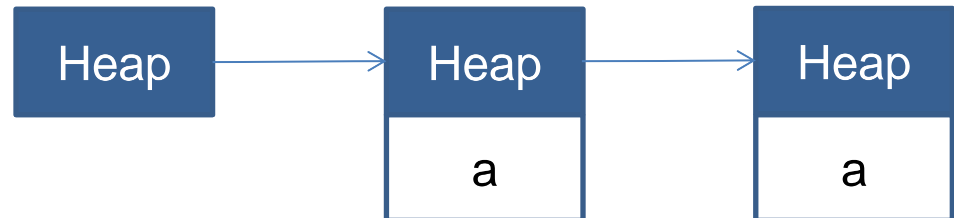
SCAI
SuperComputing Applications and Innovation



Uso della memoria

La memoria di heap è un'alternativa all'utilizzo dello stack.
L'heap viene utilizzato nell'allocazione dinamica della memoria. Il programmatore alloca esplicitamente le variabili e le dealloca esplicitamente.

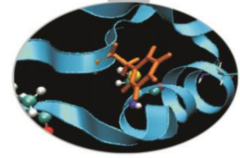
```
void x(int *a)
{
a= (int*)malloc(sizeof(int);
return ;
}

int main()
{
        int *a =NULL;
        x(a);
       if(a)free(a);
        return
}
```



La memoria viene allocata e deallocata dal programmatore sull'heap, favorendone il riutilizzo. Le variabili vengono distrutte esplicitamente

CINECA

Allocazione dinamica della memoria in C

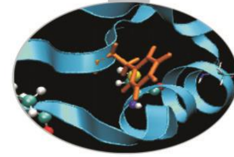


- L'allocazione dinamica permette, in generale, la gestione della memoria heap.

- 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.

NOTA: 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



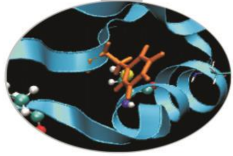
```
void *malloc(size_t number_of_bytes);
```

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 .

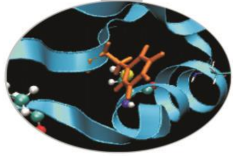
Esempio:

```
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.

SuperComputing Applications and Innovation



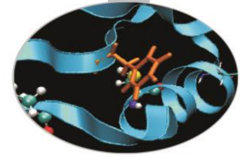
calloc e realloc

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);
```

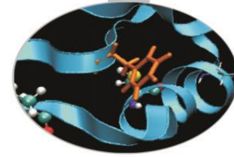
La funzione free libera una porzione di memoria, allocata dinamicamente, a partire dalla locazione specificata da ptr.

Esempio:

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

Necessaria per evitare *memory leaks* nel codice

Esempio



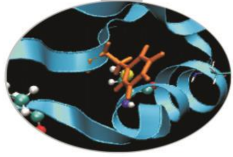
esempio: uso di malloc e free

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

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

- output:

```
Integer: 90
```

Esempio: uso di calloc e free

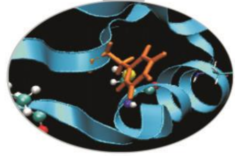


```
#include<stdio.h>
#include<stdlib.h>
int main(int argc, char* argv[]){
        double* pd;
        int i;
        int k=atoi(argv[1]); // conversione da char a int

    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:

```
>> ./a.out 5
90.000000
91.000000
92.000000
93.000000
94.000000
```

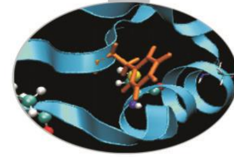
L'allocazione dinamica in C++



- 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.

- 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:

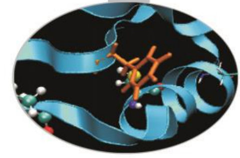
```
int* ptr_int;
ptr_int = new int;
*ptr_int = 20;
delete ptr_int;
```

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

Esempio:

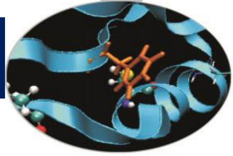
```
double* db_arr = new double [10];
delete[] db_arr;
```

Esempio: new e delete



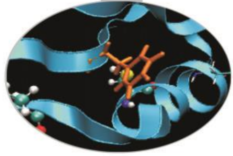
```
#include<iostream>
using namespace std;
int main ()
{
  int *pi;
  pi = new int (90) ;  /* allocazione dinamica con inizializzazione */
  if (!pi)
  {
       printf("Not enough memory\n");
       return 1;
  }
  printf("%d \n",*pi);
  delete pi;           /* deallocazione esplicita della memoria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

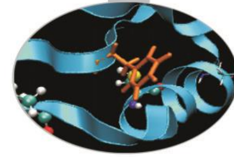
Le funzioni: i tipi restituiti



- 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:

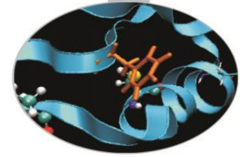
```
void  nome_funzione(lista argomenti);
```

Esempio



esempio1: scriviamo un semplice programma che esegue la somma di due numeri e la ritorna per reference o per puntatore

```
#include <iostream>
using namespace std;

int * sommaPointer(int,int);
int & sommaRef(int,int);

int main() {
   int number = 100;
   cout << number << endl;
   cout << *sommaPointer(number,number) << endl;   // ??
   cout << sommaRef (number,number) << endl;   // ??
}

int * squarePtr(int number,int number2) {
   int localSum = number + number2;
   return & localSum;
}
int & squareRef(int number,int number2) {
   int localSum = number + number2;
   return localSum ;
}
```



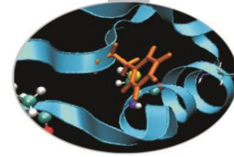
Esempio

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 puntatore a intero
   int *sum = new int;          // allocazione dinamica
   *sum = a1+a2;
   return sum;
}
```

- **output:**

```
The sum is: 6
```



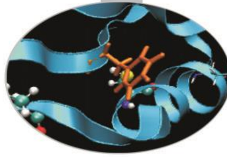
Esempio

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 reference a intero
  static int sum;         // sum è dichiarata static
  sum = a1 + a2;
  return sum;
}
```

- **output:**

```
The sum is: 6
```

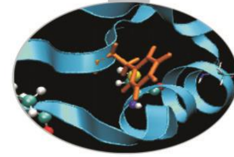


Restituzione di reference

- 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



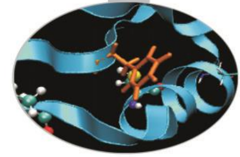
```
// definizione di fun_val
int& fun_val() { return value; }
```

```
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.

Nota: 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:

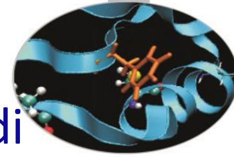
```
tipo_restituito nome_funzione(tipo_array[ ], int dim_array);
```

oppure:

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

ove la dimensione dell'array è un parametro opzionale.

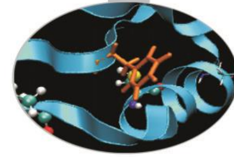
Esempio



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

```
#include<iostream.h>
Using namespace std
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; }
```

SuperComputing Applications and Innovation



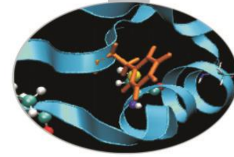
Esempio

```
// 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

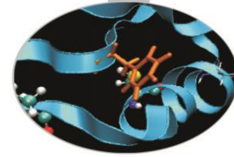


Nel caso di array bidimensionali per allocare/deallocare dinamicamente memoria si faccia riferimento alla seguente sintassi

Sintassi C

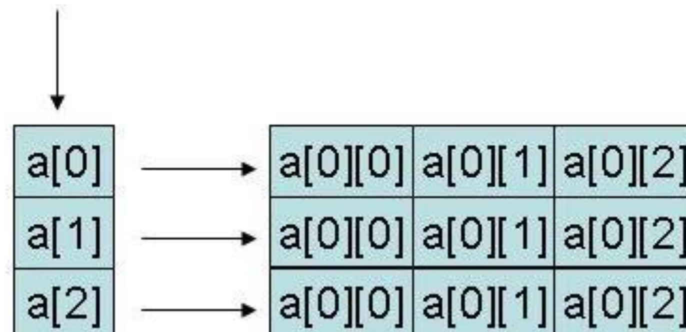
```
#include <stdio.h>
#include <stdlib.h>

int main(int argc, char* argv[]){
int i;
int nr=3, nc=4;
int **array;
printf("Allocazione della matrice\n");
 array = (int**) malloc(nr * sizeof(int *));
 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]= (int*)malloc(nc * sizeof(int));
   if(array[i]==NULL){
     printf("Impossibile allocare memoria\n");
     exit(1);
   }  // segue nella slide successiva
```

SuperComputing Applications and Innovation

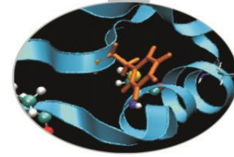


Esempio

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

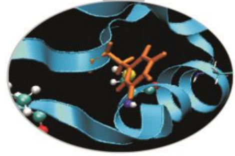


Array Multidimensionali



Sintassi C++

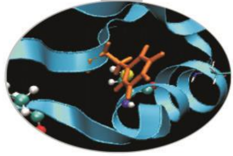
```
#include <iostream>
#
int main(int argc, char* argv[]){
int i;
int nr=3, nc=4;
int **array;
printf("Allocazione della matrice\n");
 array = new int*[nr];
 if(array==NULL)
{
   printf("Impossibile allocare  memoria\n");
   exit(1);
 }
 printf("Allocazione array di %d puntatori\n",nr);
 for(i=0;i<nr;i++)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



Array Multidimensionali

```
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;
}
```

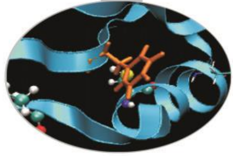
Esempio



OUTPUT

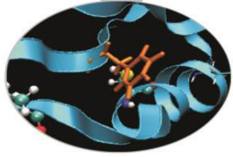
```
Allocazione della matrice
Allocazione array di 3 puntatori
Allocazione puntatore 0 di 4 elementi
Allocazione puntatore 1 di 4 elementi
Allocazione puntatore 2 di 4 elementi
Deallocazione puntatore  0 di 4 elementi
Deallocazione puntatore  1 di 4 elementi
Deallocazione puntatore  2 di 4 elementi
Deallocazione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

Le funzioni inline



- Ogni chiamata di funzione richiede un certo tempo di elaborazione.
- In C lo strumento che viene fornito dal linguaggio per limitare le chiamate a funzione è la macro **#define** introdotta in precedenza.
- In C++ quando un programma contiene funzioni piccole (cioè costituite da poche istruzioni) che vengono invocate spesso può essere conveniente definirle **inline**. Il qualificatore inline, posto innanzi al tipo di dato restituito dalla funzione, dice al compilatore di scrivere il corpo della funzione in ogni punto del programma in cui essa è chiamata invece di effettuare ogni volta un’autentica chiamata alla funzione stessa.

Le funzioni inline

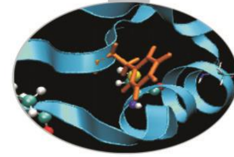


- La *definizione* di una funzione inline appare, di solito, nella forma seguente:

```
inline tipo_restituito nome_funzione(argomenti)
   {
      corpo della funzione
   }
```

- Da quanto detto emerge che l'uso delle funzioni inline permette di ridurre il tempo di esecuzione di un programma e di evitare l'allocazione di memoria avendo il solo (trascurabile) svantaggio di aumentare le dimensioni del codice eseguibile.

Esempio



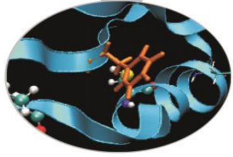
Esempio: confronto fra #define ed inline

/* C-file */

```
#include<stdio.h>
#define square(var) var*var

int main(){
      int v_i=4;
      double v_d=2.2;
      printf("v_i^2= %d \n",square(v_i));
      printf("v_d^2= %f \n",square(v_d));
      return 0;
}
```

output:

```
v_i^2= 16
v_d^2= 4.840000
```



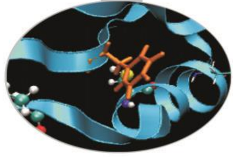
Esempio

```
// C++ file

#include<iostream>
using namespace std;
inline int square(int i){return i*i;}
inline double square(double d){
   return d*d;
}

int main(){
        int v_i=2;
        double v_d=2.2;
        cout << "v_i^2= " << square(v_i) << endl;
        cout << "v_d^2= " << square(v_d) << endl;
        return 0;
}
```

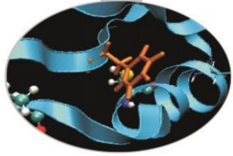
output:

```
v_i^2= 4
v_d^2= 4.84
```

Le funzioni ricorsive



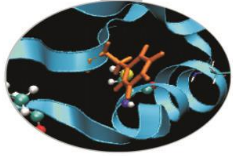
Una funzione si dice *ricorsiva* quando richiama se stessa. Questo comportamento è permesso dal C/C++ senza dover specificare nessun qualificatore particolare (a differenza del Fortran90, per es.).



Le funzioni ricorsive

Tipico esempio di funzione ricorsiva è fornito dal calcolo del fattoriale di un intero:

```
#include<iostream>
using namespace std;
unsigned long factorial(unsigned long);
int main(){
      long num;
      cout << "Insert an integer" << endl;
      cin  >> num;
      cout << "The factorial of  " << num <<  " is "
           << factorial(num) << endl;
      return 0;
}
```



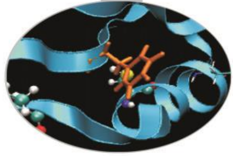
Le funzioni ricorsive

```
unsigned long factorial(unsigned long number){
    if( number <=1 )
       return 1;
    else
       return number * factorial(number-1);
}
```

Dando come input il numero 10 otteniamo in uscita:

```
>> The factorial of 10 is 3628800
```

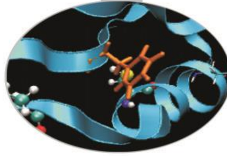
Overloading di funzioni (C++)



- In C++, ma non in C, è possibile definire più funzioni con lo stesso nome purché queste differiscano per la lista degli argomenti.
Esaminando, infatti, il numero ed il tipo degli argomenti presenti nella chiamata,il compilatore è in grado di scegliere la versione corretta della funzione.

- A volte, tuttavia, il programma può contenere istruzioni ambigue per le quali il compilatore non è in grado di scegliere fra le differenti versioni della funzione soggetta a overload. L'ambiguità dipende, solitamente, dalla conversione automatica di tipo presente nel C++.

Esempio

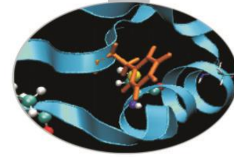


esempio1: overload, senza ambiguità, della funzione “stampa” per la scrittura su standard output

```
#include <iostram>
using namespace std;
void stampa() {cout << “Nessun argomento” << endl;}
int stampa(int num){return num * 2 ;}
double stampa(double num){return num / 2 ;}
int main(){
      stampa() ;
      cout << stampa(4) << endl;
      cout << stampa(2.5) << endl;
      return 0;
}
```

Output:
Nessun argomento
8
1.25

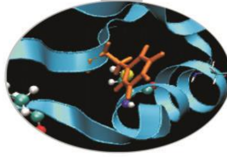
Esempio



Esempio 2: overload, con ambiguità, della funzione “stampa” per la scrittura su standard output

```
#include <iostram>
using namespace std;
void stampa() {cout << “Nessun argomento” << endl;}
float stampa(float num){return num * 2 ;}
double stampa(double num){return num / 2 ;}
int main(){
      stampa() ;
      cout << stampa(4) << endl;
      cout << stampa(2.5) << endl;
      return 0;
}
```

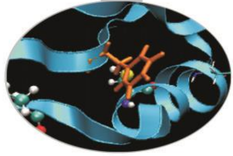
Esempio



Il nostro compilatore segnala l'ambiguità con un messaggio di errore:

```
>>g++ esempio2.cpp -o esempio2.x esempio2.cpp:
In function â€îint main()â€™: esempio2.cpp:11:
error: call of overloaded â€šstampa(int)â€™ is ambiguous
esempio2.cpp:5: note: candidates are:
 float stampa(float) esempio2.cpp:7:
 note: double stampa(double)
```

e non può generare un file eseguibile. In realtà il problema deriva dalla mancanza di una versione della funzione stampa adatta ad argomenti di tipo int.

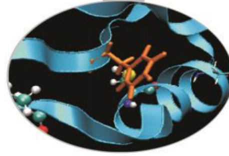


Argomenti di Default

Gli argomenti di default non sono supportati in C

• E' possibile, in C++, assegnare, ad uno o più argomenti di una funzione, dei valori di default che vengono utilizzati automaticamente quando tali argomenti sono assenti nella chiamata della funzione.
• I valori di default devono essere specificati una sola volta, ovvero quando la funzione viene dichiarata all'interno del file. Nel prototipo i parametri che accettano valori di default devono seguire quelli che non li accettano.
• E' permesso specificare argomenti di default diversi per ogni versione di una funzione soggetta a overload

Esempio

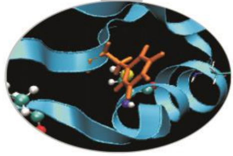


esempio: calcolo dell'area del trapezio.

l valore di default dell'altezza e delle due basi è posto uguale ad uno.

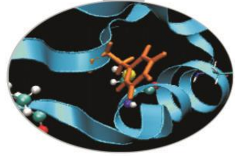
```
#include<iostream>
using namespace std;
// prototipo della funzione a_trap:
// contiene argomenti di default
double a_trap(double b_maj=1, double b_min=1, double height=1);

int main(){
        // nessun argomento
        cout << a_trap() << endl;
        // 1 solo argomento
        cout << a_trap(2.5) << endl;
        // solo 2 argomenti
        cout << a_trap(4, 1.5) << endl;
        // tutti gli argomenti
        cout << a_trap(6, 2, 3.2) << endl;
        return 0;
}
```

Esempio



```
// definizione della funzione a_trap
double a_trap (double b_maj, double b_min, double height)
 {
       double area;
       area=(b_maj+b_min)*height/2;
       cout << "Major base: " << b_maj << endl;
       cout << "Minor base: " << b_min << endl;
       cout << "Height: " << height << endl;
       cout << "The area is: ";
       return area;
}
```

Puntatori a funzione e callback



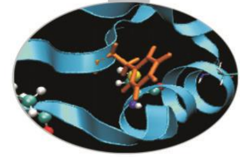
- In C è possibile utilizzare dei puntatori a funzioni, ovvero delle variabili a cui possono essere assegnati gli indirizzi in cui risiedono le funzioni, e tramite questi puntatori a funzione, le funzioni puntate possono essere chiamate all'esecuzione.

- Un puntatore a funzione punta sempre a una funzione con una specifica signature.

- Lo stesso puntatore può indirizzare funzioni diverse ma con gli stessi (tipi di) parametri e stesso tipo di valore di ritorno

- Un uso tipico dei puntatori a funzioni è passarli come argomenti ad altre funzioni

Puntatori a funzione e callback



Prototipo della funzione

```
int myFunc(int val);
```

Dichiarazione di un puntatore a funzione che resituisce un intero e prende un intero

```
int (*myFuncPointer)(int);
```

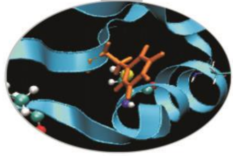
Dichiarazione di un tipo POINTERFUNC di puntatori a funzione che restituiscono un intero e prendono un intero

```
typedef  int(*POINTERFUNC)(int);
```

Dichiarazione di una variabile i tipo POINTERFUNC:

```
POINTERFUNC myPtr;
myPtr=myFunc;
int result=(*myPtr)(3); //Chiamata della funzione tramite il pointer
```

Puntatori a funzione e callback

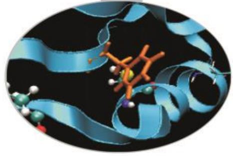


NOTA

L'istruzione typedef introduce sinonimi per tipi costruibili in C.
Spesso viene usata per semplificare dichiarazioni complesse e/o per rendere più intuitivo l'uso di un tipo in una particolare accezione.

```
typedef int* myPtrInt;
typedef float myFloat;

float a;
myFloat b=a;
int  i=3;
myPtrInt p=&i;
```

Puntatori a funzione e callback



Una funzione di callback è una funzione chiamata tramite un puntatore a funzione.

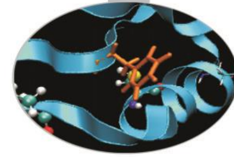
- Supponiamo che a una funzione venga passato come argomento il puntatore (indirizzo) di un'altra funzione.

- Quando la prima funzione usa il puntatore a funzione per chiamare la seconda funzione, viene effettuata una callback.

- Molto utili per rendere il codice più generale, disaccoppiando la funzione chiamante dalla funzione chiamata.

- Può essere utile nella definizione di librerie di funzioni.

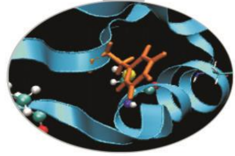
Esempio



Mergesort accetta un puntatore a una funzione per la comparazione di due elementi come ulteriore argomento

```
void mergesort(int *a, int *b, int l, int r, char (*comp)(int,
int))
{
        int i,j,k,m;
        if(r > l)
      {
          m = (r+l)/2;
           mergesort(a,b,l,m,comp);
          mergesort(a,b,m+1,r,comp);
          for(i = m+1; i > l; i--)
                      b[i-1] = a[i-1];
          for(j = m; j < r; j++)
                      b[r+m-j] = a[j+1];
          for(k = l; k <= r; k++)
                      a[k] = (*comp)(b[i],b[j]) ? b[i++] :
                      b[j--];
        }
}
```

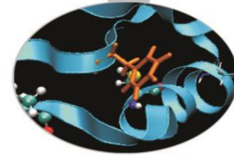
Esempio



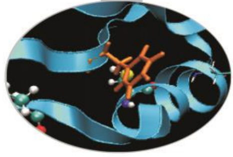
Usiamo il mergesort con le seguenti funzioni di comparazione

```
char smaller(int a, int b) { return a < b; }
char greater(int a, int b) { return a > b; }
```

Esempio



```
typdef  (char* PTR)(int,int);
int main(void)
{
    int i,e,
    a[] = { 14, 16, 12, 11, 5, 21, 17, 14, 12 };
    float b[9];
    PTR myPrt[]= { smaller, greater};
    printf("Array da ordinare:");
    for(i = 0; i < 9; i++)
        printf("%d ",a[i]);
    putchar('\n');
    do {
        printf("Scegli:\n");
        printf("0: ordina in modo crescente\n");
        printf("1: ordina in modo decrescente\n");
        fflush(stdin); /* funz. (stdio.h) per svuotare l'input */
        if(!(e =(scanf("%d",&i) && i >= 0 & i <= 1)))
                printf("\n Immetti o 0 o 1.\n");
    }
    while(!e);
```



Esempio

```
    mergesort(a,b,0,8,choice[i]);
    for(i = 0; i < 9; i++)
        printf("%d ",a[i]);
    putchar('\n');
}
```

Possiamo estendere il codice con nuove funzioni di ordinamento e utilizzare la funzione di merge sort con queste nuove definizioni.