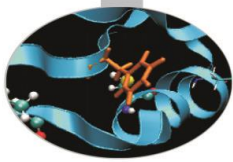
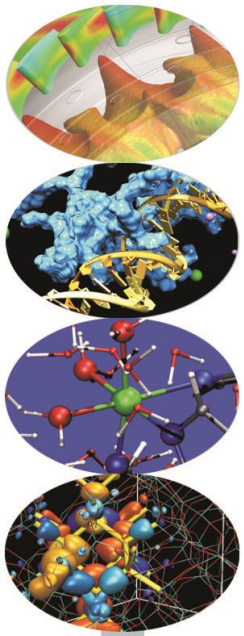
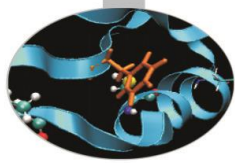


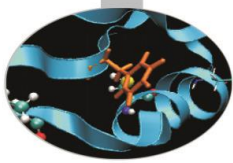
Funzioni III Parte



Indice

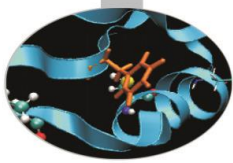


- **Funzioni inline**
- **Funzioni ricorsive**
- **Overloading di funzioni**
- **Argomenti di default**
- **Template di funzioni**
- **Inclusione di funzioni scritte con altri linguaggi**



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.

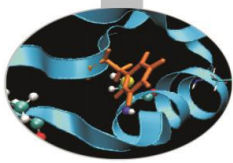


Le funzioni inline

- 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.



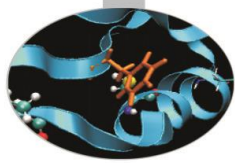
Esempio

- 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

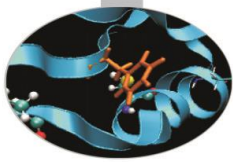
```
// c++ file
#include<iostream>
using namespace std;
inline int square(int i){return i*i;}
inline double square(double d){
    return d*d;
}

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:

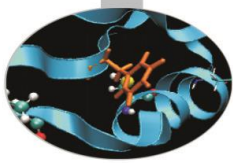
```
v_i^2= 4
v_d^2= 4.84
```

- La direttiva #define appare più versatile dell'inlining dal momento che non richiede di specificare il tipo di parametri in ingresso né quello dell'output.

Le funzioni ricorsive



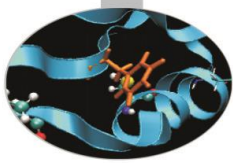
- Una funzione si dice *ricorsiva* quando richiama se stessa. Questo comportamento è permesso dal C/C++ senza dover specificare nessun qualificatore particolare (a differenza del Fortran90, per es.).



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;
}
```

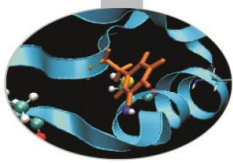


Le funzioni ricorsive

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

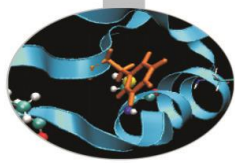
Dando come input il numero 10 otteniamo in uscita:

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



Overloading di funzioni

- 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++.



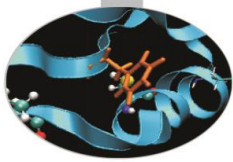
Esempio

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

```
#include<iostream>
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:

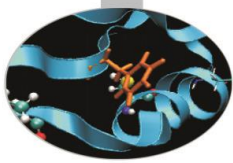
```
Nessun argomento
8
1.25
```



Esempio

esempio2: overload, con ambiguità, della funzione “stampa” per la scrittura su standard output

```
#include<iostream>
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; // ambiguità: a
cosa
                                 // convertire un int?
      cout << stampa(2.5) << endl;
      return 0;
}
```

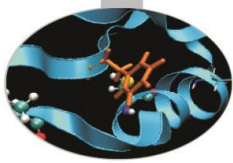


Esempio

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

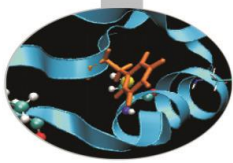
```
>g++ esempio2.cpp -o esempio2.x
esempio2.cpp: In function â€ĩint main()â€™:
esempio2.cpp:11: error: call of overloaded â€štampa(int)â€™ is ambiguous
esempio2.cpp:5: note: candidates are: float stampa(float)
esempio2.cpp:7: note:                 double stampa(double)
```

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

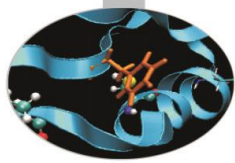
- ***Gli argomenti di default non sono supportati in C***
- E' possibile, in C++, assegnare, ad uno o più argomenti di una funzione, dei valori di *default* che vengono utilizzati automaticamente quando tali argomenti sono assenti nella chiamata della funzione.
- I valori di default devono essere specificati *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. Il valore di default dell'altezza e delle due basi è posto uguale ad uno.

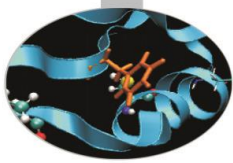
```
#include<iostream>
using namespace std;
// prototipo della funzione a_trap:
//    contiene argomenti di default
double a_trap(double b_maj=1, double b_min=1,
              double height=1);
int main(){
  cout << a_trap() << endl; // nessun argomento
  cout << a_trap(2.5) << endl; // 1 solo argomento
  cout << a_trap(4, 1.5) << endl;  // solo 2 argomenti
  cout << a_trap(6, 2, 3.2) << endl;
  return 0;}
```



Esempio

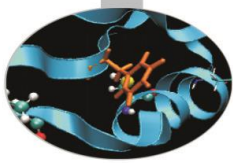
```
// definizione della funzione a_trap
double a_trap (double b_maj, double b_min,
                    double height)
{
  double area;
  area=(b_maj+b_min)*height/2;
  cout << "Major base: " << b_maj << endl;
  cout << "Minor base: " << b_min << endl;
  cout << "Height: " << height << endl;
  cout 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

Esempio



Otteniamo il seguente output:

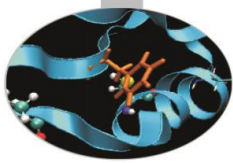
```
Major base: 1
Minor base: 1
Height: 1
The area is: 1

Major base: 2.5
Minor base: 1
Height: 1
The area is: 1.75

Major base: 4
Minor base: 1.5
Height: 1
The area is: 2.75

Major base: 6
Minor base: 2
Height: 3.2
The area is: 12.8
```



Template di funzioni

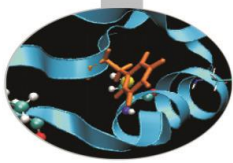
- **Template di funzioni non sono supportati dal C**
- Quando si rende necessario definire delle funzioni le cui operazioni siano le stesse per ogni tipo di dato, piuttosto che ricorrere all'overloading è preferibile utilizzare un ***template di funzioni*** o ***funzione generica*** che definisce, di fatto, un'intera famiglia di funzioni.
- Sulla base del tipo di argomenti presenti nella chiamata alla funzione associata al template, il compilatore è in grado, partendo dalla definizione generica del template stesso, di creare la funzione corrispondente al tipo di parametri presenti nel codice.

```
template <typename tipo_generico>
tipo_restituito nome_funzione(lista argomenti){

      corpo della funzione generica
}
```



Template di funzioni

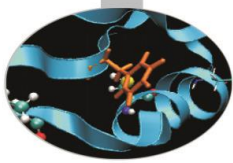
- Naturalmente il tipo restituito può coincidere con il tipo generico e la lista degli argomenti può contenere parametri generici e non.
- Nella definizione di un template la parola chiave ***typename*** può essere sostituita da ***class***.



Esempio

esempio1: scriviamo un programma che calcoli il minimo tra due variabili di qualunque tipo.

```
#include<iostream>

using namespace std;

template <typename T>

void minimo( T val_1, T val_2){

  T min = val_1 ;

  if (val_2 < val_1)

      min = val_2;

  cout << "Il minimo tra " << val_1 << " e "

       << val_2 << " e' "<< min << endl;

}     // continua
```



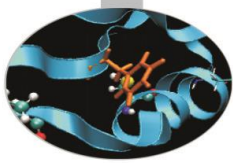
Esempio

```
// … segue
int main(){
   int int_1=2, int_2=20;
   double dbl_1=7.6, dbl_2=1.34 ;
   char c_1='m', c_2='b';
   minimo(int_1, int_2);
   minimo(dbl_1, dbl_2);
   minimo(c_1, c_2);
   return 0; }
```

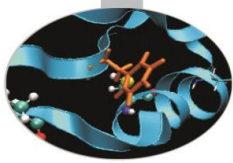
output:

```
Il minimo tra 2 e 20 e' 2

Il minimo tra 7.6 e 1.34 e' 1.34

Il minimo tra m e b e' b
```

Template di funzioni



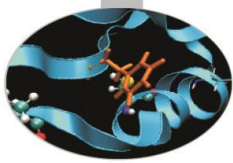
- E' possibile eseguire l'*overloading* di una funzione generica: in tal caso il tipo di dati passati viene dichiarato esplicitamente. Quando il compilatore incontra una chiamata ad una funzione generica per i cui parametri esiste una versione "overloaded", è quest'ultima ad essere invocata.



Esempio

esempio2: riscriviamo il programma precedente aggiungendo l'overloading della funzione minimo per passaggio di dati di tipo char.

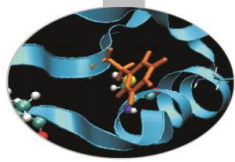
```
#include<iostream>
using namespace std;
template <typename T>

void minimo( T val_1, T val_2)
{
  T min = val_1 ;
  if (val_2 < val_1)
    min = val_2;
  cout << "Il minimo tra " << val_1
       << " e " << val_2
       << " e' "<< min << endl; } // continua …
```



Esempio

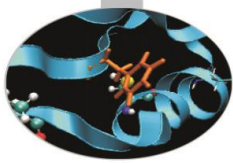
```
// … segue
// overloading della funzione minimo per il passaggio di
// dati di tipo char
void minimo(char c1, char c2){
   char min = c1;
   if(c2 < c1)
     min = c2;
   cout << "Tra " << c1 << " e " << c2
        << " viene prima " << min << endl;}
int main(){
   int int_1=2, int_2=20;
   double dbl_1=7.6, dbl_2=1.34 ;
   char c_1='m', c_2='b';
   minimo(int_1, int_2);
   minimo(dbl_1, dbl_2);
   minimo(c_1, c_2);
   return 0;}
```

Esempio



- L'output del programma è, in questo caso, decisamente più appropriato:

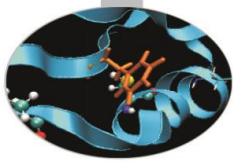
```
Il minimo tra 2 e 20 e' 2
Il minimo tra 7.6 e 1.34 e' 1.34
Tra m e b viene prima b
```



Template di funzioni

- Un template di funzioni può contenere più di un dato generico (*multitemplate*). La definizione di un *multitemplate* di funzioni diventa:

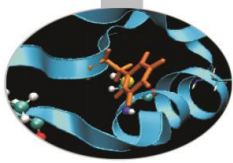
```
template<class tipo_generico1, … ,class tipo_genericoN>
tipo_restituito  nome_funzione(lista argomenti){
        corpo della funzione generica
}
```



Esempio

esempio3: scriviamo un programma che calcoli la somma di due elementi di tipo diverso. La somma dovrà essere del tipo associato al primo elemento nella lista dei parametri della funzione.

```
#include<iostream>
using namespace std;
template<class F, class S>
F sum(F first, S second) {
        F somma;
        somma=first+second;
        return somma;
}
int main() {
        int v_int=20;
        char v_char='a';
        float v_double=32.4;          // continua …
```



Esempio

```
// … segue
cout << "The sum of " << v_char << " and " << v_int
     << " is " << sum(v_char, v_int) << endl;
                          // sum restituisce un char
cout << "The sum of " << v_int << " and " <<
v_double
     << " is " << sum(v_int, v_double) << endl;
                          // sum restituisce un int
cout << "The sum of " << v_double << " and " <<
v_char
     << " is " << sum(v_double, v_char) << endl;
                          // sum restituisce un double
return 0;
}
```

output:

```
The sum of a and 20 is u
The sum of 20 and 32.4 is 52
The sum of 32.4 and a is 129.4
```