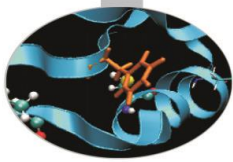
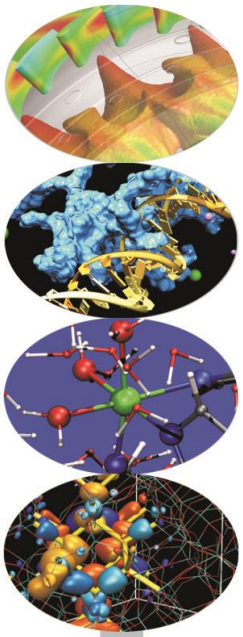
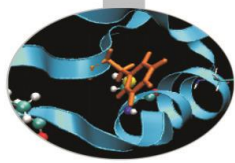


Template

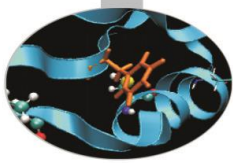




Indice

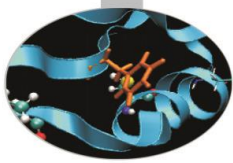
- Definizione.
- Utilizzo dei templates di classe.
- Esempi.
- Commento sulla performance della programmazione OO e l'uso dei templates.

Templates

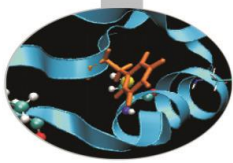


- Seguendo l'idea per cui concetti separati devono essere rappresentati separatamente e combinati solo all'occorrenza, i Templates permettono, in maniera semplice e pulita, di rappresentare e combinare tra loro un ampio spettro di concetti generali.
- Il C++ fornisce un meccanismo, i templates, che permette ad un tipo di dato di essere un parametro per la definizione di una classe o di una funzione.
- In questo senso i templates forniscono un supporto diretto alla programmazione generica, o anche al poliformismo a compile time.

Templates



- Intuitivamente si pensi ad un template di classe per gestire in maniera omogenea e generale la definizione di vettori che contengono interi, double, long double…
- Il meccanismo dei templates è differente da quello dell'overloading che invece richiede differenti implementazioni alternative.
- Tuttavia come l'overloading anche il template richiede che tutti i parametri siano poi noti a tempo di compilazione.
- I templates hanno caratteristiche tali per cui, se usati seguendo alcuni paradigmi, permettono di ovviare a problemi di efficienza presenti ad esempio nel polimorfismo in esecuzione.

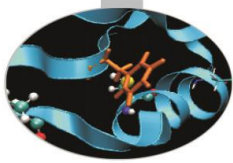


class templates

La sintassi è la seguente:

```
template <class Tipo_fittizio> class nome_classe {
                //corpo del template: funzioni,
                //variabili
                        };
//alternativamente:
template<typename T> class nome_classe{
                 //corpo del template: funzioni,
                       //variabili
                };
```

In questo modo si dichiara un template di parametro T.



class templates

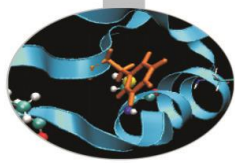
- Una volta creata una classe generica per generare una istanza specifica si utilizza la sintassi:

```
nome_classe<tipo_effettivo> nome_oggetto;
```

- Il numero di istanze possibili che differiscono per il tipo effettivo è illimitato e dipende solo dalla necessità del problema:

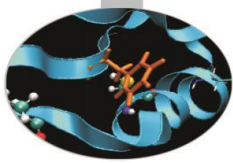
```
nome_classe<tipo_effettivo1> nome_oggetto1;
nome_classe<tipo_effettivo2> nome_oggetto2;
nome_classe<tipo_effettivo3> nome_oggetto3;
```

- Le funzioni membro di una classe generica sono anch'esse generiche senza specificarlo tramite la parola chiave template.
- All'interno dello scope definito dal costrutto di template non è necessario utilizzare gli specificatori di scopo (nome_classe<T>::) (come per le classi normali) ma la ridondanza non è ovviamente errore.



Esempio

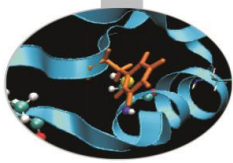
```
#include<stdlib.h>
#include <iostream.h>;
template <class T> class stack {
   T v[1000];
   int top;
 public:
   stack() { top=0; }
   void push(const T &s) { v[top] = s; top++; }
   T pop() { top --; return v[top]; }
   bool empty() { return (top==0); }
   void unused(){cout<<"i'm just a place keeper\n";}
 };
```



Esempio

```
int main(int argc, char **argv){
   stack<float> s;
   for (int i=1; i<argc; i++) s.push(atof(argv[i] ));
   while ( !s.empty( )) cout << s.pop ( ) << endl;

}
```

- Il compilatore genera la dichiarazione dell'oggetto `stack<float>` con il costruttore, e tutte le funzioni utilizzate.
- Le funzioni non utilizzate (`void unused()`) non vengono generate.

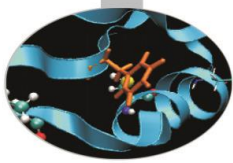


Parametri

- Come per le funzioni anche per le classi generiche è possibile avere più di un tipo di dato generico alla volta.

```
template<class A, class B, C val, int i> class multi{
   A v[i];
   int size;
   C d = val;
public:
multi(): size(i) {cout << "generato oggetto multi\n";}
};
```

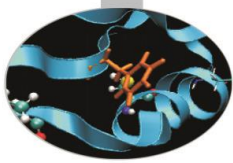
- L'unico vincolo sui parametri da inserire nella dichiarazione di un template è che questi siano determinabili a tempo di compilazione.



Equivalenza di tipi

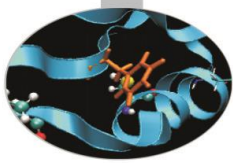
- Dato un template possono essere definiti diverse specializzazioni (in base ai tipi passati) e ognuna di queste è di fatto un tipo differente, tranne quelli definiti tramite degli alias.

```
stack<long double> ldstk;
stack<unsigned int> uistk(10);
stack<int> istk(10);
typedef unsigned int Uint;
stack<Uint> uistk(10); //è lo stesso tipo di
                       //stack<unsigned int>
```



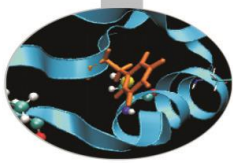
Esempio

```
//esempio, array con controllo sul limite
#include<iostream.h>
#include < stdlib.h>
const int size=10 ;
template <class T> class tipo {
   T v[size];
 public:
   tipo ( ) {  // costruttore
   register int i;
   for (i=0; i< size; i++) v[i]=i;}
   T  &operator[ ] (int i);    // overload di[ ]
};
```



Esempio

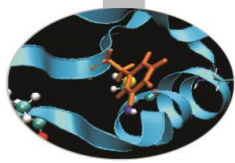
```
/* questa implementazione dell'overload di operatore  [ ]
   fornisce il controllo desiderato sul limite dell'array */

template< class T> T & tipo <T> :: operator [ ] (int i){
   if (i<0 || i>size -1) {
      cout << i << "!!! out of limit !!! " << endl;
      exit (1);
        }
  return v[i];
}
```



Esempio

```
int main ( ) {
   tipo < int> obj_int;
   tipo <char> obj_char;
   int j;
   for (j=0; j<size; j++) {
        cout << obj_int[j] << endl;
        obj_char [j] = 'x';
        }
   for (j=0; j<size; j++) cout << obj_char [j];
   cout << endl;
   obj_char [11]= 'a';  /* genererebbe un errore run-
                           time senza controllo */
}
```

Organizzazione del codice

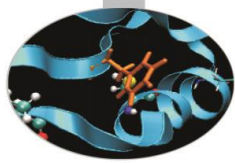


Ci sono 2 modalità di compilazione nel codice dei template:

- Per inclusione;
- Per separazione;

Nel primo caso si pone sia la dichiarazione che la definizione del template di classe nel file header.

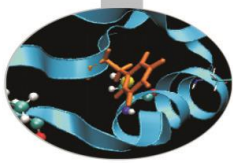
Nel secondo caso si distingue tra dichiarazione e definizione del template di classe. Nel file header vengono poste la dichiarazione (come **export**) e le definizioni delle funzioni inline. Nel file di porgramma sono invece poste le definizioni delle funzioni non inline.



Organizzazione del codice

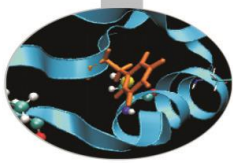
```
// file notifica.h: solo la dichiarazioneed export
export template<class T> void notifica(const T & );

// file notifica.cpp: definizione
#include <cstdlib>
#include <iostream>
#include "notifica.h"
template<class T> void notifica(const T & t) {
std::cout <<"\nNotificato valore:\n "<< t <<endl;
std::exit(1);
}
```

Specializzazioni



- Esistono contesti in cui non è possibile ottenere un carattere di generalità per tutti i tipi di dati istanziabili a partire da un template.
- Ad esempio alcune funzioni, o alcune operazioni contenute nelle funzioni potrebbero non avere senso.
- In questo caso la soluzione che viene fornita dal linguaggio consiste nel definire per quelle funzioni delle versioni specializzate su un particolare tipo di dato.
- Sarà compito del compilatore utilizzare queste versioni specializzate al posto di quelle generali.

```
template<class T> class stack; // generale
template<class T> class stack< complex<T> >;
//specializzazione per i complessi
```

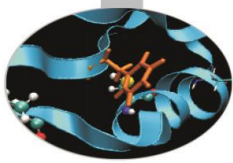


Specializzazioni

Possono peraltro esistere diversi livelli di specializzazione:

```
template<class T> class stack; // generale
template<class T> class stack< complex<T> >;
// specializzazione per i complessi
template<> class stack< complex<float> >;
// specializzazione per i complessi in singola precisione

//chiamata alla prima specializzazione
stack< complex<double> > cdstk(10);
//chiamata alla seconda specializzazione
stack< complex<float> > cdfstk(10);
```

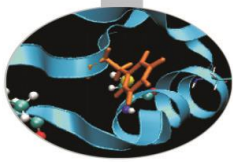
Specializzazioni



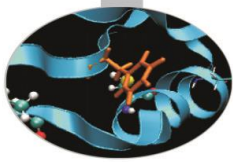
- Le specializzazioni complete di un template di classe impongono di ridefinire tutte le funzioni ed i dati membro della classe template generale.
- Esistono contesti in cui si vuole fornire solo la specializzazione di alcune funzioni membro. In questo caso si parla di specializzazioni esplicite

```
template<> LongDouble stack<LongDouble>::min(){
//corpo della funzione
}
/* le istanze di oggetti di classe stack<LongDouble> fanno uso di
   questa definizione di min()e non di quella presente nella
   definizione generale del template */
```

Performance della programmazione OO e uso dei templates



- Le funzioni virtuali ed il polimorfismo dinamico sono uno strumento potente e agile per la programmazione di sistemi complessi; tuttavia possono presentare problemi di efficienza qualora le chiamate alle funzioni polimorfiche siano troppo frequenti e le istruzioni eseguite troppo semplici.
- In questi casi esistono tecniche alternative, basate sull'uso dei template di classe per superare questi ostacoli.
- Il risultato che si ottiene è OVVIAMENTE un polimorfismo statico e non più dinamico.
- Di seguito un esempio legato alla computazione (problemi in cui la dinamica evolve in maniera abbastanza prevedibile, diversamente dai S.O) illustra due proposte.



Performance della programmazione OO e uso dei templates

- Si supponga di avere una funzione per risolvere un sistema lineare per una matrice generica e che la si voglia poter utilizzare anche per matrici particolari (sparse, a banda, diagonali)
- Non volendola definire come virtuale ed attivare il meccanismo del polimorfismo dinamico, è possibile piuttosto definire una template di classe base per le matrici generiche parametrica nel parametro T che identifica il tipo di matrice e poi derivare da essa altri template di classe per matrici particolari.
- Alternativamente e molto più semplicemente è possibile definire un template di funzione in cui il parametro è qualunque tipo di matrice.