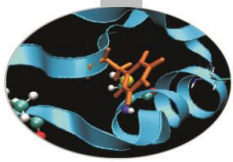
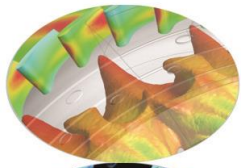
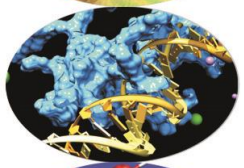
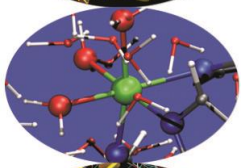
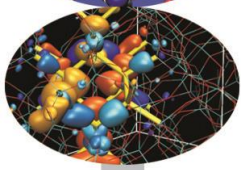
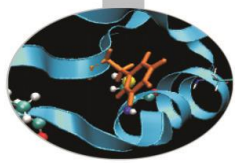


Template

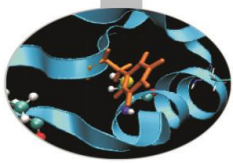





Indice



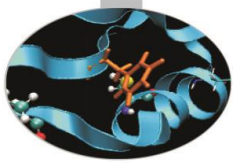
- Definizione.
- Utilizzo dei template di classe.
- Esempi.
- Commento sulla performance della programmazione OO e l'uso dei template.

Template

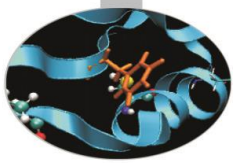


- Il meccanismo dei template rende disponibile la tecnica della programmazione generica. Un template non è altro che codice parametrico
- Con i templates un tipo di dato diventa un parametro per la definizione di una classe.
- Con un'unica definizione si hanno a disposizione versioni diverse della classe in grado di operare su differenti tipi di dati
- In questo senso i templates forniscono un supporto diretto alla programmazione generica, o anche al poliformismo a compile time (statico).

Template



- Sono utili quando una classe contiene operazioni logiche generalizzabili. Per esempio strutture dati di tipo stack di interi, caratteri o stringhe, funzionano con gli stessi algoritmi.
- 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 permettono di ovviare a problemi di efficienza presenti ad esempio nel polimorfismo in esecuzione.

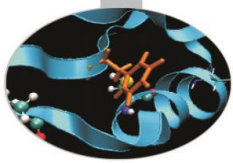


Template di classe

La sintassi è la seguente:

```
template<typename T> class nome_classe{

                //corpo del template: funzioni,
                      //variabili

               };


//alternativamente:

template <class T> class nome_classe {

               //corpo del template: funzioni,
               //variabili

                      };
```

In questo modo si dichiara un template di parametro T dove T è un segnaposto a cui verrà sostituito il nome del tipo al momento della creazione di un'istanza della classe.



Template di classe

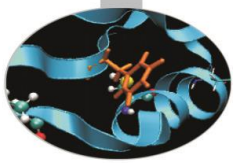
- Una volta creata una classe generica per generare una istanza specifica si utilizza la sintassi:

```
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:

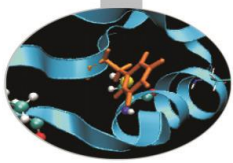
```
nome_classe<tipo_effettivo1> nome_oggetto1;
nome_classe<tipo_effettivo2> nome_oggetto2;
nome_classe<tipo_effettivo3> nome_oggetto3;
```

- Le funzioni membro di una classe generica sono anch'esse generiche senza specificarlo tramite la parola chiave template.



Esempio

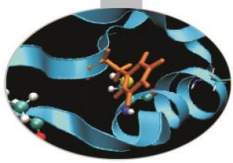
```
#include<stdlib.h>
#include <iostream.h>;
template <typename T> class stack {
   T v[1000];
   int top;
 public:
   stack() { top=0; }
   void push(const T &s) { v[top] = s; top++; }
   T pop() { top --; return v[top]; }
   bool empty() { return (top==0); }
 };
```



Esempio

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

}
```

- Il compilatore genera la dichiarazione dell'oggetto `stack<float>` con il costruttore, e tutte le funzioni utilizzate.

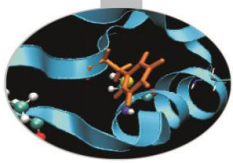


Parametri

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

```
template<typename A, typename B, int i, int val> class multi{
   A v[i];
   B v2[val];
   int size;
public:
multi(): size(i) {
   cout << "generato oggetto multi\n";
}
};
```

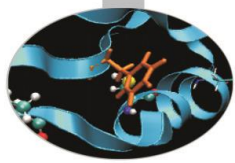
- L'unico vincolo sui parametri da inserire nella dichiarazione di un template è che questi siano determinabili a tempo di compilazione.



Equivalenza di tipi

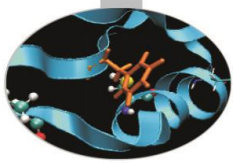
- Dato un template possono essere definiti diverse specializzazioni (in base ai tipi passati) e ognuna di queste è di fatto un tipo differente, tranne quelli definiti tramite degli alias.

```
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>
```



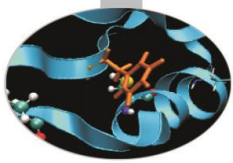
Esempio

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



Esempio

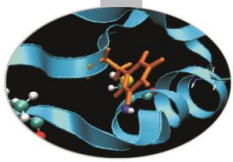
```
/* questa implementazione dell'overload di operatore  [ ]
   fornisce il controllo desiderato sul limite dell'array */

template< typename 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

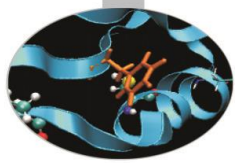


Ci sono 2 modalità di compilazione nel codice dei template:

- Per inclusione;
- Per separazione;

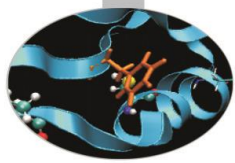
Nel primo caso si pone sia la dichiarazione che la definizione del template di classe nel file header (.hpp).

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



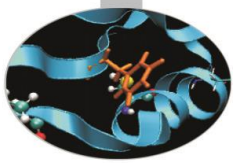
Organizzazione del codice

```
// file myTemplate.hpp:
#ifndef TEMPLATE_H
#define TEMPLATE_H
#include <iostream>
template <typename T> class myTemplate
{
  private:
  T data;
  public:
  myTemplate();
  myTemplate(T t);
  T getData() const;
  void displayData() const;
  static int someValue;
};
#endif
```



Organizzazione del codice

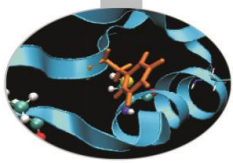
```
#include <iostream>
#include "myTemplate.hpp"

//template functions
template<typename T> myTemplate<T>::myTemplate():data(){}
template<typename T> myTemplate<T>::myTemplate(T t)
{ data = t;}
template <typename T> T myTemplate<T>::getData() const
{return data;}
template <typename T>void myTemplate<T>::displayData() const
{std::cout << data <<std::endl;}
template<typename T> int myTemplate<T>::someValue = 100;
//The explicit instantiation part
template class myTemplate<int>;
template class myTemplate<float>;
```



Organizzazione del codice

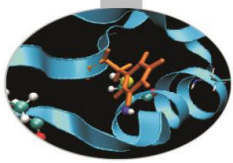
```
#include <iostream>
#include "myTemplate.hpp"
using namespace std;
int main()
{
    myTemplate<int> myTi(5);
    myTi.displayData();
    myTemplate<float> myTf(3.5);
    myTf.displayData();

    return 0;
}
```

Specializzazione



- 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<typename T> class stack{…}; // generale
template<typename T> class stack< complex<T> >{…};
//specializzazione parziale per i complessi
```

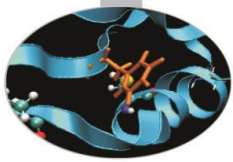


Specializzazione

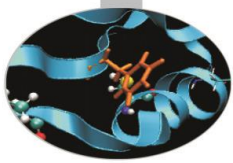
Possono peraltro esistere diversi livelli di specializzazione:

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

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

Specializzazione – Template di classe

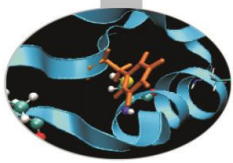


```
template<typename T> class Vcr {
int lenth;
T* vr;
public:
Vcr(int, T*); // costruttore
Vcr(const Vcr&); // costruttore di copia
~Vcr(); // distruttore
int size();
T& operator[](int i); // overloading operatore[]
T maxnorm () const;
T twonorm() const;
};
```



Specializzazione parziale

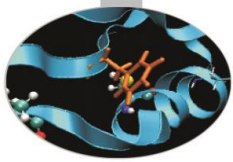
```
template<typename T> class Vcr< complex<T> > {
int lenth;
Complex<T>* vr;
public:
Vcr(int, complex<T>*); // costruttore
Vcr(const Vcr&); // costruttore di copia
~Vcr(); // distruttore
int size();
complex<T>& operator[](int i); // overloading operatore[]
T maxnorm () const;
T twonorm() const;
};
```



Specializzazione completa

```
template<> class Vcr< complex<double> > {
int lenth;
Complex<double>* vr;
public:
Vcr(int, complex<double>*); // costruttore
Vcr(const Vcr&); // costruttore di copia
~Vcr(); // distruttore
int size();
complex<double>& operator[](int i); // overloading
   operatore[]
T maxnorm () const;
T twonorm() const;
};
```

Commento sulla performance della programmazione OO e l'uso dei templates



- Le funzioni virtuali ed il polimorfismo dinamico sono uno strumento potente e versatile. 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 un polimorfismo statico e non più dinamico.