



UNIVERSITÀ
degli STUDI
di CATANIA

Classi template

Corso di programmazione I AA 2019/20

Corso di Laurea Triennale in Informatica

Prof. Giovanni Maria Farinella

Web: <http://www.dmi.unict.it/farinella>

Email: gfarinella@dmi.unict.it

Dipartimento di Matematica e Informatica

Introduzione ai template in C++

In C++ è possibile definire **classi e funzioni** template, che operano su **tipi generici**.

I template consentono al programmatore di trattare i tipi dei dati sui quali operazioni funzioni e classi come **parametri**.

I template esistono per consentire la riusabilità del codice **rispetto a tipi differenti** (user-defined e primitivi):

- strutture dati (ES: pile, liste, code)
- algoritmi (ordinamento, ricerca, etc)
- ...

Definizione e uso di funzioni template

Una **funzione generica** o funzione template opera su un tipo di dato "generico".

Si usa la parola chiave **template**, seguita dalla lista di tipi generici che si intende utilizzare.

```
1  template < class T, class C >
2      void foo1(T &a, C &b) { a=a/2; b*=2; }
3  template < typename T, typename C >
4      void foo2(const T& a, const C &b) {
5          cout << a << b << endl; }
6  template < class T, typename C >
7      void foo3(T &a, C b) {
8          a=a*2; cout << a << b << endl; }
```

Definizione e uso di funzioni template

Una funzione template è una sorta di **modello** dal quale il compilatore genera opportune **specializzazioni** in **corrispondenza di chiamate a funzione** con parametri differenti.

Il processo di generazione delle specializzazioni si chiama **template instantiation**.

```
1  template < class T, class C >
2    void foo(T &a, C &b) { a=a/2; b*=2; }
3  //...
4  foo(4, 10); // specialization: foo(int, int)
5  foo(3.4f, 6.7f); // specialization: foo(float, float)
6  foo(8.10f, 8.8f);
```

Osservazioni:

- Il programmatore **non deve predisporre una intera famiglia di funzioni overloaded** in corrispondenza di tutti i possibili tipi.
 - Di conseguenza si riduce il lavoro del programmatore.
- Le specializzazioni saranno **generate solo in corrispondenza di opportune chiamate:**
 - Di conseguenza il codice generato dal compilatore si riduce parecchio

Definizione e uso di funzioni template

ESEMPIO

```
1  template <class T> void swap(T &x, T&y){
2      T tmp = x;
3      x = y;
4      y = tmp;
5  }
```

VS

```
1  void swap(char &x, char &y){ char tmp = x; /* ... */ }
2  void swap(int &x, int &y){ int tmp = x; /* ... */ }
3  void swap(double &x, double &y){ double tmp = x; /* ... */ }
```

Definizione e uso di funzioni template

Binding tipo del parametro attuale a tipo generico viene detto **argument deduction**.

Argument deduction avviene a tempo di compilazione in corrispondenza delle invocazioni della funzione template.

Ogni tipo generico deve essere presente nella **lista dei parametri formali** della funzione, altrimenti errore a compile-time.

```
1  template < class T >
2      char foo(int k, string s) // nessun param. di tipo T
3      { return s[k % s.length()]; }
4      //...
5      char c = foo(2, "pippo"); // Compile-time error
```

Definizione e uso di funzioni template

Ancora su argument deduction.

```
1  template <class T>
2    void print(T a, T b)/
3      { cout << a << " , " << b; }
4    //...
5    print(10.5f, 50); // Compile-time error
```

A differenza di una qualsiasi funzione “normale”, in cui il compilatore opera eventuali promozioni e/o conversioni (es: *float* → *int*, in questo caso a e b devono essere di equal tipo.

In caso contrario il compilatore non riesce a selezionare un tipo per T (float o int?).

Esempi Svolti

31_01.cpp – Esempio di funzioni template,
errori di argument deduction

Funzioni template e overloading

È possibile fare overloading anche con le funzioni template

```
1 template <class T> void print(T a, T b); // (1)
2 template <class C, class T> void print (T a, C b); // (2)
```

In questo caso il compilatore:

- genera specializzazioni dal template no.1 in corrispondenza di chiamate con argomenti a e b di egual tipo
- genera specializzazioni dal template no.2 in corrispondenza di chiamate con argomenti a e b di tipo differente

Esempi Svolti

31_02.cpp – Overloading di funzioni template

Funzioni template e overloading

È possibile definire una o più funzioni non generiche:

```
1 template <class T> void print(T a, T b); //(1)
2 template <class C, class T> void print (T a, C b); //(2)
3 void print (double a, double b); //(3)
```

In questo caso il compilatore:

1. cerca corrispondenza esatta parametri attuali con una funzione non generica;
2. tenta applicazione argument deduction su una delle funzioni template;
3. usa algoritmo di risoluzione oveload SOLO sulla famiglia di funzioni non generiche;

Funzioni template e overloading

```
1  template <class T> void print(T a, T b); //(1)
2  template <class C, class T> void print (T a, C b); //(2)
3  void print (double a, double b); //(3)
```

- il template `print(T, C)` sarà impiegato per generare tutte le specializzazioni in corrispondenza delle chiamate `print(t1, t2)` con `typeid(t1) ≠ typeid(t2)`
- il template `print(T, T)` sarà impiegato per generare tutte le specializzazioni in corrispondenza delle chiamate `print(t1, t2)` con `typeid(t1) = typeid(t2)` (**ma che non siano double**)
- la funzione `print(double, double)` sarà invocata per tutte le chiamate `print(t1, t2)` con parametri attuali di tipi `double`

Esempi Svolti

31_03.cpp – Overloading di funzioni template

Funzioni template e overloading

NOTA: Per ogni tipo T usato nelle invocazioni della funzione, operatori usati sul tipo **devono essere overloaded nel tipo T**.

```
1  template <class T>
2  T max(T p1, T p2){
3      if( p1 < p2 )
4          return p2;
5      else return p1;
6  }
```

Definizione e uso di classi template

Una classe generica specifica una **famiglia di definizioni di classi**.

Definizione analoga a quella di una funzione template.

```
1  template<typename T>
2      class Stack{
3          //...
4          T *ptr;
5          //...
6          void push(T);
7      };
```

Definizione e uso di classi template

Il tipo `T` è usato per dati interni alla classe e per i parametri e i dati locali delle funzioni membro.

```
1  template<typename T>
2      class Stack{
3          //...
4          T *ptr;
5          //...
6          void push(T);
7      };
```

Definizione e uso di classi template

Funzioni membro sono dichiarate e definite esattamente con le funzioni membro della classi “normali”.

Nel caso in cui si definisca il corpo della funzione membro fuori dalla dichiarazione, allora **va dichiarato anche il template**.

```
1  template <class T> // template della classe
2  void Stack<T>::Push(T dato)
3      {
4          if (top < dimensione)
5              ptr [top++] = dato;
6      }
```

Definizione e uso di classi template

Una classe generica può anche contenere funzioni membro generiche.

```
1  template <class T>
2  class Stack{
3      //...
4      template<class C>
5          void f(C item);
6  }
7  //...
8  template <class T> // template della classe
9      template <class C> // template della funzione
10     void Stack<T>::f(C c){
11         cout << c << " , " << Pop() << endl;
12     }
```

Definizione e uso di classi template

Per creare istanze di oggetti di classi template, basta specificare i tipi di dato **tra parentesi angolari**.

```
1  template <class T>
2  class Stack{
3      //...
4  }
5  //...
6  Stack<int> Interi;
7  Stack<double> Reali;
```

Esempi Svolti

31_04.cpp – Istanze classi template e loro uso

31_05.cpp – Istanze classi template e funzioni membro template

Definizione e uso di classi template

I parametri delle classi template possono essere anche di uno dei tipi di dati primitivi (ES: int):

- Gli argomenti corrispondenti nelle dichiarazioni di oggetti debbono essere **costanti**;
- È anche possibile specificare argomenti standard per i parametri;

```
1 template <class T, int DIM=10>
2 Stack{
3     T ptr [DIM];
4     //...
5 }
6 Stack<int, 50> Interi;
7 Stack<double> Reali; // equiv. a Stack<double, 10> Reali
```

Esempi Svolti

31_06.cpp – Parametri dei template tipi primitivi

31_07.cpp – Parametri dei template tipi primitivi e valori standard

FINE