I TEMPLATE IN C++

Roberto Nardone, Luigi Romano

ROADMAP

- Concetto di Genericità
- Esempio di Genericità con Funzioni
- Esempio di Genericità con Classi
- I Template di Funzioni
- I Template di Classi
- Modelli di Compilazione di Template
- Template e Polimorfismo

CONCETTO DI GENERICITÀ

- La genericità è una tecnica di programmazione che agevola il riuso del software
- Grazie a questa, lo sviluppatore può definire una classe (o funzione) senza specificare il tipo di dato di uno o più dei suoi membri
- Nel mondo della programmazione è fondamentale poter definire classi (o funzioni) che lavorano su tipi generici
- In questo modo la classe (o funzione) potrà essere riutilizzata per altri scopi senza doverla riscrivere
- Gli algoritmi di risoluzione di numerosi problemi non dipendono dal tipo di dato da elaborare

ESEMPIO DI GENERICITÀ CON FUNZIONI – 1/2

- Capita spesso che le funzioni spesso sono riutilizzate con parametri differenti.
- Ad esempio, una funzione che scambia tra loro i valori di due variabili si deve reimplementare per ogni tipo di coppia di dati

```
void scambia(int &m, int &n){
    int aux=m;
    m=n;
    n=aux;
}
```

- Se volessimo realizzare lo scambio per valori di tipo **char** dovremmo sovraccaricare (overload) la funzione

ESEMPIO DI GENERICITÀ CON FUNZIONI – 2/2


```
void scambia(char &m, char &n){
    char aux=m;
    m=n;
    n=aux;
}
```

- Se volessimo applicarlo a coppie di variabili **float** o **double** dovremmo scrivere ancora altro codice
- Sarebbe dunque utile avere **funzioni parametriche** che permettano di creare una funzione alla quale sia possibile passare parametri di qualunque tipo con una sintassi del genere:

```
void Scambio(tipo_variabile &m, tipo_variabile &n){
    tipo_variabile aux=m;
    m=n;
    n=aux;
}
```

Esempio di Genericità con Classi – 1/2

- Supponiamo di dover gestire una lista di contatti telefonici
- A tale scopo, realizzeremo una classe **ElencoTelefonico**
- Tale classe offre funzioni per effettuare:
  - *Ricerca, Inserimento, Cancellazione, Modifica*
- Un oggetto **ElencoTelefonico** contiene oggetti di tipo **Utente** (definito attraverso una **struct** contenente attributi come *nome*, *cognome*, *indirizzo*, numero, ecc.)
- Supponiamo ora di voler realizzare una classe **ElencoCD** che necessita delle stesse operazioni della lista di contatti telefonici
- In questo caso la classe **ElencoCD** conterrà oggetti di tipo **CD**

ESEMPIO DI GENERICITÀ CON CLASSI – 2/2

- Potremmo riutilizzare il codice scritto per **ElencoTelefonico**
- Una possibilità sarebbe di copiare la classe **ElencoTelefonico** nella classe **ElencoCD** sostituendo le occorrenze di oggetti **Utente** con oggetti di tipo **CD**
- **Problema:** in questo caso il riuso del codice aumenta la produttività ma richiede il **mantenimento di due copie di codice quasi identico!**
- Soluzione: la genericità è di aiuto in questo caso. Se definiamo **Elenco** come **classe parametrica**, significa che almeno una delle classi utilizzate all'interno di **Elenco** non deve necessariamente essere assegnata fino al momento della compilazione

I TEMPLATE DI FUNZIONI

- In C++ le **Funzioni Parametriche** sono definite attraverso i **Template di Funzioni**
- Un template di funzioni è un insieme indeterminato di funzioni sovraccaricate che descrive l'algoritmo specifico di una funzione generica.
- Ogni funzione specifica di questo insieme è un'istanza del template di funzione prodotta automaticamente dal compilatore quando necessario
- Supponiamo di voler scriver una funzione **min(a, b)** che restituisca il valore più piccolo dei suoi argomenti per qualunque tipo di coppia di dati

SINTASSI DEI TEMPLATE DI FUNZIONI

- L'idea è rappresentare il tipo di dato utilizzato dalla funzione con un nome che rappresenti "*qualunque tipo*"
- Normalmente questo "*qualunque tipo*" si rappresenta con *T*, ma si può utilizzare qualunque identificatore diverso da una parola riservata
- La sintassi di un template di funzioni ha due formati, a seconda che si utilizza la parola chiave **class** o **typename**

```
template <class T>
T func(T parametro)
{
    ...
}
```

```
template <typename T>
T func(T parametro) {
    ...
}
```

- T è detto **argomento del template** che rappresenta il tipo parametrico

REGOLE

- Le intestazioni dei template di funzioni non si possono salvare in un file header *nome.h* con le definizioni in un file *nome.cpp* compilato separatamente.
  - Devono essere entrambi nello stesso file. Vedremo meglio in seguito.
- I template di funzioni possono avere più di un parametro di tipo
- La lista di parametri non può essere vuota

USO DEI TEMPLATE DI FUNZIONI

- Scriviamo dunque il template di una funzione che restituisce il minimo tra due valori in input

```
template <typename T> T minimo(T a, T b) {
    if(a<b) return a;
    return b;
}
int main(){
  int min_value=minimo<int>(4,6); //oppure =minimo(4,6)
}
```

- Quando il compilatore troverà una chiamata della forma *minimo(a,b)* istanzia la funzione minimo() a partire dai tipi di parametri *a* e *b* utilizzati nella chiamata della funzione

IMPLEMENTAZIONE ESEMPIO DI SCAMBIO

- Si riporta un esempio sull'uso dei template di funzione per lo scambio dei valori di due variabili

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

template <class T>
    void scambia(T& v1, T& v2){

    T aux;
    aux=v1;
    v1=v2;
    v2=aux;
}

int main (){
    //Scambio di interi
    int numero1=5;
    int numero2=8;

    cout<<"valori originali: "<<numero1<<" "<<numero2<<endl;
    scambia(numero1,numero2);
    cout<<"valori scambiati: "<<numero1<<" "<<numero2<<endl;

    //Scambio di caratteri
    char car1='a';
    char car2='b';

    cout<<"valori originali: "<<car1<<" "<<car2<<endl;
    scambia(car1,car2);
    cout<<"valori scambiati: "<<car1<<" "<<car2<<endl;

    return 0;
}
```

VARIANTI DEI TEMPLATE DI FUNZIONI

- E' possibile dichiare un template che tiene conto del fatto che I parametri non saranno modificati dalla funzione:

```
template <class T> const T& min(const T& a, const T& b) {
    if(a<b) return a;
    return b;
}
```

- E' possibile dichiare due parametri di tipo T ma con la condizione che siano diversi:

```
template <class T1, class T2> T1 func(T1 a, T2 b) { ... }
```

- Anche le funzioni template possono essere dichiarate **extern**, **inline**, **static**
- **Nota** → Lo specificatore si colloca dopo la riga dei parametri formali. Mai prima della parola riservata template.

```
template <class T> extern T func(T a, T b) { ... }
```

TEMPLATE DI STRUCT

- In modo simile, lo sviluppatore può definire template di struct:

```
template <typename T> struct Punto{
            T x;
            T y;
};

int main(){
  Punto<int> pt={45,15};
}
```

- **Nota** → Osservare che quando si parla di un tipo parametrico T, questo non è limitato ai tipi di dato predefiniti ma contempla anche tipi definiti dall'utente

I TEMPLATE DI CLASSI

- In C++ le **Classi Parametriche** sono definite attraverso i **template di classi**
- Il template di classe è uno schema di classe generica, che non fa esplicito riferimento a nessun particolare tipo di dati membro
- Consente di progettare un'unica classe per implementare un tipo di dato astratto, indipendente dal tipo di dato contenuto al suo interno
- All'interno del codice del programma ci sarà una diversa classe per implementare il tipo di dato astratto per ogni possibile tipo di dato membro

SINTASSI DEI TEMPLATE DI CLASSI

- La generazione di tutte le classi e dei suoi metodi è a carico del **compilatore**. Il compilatore provvede a produrre da questo schema una diversa classe per ogni tipo di dato membro
- Un template di classi ha la seguente sintassi:

```
template <typename T> class classeParametrica { ... }
```

```
template <class T> class classeParametrica { ... }
```

- Come per i template di funzioni, T è il tipo utilizzato dal template

ESEMPIO TEMPLATE DI CLASSI

- Definizione di una classe generica Array:

```
template <typename T>
class Array{
        private:
            T* pa;
            int dimensione;
        public:
            Array(int n){
                pa= new T[n];
                dimensione=n;
            }
            ~Array(){delete [] pa;}
};
...
Array<Punto> array_di_punti(10);
Array<int> array_di_interi(32);
```

MODELLI DI COMPILAZIONE DI TEMPLATE – 1/2

- Quando si definisce un oggetto di tipo classe, la definizione della classe deve essere presente ma non sono invece necessarie le definizioni delle funzioni membro.
- Si usa quindi mettere le definizioni delle classi negli header file (.h) e le definizioni dei metodi nei file sorgenti (.cpp)
- Nel caso dei template ciò non è possibile!!!
- I template infatti sono diversi. Quando il compilatore vede un definizione di template, non genera codice immediatamente
- Esso produce istanze specifiche solo quando vede una chiamata al template.
- Per generare queste istanze il compilatore deve accedere al codice sorgente che lo definisce

Modelli di Compilazione di Template – 2/2

- Il compilatore deve dunque accedere alla definizione del template.
- La soluzione che si adotta è dunque di includere negli header file sia la dichiarazione che la definizione del template e delle sue funzioni membro.
- Alcuni compilatori supportano l'uso della parola chiave **export** che permette di effettuarne la definizione esternamente alla dichiarazione

ESEMPIO PILA CON TEMPLATE DI CLASSI

- Progettare una classe Pila che permette di gestire pile con differenti tipi di dato. Tale classe deve includere le diverse operazione previste per il tipo di dato Pila
- Il template deve essere dichiarato e definito in un file header
- L'utilizzo del template dovrà essere invece fatto in un file .cpp

PILA.H: DEFINIZIONE CLASSE GENERICA PILA


```
template <class T>
class Stack {
    public:
        Stack():top(0) {
             cout << "In Stack constructor" <<  endl;
        }
        ~Stack() {
             cout << "In Stack destructor" <<  endl;
            while ( !isEmpty() ) {
                pop();
            }
            isEmpty();
        }

        void push (const T& object);
        T pop();
        const T& topElement();
        bool isEmpty();

    private:
        struct StackNode {              // linked list node
            T data;                     // data at this node
            StackNode *next;            // next node in list

            // StackNode constructor initializes both fields
            StackNode(const T& newData, StackNode *nextNode)
                : data(newData), next(nextNode) {}
        };

        StackNode *top;                 // top of stack

};
```

PILA.H: DEFINIZIONE OPERAZIONI PILA


```
template <class T>
void Stack<T>::push(const T& obj) {
     cout << "In PUSH Operation" <<  endl;
    top = new StackNode(obj, top);
}

template <class T>
T Stack<T>::pop() {
     cout << "In POP Operation" <<  endl;
    if ( !isEmpty() ) {
        StackNode *topNode = top;
        top = top->next;
        T data = topNode->data;
        delete topNode;
        return data;
    }
    cout<<"Empty Stack"<<endl;
}
```

```
template <class T>
const T& Stack<T>::topElement() {
     cout << "In topElement Operation" <<  endl;
    if ( !isEmpty() ) {
        return top->data;
    }
}

template <class T>
bool Stack<T>::isEmpty() {
    if (top == 0) {
        return true;
    }
    else {
        return false;
    }
}
```

MAIN.CPP: DEFINIZIONE CLASSE STUDENTE


```
using namespace std;

class Student {
    private:
        string name;
        string course;
        int age;

    public:
        Student(string n, string c, int a) : name(n), course (c) {
            cout << "In STUDENT constructor" << std::endl;
            age = a;
        }

        ~Student() {
            cout << "In STUDENT destructor" << std::endl;
        }

        string getName() {
            return name;
        }

        string getCourse() {
            return course;
        }

        int getAge() {
            return age;
        }
};
```

MAIN.CPP: USO CLASSE GENERICA PILA


```
int main () {
    Stack <Student> studentStack;

    Student s1( "Student1" , "Course1", 21);
    Student s2( "Student2" , "Course2", 22);

    studentStack.push ( s1 );
    studentStack.push ( s2 );

    Student s3 = studentStack.pop();
    std::cout << "Student Details: " << s3.getName() << ", " << s3.getCourse() << ", " << s3.getAge() << endl;

    Student s4 = studentStack.pop();
    std::cout << "Student Details: " << s4.getName() << ", " << s4.getCourse() << ", " << s4.getAge() << endl;

    Student s5 = studentStack.pop();
    std::cout << "Student Details: " << s5.getName() << ", " << s5.getCourse() << ", " << s5.getAge() << endl;

}
```

TEMPLATE VS POLIMORFISMO

- Ci si potrebbe chiedere qual è la differenza tra **template** e **polimorfismo**. E' possibile sostituire l'uno con l'altro?
- I template sono utilizzati per generare codice **generico** mentre il polimorfismo è usato per generare codice **dinamico**
- L'instanziazione dei template si verifica a **tempo di compilazione** mentre il polimorfismo è risolto a **tempo di esecuzione**
- Da un punto di vista pratico:
  - Le funzioni template lavorano anche con tipi aritmetici
  - Le funzioni polimorfiche devono utilizzzare puntatori
  - La genericità polimorfica si limita a gerarchie
  - I template tendono a generare un codice eseguibile grande poichè dopo la compilazione il codice è duplicato