CPP- Course Generics

Overview

- **Introduction in C/C++ Generics**
- **Functions and template methods**
- **Explicit overloading of a template function**
- **Parameters of template functions**
- **Restrictions on template functions**
- **Template Classes**
- **Instantiation of template classes**
- **Class template members**
- **Template Class Parameters**
- **Class specializations**
- **Inheritance and composition of template classes**
- **Restrictions imposed on *template* classes**

Introduction in C/C++ Generics

The first generic mechanism from C/C++ language should be considered by macrofunctions.

In the C ++ language the ***template*** notion used for generics was introduced initially as functions or classes that are written for one or more **data types** that have not yet been specified.

For these classes or functions, the type of data they are working with is specified as a parameter. The effect is that we can create a general class or function, the way we work is determined by the nature of the data we need to operate.

The characteristics that make this type of classes and functions superior to those we have used so far are:

- Defining a way of working with a **generic type of data**, any specific type of data transmitted as a parameter being only a **customization solved by the compiler**;

- Implementation of classes and body of functions is just as easy as classical

Functions and template methods

- *Template functions* were originally considered not to belong to class, and *template methods (or member functions)*, classroom interiors.
- This convention is more or less respected by those involved in the field, the general term used being *template (class -member) functions*.
- They define a general set of operations that will be applied to an entire set of input data types. The type of data that will actually work at a given time is transmitted to the function as a parameter.
- In other words, the overloading process is eliminated, the respective functions performing this operation alone.
- The term *template* used to name this type of *class* or *function* fully reflects its use: only a "template" of class or function that describes its role, leaving the compiler to complete the necessary details.

Template syntax

- The syntax for defining such a notion uses the *template* keyword and initial, type was specified by the word *class*, but as a generalization was desired not only for classes, and now is used *typename*, which implies for all types:

*template <**class** $\boldsymbol{tipg_1}$[,…class $tipg_i$]> ret_type **func_nameT** (list_of_params){*

……………...

instructions

}

or,

*template <**typename** $\boldsymbol{tipg_1}$[,…typename $tipg_i$]> ret_type **func_nameT** (list_of_params){*

……………...

instructions

}

where $tipg_i$ is a name that is in place of the type of data used by the *func_nameT(...*)

Calling templates

- When the function is applied to a particular type of data, the $tipg_i$ will be automatically replaced by the compiler with the specific data type used at that time and generated a specific function stored in memory.

- ***tipg$_i$*** is named ***template parameter***

- ***template <typename tipg$_1$[,…typename tipg$_i$]>*** ***func_nameT****(list_of_formal_params){ …},* is **header** *template*

The **calling** of a template function is:

func_nameT*(list_of_effective_params);*

, where the type of *effective parameters* will be analyzed by the compiler to generate the function.

- The particular data type must be compatible with the generic function implementation.
- Some didactical examples will be presented in the following slides.

//generics for one type

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

template <typename T> void F(T x);

int main( ){
F(1020) ;// int
F('T') ; // char
F(10.20) ;// double
F("Salut") ;// char *
return 0;
}

template <typename T> void F(T x){
cout <<x << '\n' ;
}
```

//max generics of two values

```
#include <iostream>
using namespace std;
#include <string>//string Container from STL

template <typename T> T mmax(T x, T y) ;



int main( ){
int i=7,j=9 ;
string s1 ="Pg";
string s2 = "Gen";
//CName a,b ;// objects from CName class not able to overload ?
cout <<"\n Rezultate maxim tipuri generice\n";
int k= mmax(i,j) ;
cout <<k <<endl;
cout << mmax('A', 'B')<<endl;
cout << mmax(s1, s2) << endl;
//cout << mmax(10, 20.5) ; Err. tipuri diferite
//CName c=mmax(a,b);//nu se poate supraincarca mmax pentru obiecte cu op. ?:
return 0;
} //main

template <typename T> T mmax(T x, T y){
return ((x>y) ?(x) :(y) );
}
```

//Arguments deducted implicitly or explicitly specified

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

template <typename T> T suma(T x, T y) ;

int main( ){
cout<<"\n Argumente deduse implicit sau specificate explicit\n";
cout<<suma(10,20)<<" int implicit"<<endl;//int
cout<<suma(10.0, 20.5)<<" double implicit"<<endl;//double
//suma (10, 20.5);//Err. tipuri diferite
cout<<suma <int> (10,20.5)<<" int explicit"<<endl;// Ok, int specificare explicita
cout<<suma <double> (10, 20.5)<<" double explicit"<<endl;//Ok double
return 0;
} //main

template <typename T> T suma(T x, T y){
return x+y;
}
```

Concluding, we remember that when creating a generic function, the compiler is allowed to create as many versions of the function as needed to successfully treat them all.

Functions with several generic types

- In a statement of a template function, we can have several generic types separated by commas, exemplified below. The compiler will actually create as many versions of the function as needed, being considered a default overloading process.

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

template <typename X, typename Y> void afis(X a, Y b);

int main( ){
        afis(12, "abcd");
        afis(12.8, 'c’);
        afis(12.8, 17.7);
        return 0;
}//main

template < typename X, typename Y> void afis(X a, Y b){
        cout << "\n Prima valoare: "<<a;
        cout << "\n A doua valoare: "<<b;
}
```

- It is clear that the *afis( )* function receives two types of input data, which it then displays on the screen.

Explicit overloading of a template function

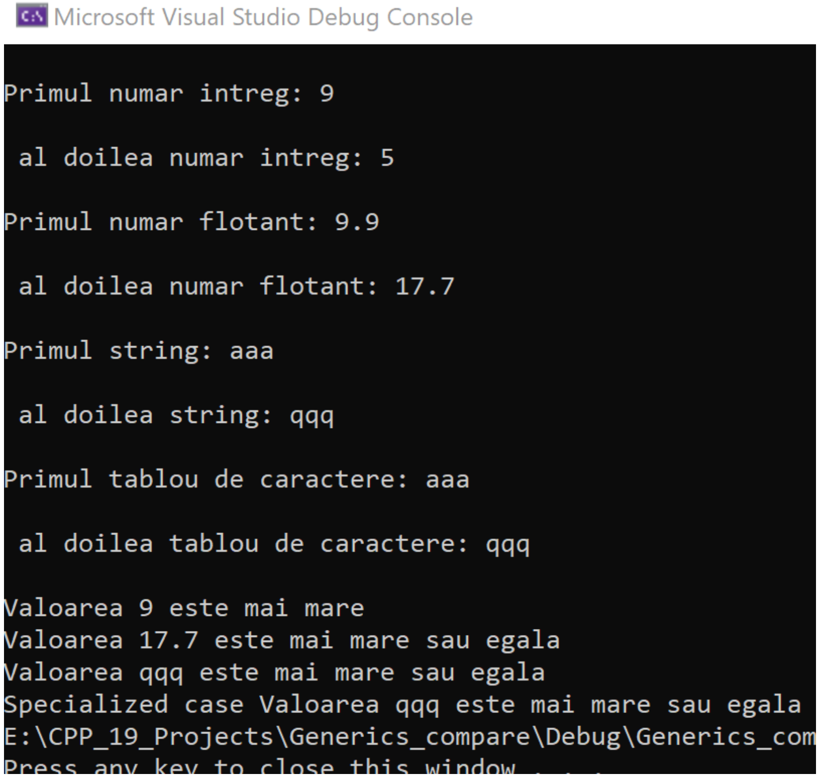
- Even though the template functions overload itself, they can be overloaded in an explicit mode. The effect is that for the data type for which the explicit overload was made, the program will follow the path that the user has defined separately.
- There is the possibility of overloading from the template header used *without specifying the name of template parameters* and using an empty < > construction before the return value of the template function. This case is called an ***explicit specialization*** of the template function.
- If the number of parameters in the case of explicit overload is equal to the one of the template function, this is considered an ***excepted case***.
- **Example:**

```
//compararea valori numerice, supraincaracare template
#include <iostream>
using namespace std;

template < typename X> void compara(X a, X b);// caz template general
void compara(int a, int b) ; // caz exceptat doar pentru int
template < > void compara (float a, float b); //specializare explicita
void compara(int a, int b, int c);//supraincarcare explicita trei parametrii
```

```
int main( ){
        int x, y, a, b, c;
        float m,n;
        double d,e;
        cout << "\nPrimul numar intreg: ";
        cin >> x;
        cout << "\n al doilea numar intreg: ";
        cin >> y;
        cout << "\nPrimul numar flotant: ";
        cin >> m;
        cout << "\n al doilea numar flotant: ";
        cin >> n;
        cout << "\n Alti trei intregi";
        cout << "\nPrimul numar intreg: ";
        cin >> a;
        cout << "\n al doilea numar intreg: ";
        cin >> b;
        cout << "\n al treilea numar intreg: ";
        cin >> c;
        cout << "\nPrimul numar double: ";
        cin >> d;
        cout << "\n al doilea numar double: ";
        cin >> e;
        compara(x,y);//supraincarcare intregi caz exceptat
        cout<<endl;
        compara(d,e);//apel template general
        cout<<endl;
        compara(a,b,c);//supraincare semnatura diferita
        cout<<endl;
        compara< >(m,n);//apel specializare explicita
        cout<<endl;
        return 0;
}
```

```
template < typename X> void compara(X a, X b){
	cout << "\n Compar date ne-intregi!-template general\n";
	if(a>b)cout << "\nValoarea "<<a<<" este mai mare.";
	else cout << "\nValoarea "<<b<<" este mai mare.";
}// caz template general

void compara(int a, int b){
	cout << "\n Compar doi intregi!- caz exceptat\n";
	if(a>b)
		cout << "\nValoarea "<<a<<" este mai mare.";
	else
		cout << "\nValoarea "<<b<<" este mai mare.";
}// caz exceptat

template < > void compara (float a, float b){
	cout<<"\nSpecializare explicita!-numere reale float\n";
	if(a>b)cout << "\nValoarea "<<a<<" este mai mare.";
		else cout << "\nValoarea "<<b<<" este mai mare.";
}//specializare explicita

void compara(int a, int b, int c){
	int sir[3];
	int i, max;
	cout << "\n Compar TREI intregi!\n";
	sir[0] = a;
	sir[1] = b;
	sir[2] = c;
	max = sir[0];
	for(i=0; i<3; i++)
		if(sir[i]>max)
			max = sir[i];
	cout << "\nValoarea "<<max<<" este mai mare.";
}//supraincarcare explicita trei parametrii
```

//compare two values




```
#include <iostream>
using namespace std;
const int dim = 20;

template <class X> void compara(X a, X b);//or typename for class
template < >void compara(char* a, char * b);//specialized case

int main( ) {
        int x, y;
        float m, n;
        string a, b;
        char aa[dim]=" ", bb[dim]=" ";
        cout << "\nPrimul numar intreg: "; cin >> x;
        cout << "\n al doilea numar intreg: "; cin >> y;
        cout << "\nPrimul numar flotant: "; cin >> m;
        cout << "\n al doilea numar flotant: "; cin >> n;
        cout << "\nPrimul string: "; cin >> a;
        cout << "\n al doilea string: "; cin >> b;
        cout << "\nPrimul tablou de caractere: "; cin >> aa;
        cout << "\n al doilea tablou de caractere: "; cin >> bb;
        compara(x, y);
        compara(m, n);
        compara(a, b);
        compara(aa, bb);//char * se va compara cu specialized altfel compara adresele
        return 0;
} //main

template <class X> void compara(X a, X b) {
        if (a > b)cout << "\nValoarea " << a << " este mai mare";
        else cout << "\nValoarea " << b << " este mai mare sau egala";
}

template < > void compara(char* a, char* b) {
        if (strcmp(a, b)>0)cout << "\nSpecialized case Valoarea " << a << " este mai mare";
        else cout << "\nSpecialized case Valoarea " << b << " este mai mare sau egala";
}
```

Parameters of template functions

- Template functions can have several types of parameters, such as:
- a) *type* template parameters introduced with *class* or *typename*
- b) *non-type* template parameters, which can be:
- -*int, enum*
- -pointer to an object or function
- -reference to an object or function
- -pointer to a member of the class
- -*C++1y/2z* considers some restrictions concerning pointers/references - https://en.cppreference.com/w/cpp/language/template_parameters

- A template function or template class can not change the value of a *non-type* parameter.
- Call arguments must be constant expressions.
- The template header may contain either *type* or *non-type* parameters.

- Example for a) and b)

```
// parametrii tip si non-tip functii generice
#include <iostream>
using namespace std;
template <typename T, int n> void afis(T t[ ]);

int main( ){
        int a[ ] ={1,2,3,4,5};
        afis<int,5> (a);
        return 0;
}

template <typename T, int n> void afis(T t[ ]){
        for (int i=0;i<n;i++) cout<< t[i] << " ";
        }
```

```
// tablou ca si parametru
#include <iostream>
using namespace std;

template< typename T> T GetAverage(T tArray[ ], int nElements);
double GetAverage(int tArray[ ], int nElements);//caz exceptat pentru rezultat double

int  main( ) {
   int  IntArray_0[5] = {100, 200, 400, 500, 700};
   int  IntArray_1[5] = {1, 2, 4, 5, 7};
   float FloatArray[3] = {1.55f, 5.44f, 12.36f};
   double DoubleArray[3] = { 1.55, 5.44, 12.36 };

   cout << "IntArray_0 average is = " << GetAverage(IntArray_0, 5) << endl;
   cout << "IntArray_1 average is = " << GetAverage(IntArray_1, 5) << endl;
   cout << "FloatArray average is = " << GetAverage(FloatArray, 3) << endl;
   cout << "DoubleArray average is = " << GetAverage(DoubleArray, 3) << endl;
 }

template< typename T> T GetAverage(T tArray[ ], int nElements){
    T tSum = T( );// = 0;
   for (int nIndex = 0; nIndex < nElements; ++nIndex) {
      tSum += tArray[nIndex];    }
   return  tSum / nElements;
}

double GetAverage(int tArray[ ], int nElements) {
   double tSum = 0.0;
   for (int nIndex = 0; nIndex < nElements; ++nIndex) {
      tSum += tArray[nIndex];
   }
   return tSum / nElements;
}
```

- c) *template-template* parameters –
- We have this case when the parameter is in its turn another template by declaring the template class within a template header.
- This template will only be entered with *class* or *typename* (*structure*, *union* is not allowed).
- A simple template header is declared by:

*template <typename **A**>*

- If *A* will be replaced with a *class template* we have:

template < *template <typename **T**> class **A**>,*

and ***A*** is a *template-template.* The **inner *T* type *may be ignored*.**

- **Example**:

c1) *template* < *template <typename T> class A> void F(A<char> a){...}*

 or, *template* < *template <typename > typename A> void F(A<char> a){...}*

c2) *template-template* for classes:

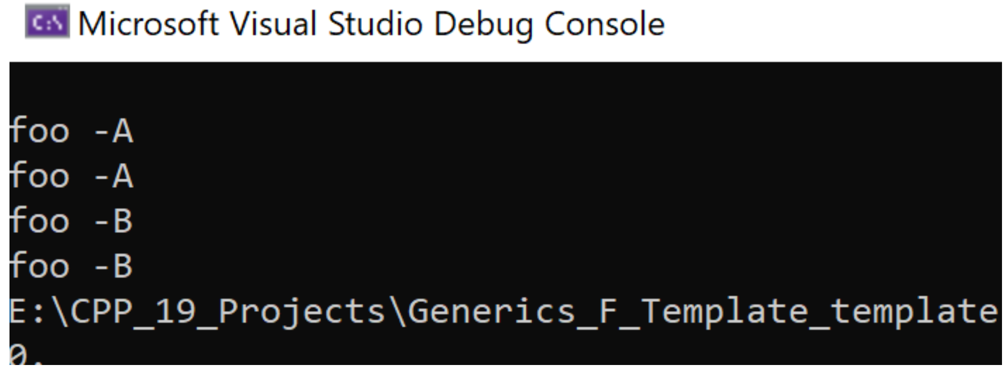
template<template < *typename T> class X> class A { …};*

template< *typename T> class B { …};*

A<B> a;

Template-template function call

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

template <typename T> class A {
public:
    static void foo(T = T( )) {
        cout << "\nfoo -A";
    }
};

template <typename T> class B {
public:
// static void foo(T = T( ))//metoda generica cu param implicit T( )
    static void foo( )    {
            cout << "\nfoo -B";
    }
};

class SomeObj { };
class SomeOtherObj { };

template <template <typename > typename T> void function( )
{
    T<SomeObj>::foo( );
    T<SomeOtherObj>::foo( );
}

int main( ){
    function<A>( );
    function<B>( );
}
```



Restrictions on template functions

- Although they offer great mobility with regard to the types of data they work with, generic functions have a major limitation: they perform the same set of operations regardless of the type of data they receive on the input. This disadvantage can be eliminated as in the case of overloading common functions, i.e. by overloading explicitly, when we can impose another approach for that method or specialization.
- **Other limitations:**

- the template functions ***do not accept the default template parameters***, but ***accept the default call parameters***
- a template function must use C++ link editors (can not use particular binding specifications).
- If a template function fits one of the specializations and an overload, then *non-template overload* is prioritized. For the external specialization call, enter < > by the construction method name.
- we have an evolution of the template functions from the versions: *'98, '0x,’1y, '2z*
- a virtual function can not be a template;
- destructors can not be templates;

Examples:

a) template <typename T1=int, int n=7> void F(T1 t) {…} //Err. no implicit params at declaration

b)//implicit params in the calling of a template function

```
#include <iostream>
using namespace std;
template <typename T1, typename T2> void F(T1 t1=10 , T2 t2 = 'A');

int main( ) {
	F<int, char>( );// valori implicite cu specificare explicita a tipului
	F<int, char>(77);// explicit 77 la T1, implicit A la T2
	F<int, char>('W');// explicit 'W'=87 la T1, implicit A la T2
	F<double, char>(77.7);// explicit 77.7 la T1, implicit A la T2
	F<int, char>(100, 'W');//explicit 100 la T1 si W la T2
F<double, string>(10.77, "World");//explicit double 10.77 la T1 si string World la T2
F<double, const char*>(10.77, "World");//explicit double 10.77 la T1 si const char * World la T2
	//F( ); //Err. nu se deduc implicit T1 si T2, trebuie specificati explicit
	F<float, int>( );//compatibili cu parametrii impliciti 10, 65='A'
	F<float, float>( );
	//F<float, string>( );//T2 incompatibil cu parametrul implicit
	return 0;
}//main

template <typename T1, typename T2> void F(T1 t1, T2 t2) {
	cout << "\nT1= " << t1;
	cout << "\tT2= " << t2;}
```

Template Classes

- In addition to defining template functions, implementing the *template classes* is another way to make use of the generalization process.
- By defining a template class, we will describe all the algorithms it uses, leaving the actual type of data to be handled to be set when creating objects in that class.
- Template classes are useful when its features and algorithms have a certain character of generality.
- In this way, ***we define a family of classes*** that depend on the template parameters. Data and methods in template classes can use template parameters as generic types.

The syntax for defining a template class

*template <**class tipg$_1$**[,… class tipg$_i$]>* ***class*** *Class_name{*

…………...

};

or

*template <**typename tipg$_1$**[,…typename tipg$_i$]>* ***class*** *Class_name{*

…………...

};

where *tipg$_i$* is a name that is in place of the type of data used by the class.

Examples:

template <typename T> ***class*** *B{*

T a;

…………...

};//B

template <typename T1, typename T2> ***class*** *X{*

T1 x;

T2 y;

…………...

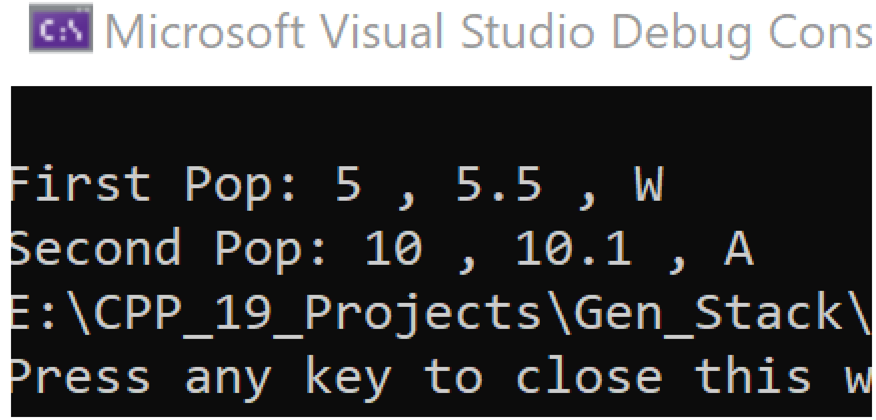
};//X

Stack example:

```
//Stiva.h

template < typename STip> class Stiva
{
private:
		int Dim;
		STip *Stack;
		int Next;
public:
		Stiva(int);//+copy constructor for management STip objects
		~Stiva( );
		int Push(STip c);
		int Pop(STip &c);
		int IsEmpty( );
		int IsFull( );
};
template < typename STip> Stiva <STip> :: Stiva(int dim_i)
{
		Next = -1;
		Dim = dim_i;
		Stack = new STip [Dim];
}
template < typename STip> Stiva <STip> :: ~Stiva( )
{
		delete  [ ]Stack;
}
```

```
// test stiva goala

template < typename STip> int Stiva <STip> :: IsEmpty( )
{
		if (Next < 0)			return 1;
		else				return 0;
}
// test stiva plina

template < typename STip> int Stiva <STip> ::IsFull( )
{
		if(Next >= Dim)		return 1;
		else						return 0;
}
// introducere in stiva
template < typename STip> int Stiva <STip> ::Push(STip c)
{
		if(IsFull( )) return 0;
		Stack[++Next] = c;
		return 1;
}
// extragere din stiva
template < typename STip> int Stiva <STip>  ::Pop(STip &c)
{
		if(IsEmpty( ))			return 0;
		c = Stack[Next--];
		return 1;
}
```

```
//main
#include <iostream>
using namespace std;
const int dim = 20;
#include "Stiva.h"

int main( ) {
        int a;
        double b;
        char c;
        Stiva<int> Si(dim);
        Stiva<double> Sd(dim / 2);
        Stiva<char> Sc(dim / 4);
        Si.Push(10); Si.Push(5);
        Sd.Push(10.10); Sd.Push(5.5);
        Sc.Push('A'); Sc.Push('W');
        Si.Pop(a);
        Sd.Pop(b);
        Sc.Pop(c);
        cout <<"\nFirst Pop: " <<a << " , " << b << " , " << c;
        Si.Pop(a);
        Sd.Pop(b);
        Sc.Pop(c);
        cout << "\nSecond Pop: " <<a << " , " << b << " , " << c;
        return 0;
}//main
```



Instantiation of template classes

- Generating functions, classes or methods using a template is called ***template instantiations*.**
- The definition created from a template instantiation to handle a **specific set of template arguments** is called a ***specialization***.
- The class thus generated from a template class is an instantiated class, which can be either implicit or explicit.

a) Implicit instantiation

- -To instantiate an object, you need the compiler instance of that class by default, in which case the template must be completely defined not just declared (methods defined for the template class).
- - For pointers to objects, there is no need for that class to be defined, it is sufficient to be declared. A class reference must be initialized by the class constructor in an *init* list.

```
template < typename T> class A; //class A declaration – we may declare pointers to the class

template < typename T> class B {
                    T n;
                    };//class B defined- we may define objects with implicit empty constructor

int main( ){
          B <int> b;//declarare obiect- instantiere implicita generare clasa B<int>
          A <char> *pc;//declarare pointer, nu e nevoie definire clasa A
          // A <float> obj;//Erroare clasa e doar declarata si trebuie definita
                    ….
          return 0;
                    }//main
```

A <T> is the template used for declaring or defining pointers/objects*.*

A obj; //is not allowed because it is not known what class to instantiate

- **b) Explicit instantiation**

You can **explicitly tell the compiler when it should generate a definition** from a template. This is called *explicit instantiation*. Explicit instantiation includes two forms: explicit instantiation **declaration** and explicit instantiation **definition**.

Explicit instantiation declaration lets you create an instantiation of a template class **without actually using** it in your code. Because this is **useful when** you are **creating library (.lib)** files that use templates for distribution, uninstantiated template definitions are not put into object (.obj) files.

Syntax:

template class template-name < argument-list > ; or

extern *template class template-name < argument-list > ;(since C++1y)*

- To do this, first the class will be defined:

```
template < typename T> class X {
                        …;
                        };//class X defined


            X <float> *p;//pointer declaration
```

template X <float>; // **explicit instance of compiler of *class X <float>*** even if the context does not require that until then I instantiate objects of that class

-If an explicit instantiation declaration of a member function or class is declared, but there is no corresponding explicit instantiation definition anywhere in the program, the compiler issues an error message.

Explicit instantiation has no effect if an **explicit specialization** appeared **before**, for the same set of template arguments.

Class template members

- Non-template classes may contain template member methods declared separately in classes. Usually in C ++, the *template functions* are defined outside the classes and the *template class functions (member functions - methods)* in classes.
- Template classes may have template or non-template members (data and methods), and if they have methods, they are by default templates.
- Methods of the template classes can be templates with the same parameters as the class, or may have their own template parameters.

a) Template methods in non-template classes

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

class A{
        public: template <typename T1, typename T2> void F(T1 t1, T2 t2){
        cout<< "\nT1= " << t1;
        cout<< "\nT2= " << t2;
        }
};//A

int main( ){
A a;
a.F<int, char>(10,'A');// specificare explicita
a.F(100, 'W');// deducere ca nu-s impliciti
        return 0;
}
```

- If in class, the template method is just declared it will be defined outside using the template header:

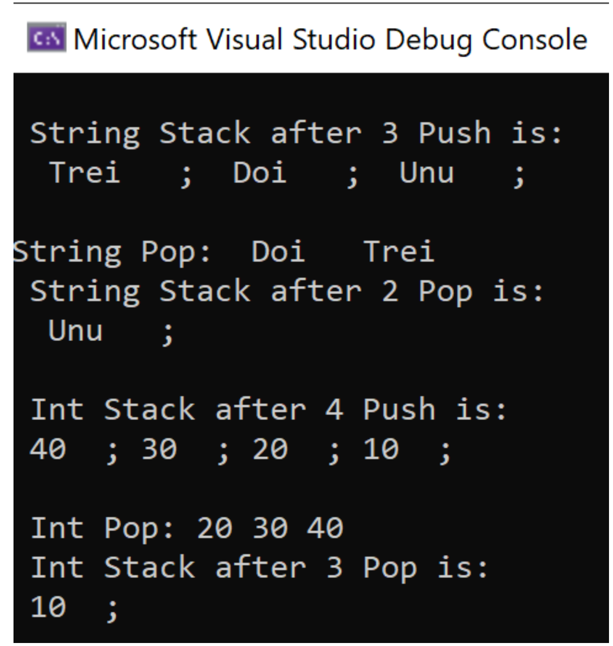
//template methods in non-template classes defined outside the class

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

class A{
        public: template <typename T1, typename T2> void F(T1 t1, T2 t2);
};//A

int main( ){
A a;
a.F<int, char>(10,'A');// specificare explicita
a.F(100, 'W');// deducere ca nu-s impliciti
        return 0;
}

template <typename T1, typename T2> void A:: F(T1 t1, T2 t2){
        cout<< "\nT1= " << t1;
        cout<< "\nT2= " << t2;
        }
```

b) Template methods in template classes - this methods of a template class are template methods with the same parameters as those of the class. Data may or may not be template.

```
//membrii template in clase template
//Stiva.h

template < typename T, int DIM> class Stiva {
        T st[DIM];
        int head;
public:
        Stiva( ):head(0){ };
        void Push (T val){ st[head++] = val;}
        T Pop ( ) {return st[--head];}
        void Display ( ) const  {
                for (int i=head-1; i>=0;i--)
                cout <<" "<< st[i] << "  ;";
                cout <<'\n';
        }
};
```

```
//main
#include <iostream>
#include <string>
using namespace std;
#include "Stiva.h"

int main( ) {
        Stiva <string, 100> S1; // Stiva de string-uri
        S1.Push(" Unu ");
        S1.Push(" Doi ");
        S1.Push(" Trei ");
        cout << "\n String Stack after 3 Push is:\n";
        S1.Display( );
        cout << "\nString Pop: " << S1.Pop( ) << " " << S1.Pop( );
        cout << "\n String Stack after 2 Pop is:\n";
        S1.Display( );
        Stiva <int, 10> S2; // Stiva de intregi
        S2.Push(10);
        S2.Push(20);
        S2.Push(30);
        S2.Push(40);
        cout << "\n Int Stack after 4 Push is:\n";
        S2.Display( );
        cout << "\n Int Pop: "<<S2.Pop( ) << " " << S2.Pop( ) << " " << S2.Pop( );
        cout << "\n Int Stack after 3 Pop is:\n";
        S2.Display( );
}
```



- -Methods defined in classes do not specify the template header, although they are considered template by default
- -The template classes accept *type* and *non-type* template parameters
- -The *const* specifier as postfix to the *Display( )* method is indicating that the method does not modify any of the member data of the class, *this** pointer does not change
- -Instances of instantiated classes include a list of arguments between < >
- -A method defined outside of the class will be indicated using the scope operator, :: and will contain the template header

Template Class Parameters

Template classes support the same types of parameters as template methods:

a) *type*

b) *non-type*

c) template-template

Example :

```
template <typename T, int n, template < typename N> class A> class B{
                    //……
                              };//B //type, non-type, template-template


template <typename T > class X {
          //          ………...
          };//X

int main ( ){
B <char, 10, X> b;…
          return 0;
          }
```

N parameter from *B* class definition may be omitted:

```
template <typename T, int n, template < typename > class A> class B{
                    //……
                              };//B
```

Implicit class *template* parameters

- Class Templates vs. Method Templates support default template parameters.
- **a) *Type* parameters**

```
//clase cu parametrii tip impliciti

 class A {
};//A

template <typename T=A> class B{
            T t;
            };//B

int main ( ){
B <int>b1;//tip int
B < > b2;// tip implicit A
//B b;
return 0;
}
```

b) *Non-type* parameters

-There are the same restrictions from the template methods, that is, they can be *int enum*, *pointers* to objects, attributes or methods, or they can be *references*.

-You can not accept real types (*float, double*), only *pointers* to these types

-The default values must be constant expressions

c) *Template-template* parameters

Appears when a template class accepts as an argument another template class.

Thus, an *A* class with an *X* template parameter can be instantiated with different class templates.

Example :

```
 //arg. implicite template-template
class D {
};//D non template

template <typename T= D> class B{…
};//B template, implicit tip D

template <typename T = int> class C {…
};//C template, implicit int

template <template <typename> class X = B > class A{
};//A clasa template template, implicit B pentru X

int main ( ){
A <C> a1; // X==C, C<T> == C<int>
A < > a2;//X==B, B<T> == B<D>
//A<D> a3; //Err. D e clasa non template
return 0;
}
```

Class specializations

The template classes admit the same specialization as the methods, and also allow partial specializations.

Specialization is made when a different definition is desired for some types.

The name of the specialization consists of the name of the class to which the list of template arguments is enclosed between the angular brackets, < >.

a) Full specialization (explicit)

The specialization is to customize the primary template parameters. It is done by preceding the name of the class by:

template < > and specification of template arguments for all parameters.

```
template <typename T> class A {
//…..
};//A template primar

template < > class A <int>{
//….
};// Specializare explicita A <int> pt. T = int

template < > class A <char>{
//….
};// Specializare explicita A <char> pt. T = char
```

The types for which the specialization is made are specified as template arguments immediately after the class name between < >. The body of the specialization should be different from the class of the primary class.

```
//specializare explicita clase template
//Str.h
const int dim=100;

template <typename T> class Str{
        T t[dim];
        int n;
public:
        void Read( ){
                cout <<"\n Enter n, and elements of the generic array: ";
                cin >>n;
                for (int i=0; i<n; i++)
                        cin >> t[i]; }//Read
        void Display( ) const {
                for (int i=0;i<n;i++)
                        cout << "\nElements are: "<< t[i] << " "; }//Display
        };//Str primar

template < > class Str <char> {
        char t[dim];
        public:
        void Read( ) const {
                cout <<"\n Enter a specialized string: ";
                cin >>(char*) t;
                        }//Read
        void Display( ) const {
                        cout << " \n The String was: " << t;
                        }//Display
};//Specializare char
```

Full (explicit) specialization must provide arguments for all template parameters

```
#include <iostream>
using namespace std;
#include "Str.h"

int main( ){
        Str <int> s1;
        cout <<"\n Generic type, int \n";
        s1.Read( );
        s1.Display( );
        Str <char> s2;//specializare
        s2.Read( );
        s2.Display( );
        return 0;
}//main
```

b) Partial specialization

If we specialize in a class that depends on at least one generic parameter then we have a *partial specialization*.

They reduce the generality of the primary template.

They are not conditioned by the existence of the complete definition of the primary template.

```
//specializare partiala clase template


template <typename T1, typename T2> class A{
        //....

};//template primar

template <typename T> class A <T, char>{
        //....

};//Specializare T2 la char

template <typename T> class A <float, T>{
        //....

};//Specializare T1 la float


int main( ){
        A <char, char> a1;// Spec. A <T, char>
        A <char, int> a2; //Sablon primar A <T1, T2>
        A <float, int> a3;//Spec. A <float, T>
        return 0;
}//main
```

Inheritance and composition of template classes

- C++ is considered a multi-paradigm language. Supports:

- Structured, procedural programming, from the C language

- O.O.P. based on inheritance, virtual methods and classes using dynamic polymorfism

- generic programming

- Functional programming with the lambda functions introduced in *C++ 1y*

- The following examples will specify elements concerning inheritance and composition.

a) the template class inherits a non-template class

```
//mostenire clasa non-template

class A {
        //....
};//A

class B: public A
{
        //...
};//B non-template


template <typename T> class C : public A
{
        T d;
};//C

int main ( ){
        B ob1;
        C <int> ob2;
        return 0;
}//main
```

b) the template class inherits a template class

```
        //mostenire template clasa template

template <typename T> class A {
        //....
};//A template



template <typename T1, typename T2> class C : public A <T2>
{
        T1 d;
};//C

int main ( ){
        C <int, char> ob2;//T1 int, T2 char
        return 0;
}//main
```

c) the name of the inherited class is one of the template parameters of the derived class

```
        //mostenire clasa de baza e parametru template la clasa derivata

template <typename T> class B {
        //....
};//B template baza



template <typename T1, typename T2> class D : public T2// T2 va fi o clasa
{
        T1 d;
};//Derivat

int main ( ){
        D <int, B<char>> ob1;//T1 int, T2 B<char>
//      D<int, char> ob2;// char nu e tip class si e ca T2
        return 0;
}//main
```

Composition

This situation is when an object contains at least one member that in turn is an object of the type that differs from that of the class.

```
        //continere
template <typename T> class A {
        T x; // obiect continut
        //....
};//A template baza


template <typename T1, typename T2> class B
{
        A<T1> a;//a obiect continut
        T2 b;// b obiect continut
};

int main ( ){
B <int, char> ob1;//T1 int, T2 char, si presupune instantierea prealabila a lui A<char>
B <int, A<char>> ob2;// T1 e int, T2 e A<char>
return 0;
}//main
```

Restrictions imposed on *template* classes

- The following restrictions are required for the implementation and use of *template* classes:

- at each instantiation of the *template* type, we are forced to replace the generic type with a specific type;

- specifying the concrete type is realized in the form of a parameter of the template class;

- at each instance, the specific type will take the place of the generic type;

- The generic types used in template class declarations can be replaced by:

- types of predefined data (*int, float*, etc.)

- types of user data defined by the programmer;

- pointers to functions;

- constant expressions.

The template class initiates an array of elements of the type that the user desires, with the values desired by the user, and then displays these values on the screen.

```
//myTemplate.h
template <typename X, int n> class myTemplate{
        X tab[n];
        int i;
        public:
                void init( ){for(i=0; i<n; i++){
                                cout << "\nElementul "<<i+1<<": ";
                                cin >> tab[i];          }
                                }//init

                void arata( ){ for(i=0; i<n; i++)
                        cout << "\nElementul "<<i+1<<": "<<tab[i];
                                }//arata
};//myTemplate
//main
#include <iostream>
using namespace std;
#include "myTemplate.h"

int main( ){
        myTemplate <int, 3> ob1;
        ob1.init( );
        ob1.arata( );
}//main
```