11 Parameterized classes and functions. The template mechanism

Data parameterization represents an important element of the data abstraction paradigm and allows the definition of some classes which contains unspecified complete data type, like functions parameters. In this way, a parameterized class represents a pattern which defines a class set.

The parameterized classes of the C++ language are like of Ada language, but the creation and utilization mechanism is different and it is based on the template notion. Moreover, C++ allows the definition and utilization of template functions, in a way likely of the template classes.

12.1 Template classes

The main containers problem in the C++ language is the data type instance of the components elements. If , for example, it was defined a stack class which defines a stack type container, for using in a program a integer stack, it must be created a distinct class, istack for example, which is derived from the stack class and from a class which defines integers.

Usually, a container is an abstract interface which defines the main operations which can be done over some component elements, without being said the type. The solution taken by Stroustrup for the C++ language is based on the reuse of the code and it is nemed *template*.

For the creation an integer stack, the parameterized stack class will have an instance at the int type. For example, if it was defined the parameterized stack class, then the next declarations defines two objects of stack type, one contains integers and the other characters:

```
stack<int> sint;
stack<char> schar;
```

So, there were created two distinct classes, even if the code written by the user was only for stack class.

12.1.1 The use and definition of parameterized classes

The definition of parameterized classes is relatively simple: the definition of the respective class is preceded by a syntactic construction as:

```
template ’<’ <parameters list> ’>’
```

Where <parameters list> represents the class parameters list. Usually, the parameters of a template class are data types, being specified by the key word class or typename.

Example. The next class defines a one-dimensional array at which the elements have a generic data type, T. This is instanced twice in the main function.

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

template <class T>
class Array {
  enum { size = 100 };
  T A[size];
public:
  T& operator[](int index) {
    return A[index];
  }
};

void main() {
  Array<int> ia;
  Array<float> fa;
  for(int i = 0; i < 20; i++) {
    ia[i] = i * i;
    fa[i] = float(i) * 0.5;
  }
  for(int j = 0; j < 20; j++)
    cout << j << ": " << ia[j] << ", " << fa[j] <<
endl;
}
```

The parameterized type T is used in the class as any known data type. The only restriction which must be is as all the classes or data types used in instances of the parameterized classes, to have functions and operators used in the parameterized class for T class.

For example, if in the Array class would be defined a function Print as:

```
template <class T>
class Array {
  // ...
  void Print() {
    for (int k=0; k<size; k++)
      A[k].Print();
  }
  // ...
};
```

it can not be created instances as Array<int> because the int type has no Print function**.**

It is possible to be specified in a parameterized class also others parameters then data types name, likely functions' parameters. So, the instances must be constant expressions which can be evaluated by the compiler.

For example, in the next one it will be rewritten Array class with two parameters, the second specifying the table dimension.

Example.

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

template <class T, unsigned size>
class Array {
  T A[size];
public:
  T& operator[](int index) {
    return A[index];
  }
};

void main() {
  Array<int, 20> ia;
  Array<float, 10> fa;
  // ...
}
```

Also, it is possible as the parameterized class parameters to have implicit values, likely to the implicit ones of the functions. For example, the next definition:

```
template <class T, unsigned size = 100>
class Array {
  // ...
};
```

it allows instances as:

```
Array<int> ia;
Array<float, 10> fa;
```

Parameterized class **instantiation** means the automatic generation by the compiler of the code for instance class. The instantiation is done, usually, in that places in the program where are defined the respective class objects. In the instantiation moment, all the arguments of the template class must be known.

Observation. When it is used a reference or a pointer to such an object, the instantiation does not take place. For example, the next declaration does not produce an instantiation of the Array class:

```
Array<int> *pa;
```

When the defining of a member function of such a class it is not made in line, but outside, the definition must be preceded by the same template construction as the class to which it belongs, which specifies the template parameters of the function. Moreover, any reference to the class to which the function belongs must be specified by an instance.

For exemplification, at the previous Array class it was added a Print function.

Example.

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

template <class T, unsigned size>
class Array {
  T A[size];
public:
  T& operator[](int index) {
    return A[index];
  }
  void Print() const;
};

template <class T, unsigned size>
void Array<T>::Print() const {
  for (int k=0; k<size; k++)
    cout << ‘ ‘;
  cout << endl;
}

void main() {
  Array<int, 10> ia;
  for(int i = 0; i < 10; i++)
    ia[i] = i * i;
  cout << “Elementele sunt:” << endl;
  ia.Print();
}
```

Usually, the definition of a parameterized class, as the implementation of the functions which are not inline is done in the same header file. The reason for this is referred to the actions which the compiler must execute when allocating memory for a parameterized class instantiation. There are also compilers which allow distinct files for class definition and implementation.

12.1.2 Typename key word

As previous said, in the template construction, it can be used the key word class, and also typename, for specifying a parameter which is a data type. Initially, it was used the class word, but typename, which was introduced letter in the language, is more suggestive.

There are situations in which these must be used. The most frequent is that in which in a parameterized class is used a data type defined in inside the class which is parameter. In this situation, the compiler does not know enough information for differentiating a name which represents a parameter class member, or a imbricate data type.

Example. Both X and Y classes contain M imprecated class. A parameterized class is an instance for each of these classes, as member, an instance of M class.

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

class X {
public:
  // ...
  class M {
  public:
    void g() { cout << "g in class X\n"; }
  };
  // ...
};

class Y {
public:
  // ...
  class M {
  public:
    void g() { cout << "g in class Y\n"; }
  };
  // ...
};

template <class T>
class A {
  typename T::M m;
public:
  void f() { m.g(); }
};
```

```
void main() {
  A<X> ax;
  A<Y> ay;
  ax.f();
  ay.f()
}
```

In the case when it was not used the typename in:

```
  typename T::M m;
```

this would be ambiguous, because T class is a generic one (if not there were no ambiguousness).

12.1.3 Template classes and friend declarations

In the parameterized classes there can be declared classes or functions as being friend to respective classes. There are three types of friend declarations (as seen later, there can be defined also template functions, like template classes):

a) class or function non-template declaration
b) template functions or classes declaration with the same parameters as the current class;
c) template functions or classes declaration with other parameters (not linked).

In the first case, the functions or classes are friend declared using the same rules as non-template classes.

In the second case, the template class parameters are used for linking class parameters (and functions) friend. In this case, an instance of a friend class (or function) has access at the private members only of an instance of the current template class with the same values of the arguments.

In the last case, the argument lists of the current template class and those of friend classes (and functions) are different, so that these are named not-linked. In this case, every instantiation of a friend class has access at the private members of every instantiation of the current template class.

Example. For example it will be defined a parameterized class friend to other parameterized class in the second case.

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

template <class T>
class A;
```

```
template <class T>
class B {
  void g() { cout << "g in class B\n"; }
  friend class A<T>;
};

template <class T>
class A {
public:
  void f(B<T> b) { b.g(); }
};

void main() {
  A<int> ax;
  A<float> ay;
  B<int> bx;
  B<float> by;

  ax.f(bx);
  ay.f(by);
  //ax.f(by);  //Conversion error of the parameter
  //ay.f(bx);  //Conversion error of the parameter
}
```

12.1.4 Parameterized classes' derivation

A parameterized class can be derived, case when the derived class will be also a parameterized class with the same parameters, at which eventually will be added others.

Example. The Derived parameterized class public inherits the Base parameterized class.

```
class A {
  int v;
public:
  A(int a = 0): v(a) {}
  void Print() const { cout << v << endl; }
};

class B {
  float v;
public:
  B(float a = 0): v(a) {}
  void Print() const { cout << v << endl; }
};
```

```
template<typename T>
class Base
{
  T& t;
public:
  Base(T& a): t(a) {}
  void Print() const { t.Print(); }
};

template<typename T, typename C>
class Derived: public Base<T>
{
  C c;
public:
  Derived(T& t, C& a): Base<T>(t), c(c)  {}
  void Print() const { Base<T>::Print(); c.Print(); }
};

void main() {
  A a(3);
  B b(3.3);
  Base<A> x(a);
  Derived<A, B> y(a, b);
  x.Print();
  y.Print();
}
```

12.1.5 Parameterized class imbrication

The bind between the parameterized classes and the imbrications notion of classes can be seen in two different ways: the define of a non-parameterized class inside a template class and the define of a template class inside another class (parameterized or not).

In the first case, the non-parameterized class imbricated inside a parameterized class becomes itself a parameterized class, being possible to use inside it the exterior class parameters.

Example. It will be redefined the Stack class as a parameterized class. The elements of such a stack are instances of the Link class, defined in the Stack class. The Link class uses the T parameter of the Stack class, beung itself a parameterized class.

```
template<class T>
class Stack {
  struct Link {
    T* data;
    Link* next;
    Link(T* dat, Link* nxt): data(dat), next(nxt) {}
  }* head;
public:
  Stack() : head(0) {}
  ~Stack(){
    while(head)
      delete pop();
  }
  void push(T* dat) {
    head = new Link(dat, head);
  }
  T* pop(){
    if(head == 0) return 0;
    T* result = head->data;
    Link* oldHead = head;
    head = head->next;
    delete oldHead;
    return result;
  }
};

class X {
  int n;
public:
  X(int a = 0): n(a) {}
  virtual ~X() { cout << "~X " << endl; }
  void Print() const { cout << n << endl; }
};

void main() {
  Stack<X> st;
  for(int i = 0; i < 10; i++)
    st.push(new X(i));
  for(int j = 0; j < 10; j++)
    st.pop()->Print();
}
```

In the second case, a imbricate class inside another class is itself a parameterized class, which can a parameterized one or not. In the case when a external class is also a parameterized class, the imbricate class can use, as in the previous example, its parameters.

Example. The B parameterized class is defined inside A parameterized class. Inside B class it can be used also its own parameters and also A's.

```
#include <iostream>
```

```
using namespace std;

class X {
  int v;
public:
  X(int a = 0): v(a) {}
  void Print() const { cout << v << endl; }
};

template <class T1, class T2>
class A {
public:
  template <class T2>
  class B {
  public:
    B(T1 a): v1(a), v2(v1) {}
    T1 v1;
    T2 v2;
    void g() { v2.Print(); }
  };
  B<T2> m;
  A(T1 s): m(s) {}
  void f() { m.g(); }
};

void main() {
  A<int, X> a(7);
  a.f();
}
```

12.2 Template functions

Template functions represent the implementation of some algorithm categories named generic algorithms. They represent the description of a problem class only by specifying its actions which execute, without to be specified the concrete data type of the elements on which they work on.

A suggestive example is the generic algorithms of sorting an elements range through a certain method. Supposing that on these elements can be applied comparison operators, a generic method quicksort describes the actions for sorting, without specifying the data type of the component elements.

The definition syntax of a template function is like the one for template class, because before the function definition it must be the same template construction.

The generic parameters of such a function can be names of data type and also elements of other data type. All the observations specified at the generic parameters of the classes remain valid in the case of template function.

As for parameterized classes, the instantiation of a template function is done at its call. There is a difference comparing to the classes: for functions it is not need to explicitly specify the instantiation values, because these are deduced automatic by the compiler.

Example. The MinVector function determines the minimal value of the elements of a integer range. In the main function are generated two instantiation of the MinVector function.

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

template <typename T >
T MinVector(T* v, unsigned n) {
  T min = v[0];
  for (unsigned k=1; k<n; k++)
    if (v[k] < min)
      min = v[k];
  return min;
}

void main() {
  int v1[] = {1, 3, 0, 8}, m1;
  double v2[] = {-1, 0, -9}, m2;
  m1 = MinVector(v1, 4);
  m2 = MinVector(v2, 3);
  cout << m1 << ", " << m2 << endl;
}
```

Argument deduction of instantiation can be done only with the support of data type of the usual call parameters of the template functions. In the previous example, the instantiation value of the generic T parameter is deduced from the data type of the first usual parameter of the MinVector function (v parameter).

It is not needed as an generic argument name to be in the usual formal parameters of the template function. Because the visibility domain of such an argument to be extended over the hole function definition, it is possible as a generic argument to be used only in the function body.

In this case, the compiler does not have sufficient information for the deduction of the instantiation arguments. From this reason, it is necessary as the instantiation to be done explicitly, like the parameterized classes instantiations. The specification of the instantiation arguments is done between the function name and the actual parameters list.

Example. Supposing that the T generic class has an implicit constructor, and Print function, the f template function has a local variable of T type, which is initialized and displays its value. In the case when the instantiation argument was not explicit, a call as:

f();

would generate an error.

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

class A {
  int n;
public:
  A(int v = 0): n(v) {}
  void Print() const { cout << n << endl; }
};

template <class T>
void f() {
  T t;
  t.Print();
}

void main(void) {
  f<A>();
}
```

Observation. Even if there are more calls, the compiler generates only one instance for each generic argument type. For example, if in the Example of Min Vect function, the main function would be:

```
void main() {
  int v1[] = {1, 3, 0, 8}, m1;
  double v2[] = {-1, 0, -9}, m2;
  int v3[] = {9, 2, 4, 7}, m3;
  m1 = MinVector(v1, 4);
  m2 = MinVector(v2, 3);
  m3 = MinVector(v3, 4);
  cout << m1 << ", " << m2 << ", " << m3 << endl;
}
```

It would be also two instances of it : MinVect <int> and MinVect<double>.