Object Oriented Programming - part 2

Emmanuelle Darles

Generic programming in C++

Table of contents

Introduction

The language C++ offers two possibilities to make generic programs :

1. *template function* : this kind of this function offers to the programmer the possibility to design function with one or more parameters generic types.
2. *template class* : in the same way of template function, template class offers to the programmer the possibility to design class with members generic type.

I Template function

1. Why ?

We want to write a function which swap two variables. The type of variables can be int, float or char.

We must write 3 functions (one for each type) :

```
1 void swap(int& a, int& b){
2     int temp = a;
3     a = b;
4     b = temp;
5 }
6
7 void swap(double& a, double& b){
8     double temp = a;
9     a = b;
10    b = temp;
11 }
12
13 void swap(char& a, char& b){
14    char temp = a;
15    a = b;
16    b = temp;
17 }
```

It's the same code for all the functions ! ***The only difference is the type of parameters !***

Solution : template function.

2. What is a template function ?

The solution consist in defining a type **T**.

T can represent an integer, a float , a double, a long long, a char, ...

```
1 void swap(T& a, T& b)
2 {
3     T temp = a;
4     a = b;
5     b = temp;
6 }
```

To inform the compiler that T can be any type, you must add the keywords template ***<class T>*** before the function

```
1 template <class T>
```

```
2 void swap(T& a, T& b)
3 {
4     T temp = a;
5     a = b;
6     b = temp;
7 }
```

This function is a template function. The type **T** can be an integer, a float, a double ...

Template function usecase

During the compilation stage, the compiler instantiates the type ***T*** and generates the function according to the type of parameters.

```
1 #include <iostream>
2 #include <swap.h>
3
4 int main(void)
5 {
6    int a = 1;
7    int b = 2;
8    std::cout << a << " " << b << std::endl;
9    swap(a,b);
10   std::cout << a << " " << b << std::endl;
11
12   float a = 1.112;
13   float b = 2.113;
14   std::cout << a << " " << b << std::endl;
15   swap(a,b);
16   std::cout << a << " " << b << std::endl;
17 }
```

1. The compiler instantiates the type `T` of `int` and generates the function **`void swap(int &, int &)`** and call the this function with `a=1` and `b=2` ;
2. The compiler instantiates the type `T` of `float` and generate the function **`void swap(float &, float &)`** and call this function with `a=1.112` and `b=2.113`

3. How to implement a template function ?

Template functions must be implemented in the `".h"` file.

Example

```
1 template <class T>
2 void swap(T& a, T& b)
3 {
4     T temp = a;
5     a = b;
6     b = temp;
7 }
```

```
1 #include <iostream>
2 #include <swap.h>
```

```
3
4 int main(void)
5 {
6    int a = 1;
7    int b = 2;
8    std::cout << a << " " << b << std::endl;
9    swap(a,b);
10   std::cout << a << " " << b << std::endl;
11
12   float a = 1.112;
13   float b = 2.113;
14   std::cout << a << " " << b << std::endl;
15   swap(a,b);
16   std::cout << a << " " << b << std::endl;
17 }
```

Example:An example of the display of a generic array

```
1 #include <iostream>
2 #ifndef T_DISPLAY
3 #define T_DISPLAY
4
5 template <class T>
6 void display (T tab[],int size) {
7     for (int i=0;i<size;i++)
8         cout << tab[i] << endl;
9 }
10 #endif
```

```
1 int main() {
2     double tab1[3] = {1.5,2.4,5.8};
3     display(tab1,3);
4     char tab2[4] = {'a','b','c','d'};
5     display(tab2,2);
6 }
```

With more generic parameters

It's possible to implement a template function with many generic parameters.

```
1 template <class T, class U>
2 void swap(T &a, U &b)
3 {
4     T temp = a;
5     a = (T) b;
6     b = (U) temp;
7 }
8
```

```
1 #include <iostream>
2 #include <swap.h>
3
4 int main(void)
5 {
6    int a = 1;
7    float b = 2.113;
```

```
    std::cout << a << " " << b << std::endl;
    swap(a,b);
    std::cout << a << " " << b << std::endl;

}
```

With C++11, it's possible to use **`variadic template function`**.

A ***variadic template function i***s a template with unknown numbers and type parameters.

```
#include <iostream>

template<typename T, typename... Args>
T add(T first, Args... args) {
  return first + adder(args...);
}

int main(void)
{
   long sum = add(1, 'ab', 3.0, 80000, 'good');
   std::cout << "sum = " << sum << std::endl;
}
```

Template function overriding

It's possible to override a template function if the code depend of the type of parameters :

```
template <class T>
T& min (T& a,T& b) {
    if (a < b) return a;
return b;
}
const char* min (const char* a, const char* b) {
    if (strcmp(a,b) <0) return a;
    return b;
}
```

```
#include <iostream.h>
#include "min.h"
int main() {

    int x=99, y=20;
    std::cout << min(x,y) << std::endl; // 20

    char c1="ok",c2="good";
    std::cout << min(c1,c2) << std::endl; // ok

}
```

4. Quiz: Computation of the square of a value

Question

Create a template function to compute the square of a value of any type (the result will have the same type).

Write a small program using this template function.

5. Quiz: Computation of the sum of array elements

Question

Create a template function to compute the sum of an array elements with any type. The number of elements is a parameter of the function.

Write a small program using this template function.

II Template class

1. What is a class template ?

A class template is a class with one or more unknown datatype attributes or functions.

Example

Let us consider a class representing an array of int.

```
1 class Array
2 {
3    private :
4      int* elements;
5      int size;
6    public :
7      Array(int* elements, int size){
8        this->elements = malloc(size*sizeof(int));
9        for (int i=0; i<size; i++){
10         this->elements[i] = elements[i];
11       this->size = size;
12     }
13     ~Array(){ free(this->elements);}
14     int* getElements(){ return elements;}
15     int  getSize(){ return size; }
16     ...
17 }
```

If we want manage arrays of float, we must implement the same class with float datatype.

```
1 class Array
2 {
3    private :
4      float* elements;
5      int size;
6    public :
7      Array(float* elements, int size){
8        this->elements = malloc(size*sizeof(int));
9        for (int i=0; i<size; i++){
10         this->elements[i] = elements[i];
11       this->size = size;
12     }
13     ...
14 }
```

A solution is to declare the attribute elements with a datatype T (T can be int, float, double, a class, ...)

```
template <class T>
class Array
{
   private :
     T* elements;
     int size;

   public :
     Array(T* elements, int size){
       this->elements = malloc(size*sizeof(T));
       for (int i=0; i<size; i++){
         this->elements[i] = elements[i];
       this->size = size;
     }
     ...
}
```

The class Array is a template class. The type of the elements will be instantiated during the compilation stage.

It's more generic and efficient !

2. The using of a template class

The using of a template is similar to the using of a template function.

To create an object, T must be made explicit.

```
#include <iostream>
#include "Array.h"

int main(void)
{
  int element_int[5]={1,2,3,4,5};
  Array<int> tabInt(elements_int,5);
  for(int i=0;i<tabInt.getSize();i++)
     std::cout << tabInt.getElement(i) << std::endl;

  float elements_float[5]={1.0,2.0,3.0,4.0,5.0};
  Array<float> tabFloat(elements_float,5);
  std::cout << tabFloat << std::endl;
  for(int i=0;i<tabInt.getSize();i++)
     std::cout << tabFloat.getElement(i) << std::endl;
}
```

During the compilation stage :

- the compiler generates an array object with elements of type int
- the compiler generates an array object with elements of type float

```
#include <iostream>
#include "Array.h"

int main(void)
{
  float elements_int[5]={1,2,3,4,5};
  Array<int> tabFloat(elements_int,5);
  for(int i=0;i<tabInt.getSize();i++)
```

```
    std::cout << tabInt.getElement(i) << std::endl;

  float elements_float[5]={1.0,2.0,3.0,4.0,5.0};
  Array<float> tabFloat(elements_float,5);
  for(int i=0;i<tabFloat.getSize();i++)
    std::cout << tabFloat.getElement(i) << std::endl;
}
```

3. Template class with different generic types

In the same way of template function, it's possible to define a template class with different generic datatype.

```
template <class T, class U>
class Array
{
   private :
     T* elements;
     U sum;
     int size;

   public :

     Array(T* elements, int size){
       this->elements = malloc(size*sizeof(T));
       for (int i=0; i<size; i++){
         this->elements[i] = elements[i];
       this->size = size;
     }
     ...
     void computeSum(){
       T sumE;
       for (int i=0; i<size; i++){
         this->elements[i] = elements[i];
       sum = (U) sumE;
}
```

```
#include <iostream>
#include "Array.h"

int main(void)
{
  float elements_float[5]={1.0,2.0,3.0,4.0,5.0};

  Array<float, long> tabFloat(elements_float,5);

  for(int i=0;i<tabFloat.getSize();i++)
    std::cout << tabFloat.getElement(i) << std::endl;

  long sum = tabFloat.getSum();
  std::cout << sum << std::endl;
}
```

4. Specialization

If we want to define a different implementation for a template when a specific datatype is passed as template parameter, we can declare a specialization of this template.

Example

```
1
2 #include <iostream>
3 using namespace std;
4
5 // class template:
6 template <class T>
7 class mycontainer {
8     T element;
9   public:
10     mycontainer (T arg)
11 {element=arg;}
12     T increase () {return
13 ++element;}
14 };
15
16 // class template specialization
17 template <>
18 class mycontainer <char> {
19     char element;
20   public:
21     mycontainer (char arg)
22 {element=arg;}
23     char uppercase ()
24 {
25 if
26 ((element>='a')&&(element<='z'))
27       element+='A'-'a';
28       return element;
29     }
30 };
31
32
33 int main () {
34   mycontainer<int> myint (7);
35   mycontainer<char> mychar
36 ('j');
37   cout << myint.increase() <<
38 endl;
39   cout << mychar.uppercase() <<
40 endl;
41 return 0; }
```

Notice the differences between the generic template class and the specialization:

```
1 template <class T> class mycontainer { ... };
2 template <> class mycontainer <char> { ... };
```

The first line is the generic template, and the second line is the specialization.

5. Quiz: Generic Point2D

Question

Write a template class to manage generic point in 2 dimension with heterogeneous type coordinates (for example x can be an int and y can be a float).

Test your template class into a small program.