References

Chapter 16- Template: C++ From Control Structures through Objects

Starting Out With C++. From Control Structures through Objects (Eighth Edition)

Just like overloading function, Template is also a kind of compile time Polymorphism. In C++ there are two types of templates: (1) function template, and (2) class template.

Function template

Function templates are special functions that can operate with *generic types*. This allows us to create a function template whose functionality can be adapted to more than one type or class without repeating the entire code for each type.

In C++ this can be achieved using *template parameters*. A template parameter is a special kind of parameter that can be used to pass a type as argument: just like regular function parameters can be used to pass values to a function, template parameters allow to pass also types to a function. These function templates can use these parameters as if they were any other regular type.

The format for declaring function templates with type parameters is:

template <class *identifier*> function_declaration;
template <typename *identifier*> function_declaration;

The only difference between both prototypes is the use of either the keyword **class** or the keyword **typename**. Its use is indistinct, since both expressions have exactly the same meaning and behave exactly the same way. For instance, with the following function template:

```
template <class myType>
myType  getMax (myType x, myType y)
{
        return (x > y? x: y);
}
```

the call like

```
cout << getMax(4,9) << endl;
```

should return the number 9.

Exercise 1:

Use the getMax() function above and test the following calls

```
cout << getMax('a', 'c') << endl << endl;

cout << getMax("a", "c") << endl << endl;

cout << getMax("ab", "ac") << endl << endl;
```

The result might be a surprise to you.

Exercise 2:

Create a **function template** named **randomPick()** with three parameters of the same type and a return value of the same type as the parameters. When called, it should **randomly pick** one of the three parameters, and return that value. For examples, if in main() you make the function calls in the following way:

```
cout << randomPick('1','2','3') << endl;  // chars

cout << randomPick (11,22,33) << endl;  // ints

cout << randomPick ("Mac","PC","Others") << endl; // strings
```

a possible output would be

```
3

22

PC
```

<u>Exercise 3:</u>

Use randomPick() to do this test.

Step 1: Move both the prototype and definition of the function to randompick.h file and include randompick.h in main.cpp. Compile and run the program. Will it work?

Step 2: Create another file named radomPick.cpp and move the definition of randomPick() to the file. Compile and run the program. Will it work? Can we include another file in main.cpp to make this work?

<u>Steps to develop a function template</u>

The following is the steps for designing and implementing a function template.

Step 1 - Develop function using usual data types, such as int, first.

Step2 - Convert the function to a template by (1) adding template prefix and convert int data type names in the function to a type parameter (*i.e.*, a T type) in the template.

Step 3 – Ensure that that the operators used in the code are appropriate for all possible typenames. For example, we can use “<“ operator to compare ints, doubles, and strings. But can we compare two objects using “<“ operators?

Class template

Declaring a class template is very similar to declaring a function template. A class template can be seen as a set of classes of the same implementation which handles different data types.

There is one import difference between regular class and class template. For a regular class, we normally separate the class into two parts: the declaration is placed in a .h (or .hpp) file while the definition (implementation) in a .cpp file. But for a class template we normally put the entire class in the header file. This is because when compiler compiles the template, it needs both the declaration and the definition. Without the definition, the entire object code for the class cannot be completed.

Let us see the code example below see why both declaration and definition have to be in .h file.

```
#include <iostream>

using namespace std;

/// .h file  -- declaration of Foo
template<typename T>
class Foo
{
   T bar;
public:
   void reset(T x);
   void add(T y);
   T display();
};

/// .cpp – definition/implementation of Foo
template<typename T>
void Foo<T>::reset(T x)
{
   bar = x;
};

template<typename T>
void Foo<T>::add(T y)
{
   bar = bar + y;
};

template<typename T>
T Foo<T>::display()
{
   return bar;
};

int main()
{
   Foo<int> f;
   f.reset(0);
   cout << f.display() << endl;
   f.add(99);
   cout << f.display() << endl;

return 0;
}
```

When compiling the line:

```
Foo<int> f;
```

in main(), the compiler generates a class of the type int for Foo. It needs the definition to complete the work.

This is the reason why we normally place the entire class template in the header file, so that when the header file is included, both the declaration and the definition are available. However, this is not the only approach. A possible alternative is include the .cpp file.

Exercise 4:

Complete a class template named myVector which works in the similar way like vector. The code is available in Sample_Template_myVector.txt.

```
#include <iostream>

using namespace std;

struct intNode
{
   int item;
   intNode * next = nullptr;
   intNode(int x):item(x){}
};

class myVector
{
private:
   intNode * head = nullptr;
   intNode * tail = nullptr;
public:
   void push_back(int item);
   void pop_back();
   void display();
};

void myVector::push_back(int item)
{
   /// TO DO: Add your code here
   return;
}

void myVector::pop_back()
{
   /// TO DO: Add your code here
   return;
}
void myVector::display()
{
   intNode *p = head;
   while (p != nullptr)
   {
      cout << p->item << " ";
      p = p ->next;
   }

}
int main()
{
   myVector myVect;

   myVect.push_back(3);
   myVect.push_back(5);
   myVect.display();

   myVect.pop_back();
   myVect.display();

   return 0;
}
```