FAST NATIONAL UNIVERSTIY OF COMPUTER AND EMERGING SCIENCES, PESHAWAR

DEPARTMENT OF COMPUTER SCIENCE

CL217 – OBJECT ORIENTED PROGRAMMING LAB

Template Function and Template Class in C++

LAB MANUAL # 12

Instructor: Fariba Laiq

SEMESTER SPRING 2023

A **template** is a simple yet very powerful tool in C++. The simple idea is to pass the data type as a parameter so that we don't need to write the same code for different data types. For example, a software company may need to sort() for different data types. Rather than writing and maintaining multiple codes, we can write one sort() and pass the datatype as a parameter.

Special functions that can interact with generic types are known as function templates. As a result, we can develop a function template whose functionality can be applied to multiple types or classes without having to duplicate the full code for each kind.

Types of Templates in C++

There are two types of templates in C++

- **Function template**
- **Class templates**

C++ Function Template Syntax

template < **class** Ttype> ret_type func_name(parameter_list)
Here, type is a placeholder name for a data type used by the function. It is used within the function definition.

The class keyword is used to specify a generic type in a template declaration.

Function Templates with Multiple Parameters

We can use more than one generic type in the template function by using the comma to separate the list.

```
1. template<class T1, class T2,.....>
2. return_type function_name (arguments of type T1, T2....)
3. {
4.     // body of function.
5. }
```

Function Template

In the following code, we have created a template function that accepts an argument of any type, add them up and returns the result of that type.

```
#include <iostream>

using namespace std;

// template function

template <class T>

T add(T t1, T t2)

{

        return t1+t2;

}

int main() {

        int result1 = add(2,3);

        float result2 = add(2.6, 3.8);

        cout<<"Result1: "<<result1<<endl;

        cout<<"Result2: "<<result2<<endl;

}
```

Output:

Result1: 5

Result2: 6.4

Function Template that works with user defined Classes

```
#include <iostream>
using namespace std;
class Distance
{
	int meter;
	public:
	Distance(int meter)
	{
		this->meter=meter;
	}
	void display()
	{
		cout<<"Distance: "<<meter<<endl;
	}
	int getMeter()
	{
		return meter;
	}
	Distance operator+(Distance d)
	{
		int m = this->meter+d.meter;
		Distance d3(m);
		return d3;
	}
};
template <class T>
T add(T t1, T t2)
{
	return t1+t2;
}
int main() {

	int result1 = add(2,3);
	float result2 = add(2.6, 3.8);
	Distance d1(3);
	Distance d2(5);
	Distance result3=add(d1, d2);
	cout<<"Result1 (after adding two ints): "<<result1<<endl;
	cout<<"Result2 (after adding two floats): "<<result2<<endl;
	cout<<"Result3 (after adding two distances): "<<result3.getMeter()<<endl;
}
```

Output:

Result1 (after adding two ints): 5

Result2 (after adding two floats): 6.4

Result3 (after adding two distances): 8

In the above code, now we can add up objects as well but for that we must overload the + operator so that it works with the objects as well.

Overloading Function Templates

This code demonstrates the use of function templates in C++ to define a function that can take multiple types of parameters.

The add function is defined as a function template with two different versions. The first version takes two parameters of the same type T, and returns the sum of those two parameters. The second version takes two parameters of different types T1 and T2, and returns the sum of those two parameters.

In main(), the add function is used to perform addition on different types of data. First, the function is called with two integers as parameters, which invokes the first version of the function template. The result of the addition is stored in an int variable and printed to the console.

Next, the function is called with an integer and a float as parameters, which invokes the second version of the function template. The result of the addition is stored in a float variable and printed to the console.

Finally, the function is called with an object of the Distance class and an integer as parameters, which again invokes the second version of the function template. The Distance object is created with an integer value of 3, and its getMeter() method is used to extract the integer value. The integer value and the integer 5 are then passed to the add function, which returns the sum of the two values. The result of the addition is stored in an int variable and printed to the console.

Note that the Distance class also overloads the + operator to allow it to be added to an int value. The add function template can call this overloaded operator to perform the addition.

```
#include <iostream>
using namespace std;
class Distance
{
	int meter;
	public:
	Distance(int meter)
	{
		this->meter=meter;
	}
	void display()
	{
		cout<<"Distance: "<<meter<<endl;
	}
	int getMeter()
	{
		return meter;
	}
	float operator+(float meter)
	{
		float m = this->meter+meter;
		return m;
	}
};
// template function
template <class T>
T add(T t1, T t2)
{
	cout<<"Func 1 called"<<endl;
	return t1+t2;
}
template <class T1, class T2>
T2 add(T1 t1, T2 t2)
{
	cout<<"Func 2 called"<<endl;
	return (t1+t2);
}

int main() {
	int result1 = add(2,3);
	float result2 = add(2, 3.8);
	Distance d1(3);
	float result3=add(d1, 5.5);
	cout<<"Result1 (after adding two ints): "<<result1<<endl;
	cout<<"Result2 (after adding one int and one float): "<<result2<<endl;
	cout<<"Result3 (after adding distance and float): "<<result3<<endl;
}
```

Func 1 called

Func 2 called

Func 2 called

Result1 (after adding two ints): 5

Result2 (after adding one int and one float): 5.8

Result3 (after adding distance and float): 8.5

C++ Class Templates

Similar to function templates, we can use class templates to create a single class to work with different data types.

Class templates come in handy as they can make our code shorter and more manageable.

Class Template Declaration

A class template starts with the keyword template followed by template parameter(s) inside <> which is followed by the class declaration.

```
template <class T>
class className {
  private:
    T var;
    ... .. ...
  public:
    T functionName(T arg);
    ... .. ...
};
```

In the above declaration, T is the template argument which is a placeholder for the data type used, and class is a keyword.

Inside the class body, a member variable var and a member function functionName() are both of type

Creating a Class Template Object

Once we've declared and defined a class template, we can create its objects in other classes or functions (such as the main() function) with the following syntax

```
className<dataType> classObject;
```

For example,

```
className<int> classObject;
className<float> classObject;
className<string> classObject;
```

This code demonstrates the use of class templates in C++ to create a class that can hold values of different data types.

The Number class is defined as a class template that takes a single type parameter T. This class has one private member variable of type T, and a public method getNum() that returns the value of the private member variable.

In the main() function, two objects of the Number class are created using different data types as the type parameter. The first object is created with an int value of 7, and the second object is created with a double value of 7.7. The getNum() method is called on each object to retrieve the stored value, which is then printed to the console.

Note that the getNum() method has a return type of T, which means it will return the same data type as the type parameter T used to create the object. In this way, the Number class can hold and return values of any data type specified when creating the object.

```
// C++ program to demonstrate the use of class templates
#include <iostream>
using namespace std;
// Class template
template <class T>
class Number {
  private:
   // Variable of type T
   T num;
  public:
   Number(T num)
           {
                   this->num=num;
           }
   T getNum() {
      return num;
   }
};

int main() {

   // create object with int type
   Number<int> numberInt(7);
   // create object with double type
   Number<double> numberDouble(7.7);
   cout << "int Number = " << numberInt.getNum() << endl;
   cout << "double Number = " << numberDouble.getNum() << endl;
   return 0;
}
```

Output:

int Number = 7

double Number = 7.7

Notice that the variable num, the constructor argument n, and the function getNum() are of type T, or have a return type T. That means that they can be of any type.

Notice the codes Number<int> and Number<double> in the code above.

This creates a class definition each for int and float, which are then used accordingly.

It is a good practice to specify the type when declaring objects of class templates. Otherwise, some compilers might throw an error.

Defining a Class Member Outside the Class Template

Suppose we need to define a function outside of the class template. We can do this with the following code:

Notice that the code template <class T> is repeated while defining the function outside of the class. This is necessary and is part of the syntax.

If we look at the code in Example 1, we have a function getNum() that is defined inside the class template Number.

We can define getNum() outside of Number with the following code:

How to define a template Function Outside the class that is declared in the class

```
// C++ program to demonstrate the use of class templates

#include <iostream>

using namespace std;

// Class template
template <class T>
class Number {
        private:
                // Variable of type T
                T num;
        public:
                Number(T num) {
                        this->num=num;
                }
                T getNum();
};
int main() {

        // create object with int type
        Number<int> numberInt(7);
        // create object with double type
        Number<double> numberDouble(7.7);
        cout << "int Number = " << numberInt.getNum() << endl;
        cout << "double Number = " << numberDouble.getNum() << endl;
        return 0;
}
template <class T>
T Number<T>::getNum() {
        return num;
}
```

Output:

int Number = 7

double Number = 7.7

C++ Class Templates With Multiple Parameters

In C++, we can use multiple template parameters and even use default arguments for those parameters. **For example:**

```
#include <iostream>
using namespace std;
// Class template with multiple and default parameters
template <class T, class U, class V = char>
class ClassTemplate {
  private:
   T var1;
   U var2;
   V var3;
  public:
   ClassTemplate(T v1, U v2, V v3) : var1(v1), var2(v2), var3(v3) {}  //
constructor
   void printVar() {
      cout << "var1 = " << var1 << endl;
      cout << "var2 = " << var2 << endl;
      cout << "var3 = " << var3 << endl;
   }
};

int main() {
   // create object with int, double and char types
   ClassTemplate<int, double> obj1(7, 7.7, 'c');
   cout << "obj1 values: " << endl;
   obj1.printVar();

   // create object with int, double and bool types
   ClassTemplate<double, char, bool> obj2(8.8, 'a', false);
   cout << "\nobj2 values: " << endl;
   obj2.printVar();
   return 0;
}
```

Output:

```
obj1 values:
var1 = 7
var2 = 7.7
var3 = c
obj2 values:
var1 = 8.8
var2 = a
var3 = 0
```

In this program, we have created a class template, named ClassTemplate, with three parameters, with one of them being a default parameter. Notice the code class V = char. This means that V is a default parameter whose default type is char.

Inside ClassTemplate, we declare 3 variables var1, var2 and var3, each corresponding to one of the template parameters.

Class Template with Array as a Data Member that can be of any primitive type

```
#include <iostream>
using namespace std;
template<class T>
class A {
	public:
		int size;
		T *arr;
		A(int size) {
			this->size=size;
			arr=new T(size);
		}
		void insert() {
			cout<<"\n";
			for (int i=0; i<size; i++) {
				cout<<"Enter a value at index: "<<i<<endl;
				cin>>arr[i];
			}
		}

		void display() {
			cout<<"\n";
			for(int i=0; i<size; i++) {
				cout << arr[i] << " ";
			}
		}
};
int main() {
	A<int> t1(4);
	t1.insert();
	t1.display();
	A<float> t2(4);
	t2.insert();
	t2.display();
	return 0;
}
```

Output:

```
Enter a value at index: 0
2
Enter a value at index: 1
3
Enter a value at index: 2
1
Enter a value at index: 3
5
2 3 1 5
Enter a value at index: 0
6.6
Enter a value at index: 1
3.3
Enter a value at index: 2
1.1
Enter a value at index: 3
7.4
6.6 3.3 1.1 7.4
```

Class Template with Array as a Data Member that can be of any primitive or Custom Datatype

This code demonstrates the use of class templates and operator overloading in C++ to create a generic array class that can hold values of any data type, including custom classes. The A class is defined as a class template with a single type parameter T. This class has a private member variable arr of type T*, which is a dynamically allocated array to hold the values. The insert() method prompts the user to input values for each element of the array, and the display() method prints the values of the array to the console. Two additional methods are defined outside of the class: the >> operator overload, which takes user input to create a Student object, and the << operator overload, which outputs the name and marks of a Student object to the console. In main(), three instances of the A class are created for int, float, and Student data types. The user is prompted to input values for each instance, and the values are then printed to the console using the display() method.

```
#include <iostream>
using namespace std;
class Student {
		string name;
		int marks;
	public:
		Student(string name, int marks) {
			this->name=name;
			this->marks=marks;
		}
			Student() {

		}
		friend ostream & operator <<(ostream &, Student &s);
		friend istream & operator >>(istream &, Student &s);
};
template<class T>
class A {
	public:
		int size;
		T *arr;
		A(int size) {
			this->size=size;
			arr=new T[size];
		}
		void insert() {
			cout<<"\n";
			for (int i=0; i<size; i++) {
				cout<<"Enter a value at index: "<<i<<endl;
				cin >> arr[i];
			}
		}

		void display() {
			cout<<"\n";
			for(int i=0; i<size; i++) {
				cout << arr[i] << " ";
			}
		}
};
```

```
ostream & operator <<(ostream &os, Student &s) {
	os<<"Name: "<<s.name<<" Marks: "<<s.marks<<endl;
	return os;
}

istream & operator >>(istream &is, Student &s) {
	cout<<"\nEnter Name: ";
	is>>s.name;
	cout<<"Enter Marks: ";
	is>>s.marks;
	return is;
}
int main() {
	cout<<"\nFor int values:"<<endl;
	A<int> t1(2);
	t1.insert();
	t1.display();

	cout<<"\nFor float values:"<<endl;
	A<float> t2(2);
	t2.insert();
	t2.display();


	cout<<"\nFor student values:"<<endl;
	A<Student> t3(2);
	t3.insert();
	t3.display();

	return 0;
}
```

Output:

For int values:
Enter a value at index: 0
2
Enter a value at index: 1
4
2 4
For float values:
Enter a value at index: 0

3.3
Enter a value at index: 1
5.5
3.3 5.5
For student values:
Enter a value at index: 0
Enter Name: Fari
Enter Marks: 50
Enter a value at index: 1
Enter Name: Ali
Enter Marks: 60
Name: Fari Marks: 50
Name: Ali Marks: 60

References:

https://www.programiz.com/cpp-programming/class-templates