OBJECT ORIENTED PROGRAMMING LAB

Lab Manual # 15

Templates (Function and Class Templates) in C++

Instructor: Muhammad Abdullah Orakzai

Semester Fall, 2021

Course Code: CL1004

Fast National University of Computer and Emerging Sciences
Peshawar

Department of Computer Science

OBJECT ORIENTED PROGRAMMING LANGUAGE

Table of Contents

C++ Templates 1

1. Function Templates 1

Function Template Declaration 1

Example 1: Function Template to sum of three values 2

Example 2: Function Template to find the largest number 3

Example 3: Swap Data Using Function Templates 5

2. Class Templates 7

Class Template Declaration 7

Creating a Class Template Object 7

Example 1: C++ Class Templates 8

Defining a Class Member Outside the Class Template 10

Example 2: Simple Calculator Using Class Templates 11

C++ Class Templates with Multiple Parameters 13

Example 3: C++ Templates with Multiple Parameters 14

References 16

C++ Templates

In this topic, you'll learn about templates in C++. You'll learn to use the power of templates for generic programming.

Templates are powerful features of C++ which allows you to write generic programs. In simple terms, you can create a single function or a class to work with different data types using templates.

Templates are often used in larger codebase for the purpose of code reusability and flexibility of the programs.

The concept of templates can be used in two different ways:

- Function Templates
- Class Templates

1. Function Templates

A function template works in a similar to a normal function, with one key difference.

A **single function template** can work with different data types at once but, a **single normal function** can only work with one set of data types.

Normally, if you need to perform identical operations on two or more types of data, you use function overloading to create two functions with the required function declaration.

However, a better approach would be to use function templates because you can perform the same task writing less and maintainable code.

In function overloading more than one function with the same name are declared and defined. This makes the program more complex and lengthy. To overcome this problem, function template is used.

Function Template Declaration

In a function template a single typeless function is defined such that arguments of any standard data type can be passed to the function.

The general syntax of a function template is:

```
template <class T>

T function_name(T  argument(s))

{

   Statement(s);

}
```

template The keyword template is used to define the function template.

class In function templates a general data name is defined by using the keyword "**class**". In the above syntax, it is represented by word "T". Any character or word can be used after the keyword "**class**" to specify the general data name.

function_name It specifies the name of the function.

argument(s) These are the arguments that are to be passed to the function. The arguments are declared of type **T**. As explained earlier, the type **T** specifies any standard data type.

statement(s) It specifies the actual statements of the function.

For example, to write a function to calculate a sum of three values either integers or float-points, the function template is written as:

```
template <class T>
T sum(T a, T b, T c)
{
  return (a+b+c);
}
```

Example 1: Function Template to sum of three values

Write a program to calculate and print the sum of three integer and floating-point values. Use the function template.

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

template <class T>
T sum(T a, T b, T c)
{
   return (a+b+c);
}

int main()
{
```

```
    cout<<"Sum of 3 integer values ="<<sum(33,44,55)<<endl;

    cout<<"Sum of 3 float values ="<<sum(33.5,44.55,55.66)<<endl;
    return 0;

}
```

Output:

```
Sum of 3 integer values =132

Sum of 3 float values =133
```

Note: In template functions, the function prototype is generally omitted.

Example 2: Function Template to find the largest number

//Program to display largest value among two numbers using function templates.

```
// If two characters are passed to function template, character with larger
ASCII value is displayed.

#include <iostream>

using namespace std;

// template function

template <class T>

T Large(T n1, T n2)

{

      return (n1 > n2) ? n1 : n2;

}

int main()

{

      int i1, i2;

      float f1, f2;

      char c1, c2;

      cout << "Enter two integers:\n";

      cin >> i1 >> i2;

      cout << Large(i1, i2) <<" is larger." << endl;
```

```
      cout << "\nEnter two floating-point numbers:\n";
      cin >> f1 >> f2;
      cout << Large(f1, f2) <<" is larger." << endl;

      cout << "\nEnter two characters:\n";
      cin >> c1 >> c2;
      cout << Large(c1, c2) << " has larger ASCII value.";
      return 0;
}
/* Output
Enter two integers:
5
10
10 is larger.
Enter two floating-point numbers:
12.4
10.2
12.4 is larger.
Enter two characters:
z
Z
z has larger ASCII value.
*/
```

- In the above program, a function template Large() is defined that accepts two arguments n1 and n2 of data type T. T signifies that argument can be of any data type.
- Large() function returns the largest among the two arguments using a simple conditional operation.
- Inside the main() function, variables of three different data types: int, float and char are declared. The variables are then passed to the Large() function template as normal functions.
- During run-time, when an integer is passed to the template function, compiler knows it has to generate a Large() function to accept the int arguments and does so.

- Similarly, when floating-point data and char data are passed, it knows the argument data types and generates the Large() function accordingly.
- This way, using only a single function template replaced three identical normal functions and made your code maintainable.

Example 3: Swap Data Using Function Templates

//Program to swap data using function templates.

```
#include <iostream>
using namespace std;
template <typename T>
void Swap(T &n1, T &n2)
{
      T temp;
      temp = n1;
      n1 = n2;
      n2 = temp;
}
int main()
{
      int i1 = 1, i2 = 2;
      float f1 = 1.1, f2 = 2.2;
      char c1 = 'a', c2 = 'b';
      cout << "Before passing data to function template.\n";
      cout << "i1 = " << i1 << "\ni2 = " << i2;
      cout << "\nf1 = " << f1 << "\nf2 = " << f2;
      cout << "\nc1 = " << c1 << "\nc2 = " << c2;
      Swap(i1, i2);
      Swap(f1, f2);
      Swap(c1, c2);
     cout << "\n\nAfter passing data to function template.\n";
      cout << "i1 = " << i1 << "\ni2 = " << i2;
```

```
	cout << "\nf1 = " << f1 << "\nf2 = " << f2;
	cout << "\nc1 = " << c1 << "\nc2 = " << c2;


	return 0;
}
/* Output
Before passing data to function template.
i1 = 1
i2 = 2
f1 = 1.1
f2 = 2.2
c1 = a
c2 = b
After passing data to function template.
i1 = 2
i2 = 1
f1 = 2.2
f2 = 1.1
c1 = b
c2 = a
*/
```

In this program, instead of calling a function by passing a value, a call by reference is issued.

The Swap() function template takes two arguments and swaps them by reference.

2. 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 `T`.

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;
```

Example 1: C++ Class Templates

```
// 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 n) : num(n) {}   // constructor


    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

*/
```

In this program. we have created a class template **`Number`** with the code

```
template <class T>
class Number {
   private:
    T num;

   public:
    Number(T n) : num(n) {}
    T getNum() { return num; }
};
```

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.

In **`main()`**, we have implemented the class template by creating its objects

```
Number<int> numberInt(7);
Number<double> numberDouble(7.7);
```

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 compulsory to specify the type when declaring objects of class templates. Otherwise, the compiler will produce an error.

```
//Error
Number numberInt(7);
Number numberDouble(7.7);
```

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:

```
template <class T>
class ClassName {
    ... .. ...
    // Function prototype
    returnType functionName();
};
// Function definition
template <class T>
returnType ClassName<T>::functionName() {
    // 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:

```
template <class T>
class Number {
    ... .. ...
    // Function prototype
    T getnum();
};
// Function definition
template <class T>
T Number<T>::getNum() {
    return num;
}
```

Example 2: Simple Calculator Using Class Templates

This program uses a class template to perform addition, subtraction, multiplication and division of two variables **num1** and **num2**.

The variables can be of any type, though we have only used **int** and **float** types in this example.

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

template <class T>
class Calculator {
   private:
    T num1, num2;

   public:
    Calculator(T n1, T n2) {
        num1 = n1;
        num2 = n2;
    }

    void displayResult() {
        cout << "Numbers: " << num1 << " and " << num2 << "." << endl;
        cout << num1 << " + " << num2 << " = " << add() << endl;
        cout << num1 << " - " << num2 << " = " << subtract() << endl;
        cout << num1 << " * " << num2 << " = " << multiply() << endl;
        cout << num1 << " / " << num2 << " = " << divide() << endl;
    }

    T add() { return num1 + num2; }
    T subtract() { return num1 - num2; }
    T multiply() { return num1 * num2; }
    T divide() { return num1 / num2; }
};
```

```
int main() {
    Calculator<int> intCalc(2, 1);
    Calculator<float> floatCalc(2.4, 1.2);

    cout << "Int results:" << endl;
    intCalc.displayResult();

    cout << endl
         << "Float results:" << endl;
    floatCalc.displayResult();

    return 0;
}
/*
```

Output

```
Int results:
Numbers: 2 and 1.
2 + 1 = 3
2 - 1 = 1
2 * 1 = 2
2 / 1 = 2

Float results:
Numbers: 2.4 and 1.2.
2.4 + 1.2 = 3.6
2.4 - 1.2 = 1.2
2.4 * 1.2 = 2.88
2.4 / 1.2 = 2
*/
```

In the above program, we have declared a class template **`Calculator`**.

The class contains two private members of type **`T`**: **`num1`** **&** **`num2`**, and a constructor to initialize the members.

We also have **`add()`**, **`subtract()`**, **`multiply()`**, and **`divide()`** functions that have the return type **`T`**. We also have a **`void`** function **`displayResult()`** that prints out the results of the other functions.

In **`main()`**, we have created two objects of **`Calculator`**: one for **`int`** data type and another for **`float`** data type.

```
Calculator<int> intCalc(2, 1);
Calculator<float> floatCalc(2.4, 1.2);
```

This prompts the compiler to create two class definitions for the respective data types during compilation.

C++ Class Templates with Multiple Parameters

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

```
template <class T, class U, class V = int>
class ClassName {
  private:
    T member1;
    U member2;
    V member3;
    ... .. ...
  public:
    ... .. ...
};
```

Example 3: C++ Templates with Multiple Parameters

```
#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:

    // constructor

    ClassTemplate(T v1, U v2, V v3) : var1(v1), var2(v2), var3(v3) {}


    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.

```
var2 = a
template <class T, class U, class V = char>
class ClassTemplate {
  // code
};
```

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 ClassTemplate {

   private:

    T var1;

    U var2;

    V var3;

    ... .. ...

    ... .. ...

};
```

In **`main()`**, we create two objects of **`ClassTemplate`** with the code

```
// create object with int, double and char types
ClassTemplate<int, double> obj1(7, 7.7, 'c');

// create object with double, char and bool types
ClassTemplate<double, char, bool> obj2(8, 8.8, false);
```

Here,

Object	T	U	V
obj1	Int	double	char
obj2	Double	char	bool

For obj1, T = int, U = double and V = char.

For obj2, T = double, U = char and V = bool.

References

https://beginnersbook.com/2017/08/cpp-data-types/

http://www.cplusplus.com/doc/tutorial/basic_io/

https://www.w3schools.com/cpp/default.asp

https://www.javatpoint.com/cpp-tutorial

https://www.geeksforgeeks.org/object-oriented-programming-in-cpp/?ref=lbp

https://www.programiz.com/

https://ecomputernotes.com/cpp/