BAKI ALİ NEFT MƏKTƏBİ
BAKU HIGHER OIL SCHOOL

Programming and Computer Applications-2

Templates

Instructor : PhD, Associate Professor Leyla Muradkhanli

Function Templates

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**

Function Templates

- All function-template definitions begin with keyword `template` followed by a list of template parameters to the function template enclosed in angle brackets (`<` and `>`); each template parameter that represents a type must be preceded by either of the interchangeable keywords `class` or `typename`, as in
    - `template<typename T>`
  - Or
    - `template<class ElementType>`
  - Or
    - `template<typename BorderType, typename FillType>`
- The type template parameters of a function-template definition are used to specify the types of the arguments to the function, to specify the return type of the function and to declare variables within the function.
- Keywords `typename` and `class` used to specify function-template parameters actually mean “any fundamental type or user-defined type.”

Function Templates

```
#include <iostream>
using namespace std;
template <typename T>
void show(T data)
{
cout<<data<<endl;
}
int main () {

   show(25);
   show(17.38);
   show("Programming");
   return 0;
 }
```

Output

```
25
17.38
Programming
```

Function Templates

```
#include <iostream>
using namespace std;
template <class T>
T square(T number)
{
   return number * number;
}
int main()
{
   int myint;
   double mydouble;
   cout << "Enter an integer and a floating-point value: ";
   cin >> myint >> mydouble;
   cout << "Here are their squares: ";
   cout << square(myint) << " and "
        << square(mydouble) << endl;
   return 0;
}
```

Function Templates

Output:

```
Enter an integer and a floating-point value:  5  7.7
Here are their squares: 25 and 59.29
```

Function Templates

```
// This program demonstrates the swapVars function template.
#include <iostream>
using namespace std;
template <typename T>
void swapVars(T &var1, T &var2)
{
   T temp;
   temp=var1;
   var1 = var2;
   var2 = temp;
}
int main()
{
   char ch1, ch2;
   int i1, i2;
   double d1, d2;
```

Function Templates

```
// Get and swapVars two chars
 cout << "Enter two characters: ";
 cin >> ch1 >> ch2;
 swapVars(ch1, ch2);
 cout << ch1 << " " << ch2 << endl;

// Get and swapVars two ints
 cout << "Enter two integers: ";
 cin >> i1 >> i2;
 swapVars(i1, i2);
 cout << i1 << " " << i2 << endl;
```

Function Templates

```
  // Get and swapVars two doubles
    cout << "Enter two floating-point numbers: ";
    cin >> d1 >> d2;
    swapVars(d1, d2);
    cout << d1 << " " << d2 << endl;
    return 0;
}
```

Output:

```
Enter two character:  a  c
c  a
Enter two integers:  3  9
9  3
Enter two floating-point numbers:  4.5  8.9
8.9  4.5
```

Overloaded Function Templates

```
// This program demonstrates an overloaded function template.
#include <iostream>
using namespace std;

template <class T>
T sum(T val1, T val2)
{
   return val1 + val2;
}

template <class T>
T sum(T val1, T val2, T val3)
{
   return val1 + val2 + val3;
}
```

Overloaded Function Templates

```
int main()
{
   double num1, num2, num3;

   // Get two values and display their sum.
   cout << "Enter two values: ";
   cin >> num1 >> num2;
   cout << "Their sum is " << sum(num1, num2) << endl;

   // Get three values and display their sum.
   cout << "Enter three values: ";
   cin >> num1 >> num2 >> num3;
   cout << "Their sum is " << sum(num1, num2, num3) << endl;
   return 0;
}
```

Overloaded Function Templates

Output:

```
Enter two values:  5  8
Their sum is 13
Enter three values: 5  8  9
Their sum is 22
```

Class Templates declaration

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

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

Inside the class body, a member variable **var** and a member function **someOperation()** are both of type **T.**

How to create a class template object?

To create a class template object, you need to define the data type inside a < > when creation.

className<dataType> classObject;

For example:

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

Example : Simple calculator using Class template

Program to add, subtract, multiply and divide two numbers using class template.

```
#include <iostream>
using namespace std;
template <class T>
class Calculator
{
private:
T num1, num2;
public:
Calculator(T n1, T n2)
{
num1 = n1;
num2 = n2;
}
```

Example : Simple calculator using Class template

```
void displayResult()
{
cout << "Numbers are: " << num1 << " and " << num2 << "." << endl;
cout << "Addition is: " << add() << endl;
cout << "Subtraction is: " << subtract() << endl;
cout << "Product is: " << multiply() << endl;
cout << "Division is: " << divide() << endl;
}
T add() { return num1 + num2; }
T subtract() { return num1 - num2; }
T multiply() { return num1 * num2; }
T divide() { return num1 / num2; }
};
```

Example : Simple calculator using Class template

```
int main()
{
Calculator<int> intCalc(5, 2);
Calculator<float> floatCalc(2.4, 1.2);
cout << "Int results:" << endl;
intCalc.displayResult();

cout << endl << "Float results:" << endl;
floatCalc.displayResult();
return 0;
}
```

Example : Simple calculator using Class template

Output:

```
Int results:
Numbers are: 5 and 2.
Addition is: 7
Subtraction is: 3
Product is: 10
Division is: 2

Float results:
Numbers are: 2.4 and 1.2.
Addition is: 3.6
Subtraction is: 1.2
Product is: 2.88
Division is: 2
```

Example : Simple calculator using Class template

In the above program, a class template **Calculator** is declared.

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

It also contains public member functions to calculate the addition, subtraction, multiplication and division of the numbers which return the value of data type defined by the user. Likewise, a function **displayResult()** to display the final output to the screen.

In the **main()** function, two different **Calculator** objects **intCalc** and **floatCalc** are created for data types: **int** and **float** respectively. The values are initialized using the constructor.

Notice we use **<int>** and **<float>** while creating the objects. These tell the compiler the data type used for the class creation.

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

Then, **displayResult()** of both objects is called which performs the Calculator operations and displays the output.

Class Template With Multiple Parameters

We can use more than one generic data type in a class template, and each generic data type is separated by the comma.

Syntax

```
template<class T1, class T2, ......>
class class_name
{
    // Body of the class.
}
```

Class Template With Multiple Parameters

```
#include <iostream>
     using namespace std;
     template<class T1, class T2>
    class A
    {
         T1 a;
         T2 b;
         public:
        A(T1 x,T2 y)
       {
           a = x;
           b = y;
        }
           void display()
          {
                 cout << "Values of a and b are : " << a<<" ,"<<b<<endl;
           }
      };
```

Class Template With Multiple Parameters

```
int main()
    {
          A<int,float> obj1(5,3.5);
          obj1.display();
  A<int,int> obj2(10,7);
          obj2.display();
     return 0;
    }
```

Nontype Template Arguments

The template can contain multiple arguments, and we can also use the non-type arguments In addition to the type T argument, we can also use other types of arguments such as strings, function names, constant expression and built-in types. Let' s see the following example:

```
template<class T, int size>
class array
{
        T arr[size]; // automatic array initialization.
};
```

Nontype Template Arguments

In the above case, the nontype template argument is size and therefore, template supplies the size of the array as an argument.

Arguments are specified when the objects of a class are created:

```
array<int, 15> t1;     // array of 15 integers.
array<float, 10> t2; // array of 10 floats.
array<char, 4> t3;  // array of 4 chars.
```

Nontype Template Arguments

```
#include <iostream>
using namespace std;
template<class T, int size>
class A
{
    public:
    T arr[size];
    void insert()
    {
        int n =1;
        for (int i=0;i<size;i++)
        {
            arr[i] = n;
            n++;
        }
    }
void display()
    {
        for(int i=0;i<size;i++)
        {
            cout << arr[i] << " ";
        }
    }
};
int main()
{
    A<int,10> obj;
    obj.insert();
    obj.display();
    return 0;
}
```