Himalaya College of Engineering

Advanced C++ Programming Lab Report

Lab 8: Templates in C++

Prepared By : Nawnit Paudel(HCE081BEI024)

Subject : Object-Oriented Programming (OOP)

Program : Bachelor of Electronics, Communication and Information Engineering

Institution : Himalaya College of Engineering

OBJECTIVE

- To understand the concept of Templates in C++.

BACKGROUND THEORY

Templates in C++:

Templates are a powerful feature in C++ that allow us to write generic programming. They enable us to define functions and classes that operate with generic types, rather than specific ones which helps in polymorphism which minimizes code length.

Types of Templates:

A. Function Templates: These are templates used to create generic functions that can operate on different data types without being rewritten for each type. The compiler generates specific versions of the function for each data type used with the template.

Syntax:

```
template <typename T>
T functionName(T arg1, T arg2) {     //
Function body operating on type T
return arg1 + arg2;
}
```

B. Class Templates: These are templates used to create generic classes that can hold or operate on different data types. You can define a class template once and then create objects of that class for various data types.

Syntax:

```
template <typename T>
class ClassName { public:
   T memberVariable;
   ClassName(T val) : memberVariable(val) {}
void memberFunction() {
      // Function body operating on type T
   }
};
```

Rules of Using Templates:

1. **Template Declaration:** Templates are declared using the template keyword followed by angle brackets <> containing one or more template parameters. These parameters are typically denoted by typename or class (they are interchangeable in this context) followed by an identifier (e.g., T, U, N).
2. **Type Deduction (for Function Templates):** For function templates, the compiler can often deduce the actual types of the template arguments from the types of the function arguments. Explicitly specifying template arguments for function templates is optional but can be done.
3. **Explicit Instantiation (for Class Templates):** For class templates, you must explicitly specify the type(s) when creating an object of the template class (e.g., ClassName<int> obj;).
4. **Specialization:** Templates can be specialized for specific types. This allows you to provide a different implementation for a particular type if the generic implementation is not suitable or needs to be optimized for that type.
5. **Non-type Template Parameters:** Besides type parameters, templates can also have non-type template parameters, which are compile-time constants (e.g., template <typename T, int N>).
6. **Templates and Inheritance:** Class templates can participate in inheritance hierarchies, just like regular classes. A derived class can inherit from a base class template, or a class template can inherit from a regular class.
7. **Header Files:** It is common practice to define template functions and class template member functions entirely within header files. This is because the compiler needs access to the template's full definition at the point of instantiation to generate the specific code for the types being used.

Example:

```
#include <iostream>
#include <string> using
namespace std;
// Function Template to find the maximum of two values
template <typename T> T findMax(T a, T b) {
   return (a > b) ? a : b;
}
// Class Template for a simple Pair template
<typename T1, typename T2>
class Pair { public:
   T1 first;
   T2 second;
   Pair(T1 f, T2 s) : first(f), second(s) {}     void printPair() const
{        cout << "First: " << first << ", Second: " << second <<
endl;
   } }; int
main() {
   // Using Function Template     int
maxInt = findMax(10, 20);     cout <<
"Max Int: " << maxInt << endl;

   double maxDouble = findMax(3.14, 2.71);
cout << "Max Double: " << maxDouble << endl;
   string str1 = "hello";     string str2 =
"world";     string maxString =
findMax(str1, str2);     cout << "Max
String: " << maxString << endl;

   // Using Class Template
Pair<int, double> p1(100, 5.67);
p1.printPair();
```

```
    Pair<string, char> p2("Template", 'T');
p2.printPair();     Pair<double, int>
p3(99.9, 42);     p3.printPair();     return
0;
}
```

LAB ASSIGNMENTS:

1. Write a function template swapValues() that swaps two variables of any data type. Demonstrate its use with int, float, and char.

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

template <typename T>
void swapValues(T& a, T& b) {
    T temp = a;
    a = b;
    b = temp;
}

int main() {
    int x = 5, y = 10;
    float f1 = 1.5f, f2 = 2.5f;
    char c1 = 'A', c2 = 'B';

    swapValues(x, y);
    cout << "Swapped int: x = " << x << ", y = " << y << endl;

    swapValues(f1, f2);
    cout << "Swapped float: f1 = " << f1 << ", f2 = " << f2 << endl;

    swapValues(c1, c2);
```

```
    cout << "Swapped char: c1 = " << c1 << ", c2 = " << c2 << endl;

    return 0;
}
```

2. Write a program to overload a function template maxValue() to find the maximum of two values (for same type) and three values (for same type). Call it using int, double, and char.

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

template <typename T>
T maxValue(T a, T b) {
    return (a > b) ? a : b;
}

template <typename T>
T maxValue(T a, T b, T c) {
    return maxValue(maxValue(a, b), c);
}

int main() {
    int a, b, c;
    char ch1, ch2, ch3;
    double d1, d2, d3;
    float f1, f2, f3;

    cout << "Enter 2 integers: ";
    cin >> a >> b;
    cout << "Enter the third integer: ";
    cin >> c;
```

```
cout << "Enter 2 characters: ";
cin >> ch1 >> ch2;
cout << "Enter the third character: ";
cin >> ch3;

cout << "Enter 2 doubles: ";
cin >> d1 >> d2;
cout << "Enter the third double: ";
cin >> d3;

cout << "Enter 2 float numbers: ";
cin >> f1 >> f2;
cout << "Enter the third float: ";
cin >> f3;

cout << "Max of 2 integers: " << maxValue(a, b) << endl;
cout << "Max of 3 integers: " << maxValue(a, b, c) << endl;

cout << "Max of 2 doubles: " << maxValue(d1, d2) << endl;
cout << "Max of 3 doubles: " << maxValue(d1, d2, d3) << endl;

cout << "Max of 2 chars: " << maxValue(ch1, ch2) << endl;
cout << "Max of 3 chars: " << maxValue(ch1, ch2, ch3) << endl;

cout << "Max of 2 floats: " << maxValue(f1, f2) << endl;
cout << "Max of 3 floats: " << maxValue(f1, f2, f3) << endl;

return 0;
```

```
}
```

3. Create a class template Calculator<T> that performs addition, subtraction, multiplication, and division of two data members of type T. Instantiate it with int and float.

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

template <typename T>
class Calculator {
   T a, b;
public:
   Calculator(T x, T y) : a(x), b(y) {}

   T add() {
      return a + b;
   }

   T subtract() {
      return a - b;
   }

   T multiply() {
      return a * b;
   }

   T divide() {
      return a / b;
   }
};

int main() {
   int a, b;
   cout << "Enter 2 integers: ";
   cin >> a >> b;
   Calculator<int> calcInt(a, b);
   cout << "Addition: " << calcInt.add() << endl;
   cout << "Subtraction: " << calcInt.subtract() << endl;
   cout << "Multiplication: " << calcInt.multiply() << endl;
   cout << "Division: " << calcInt.divide() << endl;

   float c, d;
   cout << "Enter 2 float numbers: ";
```

```
    cin >> c >> d;
    Calculator<float> calcFloat(c, d);
    cout << "Addition: " << calcFloat.add() << endl;
    cout << "Subtraction: " << calcFloat.subtract() << endl;
    cout << "Multiplication: " << calcFloat.multiply() << endl;
    cout << "Division: " << calcFloat.divide() << endl;

    return 0;
}
```

DISCUSSION:

Templates in C++ let us write functions and classes that can work with any data type. Instead of writing separate code for int, float, or char, we can use templates to handle all types with one piece of code. This makes programming easier, faster, and reduces repetition.

CONCLUSION:

Templates in C++ help us write generic programs that work with any data type. They increase code reusability and make our programs more flexible. By using templates, we save time and avoid repeating the same code for different data types.You can remember templates as a smart way to write one solution that works for many data types.