



Himalaya College of Engineering

Advanced C++ Programming Lab Report

Lab 7: TEMPLATES

Prepared By: Ankit Belbase(HCE081BEI007)

Subject: Object-Oriented Programming (OOP)

Program: Bachelor of Electronics Engineering

Institution: Himalaya College of Engineering

Date: July 13,2025

Objective:

1. To understand the concept of templates in C++ programming.
2. To observe how templates improve code reusability and maintainability.
3. To write code that works with multiple data types using a single template.
4. To learn how to create and use function templates.

Introduction:

Templates in C++ are used to create generic functions and classes. Instead of writing the same logic multiple times for different data types (like int, float, char), templates allow writing one single function or class that works with any data type. This feature is part of generic programming in C++.

Types of Templates:**1. Function Template:**

A function template works with different data types using a single code structure.

Example: You can call this function for `int`, `float`, `double`, etc.

```
template <typename T>

T add(T a, T b) {

    return a + b;

}
```

2. Class Template:

A class template allows defining a generic class where member functions and variables can work with any data type.

Example:

```
template <class T>

class Box {

    T value;

public:
```

```
void set(T v) { value = v; }  
T get() { return value; }  
};
```

Advantages of Using Templates:

- **Code Reusability:** One function/class works for all types.
- **Type Flexibility:** Supports any data type without rewriting code.
- **Compile-time Safety:** Type checking happens during compilation.
- **Easy Maintenance:** One change updates all type versions.

Real life example :

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

```
int main() {  
    cout << maximum(3, 7) << endl;    // int  
    cout << maximum(4.5, 2.1) << endl; // float  
    cout << maximum('a', 'z') << endl; // char  
    return 0;  
}
```

Templates are a powerful feature of C++ that help write efficient, reusable, and type-independent code. They are especially useful when the same logic needs to be applied to multiple data types, saving both time and effort while keeping the code clean and manageable.

LAB QUESTIONS:

Ques 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;
}

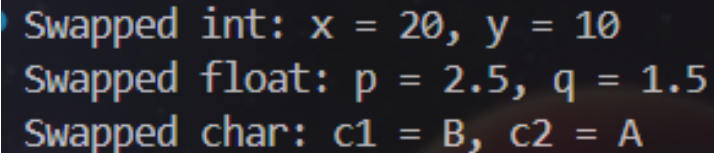
int main() {

int x = 10, y = 20;
swapValues(x, y);
cout << "Swapped int: x = " << x << ", y = " << y << endl;


float p = 1.5, q = 2.5;
swapValues(p, q);
cout << "Swapped float: p = " << p << ", q = " << q << endl;


char c1 = 'A', c2 = 'B';
swapValues(c1, c2);
cout << "Swapped char: c1 = " << c1 << ", c2 = " << c2 << endl;
return 0;
}
```

Output:



```
Swapped int: x = 20, y = 10
Swapped float: p = 2.5, q = 1.5
Swapped char: c1 = B, c2 = A
```

Ques 2. Write a program to overload a function template maxValuе() 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 maxValuе(T a,Tb)
{
    return (a > b) ? a : b;
}

template <typename
T> T maxValuе(T a, T
b, T c) {
    T max = (a > b) ? a : b;
    return (max > c) ? max : c;
}

int main() {
    int i1 = 5, i2 = 10, i3 = 7;
    cout << "Max of 2 int: " << maxValuе(i1, i2) <<
endl; cout << "Max of 3 int: " << maxValuе(i1, i2, i3)
<< endl; float f1 = 3.3, f2 = 7.2, f3 = 5.5;
    cout << "Max of 2 float: " << maxValuе(f1, f2) <<
endl; cout << "Max of 3 float: " << maxValuе(f1, f2,
f3) << endl; char c1 = 'A', c2 = 'Z', c3 = 'M';
    cout << "Max of 2 chars: " << maxValuе(c1, c2) <<
endl; cout << "Max of 3 chars: " << maxValuе(c1, c2,
c3) << endl;
    return 0;
}
```

```
Max of 2 int: 10
Max of 3 int: 10
Max of 2 float: 7.2
Max of 3 float: 7.2
Max of 2 chars: Z
Max of 3 chars: Z
```

Ques 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 b != 0 ? a / b : 0;
    }
};

int main() {
    Calculator<int> calcInt(10, 5);
    cout << "Int Addition: " << calcInt.add() << endl;
    cout << "Int Subtraction: " << calcInt.subtract() <<
    endl; cout << "Int Multiplication: " <<
    calcInt.multiply() << endl; cout << "Int Division: " <<
    calcInt.divide() << endl; Calculator<float>
    calcFloat(10.5f, 2.5f);
    cout << "Float Addition: " << calcFloat.add() << endl;
    cout << "Float Subtraction: " << calcFloat.subtract() <<
    endl; cout << "Float Multiplication: " <<
    calcFloat.multiply() << endl; cout << "Float Division: " <<
    calcFloat.divide() << endl;
    return 0;
}
{

```

```

Int Addition: 15
Int Subtraction: 5
Int Multiplication: 50
Int Division: 2
Float Addition: 13
Float Subtraction: 8
Float Multiplication: 26.25
Float Division: 4.2

```

Ques 4. Define a class template Base<T> with a protected data member and a member function to display it. Derive a class Derived<T> from it, add another data member, and display both data members. Use string and int types to test.

```
#include<iostream>
#include <string>
using namespace
std;
template <typename
T> class Base {
protected:
    T data1;
public:
    Base(T val) {
        data1 = val;
    }
    void displayData1() {
        cout << "Data1: " << data1 << endl;
    }
};
template <typename T>
class Derived : public
    Base<T> { T data2;
public:
    Derived(T val1, T val2) : Base<T>(val1) {
        data2 = val2;
    }

    void displayBoth() {
```



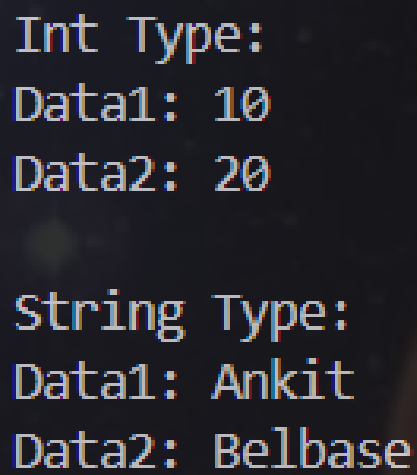
```

        this->displayData1();

        cout << "Data2: " << data2 << endl;
    }
};

int main() {
    Derived<int> objInt(10,
        20); cout << "Int Type:"
        << endl;
    objInt.displayBoth();
    Derived<string> objStr("Ankit",
        "Belbase");
    cout << "\nString Type:" << endl;
    objStr.displayBoth();
    return 0;
}

```



```

Int Type:
Data1: 10
Data2: 20

String Type:
Data1: Ankit
Data2: Belbase

```

DISCUSSION:

Templates in C++ allow writing generic functions and classes that work with any data type. They help avoid code repetition and make programs more flexible and reusable. Function and class templates improve type safety and performance by enabling compile-time polymorphism. Although templates can be harder to debug and may increase code size, they are essential for efficient and clean C++ programming.

CONCLUSION:

Templates in C++ offer a robust mechanism for writing generic, reusable, and type-safe code. They simplify the development process by allowing functions and classes to operate with various data types without rewriting code. While they can be complex and sometimes difficult to debug, the advantages in flexibility and performance make templates an essential part of modern C++ programming. Mastery of templates is key for writing efficient and maintainable code in large-scale software development.

