

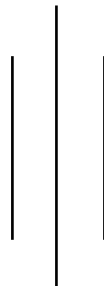


# TRIBHUVAN UNIVERSITY

## INSTITUTE OF ENGINEERING



### HIMALAYA COLLEGE OF ENGINEERING CHYASAL, LALITPUR



**Lab Report No: - Templates**  
**Title: -07**

**Submitted by: -**  
**Name: -Aditya Man Shrestha**  
**Roll NO: -HCE081BEI005**

**Submitted To: -**  
**Department Of**  
**Checked by: -**

**Date of submission: -**

## Objectives:

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

## Tools and Libraries Used:

- Programming Language: C++
- IDE: Code::Blocks
- Libraries: include <iostream>, include <string>

## 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
} };

```

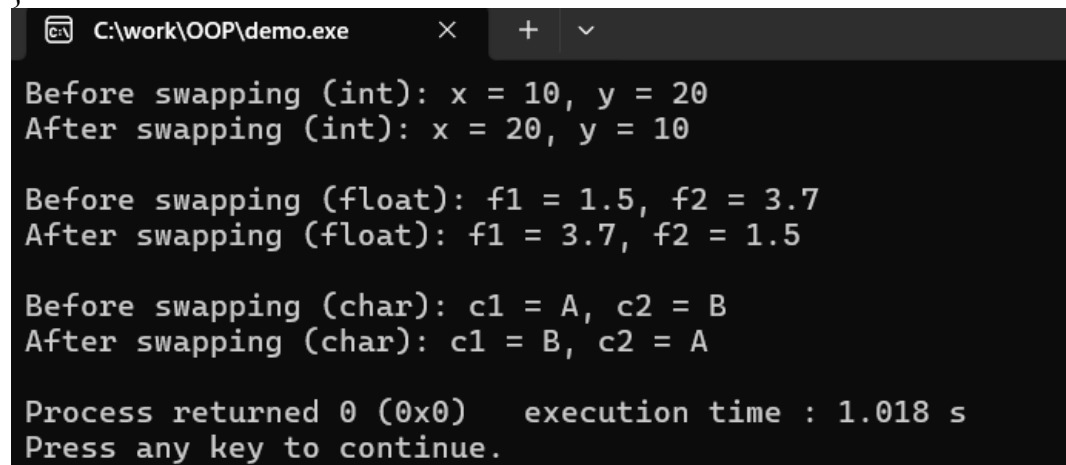
### **Rules of Using Templates:**

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.

## Lab Assignment

Qn1.

```
#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;
    cout << "Before swapping (int): x = " << x << ", y = " << y << endl;
    swapValues(x, y);
    cout << "After swapping (int): x = " << x << ", y = " << y << endl;
    float f1 = 1.5, f2 = 3.7;
    cout << "\nBefore swapping (float): f1 = " << f1 << ", f2 = " << f2 << endl;
    swapValues(f1, f2);
    cout << "After swapping (float): f1 = " << f1 << ", f2 = " << f2 << endl;
    char c1 = 'A', c2 = 'B';
    cout << "\nBefore swapping (char): c1 = " << c1 << ", c2 = " << c2 << endl;
    swapValues(c1, c2);
    cout << "After swapping (char): c1 = " << c1 << ", c2 = " << c2 << endl;
    return 0;
}
```



```
C:\work\OOP\demo.exe
Before swapping (int): x = 10, y = 20
After swapping (int): x = 20, y = 10

Before swapping (float): f1 = 1.5, f2 = 3.7
After swapping (float): f1 = 3.7, f2 = 1.5

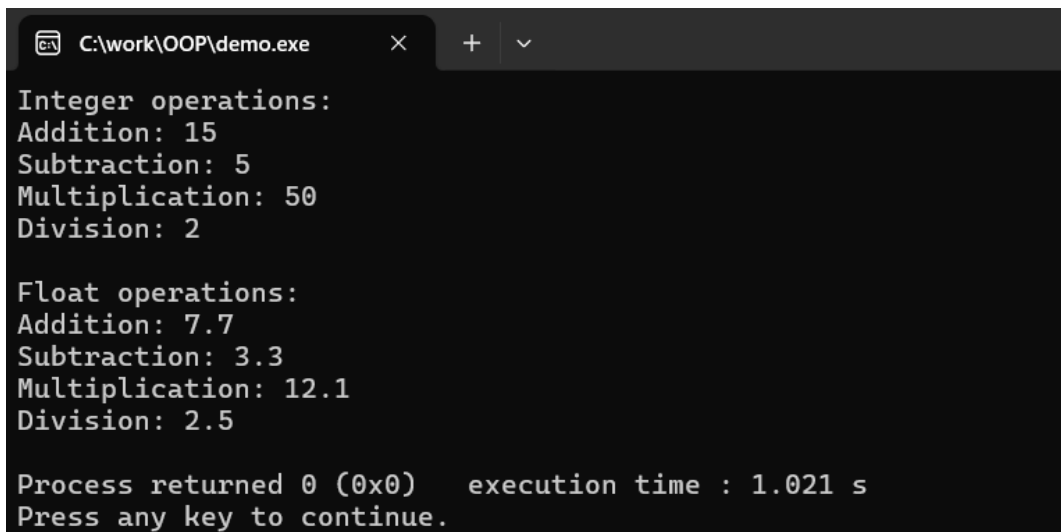
Before swapping (char): c1 = A, c2 = B
After swapping (char): c1 = B, c2 = A

Process returned 0 (0x0)    execution time : 1.018 s
Press any key to continue.
```

Qn2.

```
#include <iostream>
using namespace std;
template <typename T>
class Calculator {
private:
    T num1, num2;
public:
    Calculator(T a, T b) {
        num1 = a;
        num2 = b; }
    T add() {
        return num1 + num2; }
    T subtract() {
        return num1 - num2; }
    T multiply() {
        return num1 * num2; }
    T divide() {
        if (num2 != 0)
            return num1 / num2;
        else {
            cout << "Error: Division by zero!" << endl;
            return 0;
        }
    }
};

int main() {
    Calculator<int> intCalc(10, 5);
    cout << "Integer operations:" << endl;
    cout << "Addition: " << intCalc.add() << endl;
    cout << "Subtraction: " << intCalc.subtract() << endl;
    cout << "Multiplication: " << intCalc.multiply() << endl;
    cout << "Division: " << intCalc.divide() << endl;
    cout << endl;
    Calculator<float> floatCalc(5.5f, 2.2f);
    cout << "Float operations:" << endl;
    cout << "Addition: " << floatCalc.add() << endl;
    cout << "Subtraction: " << floatCalc.subtract() << endl;
    cout << "Multiplication: " << floatCalc.multiply() << endl;
    cout << "Division: " << floatCalc.divide() << endl;
    return 0;
}
```



```
C:\work\OOP\demo.exe
Integer operations:
Addition: 15
Subtraction: 5
Multiplication: 50
Division: 2

Float operations:
Addition: 7.7
Subtraction: 3.3
Multiplication: 12.1
Division: 2.5

Process returned 0 (0x0)   execution time : 1.021 s
Press any key to continue.
```

## DISCUSSION

In this lab, we were able to understand the core concept of templates, which are a powerful feature in C++ that enable generic programming. We learnt how function templates and class templates allow us to write flexible and reusable code for different data types without duplication. Additionally, we explored the concept of template specialization and overloading templates to handle specific cases. Although understanding template syntax and its implementation was challenging at first, with the help of examples and online resources, I was able to understand and implement it.

## CONCLUSION

From this lab, we can conclude that the use of templates enhances code maintainability, reduces the need for rewriting code, and facilitates polymorphism which is a core concept in OOP.