



TRIBHUVAN UNIVERSITY

INSTITUTE OF ENGINEERING



HIMALAYA COLLEGE OF ENGINEERING

CHYASAL, LALITPUR



Lab Report No: -07

Title: - Template

Submitted by: -

Name: Aastha Gaire

Submitted To: -

**Department of Electronics and
Computer Engineering**

Roll NO: - HCE081BEI003

Checked by: -

Date of submission: - 2082/03/21

OBJECTIVE

- To comprehend C++ templates and their function in generic programming.
- To become proficient in writing type-independent functions by creating and utilizing function templates.
- to put into practice and illustrate how to create generic data structures or classes using class templates.
- To investigate how templates can improve reusability and decrease code redundancy.

THEORY

Generic programs are written in C++ using templates. This eliminates the need to write distinct code for int, float, char, and other data types and allows us to write a single piece of code that works with all of them. We may boost reusability and decrease code duplication by using templates.

In C++, there are two primary categories of templates:

1) Function template

A function template allows a function to work with different data types without rewriting the code for each type.

Example:

```
template <typename T>

T swapValues(T a, T b) {

    T temp = a;
    a = b;
    b = temp;
    return a;
}
```

Advantages:

- Saves time by avoiding repeated code.
- Ensures consistency across all versions of the function.

2. Class Templates:

A class template works the same way but is used to create classes that can store or operate on any data type.

Example:

```
template <class T>
class Calculator {
    T num1, num2;
public:
    Calculator(T a, T b) : num1(a), num2(b) {}
    T add() { return num1 + num2; }
    T multiply() { return num1 * num2; }
};
```

Why Use Templates?

- **Generic Code:** Write once, use for many types.
- **Type Safety:** The compiler checks types at compile time.
- **Maintainability:** Easier to update or fix one version of code.
- **Used in STL:** Templates are heavily used in the Standard Template Library (like vector, list, map).

Real-World Use:

The C++ STL uses templates for many useful containers:

- `vector<int>`: A list of integers
- `vector<float>`: A list of floats
- `stack<string>`: A stack of strings

LAB PROGRAMS

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

```
#include <iostream>

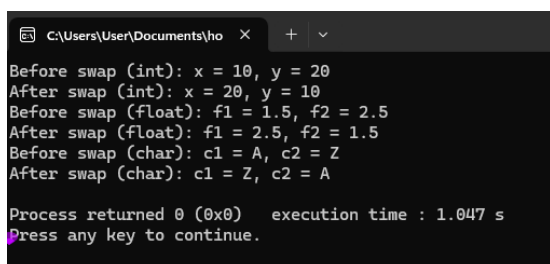
using namespace std;

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

int main() {
    int x = 10, y = 20;
    cout << "Before swap (int): x = " << x << ", y = " << y << endl;
    SwapValues(x, y);
    cout << "After swap (int): x = " << x << ", y = " << y << endl;
    float f1 = 1.5f, f2 = 2.5f;
    cout << "Before swap (float): f1 = " << f1 << ", f2 = " << f2 << endl;
    SwapValues(f1, f2);
    cout << "After swap (float): f1 = " << f1 << ", f2 = " << f2 << endl;

    char c1 = 'A', c2 = 'Z';
    cout << "Before swap (char): c1 = " << c1 << ", c2 = " << c2 << endl;
    SwapValues(c1, c2);
    cout << "After swap (char): c1 = " << c1 << ", c2 = " << c2 << endl;
    return 0;
}
```

Output:



```
C:\Users\User\Documents\ho  X + v
Before swap (int): x = 10, y = 20
After swap (int): x = 20, y = 10
Before swap (float): f1 = 1.5, f2 = 2.5
After swap (float): f1 = 2.5, f2 = 1.5
Before swap (char): c1 = A, c2 = Z
After swap (char): c1 = Z, c2 = A

Process returned 0 (0x0)   execution time : 1.047 s
Press any key to continue.
```

B) 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 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 T{}; // Return default value of T
        }
    }
};

int main() {
    // Instantiate Calculator with int
    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;
```

```

    cout << endl;

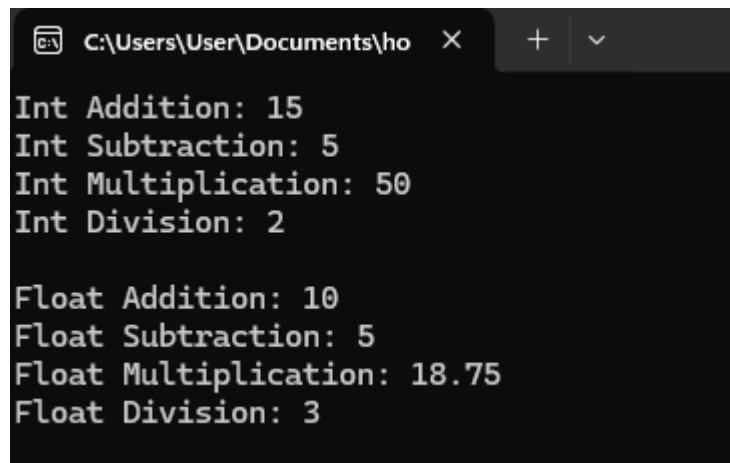
    Calculator<float> calcFloat(7.5f, 2.5f);

    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;
}

```

OUTPUT:



```

C:\Users\User\Documents\ho X + v
Int Addition: 15
Int Subtraction: 5
Int Multiplication: 50
Int Division: 2

Float Addition: 10
Float Subtraction: 5
Float Multiplication: 18.75
Float Division: 3

```

Discussion:

In this lab, we discovered how C++ templates facilitate the creation of classes and functions that can handle any kind of data. We improved reusability and prevented code redundancy by utilizing function and class templates. The programs demonstrated the versatility of templates by functioning well for a variety of types, including int, float, and char. We also realized that STL containers like vector and stack are built on templates. Overall, the lab demonstrated how templates improve the efficiency and type safety of C++ programming.

Conclusion:

We were able to comprehend the significance of generic programming in C++ thanks to the templates lab. We were able to create reusable, adaptable code that works with various data types by utilizing function and class templates. This enhanced maintainability and decreased code duplication. Additionally, we observed how templates are utilized in the Standard Template Library (STL), which makes them an essential component of practical C++ programming. All things considered, the lab improved our comprehension of how templates support reliable and effective software development.

