TRIBHUVAN UNIVERSITY

INSTITUTE OF ENGINEERING

HIMALAYA COLLEGE OF ENGINEERING

CHYASAL, LALITPUR

Lab Report No: -07

Title: - Template

Submitted by: -

Name: - Diwas Pokhrel

Roll NO: - HCE081BEI014

Submitted To: -

Department of Electronics and Computer Engineering

Checked by: -

Date of submission: - 2082/03/21

OBJECTIVE

- To understand the concept of templates in C++ and their role in generic programming.
- To learn how to create and use function templates to write type-independent functions.
- To implement and demonstrate the use of class templates for building generic data structures or classes.
- To explore the advantages of templates in reducing code redundancy and increasing reusability.

THEORY

In C++, templates are used to write generic programs. This means we can write a single piece of code that works with any data type instead of writing separate versions for int, float, char, etc. Templates help us reduce code duplication and increase reusability**.**

There are two main types of templates in C++:

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.

```cpp
#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;
}
```

Output

```
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


=== Code Execution Successful ===
```

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.

```cpp
#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;
}
```

Output

```
Integer operations:
Addition: 15
Subtraction: 5
Multiplication: 50
Division: 2

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


=== Code Execution Successful ===
```

Discussion:

In this lab, we learned how templates in C++ help create functions and classes that work with any data type. By using function and class templates, we avoided code repetition and increased reusability. The programs worked correctly for multiple types like int, float, and char, proving the flexibility of templates. We also understood that templates are the foundation of STL containers like vector and stack. Overall, the lab showed how templates make C++ programming more efficient and type-safe.

Conclusion:

The templates lab helped us understand the importance of generic programming in C++. By using function and class templates, we were able to write flexible and reusable code that works with different data types. This reduced code duplication and improved maintainability. We also saw how templates are used in the Standard Template Library (STL), making them a key feature in real-world C++ programming. Overall, the lab enhanced our understanding of how templates contribute to efficient and robust software development**.**