TRIBHUVAN UNIVERSITY

INSTITUTE OF ENGINEERING

HIMALAYA COLLEGE OF ENGINEERING

CHYASAL, LALITPUR

Lab Report No: - 07

Title: - Templates

Submitted by: -

Name: -Gaurab Pandey

Roll NO: - 15

Submitted To: -

Department Of

Checked by: -

Date of submission: -

Objectives:

1.To understand and implement basic C++ templates for creating generic functions and classes that can operate with different data types.

Theory:

Templates in C++ allow for generic programming by enabling functions and classes to work with different data types without rewriting code. They are defined using the template keyword followed by template parameters (e.g., typename T).

1.Function Templates

A single function template can handle multiple data types.

Syntax:

```
template <typename T>  // or 'template <class T>'
return_type function_name(T parameter1, T parameter2, ...) {
// Function body (uses type T)
}
```

2.Class Templates

Class templates allow you to define a generic class that can work with different data types. They useful for creating data structures.

Syntax:

```
template <class T>  // or 'template <typename T>'
class ClassName {
private:
    T member_variable;
public:
    ClassName(T arg) : member_variable(arg) {}
    T getValue() { return member_variable; }
    // Other member functions...
};
```

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

Code:

```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 = 2.5, f2 = 3.3;
    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 = 2.5, f2 = 3.3
After swapping (float): f1 = 3.3, f2 = 2.5

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

2. 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.

Code:

```
#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;
        }
    }
};
```

```cpp
int main() {
    Calculator<int> intCalc(120, 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(7.5f, 6.2f);
    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: 125
Subtraction: 115
Multiplication: 600
Division: 24

Float operations:
Addition: 13.7
Subtraction: 1.3
Multiplication: 46.5
Division: 1.20968
```

Discussion

In this lab, we explored the implementation and application of C++ templates, which enable generic programming by allowing functions and classes to operate on different data types without code duplication. The lab demonstrated how templates enhance code reusability and type safety, as the compiler generates specialized versions for each data type at compile time.

Conclusion

The lab effectively demonstrated the significance of templates in C++ for achieving generic, reusable, and type-safe programming. By implementing both function and class templates, we recognized their role in reducing redundancy and improving maintainability—key advantages in large-scale software development.