Objectives:

- To gain an understanding of Templates in C++.

Tools and Libraries Used:

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

Theory:

Templates in C++ are a powerful feature that allows writing generic and reusable code. They enable functions and classes to operate with any data type without rewriting code for each type.

1. Function Templates

Function templates allow defining a generic function that works with different data types.

Syntax:

```
template <typename T>
T add(T a, T b) {
   return a + b;
}
```

Here, T is a placeholder type that gets replaced during compilation based on the actual arguments passed.

2. Class Templates

Class templates enable the creation of classes that can work with any data type.

Syntax:

```
template <class T>
class Box {
   T value;
```

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

Key Features:

- Code Reusability: Write once, use for any type.
- Type Safety: Type checking is done at compile time.
- Flexibility: Supports custom data types and user-defined classes.

Types of Templates:

- Single-type parameter: One generic type.
- Multiple-type parameters: Can use multiple typename or class keywords.
- Default template arguments: Allow providing default types.

Lab Questions:

Q no 1:

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

Code:

```
1.  #include <iostream>
2.  using namespace std;
3.  template <typename T>
4.  void swapValues(T &a, T &b) {
5.      T temp = a;
6.      a = b;
7.      b = temp;
8.  }
9.  int main() {
10.     int x = 10, y = 20;
11.     cout << "Before swapping (int): x = " << x << ", y = " << y <<
    endl;
12.     swapValues(x, y);
13.     cout << "After swapping (int): x = " << x << ", y = " << y <<
    endl;
14.     float f1 = 1.5, f2 = 3.7;
15.     cout << "\nBefore swapping (float): f1 = " << f1 << ", f2 = " <<
    f2 << endl;
16.     swapValues(f1, f2);
17.     cout << "After swapping (float): f1 = " << f1 << ", f2 = " << f2
    << endl;
18.     char c1 = 'A', c2 = 'B';
19.     cout << "\nBefore swapping (char): c1 = " << c1 << ", c2 = " << c2
    << endl;
20.     swapValues(c1, c2);
21.     cout << "After swapping (char): c1 = " << c1 << ", c2 = " << c2 <<
    endl;
22.     return 0;
23. }
```

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
```

Q no 2:

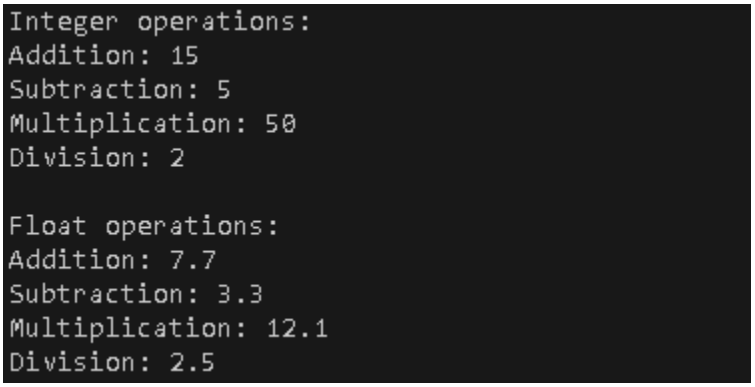
Create a class template Calculator<T> that performs addition, subtraction, multiplication, and division of two data members of type T. Instantiate it with int and float.

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;
        }
    }
};
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:



Conclusion:

Understanding templates in C++ is essential for writing generic, reusable, and type-safe code. Through this lab, we learned how to:

- Define function and class templates to work with multiple data types.
- Use template syntax to reduce code duplication.
- Apply templates in real scenarios for flexibility and efficiency.

These concepts are a key part of generic programming and are widely used in the Standard Template Library (STL). Mastery of templates helps in building scalable and maintainable software by promoting code reusability and abstraction.