TRIBHUVAN UNIVERSITY

INSTITUTE OF ENGINEERING

HIMALAYA COLLEGE OF ENGINEERING

CHYASAL, LALITPUR

Lab Report No: -06

Title: - Virtual Functions & Templates

Submitted by: -	**Submitted To: -**
Name: - Bikram Panthi	**Department Of Electronics Engineering**
Roll NO: - 13	**Checked by: -**
	Date of submission: -

Objectives:

- To understand and apply the concept of runtime polymorphism using virtual functions and abstract classes in C++.
- To implement pure virtual functions and virtual destructors for safe object cleanup in inheritance.
- To learn how to create and use function and class templates for generic programming with different data types.
- To develop reusable and flexible code using templates and polymorphic behavior for various operations.

Theory

Virtual Functions

In C++, virtual functions are used to achieve runtime polymorphism, which means that the decision about which function to call is made at runtime based on the type of object, not the type of pointer. Normally, if we have a base class pointer pointing to a derived class object, and we call a function, the base class version of the function is called. But if the function is declared as virtual in the base class, then the derived class version will be called instead. This allows us to write flexible and reusable code that works correctly with inheritance.

A virtual function is declared by placing the virtual keyword before the function definition inside the base class. The function can then be overridden in the derived class using the same function name and signature. When we use a base class pointer or reference to point to a derived class object, the derived version of the virtual function is called, not the base version. This behavior is known as late binding or dynamic binding, and it is one of the key features of object-oriented programming.

Here's a simple syntax example of a virtual function:

```
class Base {

public:

   virtual void show() {

      cout << "Base class" << endl;

   }

};
```

In this example, show() is a virtual function. If a derived class overrides this function and we call it using a base class pointer pointing to a derived class object, the overridden version in the derived class will be executed.

Virtual functions are especially useful when working with arrays or collections of base class pointers that actually point to objects of different derived classes. They help us write cleaner and more general code. One more important point is that if we are using virtual functions, the destructor of the base class should also be made virtual. This ensures that when we delete a derived class object through a base class pointer, the derived class's destructor is also called properly, which helps in cleaning up memory correctly and avoiding memory leaks.

So, in short, virtual functions allow us to override base class functions in derived classes, and they make sure that the correct function is called at runtime, depending on the object type. They make inheritance and polymorphism more powerful and useful in real-world programming.

Templates

In C++, templates allow us to write generic and reusable code. Instead of writing the same function or class multiple times for different data types (like int, float, double, etc.), we can write one version using a template, and the compiler will automatically generate the correct version when the program runs.

There are two main types of templates in C++ — function templates and class templates. A function template is used when we want the same logic to work with different data types, like swapping two values or finding the maximum of two numbers. For example, instead of writing separate functions to swap integers, floats, and characters, we can write one function template that works for all. We use the keyword template followed by <typename T> or <class T> before the function definition. T is a placeholder for the actual data type, which gets replaced during compilation depending on the input.

A class template, on the other hand, is used to create classes that work with any data type. This is useful when we want to design classes like stacks, queues, calculators, or data containers that should support various types without rewriting the entire class for each type. For example, we can create a calculator class template that works with both integers and floating-point numbers by using a single definition.

Here is a basic syntax of a function template:

```
template <typename T>

T add(T a, T b) {

   return a + b;

}
```

In this example, add() can be used to add two integers, two floats, or even two doubles — the type T is decided at the time the function is called.

Templates are very helpful in generic programming, and they make our code shorter, more flexible, and easier to maintain. They also help avoid redundancy by removing the need to write multiple versions of the same logic. One thing to note is that templates are mostly resolved at compile-time, which makes them efficient but also means errors in template code can be harder to read.

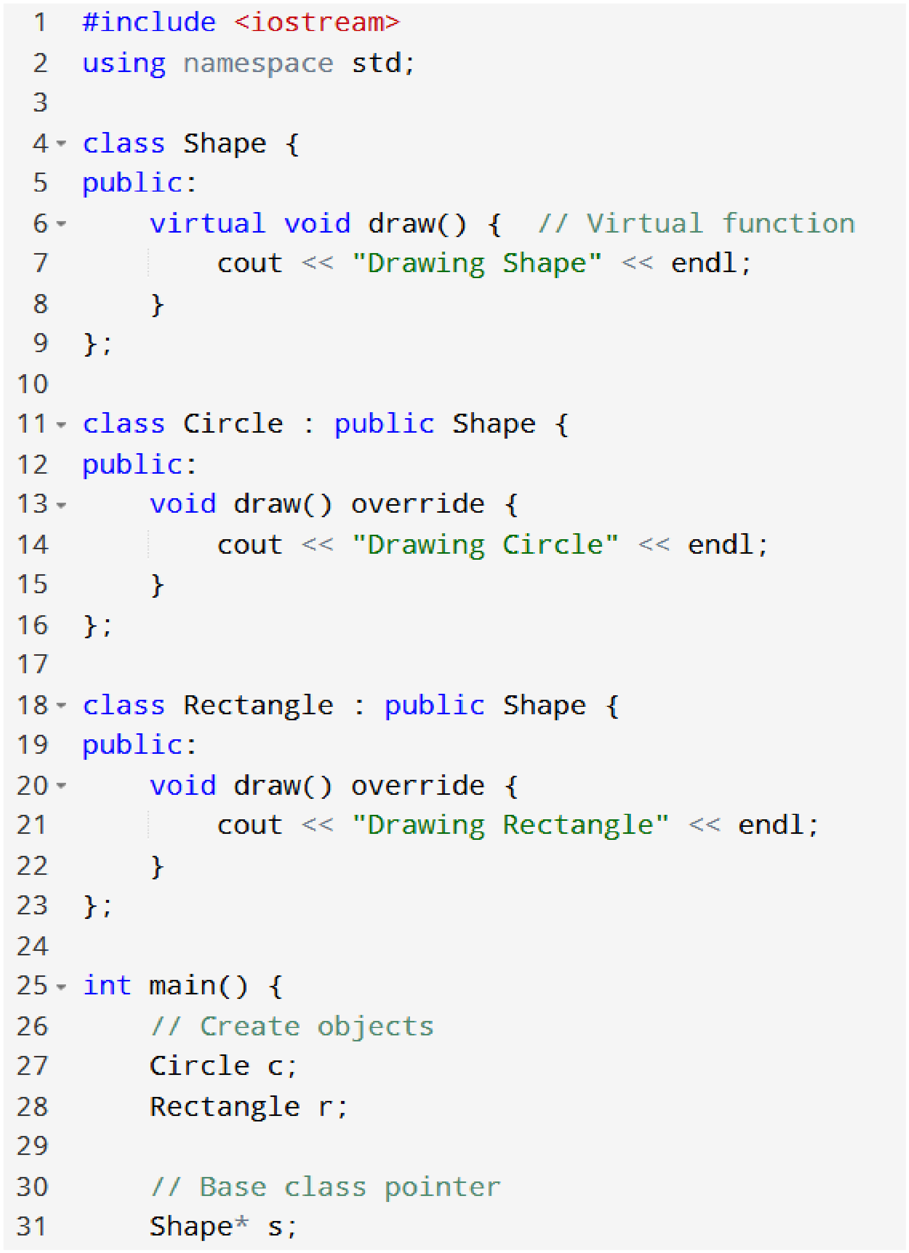
In conclusion, templates in C++ are a powerful feature that allow us to write code that can work with any data type. They are commonly used in the Standard Template Library (STL) as well — where things like vector, stack, map, etc., are all based on templates. Using templates saves time and makes your programs more scalable and reusable.

Questions

1. Create a base class Shape with a virtual function draw() that prints "Drawing Shape". Derive two classes, Circle and Rectangle, each overriding draw() to print "Drawing Circle" and "Drawing Rectangle", respectively. In the main function:

- Create Circle and Rectangle objects.
- Use a Shape* pointer to call draw() on both objects to show polymorphic behavior.
- Create an array of Shape (not using virtual keyword for draw()) and repeat the experiment. Compare outputs to explain why virtual functions are needed.
- Use a Circle* pointer to call draw() on a Circle object and compare with the base class pointer's behavior.

Code:



```cpp
#include <iostream>
using namespace std;

class Shape {
public:
    virtual void draw() {  // Virtual function
        cout << "Drawing Shape" << endl;
    }
};

class Circle : public Shape {
public:
    void draw() override {
        cout << "Drawing Circle" << endl;
    }
};

class Rectangle : public Shape {
public:
    void draw() override {
        cout << "Drawing Rectangle" << endl;
    }
};

int main() {
    // Create objects
    Circle c;
    Rectangle r;

    // Base class pointer
    Shape* s;

    s = &c;
    s->draw();  // Output: Drawing Circle

    s = &r;
    s->draw();  // Output: Drawing Rectangle

    // Circle pointer
    Circle* cp = &c;
    cp->draw();  // Output: Drawing Circle

    return 0;
}
```

Output:

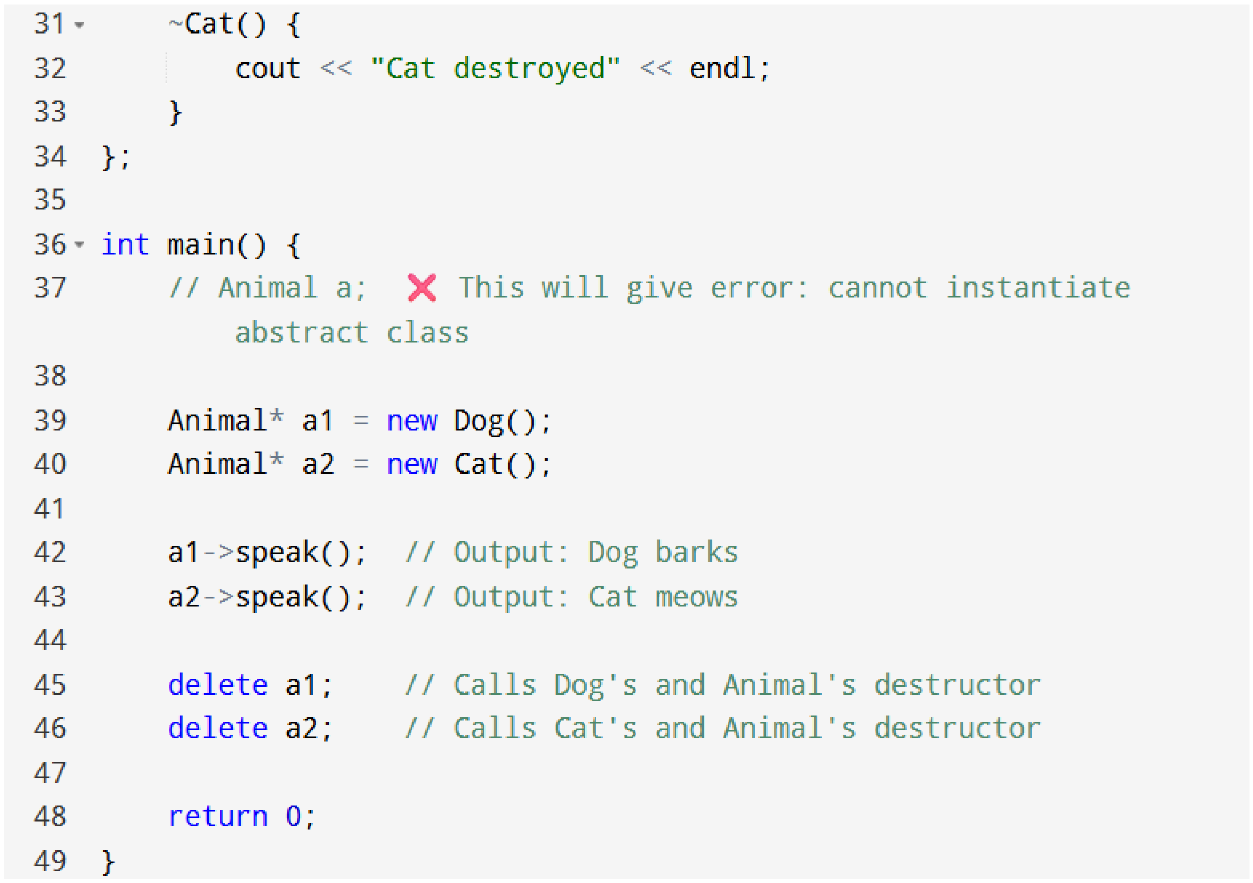
```
Drawing Circle
Drawing Rectangle
Drawing Circle


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

2. Create an abstract base class Animal with a pure virtual function speak() and a virtual destructor. Derive two classes, Dog and Cat, each implementing speak() to print "Dog barks" and "Cat meows", respectively. Include destructors in both derived classes that print "Dog destroyed" and "Cat destroyed". In the main function:

- Attempt to instantiate an Animal object (this should fail).
- Create Dog and Cat objects using Animal* pointers and call speak().
- Delete the objects through the Animal* pointers and verify that derived class destructors are called.
- Modify the Animal destructor to be non-virtual, repeat the deletion, and observe the difference.

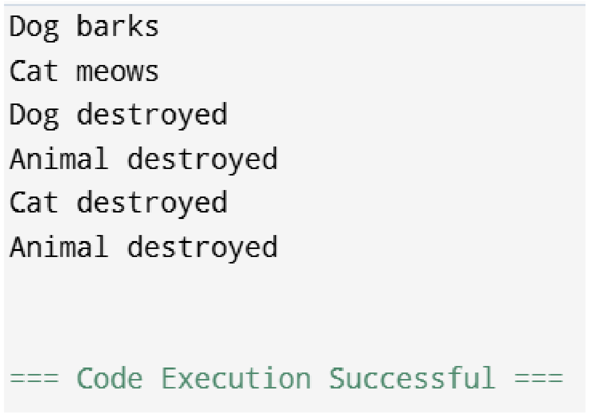
Code:

```cpp
#include <iostream>
using namespace std;

class Animal {
public:
    virtual void speak() = 0;  // Pure virtual function

    // Virtual destructor
    virtual ~Animal() {
        cout << "Animal destroyed" << endl;
    }
};

class Dog : public Animal {
public:
    void speak() override {
        cout << "Dog barks" << endl;
    }

    ~Dog() {
        cout << "Dog destroyed" << endl;
    }
};

class Cat : public Animal {
public:
    void speak() override {
        cout << "Cat meows" << endl;
    }

```



```cpp
    ~Cat() {
        cout << "Cat destroyed" << endl;
    }
};

int main() {
    // Animal a;  ❌ This will give error: cannot instantiate abstract class

    Animal* a1 = new Dog();
    Animal* a2 = new Cat();

    a1->speak();  // Output: Dog barks
    a2->speak();  // Output: Cat meows

    delete a1;    // Calls Dog's and Animal's destructor
    delete a2;    // Calls Cat's and Animal's destructor

    return 0;
}
```

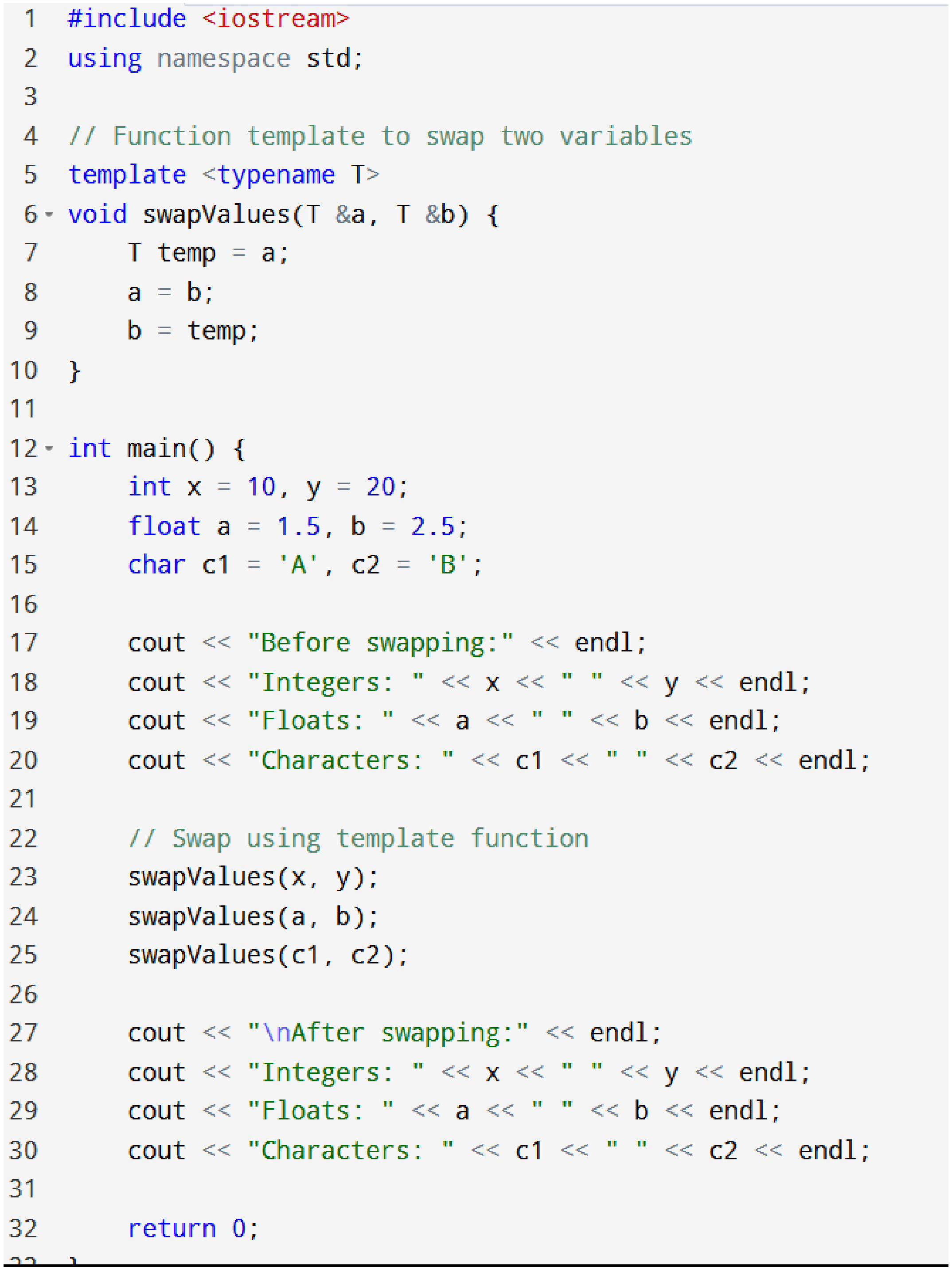
Output:



```
Dog barks
Cat meows
Dog destroyed
Animal destroyed
Cat destroyed
Animal destroyed


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

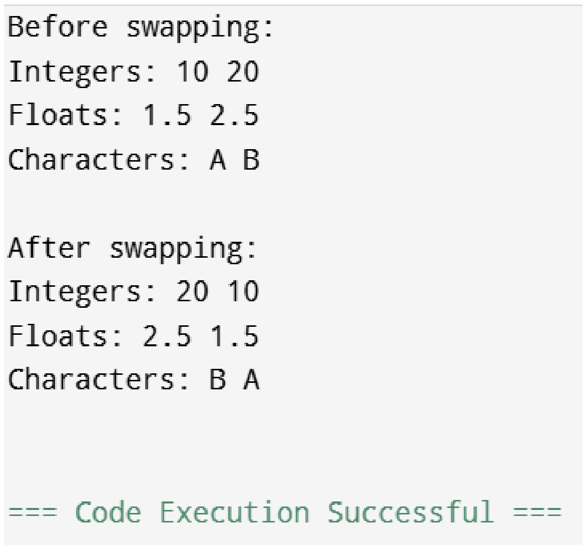
3. Write a function template swapValues() that swaps two variables of any data type. Demonstrate its use with int, float, and char.

Code:



```cpp
#include <iostream>
using namespace std;

// Function template to swap two variables
template <typename T>
void swapValues(T &a, T &b) {
    T temp = a;
    a = b;
    b = temp;
}

int main() {
    int x = 10, y = 20;
    float a = 1.5, b = 2.5;
    char c1 = 'A', c2 = 'B';

    cout << "Before swapping:" << endl;
    cout << "Integers: " << x << " " << y << endl;
    cout << "Floats: " << a << " " << b << endl;
    cout << "Characters: " << c1 << " " << c2 << endl;

    // Swap using template function
    swapValues(x, y);
    swapValues(a, b);
    swapValues(c1, c2);

    cout << "\nAfter swapping:" << endl;
    cout << "Integers: " << x << " " << y << endl;
    cout << "Floats: " << a << " " << b << endl;
    cout << "Characters: " << c1 << " " << c2 << endl;

    return 0;
}
```

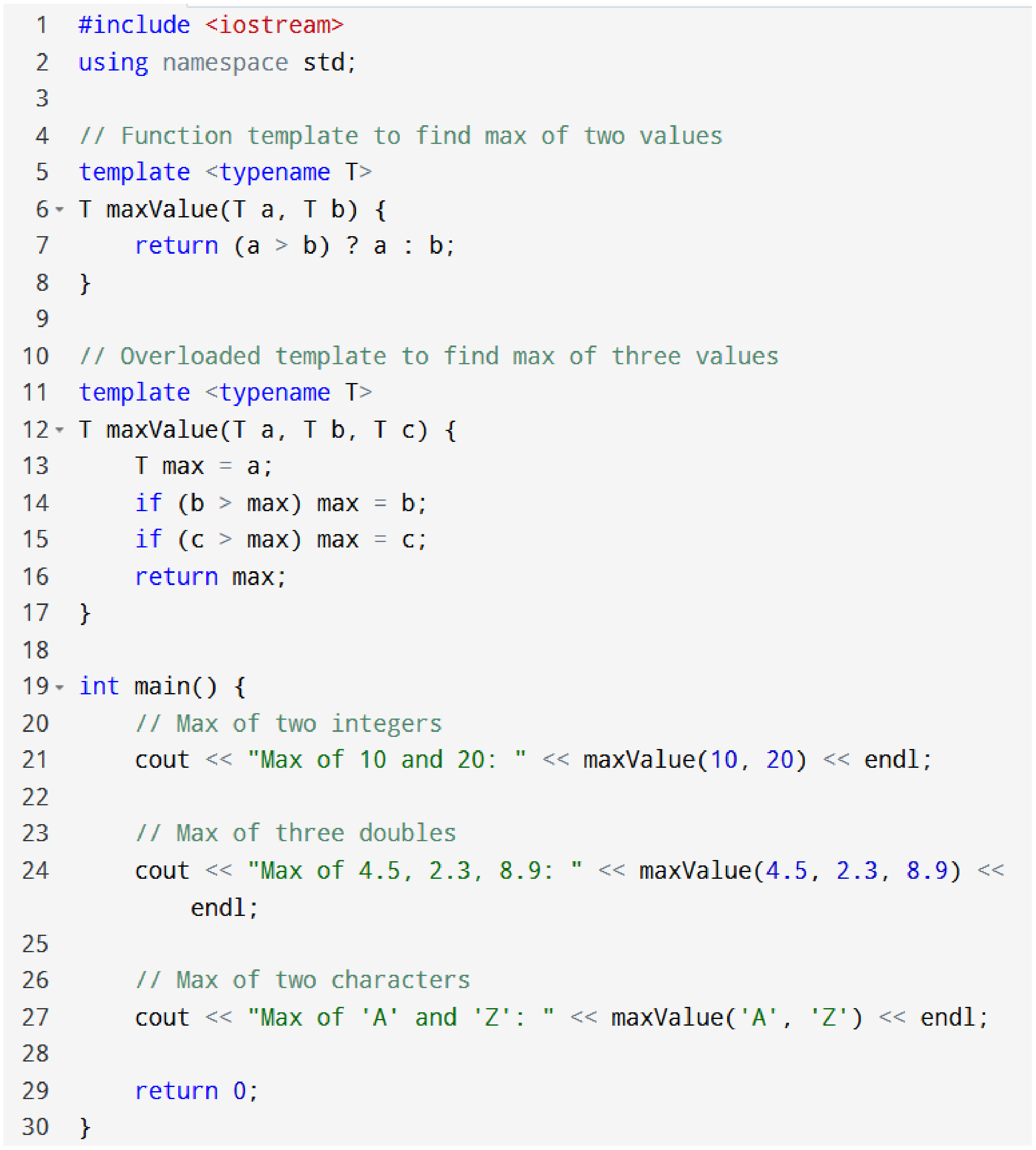
Output:



```
Before swapping:
Integers: 10 20
Floats: 1.5 2.5
Characters: A B

After swapping:
Integers: 20 10
Floats: 2.5 1.5
Characters: B A



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

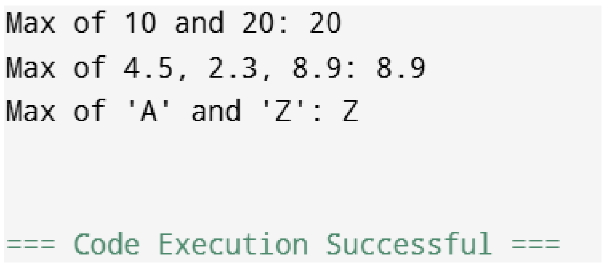
4. Write a program to overload a function template maxValue() to find the maximum of two values (for same type) and three values (for same type). Call it using int, double, and char.

Code:



```cpp
#include <iostream>
using namespace std;

// Function template to find max of two values
template <typename T>
T maxValue(T a, T b) {
    return (a > b) ? a : b;
}

// Overloaded template to find max of three values
template <typename T>
T maxValue(T a, T b, T c) {
    T max = a;
    if (b > max) max = b;
    if (c > max) max = c;
    return max;
}

int main() {
    // Max of two integers
    cout << "Max of 10 and 20: " << maxValue(10, 20) << endl;

    // Max of three doubles
    cout << "Max of 4.5, 2.3, 8.9: " << maxValue(4.5, 2.3, 8.9) << endl;

    // Max of two characters
    cout << "Max of 'A' and 'Z': " << maxValue('A', 'Z') << endl;

    return 0;
}
```

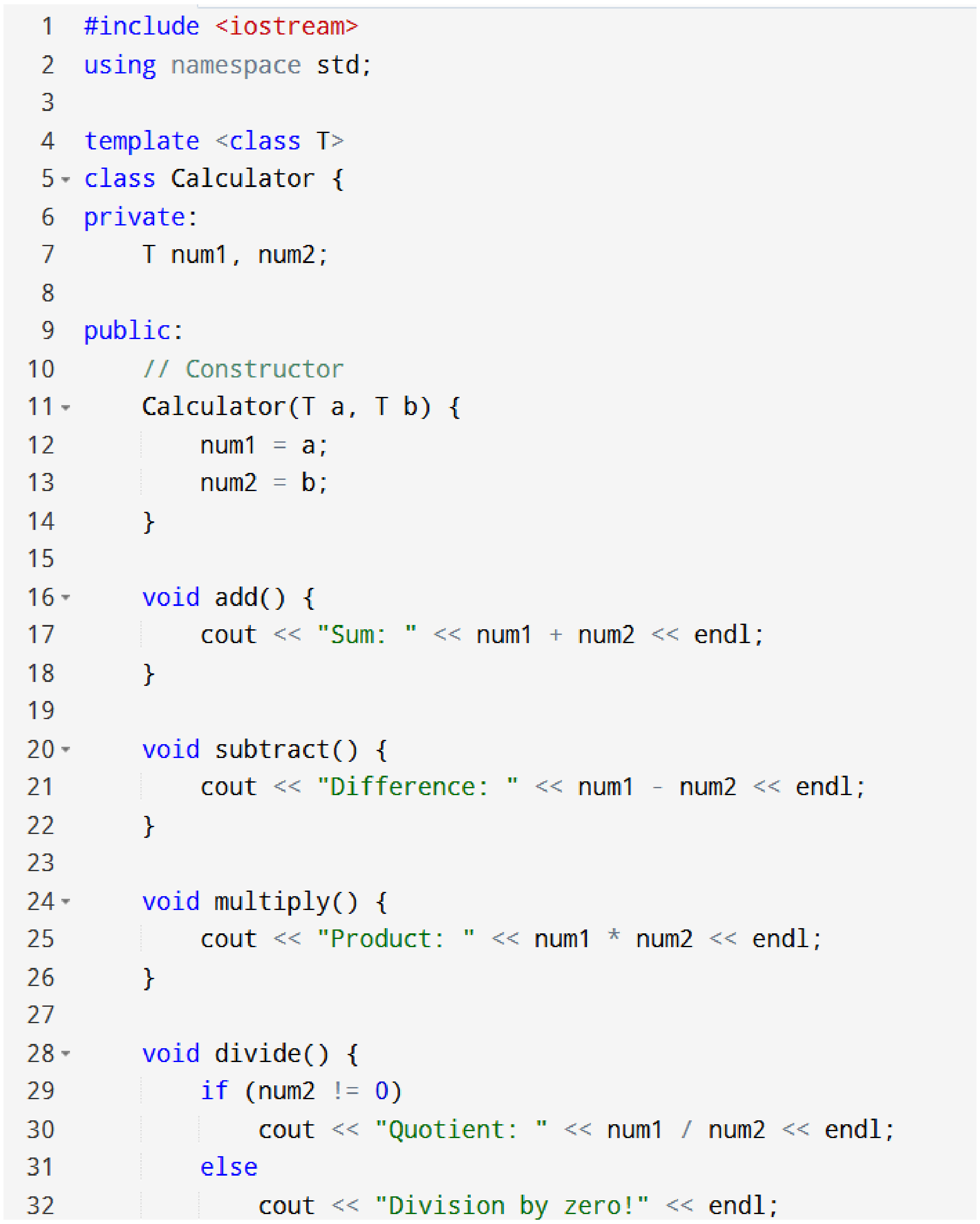
Output:



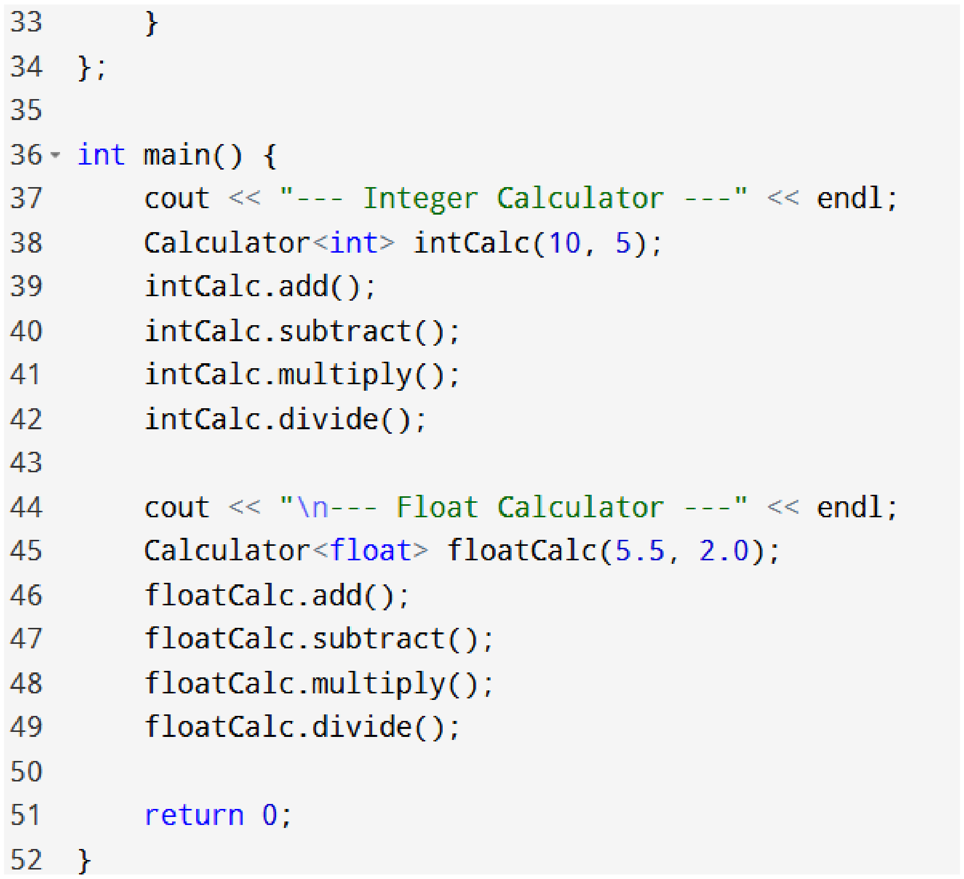
```
Max of 10 and 20: 20
Max of 4.5, 2.3, 8.9: 8.9
Max of 'A' and 'Z': Z


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

5. 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:

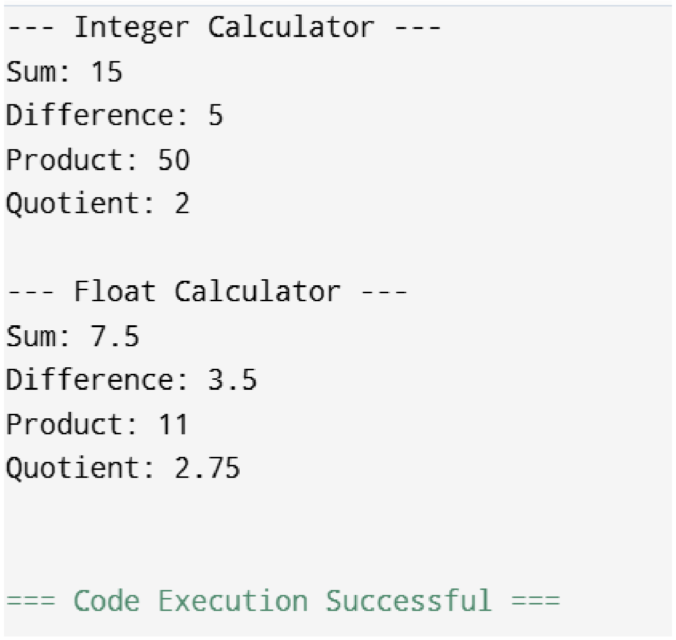


```cpp
#include <iostream>
using namespace std;

template <class T>
class Calculator {
private:
    T num1, num2;

public:
    // Constructor
    Calculator(T a, T b) {
        num1 = a;
        num2 = b;
    }

    void add() {
        cout << "Sum: " << num1 + num2 << endl;
    }

    void subtract() {
        cout << "Difference: " << num1 - num2 << endl;
    }

    void multiply() {
        cout << "Product: " << num1 * num2 << endl;
    }

    void divide() {
        if (num2 != 0)
            cout << "Quotient: " << num1 / num2 << endl;
        else
            cout << "Division by zero!" << endl;
```



```cpp
    }
};

int main() {
    cout << "--- Integer Calculator ---" << endl;
    Calculator<int> intCalc(10, 5);
    intCalc.add();
    intCalc.subtract();
    intCalc.multiply();
    intCalc.divide();

    cout << "\n--- Float Calculator ---" << endl;
    Calculator<float> floatCalc(5.5, 2.0);
    floatCalc.add();
    floatCalc.subtract();
    floatCalc.multiply();
    floatCalc.divide();

    return 0;
}
```

Output:



```
--- Integer Calculator ---
Sum: 15
Difference: 5
Product: 50
Quotient: 2

--- Float Calculator ---
Sum: 7.5
Difference: 3.5
Product: 11
Quotient: 2.75


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

Discussion:

In this lab, we explored two important features of C++: virtual functions and templates. Virtual functions helped us understand how runtime polymorphism works using base class pointers and overridden functions in derived classes. We saw that when the virtual keyword is used, the correct version of the function is called based on the object type, not the pointer type. We also observed how pure virtual functions make a class abstract and how virtual destructors ensure proper object destruction when using base class pointers.

On the other hand, templates allowed us to write generic code for both functions and classes. With function templates, we performed the same operation (like swapping or finding max) for different data types without rewriting code. Class templates helped us build flexible classes like a calculator that work for both int and float. This lab gave us a practical understanding of how virtual functions support polymorphism and how templates promote code reusability.

Conclusion:

Through this lab, we gained hands-on experience with two powerful concepts in C++. Virtual functions make object-oriented programs more flexible and maintainable by supporting runtime polymorphism. We also understood the importance of declaring destructors as virtual to avoid unexpected behavior during deletion. Templates, both function and class types, helped us write cleaner and reusable code that works with multiple data types. Overall, this lab enhanced our understanding of how to design and implement robust and efficient C++ programs using object-oriented and generic programming techniques.