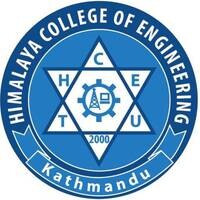


**TRIBHUVAN UNIVERSITY**

**INSTITUTE OF ENGINEERING**



**HIMALAYA COLLEGE OF ENGINEERING**

**CHYASAL, LALITPUR**

**Lab Report No: - 6**

**Title: - Virtual functions and Templetes**

**Submitted by: - Submitted To: -**

**Name: - Atul Bhattarai Department Of Computer and Electronics**

**Roll NO: - HCOE 081 BEI 011 Checked by: -**

**Date of submission: -**

### **Objective**

* To understand the concept and usage of virtual functions in C++.
* To implement function and class templates for generic programming.

**Theory**

#### **Virtual Functions:**

#### Virtual functions allow **polymorphism** in C++, enabling **dynamic dispatch**. When a base class declares a function as virtual, the derived class can **override** it, and the function call is resolved **at runtime** using a **vtable (virtual table).** This ensures that the correct function is executed, based on the **actual type** of the object, not the type of the pointer.

**Syntax:**

class Base {

public:

virtual void show() {

cout << "Base class";

}

};

class Derived : public Base {

public:

void show() override {

cout << "Derived class";

}

};

**Templates:**

Templates are used to write **generic and reusable code**. They allow functions and classes to operate with any data type without rewriting the code for each type.

* **Function Template**:

template <typename T>

T add(T a, T b) {

return a + b;

}

* **Class Template**:

template <class T>

class Calculator {

T a, b;

public:

Calculator(T x, T y) : a(x), b(y) {}

T multiply() { return a \* b; }

};

Templates promote **type independence**, **code reuse**, and **reduced redundancy**.

**Lab assignments:**

**Q1. 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 = 5, y = 10;

    float a = 1.2f, b = 3.4f;

    char c1 = 'A', c2 = 'Z';

    cout << "Before swapping:\n";

    cout << "Integers: x = " << x << ", y = " << y << endl;

    cout << "Floats: a = " << a << ", b = " << b << endl;

    cout << "Chars: c1 = " << c1 << ", c2 = " << c2 << endl;

    swapValues(x, y);

    swapValues(a, b);

    swapValues(c1, c2);

    cout << "\nAfter swapping:\n";

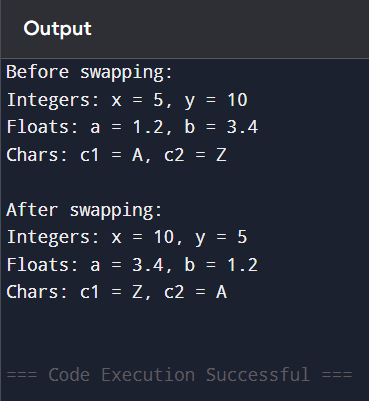
    cout << "Integers: x = " << x << ", y = " << y << endl;

    cout << "Floats: a = " << a << ", b = " << b << endl;

    cout << "Chars: c1 = " << c1 << ", c2 = " << c2 << endl;

    return 0;

}

****

**Q2. 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 <class 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()     { return (num2 != 0) ? num1 / num2 : 0; }

};

int main() {

    Calculator<int> intCalc(10, 5);

    cout << "Integer Calculation:\n";

    cout << "Add: " << intCalc.add() << endl;

    cout << "Subtract: " << intCalc.subtract() << endl;

    cout << "Multiply: " << intCalc.multiply() << endl;

    cout << "Divide: " << intCalc.divide() << endl;

    Calculator<float> floatCalc(10.5, 2.5);

    cout << "\nFloat Calculation:\n";

    cout << "Add: " << floatCalc.add() << endl;

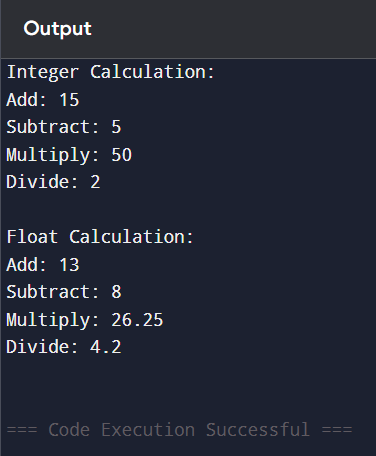
    cout << "Subtract: " << floatCalc.subtract() << endl;

    cout << "Multiply: " << floatCalc.multiply() << endl;

    cout << "Divide: " << floatCalc.divide() << endl;

    return 0;

}

****

**Q3. Create a base class Base with a virtual method display(). Create a derived class Derived that overrides the display() method. Implement a main() function where you create a Base pointer pointing to a Derived object and call the display() method.**

#include <iostream>

using namespace std;

class Base {

public:

    virtual void display() {

        cout << "Display from Base class" << endl;

    }

};

class Derived : public Base {

public:

    void display() override {

        cout << "Display from Derived class" << endl;

    }

};

int main() {

    Base\* basePtr;

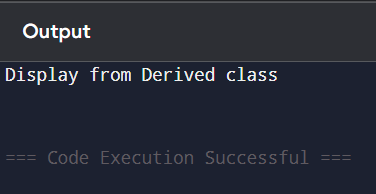
    Derived derivedObj;

    basePtr = &derivedObj;

    basePtr->display();

    return 0;

}

****

**Q4. Create a base class Person with a virtual method show(). Create another base class Employee with a virtual method show(). Create a derived class Manager that inherits from both Person and Employee, and overrides the show() method. Implement a main() function to demonstrate the usage of these classes.**

#include <iostream>

using namespace std;

class Person {

public:

    virtual void show() {

        cout << "This is a Person." << endl;

    }

};

class Employee {

public:

    virtual void show() {

        cout << "This is an Employee." << endl;

    }

};

class Manager : public Person, public Employee {

public:

    void show() override {

        Person::show();

        Employee::show();

        cout << "This is a Manager." << endl;

    }

};

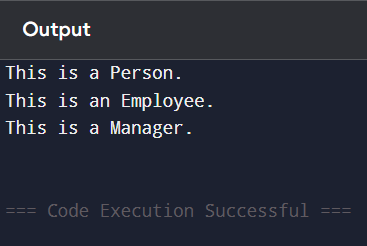
int main() {

    Manager mgr;

    mgr.show();

    return 0;

}



**Discussion**

This lab illustrated the practical usage of virtual functions for polymorphism and templates for generic programming. Virtual functions provided flexibility in method overriding, and templates minimized code duplication for similar operations on different data types.

**Conclusion**

The experiment successfully demonstrated key object-oriented and generic programming concepts in C++. Virtual functions enable dynamic method binding, critical for polymorphic behavior. Templates improve code reusability and scalability by supporting type-independent programming.