# TRIBHUVAN UNIVERSITY

## INSTITUTE OF ENGINEERING

## HIMALAYA COLLEGE OF ENGINEERING

### CHYASAL, LALITPUR

**Lab Report No: -06**

**Title: - Virtual Functions**


**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 virtual functions in C++ and their role in achieving runtime polymorphism.
- To implement virtual functions in a class hierarchy to observe dynamic binding.
- To demonstrate the use of the virtual keyword in base and derived classes.
- To explore how virtual functions enable function overriding in inheritance.
- To analyze the behavior of virtual functions with different types of object pointers and references.
- To understand the significance of virtual destructors in preventing memory leaks.

## THEORY

Virtual functions in C++ are a fundamental feature of object-oriented programming that facilitate runtime polymorphism, enabling a derived class to override a function defined in its base class. By declaring a function as virtual in the base class, the C++ runtime determines which function to call based on the actual type of the object, rather than the type of the pointer or reference used to access it. Virtual functions are critical in inheritance hierarchies, allowing derived classes to provide specialized implementations of a base class's interface. Additionally, virtual destructors are essential when deleting objects through a base class pointer to ensure proper cleanup of derived class resources, preventing memory leaks. Pure virtual functions, declared using = 0, make a class abstract, meaning it cannot be instantiated and must be inherited by a derived class that provides implementations for all pure virtual functions. Virtual functions introduce a slight performance overhead due to vtable lookups but provide significant flexibility and extensibility, enabling polymorphic behavior in complex programs.

## Code Example

C++ program demonstrating virtual functions, virtual destructors, and a pure virtual function in a class hierarchy is given below :

```
#include <iostream>
using namespace std;
class Base {
public:
   // Virtual function
   virtual void display() {
      cout << "Display from Base class" << endl;
   }
```

```
                                                    // Pure virtual function
    virtual void pureVirtual() = 0;
                                                    // Virtual destructor
    virtual ~Base() {
        cout << "Base class destructor" << endl;
    }
};
class Derived : public Base {
public:
                                                    // Overriding the virtual function
        void display() override {
        cout << "Display from Derived class" << endl;
    }
                                                    // Implementing the pure virtual function
        void pureVirtual() override {
        cout << "Pure virtual function implemented in Derived class" << endl;
    }
    // Destructor
    ~Derived() {
        cout << "Derived class destructor" << endl;
    }
};
int main() {
    // Pointer of Base type pointing to Derived object
    Base* ptr = new Derived();

    // Call virtual function (resolves to Derived's display)
    ptr->display();

    // Call pure virtual function
    ptr->pureVirtual();

    // Delete object to invoke destructors
```

```
 delete ptr;
    return 0;
}
```

The output is :

| Display from Derived class |
| --- |
| Pure virtual function implemented in Derived class |
| Derived class destructor |
| Base class destructor |

The display() function call through the base class pointer resolves to the Derived class's implementation due to the virtual keyword.The virtual destructor ensures that both the Derived and Base class destructors are called in the correct order, preventing resource leaks.

## LAB PROGRAMS

A) 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

```cpp
#include<iostream>
using namespace std;
class base{
    public:
    virtual void display () {
        cout<<"Displayed in base class";
    }
};
class derived : public base {
    public:
    void display() override {
        cout<<"Displayed in derived class";
    }
};
int main() {
    base *b;
    derived d;
    b=&d;
    b->display();
    return 0;
}
```

```
Output
Displayed in derived class

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

B.Create a base class Shape with a virtual destructor.

- Create a derived class Circle that has a constructor and destructor.
- Implement a main() function to demonstrate the use of virtual destructors by creating a
- Shape pointer pointing to a Circle object.

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

class shape {
public:
    virtual ~shape() {
        cout << "Destructor of class shape." << endl;
    }
};

class circle : public shape {
public:
    circle() {
        cout << "Constructor in class circle." << endl;
    }
    ~circle() {
        cout << "Destructor in class circle." << endl;
    }
};

int main() {
    shape* s = new circle();
    delete s;
    return 0;
}
```

Output

```
Constructor in class circle.
Destructor in class circle.
Destructor of class shape.


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

C) 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 ===
```

D) 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:

This lab demonstrated how virtual functions enable runtime polymorphism in C++. By overriding virtual methods in derived classes and using base class pointers, we saw dynamic binding in action. The use of virtual destructors ensured proper object destruction and memory management. Additionally, templates allowed us to write generic functions and classes that work with any data type, promoting code reusability and flexibility.

## Conclusion:

The lab helped us understand the importance of virtual functions and templates in C++. Virtual functions enable dynamic behavior in object-oriented design, while templates allow type-independent programming. Both concepts are vital for writing flexible and reusable code in C++.