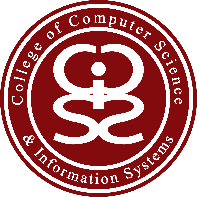
A logo with a letter m

Description automatically generated



**FALL 2023**

Lab Manual

**OBJECT ORIENTED PROGRAMMING**

CSC-213

Lab Manual

**Programming Fundamentals**

**Semester : Fall 2023**

**Program : BS**

**Course Title and Name : CSC 213**

**Credits : 0+1**

**Faculty : Mehar Khan Niazi**

**Student Name : Abdullah Maroof**

**Student ID :20221-33333**

**Total Marks : 100**

**Obtained Marks :**

**Submitted Date :**

**TABLE OF CONTENTS**

|  |  |  |
| --- | --- | --- |
| **Lab No.** | **Experiment Title** | **Date** |
| **1** | **Transition from C to C++**  **●** Understanding the shift from C to C++  ● Setting up the development environment (IDE)  ● Basic syntax and structure transition  ● Handson experience with cin, cout, std::  Namespace, variables, loops, and conditions |  |
| **2** | **Overview of programming paradigms/ Introduction to Classes and Objects**  ● Importance of OOP, Why OOP, Procedural  Programing vs OOP  ● Basic syntax and structure of a class in C++  ● Introduction to classes, objects  ● Access Specifiers in Classes |  |
| **3** | **Functions and Constructors**   * Introduction to Constructors * Default Constructors * Passing and Returning Objects to/from Functions * Private Functions and Variable Scopes |  |
| **4** | **Overloading**   * Method Overloading * Constructor Overloading (Parameterized Constructors) * Introduction to Copy Constructors |  |
| **5** | **Classes and Objects Continued**   * Working with Arrays of Objects * Destructors * Constant Member Functions & Objects |  |
| **6** | **Inheritance**   * Creating Derived and Base Classes * Derived Class Constructors * Understanding the Protected Specifier * Function Overriding * Scope Resolution with Overridden Functions |  |
| **7** | **Level of Inheritance**   * Single Inheritance * Multilevel Inheritance * Multiple Inheritance |  |
| **8.** | **MidTerm Break** |  |
| **9, 10.** | **Version control**  ● Version Control with Git and GitHub  ● Collaborative software development using version control.  **Polymorphism**  ● Virtual Functions  ● Late and Early Binding  ● Friend Functions  ● Abstract Classes in Terms of Polymorphism  ● Function Overloading vs. Overriding  ● Type Casting  ● Handling Multiple Inheritance Issues |  |

**Lab Manual**

**Week 9B Lab: Polymorphism and Associated Concepts**

**Lab Objective:**

The objective of this week 9 and 10 Lab is to introduce students with the concept of polymorphism in Object Oriented Programming (OOP).

The students will learn about:

* Virtual functions and their usage
* Inline and static functions
* Distinction between late binding and early binding
* Implementation and use of friend functions
* Abstract classes in relation to polymorphism
* Differentiating between function overloading and overriding
* Type casting in OOP
* Handling complications in multiple inheritance

**Tools/Software Requirement:**

* DevC++ / Online C++ Compiler

**Virtual functions and Pure virtual functions**

**Objective:**

The objective of this lab is to understand and implement virtual functions and pure virtual functions, which play a crucial role in achieving polymorphism - one of the core concepts of Object-Oriented Programming (OOP). By the end of this lab, students should be able to explain the difference between static linkage and dynamic linkage and how to use virtual functions to enable runtime polymorphism. Additionally, students will learn to declare pure virtual functions to create abstract classes that serve as appropriate base classes for derived classes with concrete implementations.

**Theory:**

**1. Virtual Functions:**

Virtual functions enable a program to decide at runtime which function to call based on the type of the object pointed to by the base pointer. This concept is known as runtime polymorphism or dynamic linkage. When a function is declared as virtual in a base class, it can be overridden in any derived class. The compiler makes sure that the overridden function is called when a derived class object is referred to through a base class pointer or reference.

2**. Pure Virtual Functions and Abstract Classes:**

Sometimes, the base class is only meant to define an interface for its derived classes. In such cases, we declare a virtual function to be pure by assigning 0 to it (as shown in the syntax `virtual int area() = 0;`). A class containing pure virtual functions cannot be instantiated and is considered an abstract class. This is useful when we have a base class that should not be instantiated on its own, but only used as a base for other classes.

Code Examples:

1. Virtual Functions Example:

#include <iostream>

using namespace std;

// Base class

class Shape {

protected:

    int width, height;

public:

    Shape(int a = 0, int b = 0) {

        width = a;

        height = b;

    }

    virtual int area() {

        cout << "Base class area :" << width \* height << endl;

        return width \* height;

    }

};

// Derived class

class Rectangle: public Shape {

public:

    Rectangle(int a = 0, int b = 0):Shape(a, b) { }

    int area() {

        cout << "Rectangle class area :" << width \* height << endl;

        return width \* height;

    }

};

// Main function for the program

int main() {

    Shape \*shape;

    Rectangle rec(10,7);

    // store the address of Rectangle

    shape = &rec;

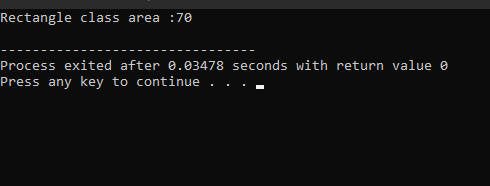
    // call rectangle area.

    shape->area();

    return 0;

}

**Output and Reason:**



This C++ program demonstrates the usage of a virtual function for achieving runtime polymorphism in the program.

The program defines a base class called Shape which has width and height as protected members. It also has a constructor with default values for width and height. Additionally, it defines a virtual function called area() that calculates and returns the area of the shape.

The program also defines a derived class called Rectangle that inherits from the Shape class. It also has a constructor that sets the width and height of the rectangle to the given values. It overrides the area() function of the Shape class to calculate the area of the rectangle and prints out the result.

The main function creates an object of the Rectangle class with given dimensions. It then creates a Shape pointer named shape and assigns the address of the Rectangle object to it. Finally, it calls the area() function using the shape pointer.

Since the area() function is defined as virtual in the Shape class and overridden in the Rectangle class, the area() function called will be dependent on the type of object that the pointer points to at runtime. In this case, it will call the area() function of the Rectangle class.

The program outputs the area of the Rectangle, which is calculated using the overridden area() function of the Rectangle class.

2. Pure Virtual Function Example:

#include <iostream>

using namespace std;

// Base class

class Shape {

protected:

    int width, height;

public:

    Shape(int a = 0, int b = 0) {

        width = a;

        height = b;

    }

    // Pure virtual function

    virtual int area() = 0;

};

// Derived class

class Rectangle : public Shape {

public:

    Rectangle(int a = 0, int b = 0) : Shape(a, b) { }

    int area() {

        cout << "Rectangle class area: " << width \* height << endl;

        return width \* height;

    }

};

// Derived class

class Triangle : public Shape {

public:

    Triangle(int a = 0, int b = 0) : Shape(a, b) { }

    int area() {

        cout << "Triangle class area: " << (width \* height) / 2 << endl;

        return (width \* height) / 2;

    }

};

// Main function for the program

int main() {

    Rectangle rec(10, 7);

    Triangle tri(10, 5);

    // Polymorphic array

    Shape \*shapes[2] = {&rec, &tri};

    // Call area for all shapes.

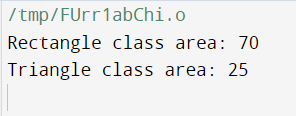
    while (Shape \*shape : shapes) {

        shape->area();

    }

    return 0;

}



**Output and Reason:**

**Output:**

Rectangle class area :70

Triangle class area :25

In this example, the `Shape` class has a pure virtual function, making it an abstract class. The `Rectangle` and `Triangle` classes override the `area` function to provide their specific implementations. Notice that we cannot instantiate the `Shape` class, and the `area` method must be implemented by the derived classes.

**Lab Tasks:**

**Requirements (Code, Output, and Reason/Explanation)**

**Task 1: Pure Virtual Function Usage**

Create a base class called Shape class to be an abstract class by making the draw() function a pure virtual function. And derive two classes from it: Circle and Square. Implement draw() in both derived classes to print out which shape is being drawn. Use Shape pointers to demonstrate polymorphic behavior.

**For each task, students are required to provide the following:**

**Code:** The full source code of the implementation.

Here's an example implementation of the modified Shape class, along with the Circle and Square classes:

#include<iostream>

using namespace std;

// Base class

class Shape {

public:

virtual void draw() = 0; // pure virtual function

};

// Derived class 1 - Circle

class Circle: public Shape {

public:

void draw() {

cout << "Drawing a Circle" << endl ;

}

};

// Derived class 2 - Square

class Square: public Shape {

public:

void draw() {

cout << "Drawing a Square" << endl ;

}

};

int main() {

Shape \*shape1, \*shape2; // declaring two pointers of type Shape

Circle circle; // creating Circle object

Square square; // creating Square object

shape1 = &circle; // assigning Circle object address to shape1

shape2 = &square; // assigning Square object address to shape2

shape1->draw(); // calling draw() using shape1 (polymorphic behavior)

shape2->draw(); // calling draw() using shape2 (polymorphic behavior)

return 0;

}

**Reason/Explanation:** An explanation of how the code works and how it demonstrates the given task concept.

Here, we declare a Shape pointer named `shape1`. We then create a Circle object named `circle`, and assign the address of `circle` to `shape1`. This is possible because Circle is derived from Shape, and therefore can be treated as a Shape object.

We do the same with a Square object and a Shape pointer named `shape2`.

We are then able to call the `draw()` function for both objects using their respective pointers. This demonstrates polymorphic behavior, as the `draw()` function called depends on the type of object pointed to by each pointer.

**Output:** A screenshot or text copy of the output from running the program.

