DAY -5 LAB Assignments

#include <iostream>

using namespace std;

#include <string>

/\* void swapByValue(int x, int y) {

int temp = x;

x = y;

y = temp;

int result = x+y;

cout << "result : = " << result<<endl;

}

int main() {

int a = 2, b = 1;

cout << "Before swap : a = " << a << ", b = " << b << std::endl;

swapByValue(a, b);

cout << "After swap : a = " << a << ", b = " << b << std::endl;

return 0;

} \*/

/\* void swapByValue(int& x, int& y) {

int temp = x;

x = y;

y = temp;

}

int main() {

int a = 2, b = 1;

cout << "Before swap : a = " << a << ", b = " << b << std::endl;

swapByValue(a, b);

cout << "After swap : a = " << a << ", b = " << b << std::endl;

return 0;

} \*/

// Type-1 Pass by Value and Pass by Reference Ambiguity

/\*void swap (int a,int b)

{

int temp=a;

a=b;

b=temp;

cout<<"x is:"<<a<<"y is:"<<b<<endl;

}

void swap (int& a,int& b){

int temp = a;

a = b;

b = temp;

cout<<"x is:"<<a<<"y is:"<<b<<endl;

}

int main()

{

int x=5,y=6;

swap(x,y);

swap(x,y);

return 0;

} \*/

// Type-2 Parameter passing Ambiguity (Default Argument)

/\* void test(int a)

{

cout<<"x is"<<a<<endl;

}

void test(int a,int b=5) // Default Argument

{

cout<<"x is : "<<a<<"y is:"<<b<<endl;

}

int main()

{

int x=5,y=6;

test (x);

test(x,y);

return 0;

} \*/

// Type 3 -Type Mismatch Ambiguity

/\*void test(float a,float b) // Declared float here

{

cout<<"x is"<<a<<"y is"<<b<<endl;

}

void test(int a,int b) // Declared int here

{

cout<<"x is : "<<a<<"y is:"<<b<<endl;

}

int main()

{

float x,y; // float data type declared here

// int x,y;

test (5.5,6.9);

test (5.5,6.9);

// test (5,6);

return 0;

} \*/

// Type-4 Multiple inheritence Ambiguity

/\* class A{

public:

void show(){

cout<<"This is class 'A'"<<endl;

}

};

class B{

public:

void show(){

cout<<"This is class 'B'"<<endl;

}

};

class C : public A,public B{

};

int main(){

C obj;

// obj.show(); //Ambigious: show() is present in both A and both

//Resolving Ambiguity using Scope resolution operator

obj.A::show(); // Calls show() from class A

obj.B::show(); // Calls show() from class B

return 0;

} \*/

// Ambiguity in case of Operator Overloading

/\* class A {

public:

void operator+(const A&) {

cout << "Operator+ for class A" <<endl;

}

};

class B {

public:

void operator+(const B&) {

cout << "Operator+ for class B" <<endl;

}

};

class C : public A, public B {

public:

// Resolving ambiguity by explicitly defining operator+ in class C

void operator+(const C&) {

cout << "Operator+ for class C" <<endl;

}

};

int main() {

C obj1, obj2;

obj1 + obj2; // Ambiguous: operator+ for different types A and B

// Resolving ambiguity using static\_cast

static\_cast<A&>(obj1) + static\_cast<A&>(obj2); // Calls operator+ for class A

static\_cast<B&>(obj1) + static\_cast<B&>(obj2); // Calls operator+ for class B

// Resolving ambiguity by defining operator+ in class C

obj1 + obj2; // Calls operator+ for class C

return 0;

} \*/

/\* #include <iostream>

class Matrix {

public:

// Constructor to initialize matrix with dimensions and default values

Matrix(int rows, int cols, double defaultValue = 0.0) : rows\_(rows), cols\_(cols) {

data\_ = new double\*[rows];

for (int i = 0; i < rows; ++i) {

data\_[i] = new double[cols];

for (int j = 0; j < cols; ++j) {

data\_[i][j] = defaultValue;

}

}

}

// Destructor to free allocated memory

~Matrix() {

for (int i = 0; i < rows\_; ++i) {

delete[] data\_[i];

}

delete[] data\_;

}

// Function to input matrix elements from user

void input() {

for (int i = 0; i < rows\_; ++i) {

for (int j = 0; j < cols\_; ++j) {

std::cout << "Enter element (" << i << ", " << j << "): ";

std::cin >> data\_[i][j];

}

}

}

// Overload the + operator for matrix addition

Matrix operator+(const Matrix& other) const {

if (rows\_ != other.rows\_ || cols\_ != other.cols\_) {

throw std::runtime\_error("Matrix dimensions must be equal for addition");

}

Matrix result(rows\_, cols\_);

for (int i = 0; i < rows\_; ++i) {

for (int j = 0; j < cols\_; ++j) {

result.data\_[i][j] = data\_[i][j] + other.data\_[i][j];

}

}

return result;

}

// Overload the \* operator for matrix multiplication

Matrix operator\*(const Matrix& other) const {

if (cols\_ != other.rows\_) {

throw std::runtime\_error("Incompatible dimensions for multiplication");

}

Matrix result(rows\_, other.cols\_);

for (int i = 0; i < rows\_; ++i) {

for (int j = 0; j < other.cols\_; ++j) {

result.data\_[i][j] = 0;

for (int k = 0; k < cols\_; ++k) {

result.data\_[i][j] += data\_[i][k] \* other.data\_[k][j];

}

}

}

return result;

}

// Function to display the matrix elements

void display() const {

for (int i = 0; i < rows\_; ++i) {

for (int j = 0; j < cols\_; ++j) {

std::cout << data\_[i][j] << " ";

}

std::cout << std::endl;

}

}

private:

int rows\_;

int cols\_;

double\*\* data\_;

};

int main() {

// Test case 1: Matrices with same dimensions

Matrix m1(2, 2);

Matrix m2(2, 2);

std::cout << "Enter elements for matrix 1:\n";

m1.input();

std::cout << "Enter elements for matrix 2:\n";

m2.input();

std::cout << "Matrix 1:\n";

m1.display();

std::cout << "Matrix 2:\n";

m2.display();

try {

Matrix sum = m1 + m2;

std::cout << "Sum of matrices:\n";

sum.display();

} catch (const std::runtime\_error& e) {

std::cerr << "Error: " << e.what() << std::endl;

}

try {

Matrix product = m1 \* m2;

std::cout << "Product of matrices:\n";

product.display();

} catch (const std::runtime\_error& e) {

std::cerr << "Error: " << e.what() << std::endl;

}

} \*/

/\* #include <iostream>

class Distance {

public:

// Constructor with default values (0 feet, 0 inches)

Distance() : feet\_(0), inches\_(0) {}

// Constructor with specified feet and inches

Distance(int feet, int inches) : feet\_(feet), inches\_(inches) {

// Handle negative values and convert inches to feet if needed

if (inches\_ < 0) {

feet\_--;

inches\_ += 12;

}

inches\_ %= 12; // Ensure inches are between 0 and 11

}

// Display the distance in feet and inches format

void display() const {

std::cout << feet\_ << " ft " << inches\_ << " in" << std::endl;

}

// Overload the + operator for addition of distances

Distance operator+(const Distance& other) const {

int total\_inches = inches\_ + other.inches\_;

int total\_feet = feet\_ + other.feet\_ + total\_inches / 12;

return Distance(total\_feet, total\_inches % 12);

}

// Overload the - operator for subtraction of distances

Distance operator-(const Distance& other) const {

int inches\_diff = inches\_ - other.inches\_;

int feet\_diff = feet\_ - other.feet\_;

if (inches\_diff < 0) {

inches\_diff += 12;

feet\_diff--;

}

return Distance(feet\_diff, inches\_diff);

}

// Overload comparison operators for distance comparisons

bool operator==(const Distance& other) const {

return (feet\_ == other.feet\_) && (inches\_ == other.inches\_);

}

bool operator!=(const Distance& other) const {

return !(\*this == other); // Use overloaded == for negation

}

bool operator<(const Distance& other) const {

if (feet\_ < other.feet\_) {

return true;

} else if (feet\_ > other.feet\_) {

return false;

} else { // Feet are equal, compare inches

return inches\_ < other.inches\_;

}

}

bool operator>(const Distance& other) const {

return other < \*this; // Use overloaded < for reversed comparison

}

bool operator<=(const Distance& other) const {

return \*this == other || \*this < other;

}

bool operator>=(const Distance& other) const {

return \*this == other || \*this > other;

}

private:

int feet\_;

int inches\_;

};

int main() {

Distance d1(5, 7); // 5 feet 7 inches

Distance d2(3, 10); // 3 feet 10 inches

std::cout << "Distance 1: ";

d1.display();

std::cout << "Distance 2: ";

d2.display();

Distance sum = d1 + d2;

std::cout << "Distance 1 + Distance 2: ";

sum.display();

Distance difference = d1 - d2;

std::cout << "Distance 1 - Distance 2: ";

difference.display();

if (d1 == d2) {

std::cout << "Distance 1 and Distance 2 are equal.\n";

} else {

std::cout << "Distance 1 and Distance 2 are not equal.\n";

}

if (d1 > d2) {

std::cout << "Distance 1 is greater than Distance 2.\n";

} else if (d1 < d2) {

std::cout << "Distance 1 is less than Distance 2.\n";

} else {

std::cout << "Distance 1 and Distance 2 are equal.\n";

}

return 0;

} \*/

/\* Task 1- Calculator Using Method Overloading (Time-12.26 p.m)

You are required to implement a program that calculates the area of different shapes using compile-time polymorphism (method overloading) in C++. The program should support calculation of areas for the following shapes:

Rectangle

Circle

Triangle

Requirements:

Shape Class: Implement a Shape class as a base class with virtual functions

to calculate and display the area of each shape.Derived Classes: Implement

derived classes Rectangle, Circle, and Triangle, inheriting from Shape, each

with overridden functions to calculate and display their respective areas.

Method Overloading: Use method overloading in the Shape class to define

multiple calculateArea functions, each specific to one shape.

Input and Output: Implement a main() function to test the implemented classes

by creating instances of each shape, inputting dimensions,

and displaying their calculated areas. \*/

/\*class Shape {

public:

// Pure virtual function to enforce area calculation in derived classes

virtual double calculateArea() const = 0;

// Method overloading to display area for consistency (can be overridden)

void displayArea() const {

cout << "Area of Figure are : " << calculateArea() << endl;

}

};

class Rectangle : public Shape {

public:

Rectangle(double length, double width) : length\_(length), width\_(width) {}

private:

double length\_;

double width\_;

// Override calculateArea for rectangle

double calculateArea() const override {

return length\_ \* width\_;

}

};

class Circle : public Shape {

public:

Circle(double radius) : radius\_(radius) {}

private:

double radius\_;

// Override calculateArea for circle

double calculateArea() const override {

return 3.14159 \* radius\_ \* radius\_;

}

};

class Triangle : public Shape {

public:

Triangle(double base, double height) : base\_(base), height\_(height) {}

private:

double base\_;

double height\_;

// Override calculateArea for triangle

double calculateArea() const override {

return 0.5 \* base\_ \* height\_;

}

};

int main() {

// Create shapes and calculate areas using base class pointer

Shape\* shape1 = new Rectangle(12, 4);

shape1->displayArea();

Shape\* shape2 = new Circle(7);

shape2->displayArea();

Shape\* shape3 = new Triangle(14, 21);

shape3->displayArea();

// Deallocate memory (remember to delete for each new)

delete shape1;

delete shape2;

delete shape3;

return 0;

} \*/

// Code for Function overloading

/\*class Animal{

public:

void eat(){

cout<<"Eating.....";

}

};

class Dog : public Animal{

public:

void eat(){

cout<<"Eating Bread.....";

}

};

int main (){

Dog d=Dog();

//d.eat();

d.Animal::eat();

return 0;

} \*/

// Code to print the data of base class without using base scope resolution opertor

/\*class A{

int x=5;

public:

void display(){

cout<<"Value of x is:"<<x<<endl;

}

};

class B:public A{

int y = 10;

public:

void display()

{

cout<<"Value of y is: "<<y<<endl;

}

};

int main(){

A \*a;

B b;

a =&b;

a->display();

b.B::display(); // Explicitly display the data of Derived class

return 0;

} \*/

// Use virtual functions for above class

/\* class A{

int x=5;

public:

virtual void display(){

cout<<"Value of x is:"<<x<<endl;

}

};

class B:public A{

int y = 10;

public:

void display()

{

//A::display();

cout<<"Value of y is: "<<y<<endl;

}

};

int main(){

A \*a;

B b;

a =&b;

a->display();

return 0;

} \*/

/\* class A{

public:

virtual void display(){

cout<<"Base class is invoked"<<endl;

}

};

class B:public A{

public:

void display()

{

cout<<"Derived class is invoked"<<endl;

}

};

int main(){

A\* a; // pointer of base class

B b; // object of derived class

a =&b;

a->display(); // Late binding occurs

return 0;

} \*/

// Code Assignment Question 1: Shape Hierarchy time - 2:52 p.m

/\* Create a base class Shape with a pure virtual function draw() that has

no implementation. Derive classes Square, Circle, and Triangle from Shape.

Each derived class should override draw() to provide its specific drawing

behavior (e.g., printing "\*" for square, "OOO" for circle, etc.).

Write a function printShape(Shape shape) that takes a base class pointer

and calls draw() on it. Demonstrate polymorphism by creating objects of the

derived classes, storing them in a Shape\* array, and calling printShape()

on each element. \*/

// Base class

/\* class Shape {

public:

virtual void draw(){ // Pure virtual function

cout << "Base class is called "<<endl;

};

};

// Derived class Square

class Square : public Shape {

public:

void draw() override {

cout << "Square class is called "<<endl;

}

};

// Derived class Circle

class Circle : public Shape {

public:

void draw() override {

cout << "Circle class is called"<<endl;

}

};

// Derived class Triangle

class Triangle : public Shape {

public:

void draw() override {

std::cout << "Triangle class is called"<<endl;

}

};

// Function to print shape

void printShape(Shape\* shape) {

shape->draw();

}

int main() {

// Create objects of derived classes

Square square;

Circle circle;

Triangle triangle;

// Store them in a Shape\* array

Shape\* shapes[] = { &square, &circle, &triangle };

// Call printShape on each element

for (Shape\* shape : shapes) {

printShape(shape);

cout <<endl;

}

return 0;

} \*/

//Question 2: Animal Sounds

/\* Design a base class Animal with a pure virtual function makeSound()

that returns a string representing the animal's sound. Derive classes

like Dog, Cat, and Bird from Animal, each overriding makeSound() with

the appropriate sound ("Woof!", "Meow!", "Chirp!"). Create a function

playAnimalSound(Animal\* animal) that takes an Animal pointer and calls

makeSound(). Populate an Animal\* array with various animal objects and

use playAnimalSound() to hear their sounds polymorphically. \*/

// Base class

/\* class Animal {

public:

virtual string makeSound() {

return" This is the Base class";// Pure virtual function

}

};

// Derived class Dog

class Dog : public Animal {

public:

string makeSound() override {

return "Woof!";

}

};

// Derived class Cat

class Cat : public Animal {

public:

string makeSound() override {

return "Meow!";

}

};

// Derived class Bird

class Bird : public Animal {

public:

string makeSound() override {

return "Chirp!";

}

};

// Function to play animal sound

void playAnimalSound(Animal\* animal) {

cout << animal->makeSound() << endl;

}

int main() {

// Create objects of derived classes

Dog dog;

Cat cat;

Bird bird;

// Store them in an Animal\* array

Animal\* animals[] = { &dog, &cat, &bird };

// Call playAnimalSound on each element

for (Animal\* animal : animals) {

playAnimalSound(animal);

}

return 0;

} \*/

// Code on "Copy Constructor"

class Point{

public:

double x,y;

Point(){

x = 0.0;

y = 0.0;

cout<<" Default Constructor"<<endl;

}

Point(double nx,double ny){

x = nx;

y = ny;

cout<<" 2-Parameter Constructor"<<endl;

}

} ;

int main()

{

Point q(1.0,2.0); // 2- Parameter Constructor

Point r= q; // Invoking the copy Constructor/

}

// Code on " Custom Copy Constructor ""

// One can define its own copy Constructor

/\* class Point{

public:

double x,y;

Point(double nx,double ny){

x = nx;

y = ny;

cout<<" 2-Parameter Constructor"<<endl;

}

Point(Point &o){

x = o.x;

y = o.y;

cout<<" Custom Copy Constructor"<<endl;

}

} ;

int main()

{

Point q(1.0,2.0);

Point r= q; // Custom copy Constructor

} \*/