#include <iostream>

using namespace std;

void passbyVal(int a, int b) {

int temp = a;

a = b;

b = temp;

}

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

int temp = a;

a = b;

b = temp;

}

int main() {

int a = 20, b = 30;

cout << "Original values: a = " << a << ", b = " << b << endl;

passbyVal(a, b);

cout << "After passbyVal: a = " << a << ", b = " << b << endl;

passbyRe(a, b);

cout << "After passbyRe: a = " << a << ", b = " << b << endl;

return 0;

}

**Problem Statement: Distance Calculation Using Operator Overloading**

**You are required to implement a program that calculates distances using operator overloading in C++. The program should be able to perform the following operations on distances:**

**Addition of Distances:**

**Implement an addition operator (+) that adds two distances together.**

**The distance should be represented in feet and inches.**

**Subtraction of Distances:**

**Implement a subtraction operator (-) that subtracts one distance from another.**

**Ensure that the subtraction operation handles cases where the result may involve negative values or borrowing (like in subtraction of inches).**

**Comparison of Distances:**

**Implement comparison operators (==, !=, <, >, <=, >=) to compare distances based on their total length (combined feet and inches).**

**Use these operators to determine which distance is greater, less than, or equal to another.**

**Requirements:**

**Distance Class: Implement a Distance class with appropriate member variables (feet and inches).**

**Constructors: Implement constructors to initialize distances.**

**Member Functions: Implement member functions for display and any other necessary operations.**

**Operator Overloading: Overload the necessary operators (+, -, ==, !=, <, >, <=, >=) inside the Distance class to perform the specified operations.**

**Testing: Create a main() function to test the implemented Distance class and its operator overloading functionality. Test various scenarios including addition, subtraction, and comparison of distances.**

#include <iostream>

using namespace std;

class Distance {

private:

int feet;

float inches;

public:

Distance() : feet(0), inches(0.0) {}

Distance(int ft, float in) : feet(ft), inches(in) {}

void displayDistance() {

cout << "Feet: " << feet << " Inches: " << inches << endl;

}

Distance operator+(const Distance& d2) {

int totalFeet = feet + d2.feet;

float totalInches = inches + d2.inches;

if (totalInches >= 12.0) {

totalInches -= 12.0;

totalFeet++;

}

return Distance(totalFeet, totalInches);

}

Distance operator-(const Distance& d2) {

int totalFeet = feet - d2.feet;

float totalInches = inches - d2.inches;

if (totalInches < 0) {

totalInches += 12.0;

totalFeet--;

}

return Distance(totalFeet, totalInches);

}

bool operator==(Distance& d2) {

return (feet == d2.feet && inches == d2.inches); }

bool operator!=(Distance& d2) {

return !(\*this == d2); }

bool operator<(Distance& d2) {

float thisTotal = feet + (inches / 12.0);

float d2Total = d2.feet + (d2.inches / 12.0);

return thisTotal < d2Total; }

bool operator>(Distance& d2) {

return !(\*this < d2 || \*this == d2); }

bool operator<=(Distance& d2) {

return (\*this < d2 || \*this == d2); }

bool operator>=(Distance& d2) {

return !(\*this < d2); }

};

int main() {

Distance d1(10, 6.5);

Distance d2(5, 3.25);

Distance sum = d1 + d2;

cout << "Sum of distances: ";

sum.displayDistance();

Distance diff = d1 - d2;

cout << "Difference of distances: ";

diff.displayDistance();

if (d1 == d2)

cout << "Distances are equal" << endl;

else if (d1 != d2)

cout << "Distances are not equal" << endl;

if (d1 < d2)

cout << "Distance d1 is less than d2" << endl;

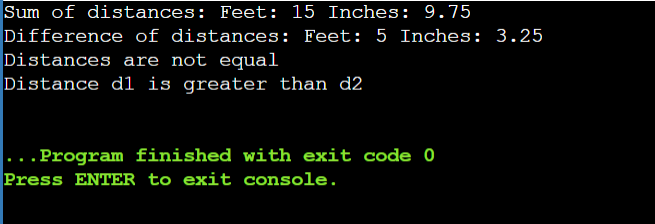
else if (d1 > d2)

cout << "Distance d1 is greater than d2" << endl;

return 0;

}

**Output**

****

**Q.2 >>Problem Statement: Shape Area Calculator Using Method Overloading**

**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.**

#include <iostream>

#include <cmath> // For M\_PI constant

using namespace std;

// Base Shape class

class Shape {

public:

// Virtual function for calculating area (to be overridden)

virtual void calculateArea() {

cout << "Calculating area of generic shape" << endl;

}

// Virtual function for displaying area (to be overridden)

virtual double getArea() const {

return area;

}

protected:

double area;

};

// Derived Rectangle class

class Rectangle : public Shape {

public:

Rectangle(double l, double b) : length(l), breadth(b) {}

// Override calculateArea for Rectangle

void calculateArea() override {

area = length \* breadth;

}

private:

double length;

double breadth;

};

// Derived Circle class

class Circle : public Shape {

public:

Circle(double r) : radius(r) {}

// Override calculateArea for Circle

void calculateArea() override {

area = M\_PI \* radius \* radius; // Using M\_PI for pi constant

}

private:

double radius;

};

// Derived Triangle class

class Triangle : public Shape {

public:

Triangle(double b, double h) : base(b), height(h) {}

// Override calculateArea for Triangle

void calculateArea() override {

area = 0.5 \* base \* height;

}

private:

double base;

double height;

};

int main() {

// Example usage:

Rectangle rect(5.0, 3.0);

Circle circle(4.5);

Triangle triangle(4.0, 2.5);

// Calculate areas

rect.calculateArea();

circle.calculateArea();

triangle.calculateArea();

// Display areas

cout << "Rectangle Area: " << rect.getArea() << endl;

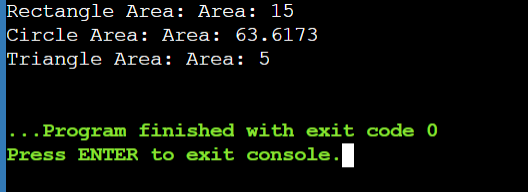
cout << "Circle Area: " << circle.getArea() << endl;

cout << "Triangle Area: " << triangle.getArea() << endl;

return 0;

}

**Output**

****

**Overload the following operators for the Matrix class:**

**+ (addition): Allows addition of two matrices.**

**\* (multiplication): Allows multiplication of two matrices.**

**Ensure that the operations adhere to matrix addition and multiplication rules (e.g., matrix dimensions compatibility).**

**Input and Output:**

**Implement a function to input matrices from the user or use predefined matrices for testing purposes.**

**Display the result of matrix addition and multiplication operations using overloaded operators.**

**Testing and Output:**

**Create a main() function to test your Matrix class and its operator overloading functionality.**

**Test with multiple matrices of different dimensions to demonstrate compile-time polymorphism through operator overloading.**

#include <iostream>

using namespace std;

class Matrix {

private:

int m[10][10];

int rows;

int cols;

public:

Matrix(int r, int c) : rows(r), cols(c) {

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

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

m[i][j] = 0;

}

}

}

void inputMatrix() {

cout << "Enter matrix elements row-wise:" << endl;

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

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

cin >> m[i][j];

}

}

}

void displayMatrix() {

cout << "Matrix:" << endl;

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

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

cout << m[i][j] << " ";

}

cout << endl;

}

}

Matrix operator+(Matrix& matrix) {

Matrix result(rows, cols);

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

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

result.m[i][j] = m[i][j] + matrix.m[i][j];

}

}

return result;

}

Matrix operator\*(Matrix& matrix) {

Matrix result(rows, matrix.cols);

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

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

result.m[i][j] = 0;

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

result.m[i][j] += m[i][k] \* matrix.m[k][j];

}

}

}

return result;

}

};

int main() {

Matrix A(3, 3);

Matrix B(3, 3);

cout << "Enter elements for matrix A:" <<endl;

A.inputMatrix();

cout << "Enter elements for matrix B:" <<endl;

B.inputMatrix();

cout << "Matrix A:" <<endl;

A.displayMatrix();

cout << "Matrix B:" <<endl;

B.displayMatrix();

Matrix C = A + B;

cout << "Result of A + B:" <<endl;

C.displayMatrix();

Matrix D = A \* B;

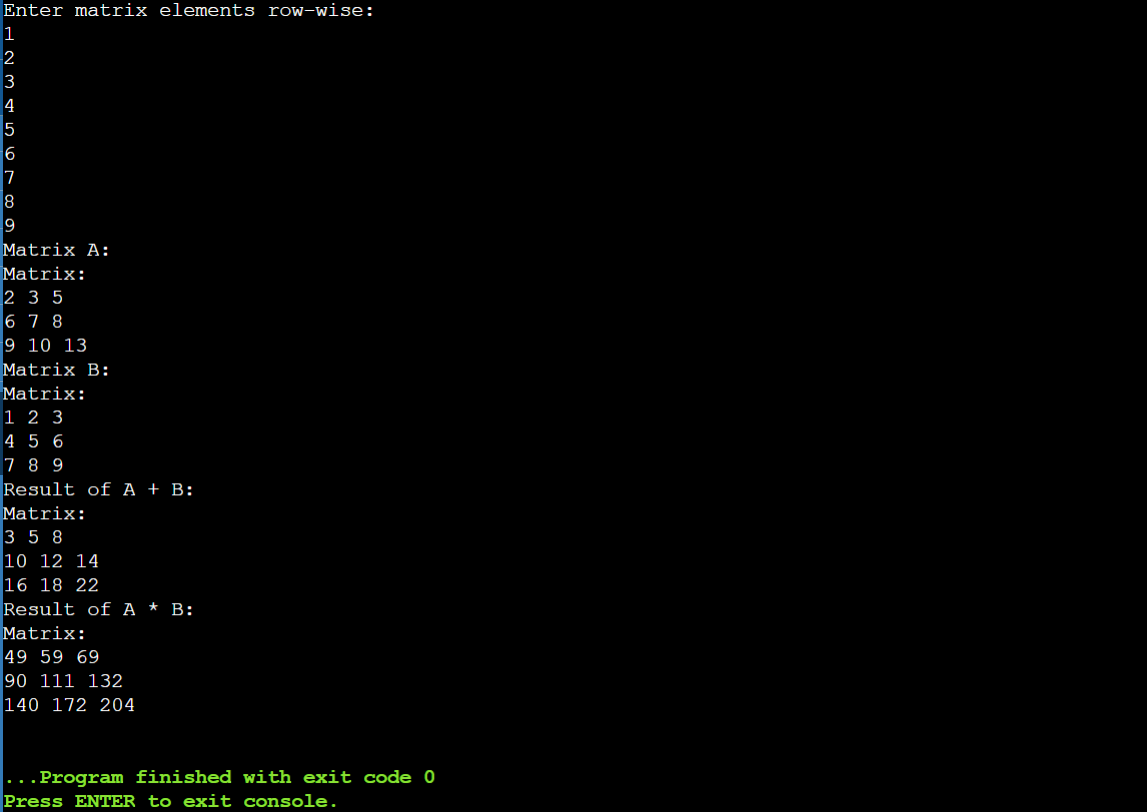
cout << "Result of A \* B:" <<endl;

D.displayMatrix();

return 0;

}

**Output**

****

**Function Overriding**

#include <iostream>

using namespace std;

class Animal {

public:

void eat(){

cout<<"Eating....";

}

};

class Dog:public Animal

{

public:

void eat()

{

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

}

};

int main(void){

Dog d = Dog();

d.eat();

return 0;

}

#include <iostream>

using namespace std;

class A{

public:

virtual 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();

return 0;

}

**Question 1: Shape Hierarchy**

**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.**

#include <iostream>

using namespace std;

// Base class Shape with a pure virtual function draw()

class Shape {

public:

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

virtual ~Shape() {} // Virtual destructor

};

// Derived class Square

class Square : public Shape {

public:

void draw() override {

cout << "Drawing a Square: \*\*\*\*\*" << endl;

}

};

// Derived class Circle

class Circle : public Shape {

public:

void draw() override {

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

}

};

// Derived class Triangle

class Triangle : public Shape {

public:

void draw() override {

cout << "Drawing a Triangle: /\\ /\\ /\\ /\\ /\\ " << endl;

}

};

// Function to print shape using a base class pointer

void printShape(Shape\* shape) {

shape->draw();

}

int main() {

// Creating objects of derived classes

Square square;

Circle circle;

Triangle triangle;

// Creating an array of Shape pointers

Shape\* shapes[3];

shapes[0] = &square;

shapes[1] = &circle;

shapes[2] = &triangle;

// Demonstrating polymorphism

for (int i = 0; i < 3; ++i) {

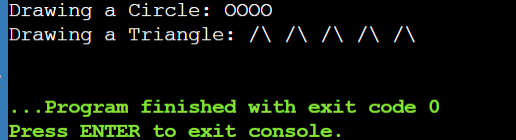
printShape(shapes[i]);

}

return 0;

}

**Output**

****

**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.**

#include <iostream>

#include <string>

using namespace std;

// Base class Animal

class Animal {

public:

virtual string makeSound() = 0; // 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;

// Array of Animal pointers

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

// Call playAnimalSound() on each element

for (int i = 0; i < 3; ++i) {

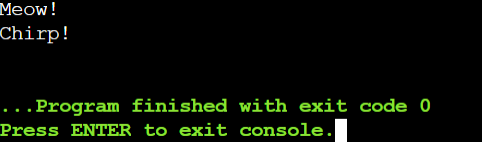
playAnimalSound(animals[i]);

}

return 0;

}

**Output**

****

**Copy Constructor**

#include <iostream>

using namespace std;

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-paramerter constructor" << endl;

}

};

int main() {

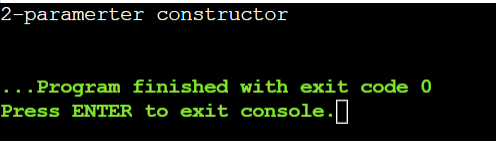
Point q(1.0, 2.0);

Point r = q;

// r.x is 1.0, r.y is 2.0)

}

**Output**

****

**You can define your own copy constructor**

#include <iostream>

using namespace std;

class Point {

public:

double x, y;

Point(double nx, double ny) {

x = nx;

y = ny;

cout << "2-paramerter constructor" << endl;

}

Point(Point &o) {

x = o.x;

y = o.y;

cout << "custom copy constructor" << endl;

}

};

int main() {

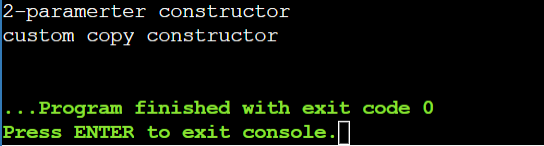
Point q(1.0, 2.0);//2-paramerter constructor

Point r = q;// custom copy constructor

// r.x is 1, r.y is 2)

}

**Output**

****