Ambiquious error :

#include <iostream>

using namespace std;

void test(float a,float b)

{

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

}

void test(int a, int b)

{

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

}

int main()

{

double x=6.0,y=8.0;

test(x,y);

return 0;

}

}

2. 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;

// Distance class definition

class Distance {

private:

int feet;

int inches;

public:

// Constructor to initialize distance

Distance(int f = 0, int i = 0) : feet(f), inches(i) {

// Normalize the distance so inches are less than 12

while (inches >= 12) {

feet++;

inches -= 12;

}

}

// Member function to display the distance

void display() const {

cout << feet << " feet " << inches << " inches" << endl;

}

// Overload + operator for addition

Distance operator+(const Distance &d) const {

int newFeet = feet + d.feet;

int newInches = inches + d.inches;

return Distance(newFeet, newInches);

}

// Overload - operator for subtraction

Distance operator-(const Distance &d) const {

int totalInches1 = feet \* 12 + inches;

int totalInches2 = d.feet \* 12 + d.inches;

int diffInches = totalInches1 - totalInches2;

return Distance(diffInches / 12, diffInches % 12);

}

// Overload == operator for equality comparison

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

return (feet == d.feet && inches == d.inches);

}

// Overload != operator for inequality comparison

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

return !(\*this == d);

}

// Overload < operator for less-than comparison

bool operator<(const Distance &d) const {

if (feet < d.feet) return true;

if (feet == d.feet && inches < d.inches) return true;

return false;

}

// Overload > operator for greater-than comparison

bool operator>(const Distance &d) const {

return d < \*this;

}

// Overload <= operator for less-than or equal comparison

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

return !(\*this > d);

}

// Overload >= operator for greater-than or equal comparison

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

return !(\*this < d);

}

};

// Main function to test the Distance class

int main() {

Distance d1(5, 10); // Create a Distance object d1 with 5 feet 10 inches

Distance d2(3, 11); // Create a Distance object d2 with 3 feet 11 inches

// Display the distances

cout << "Distance 1: ";

d1.display();

cout << "Distance 2: ";

d2.display();

// Test addition

Distance d3 = d1 + d2;

cout << "After addition (d1 + d2): ";

d3.display();

// Test subtraction

Distance d4 = d1 - d2;

cout << "After subtraction (d1 - d2): ";

d4.display();

// Test comparison operators

cout << "d1 == d2: " << (d1 == d2) << endl;

cout << "d1 != d2: " << (d1 != d2) << endl;

cout << "d1 < d2: " << (d1 < d2) << endl;

cout << "d1 > d2: " << (d1 > d2) << endl;

cout << "d1 <= d2: " << (d1 <= d2) << endl;

cout << "d1 >= d2: " << (d1 >= d2) << endl;

return 0;

}

FUNTION OVERRING:

#include<iostream>

using namespace std;

class animal{

public:

void eat(){

cout<<"eting";

}

};

class dog: public animal

{

public:

void eat()

{

cout<<"eating bread";

}

};

int main(void){

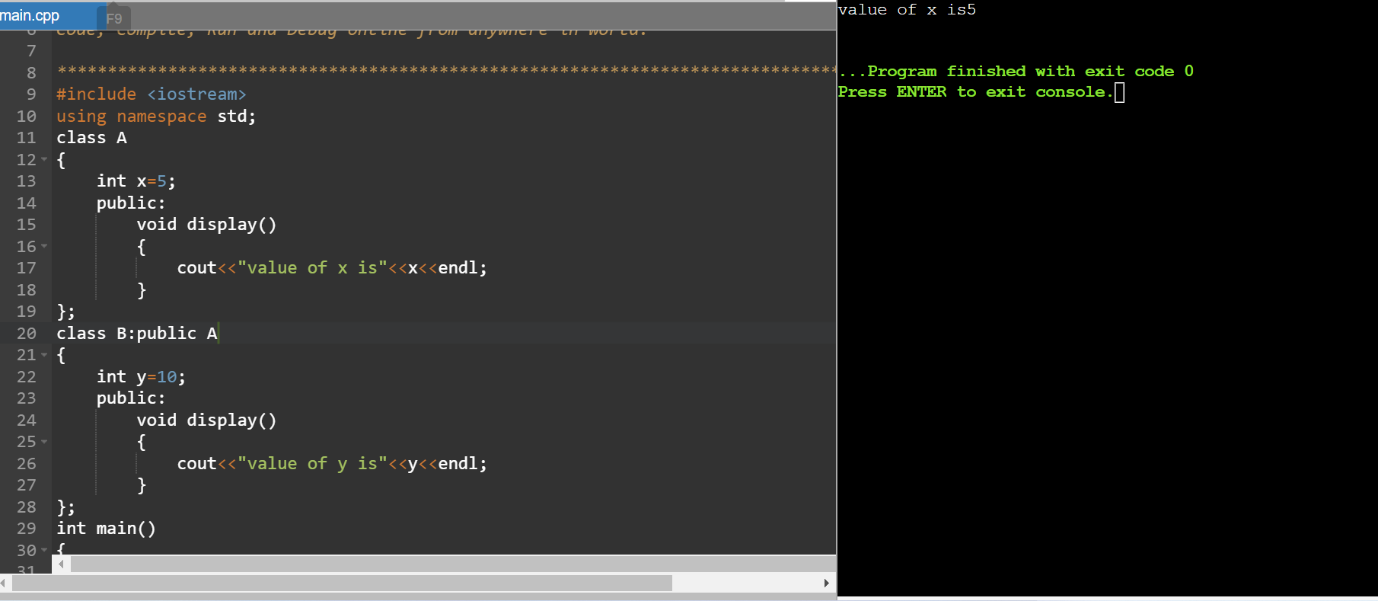
dog d = dog();

d.eat();

return 0;

}

VIRTUAL FUNTION:



Sir questions:

#include <iostream>

using namespace std;

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

class Shape {

public:

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

};

// Derived class Square

class Square : public Shape {

public:

void draw() const override {

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

}

};

// Derived class Circle

class Circle : public Shape {

public:

void draw() const override {

cout << "Drawing a circle: OOO" << endl;

}

};

// Derived class Triangle

class Triangle : public Shape {

public:

void draw() const override {

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

}

};

// Function to print the shape

void printShape(const Shape\* shape) {

shape->draw();

}

int main() {

// Create objects of derived classes

Square square;

Circle circle;

Triangle triangle;

// Create an array of Shape pointers

Shape\* shapes[3];

shapes[0] = &square;

shapes[1] = &circle;

shapes[2] = &triangle;

// Demonstrate polymorphism

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

printShape(shapes[i]);

}

return 0;

}

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 with a pure virtual function makeSound()

class Animal {

public:

virtual string makeSound() const = 0; // Pure virtual function

};

// Derived class Dog

class Dog : public Animal {

public:

string makeSound() const override {

return "Woof!";

}

};

// Derived class Cat

class Cat : public Animal {

public:

string makeSound() const override {

return "Meow!";

}

};

// Derived class Bird

class Bird : public Animal {

public:

string makeSound() const override {

return "Chirp!";

}

};

// Function to play the animal sound

void playAnimalSound(const Animal\* animal) {

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

}

int main() {

// Create objects of derived classes

Dog dog;

Cat cat;

Bird bird;

// Create an array of Animal pointers

Animal\* animals[3];

animals[0] = &dog;

animals[1] = &cat;

animals[2] = &bird;

// Demonstrate polymorphism

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

playAnimalSound(animals[i]);

}

return 0;

}

COPY CONSTRUCTOR:

1. #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)

}

2. #include<iostream>

using namespace std;

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;

}