Create a program that simulates a zoo with various animals. Each animal should have a common method called "speak" that makes a sound specific to the animal type.

Objective:

Utilize runtime polymorphism to achieve the following:

Define an abstract base class named Animal with a method speak that doesn't have an implementation (declare it abstract).

Create subclasses for different animals like Lion, Elephant, etc., inheriting from Animal.

Override the speak method in each subclass to define the specific sound of the animal (e.g., Lion roars, Elephant trumpets).

In the main program, create an array of Animal references. Populate this array with objects of different animal subclasses.

Loop through the animal array and call the speak method on each reference. Since the references are of the base class type, runtime polymorphism will determine the actual subclass and invoke the appropriate overridden speak method.

This exercise will demonstrate runtime polymorphism by:

Highlighting the separation between declared type (reference variable type) and actual type (object type).

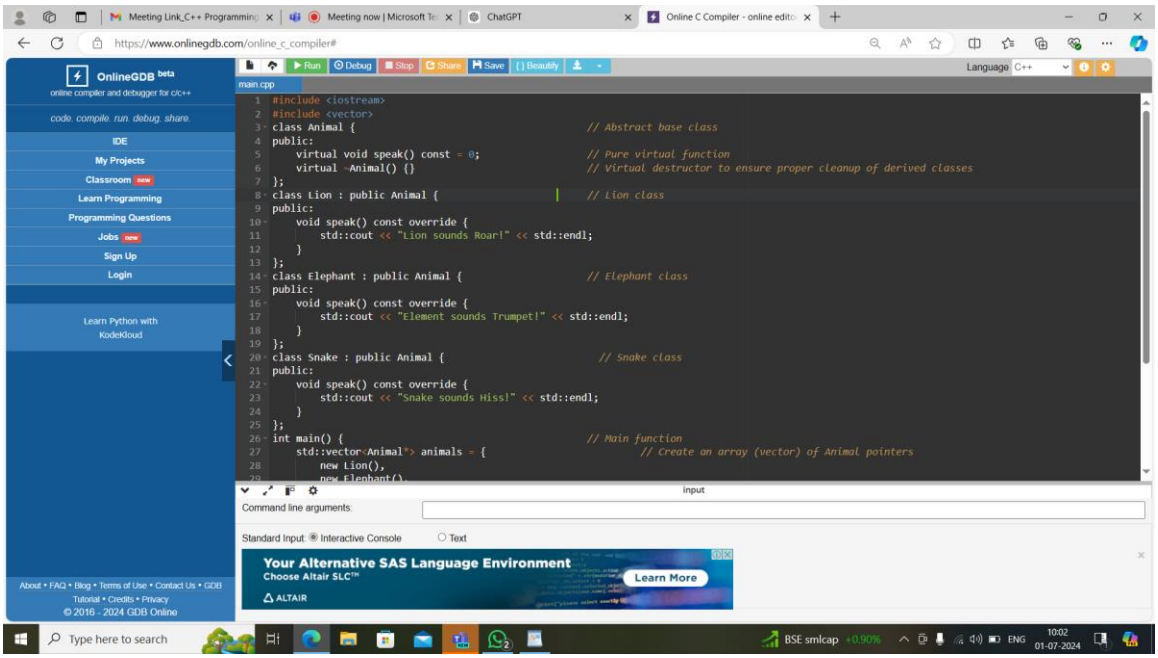
Showing how the method call is resolved at runtime based on the actual object.

```
#include <iostream>

#include <vector>

class Animal {                                          // Abstract base class

public:

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

    virtual ~Animal() {}                                // Virtual destructor to ensure proper
cleanup of derived classes

};

class Lion : public Animal {                            // Lion class

public:

    void speak() const override {
```

```
        std::cout << "Lion sounds Roar!" << std::endl;
    }
};
class Elephant : public Animal {                                    // Elephant class
public:
    void speak() const override {
        std::cout << "Elephant sounds Trumpet!" << std::endl;
    }
};
class Snake : public Animal {                                       // Snake class
public:
    void speak() const override {
        std::cout << "Snake sounds Hiss!" << std::endl;
    }
};
int main() {                                                        // Main function
    std::vector<Animal*> animals = {                                // Create an array (vector) of
Animal pointers
        new Lion(),
        new Elephant(),
        new Snake()
    };
    for (const auto& animal : animals) {                            // Iterate through the array and
make each animal speak
        animal->speak();
    }
```

```
    for (const auto& animal : animals) {                    // Clean up dynamically allocated memory
        delete animal;
    }
    return 0;
}
```

OUTPUT :-

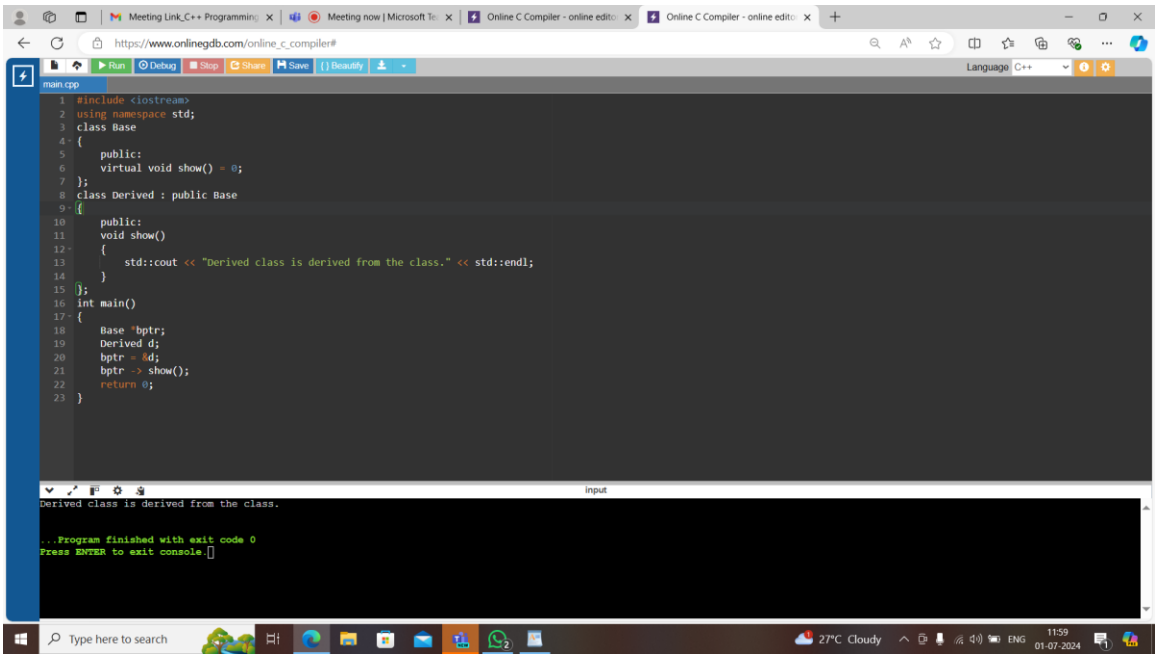


Virtual Function :-

```
#include <iostream>
using namespace std;
class Base
{
    public:
    virtual void show() = 0;
};
class Derived : public Base
```

```
{
        public:
        void show()
        {
                std::cout << "Derived class is derived from the class." << std::endl;
        }
};
int main()
{
        Base *bptr;
        Derived d;
        bptr = &d;
        bptr -> show();
        return 0;
}
```

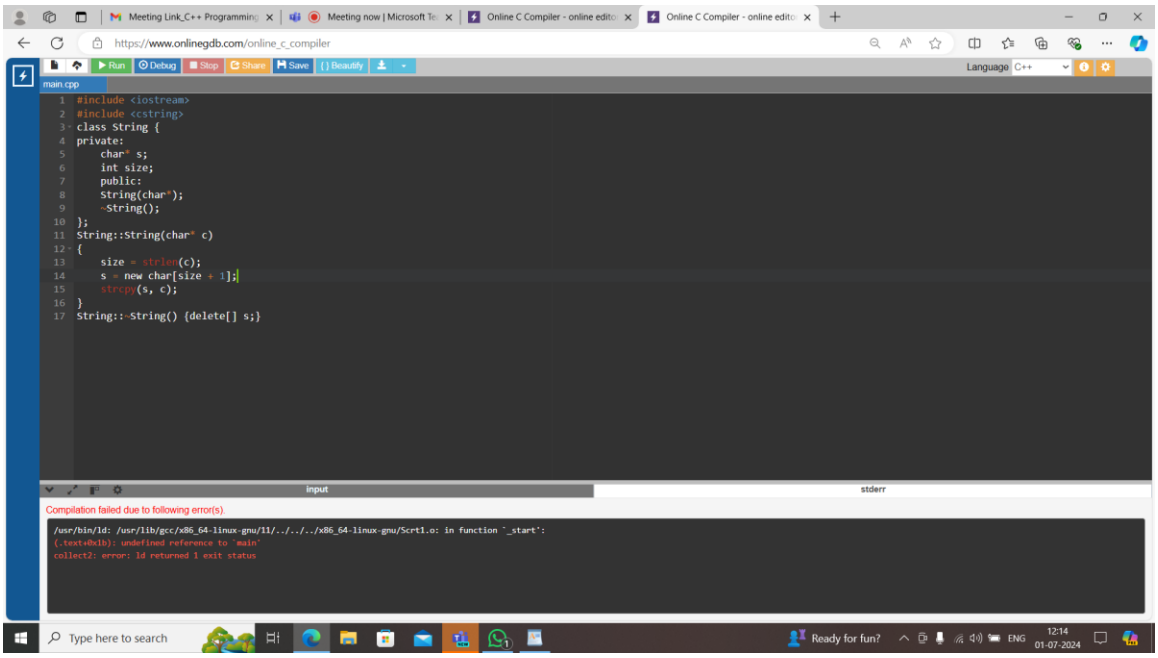
OUTPUT :-



DESTRUCTORS :-

```
#include <iostream>
#include <cstring>
class String {
private:
	char* s;
	int size;
	public:
	String(char*);
	~String();
};
String::String(char* c)
{
	size = strlen(c);
	s = new char[size + 1];
	strcpy(s, c);
}
String::~String() {delete[] s;}
```

OUTPUT :-

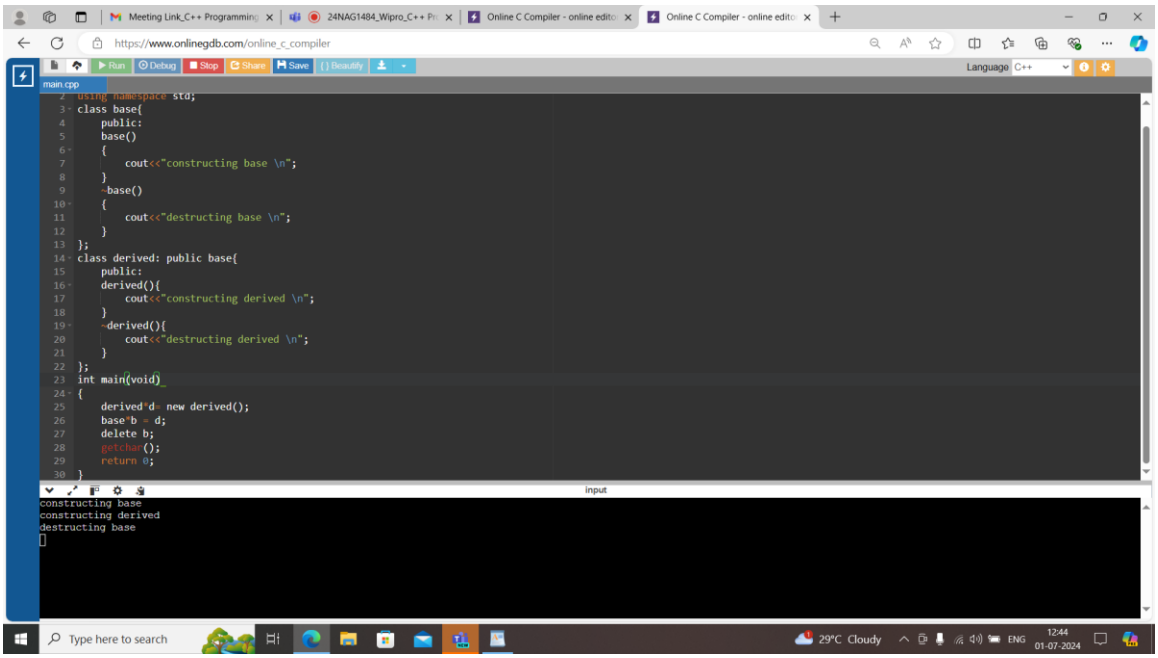


VIRTUAL DESTRUCTOR :-

```
#include<iostream>
using namespace std;
class base{
    public:
    base()
    {
        cout<<"constructing base \n";
    }
    ~base()
    {
        cout<<"destructing base \n";
    }
};
class derived: public base{
    public:
```

```
    derived(){
        cout<<"constructing derived \n";
    }
    ~derived(){
        cout<<"destructing derived \n";
    }
};
int main(void)
{
    derived*d= new derived();
    base*b = d;
    delete b;
    getchar();
    return 0;
}
```

OUTPUT :-

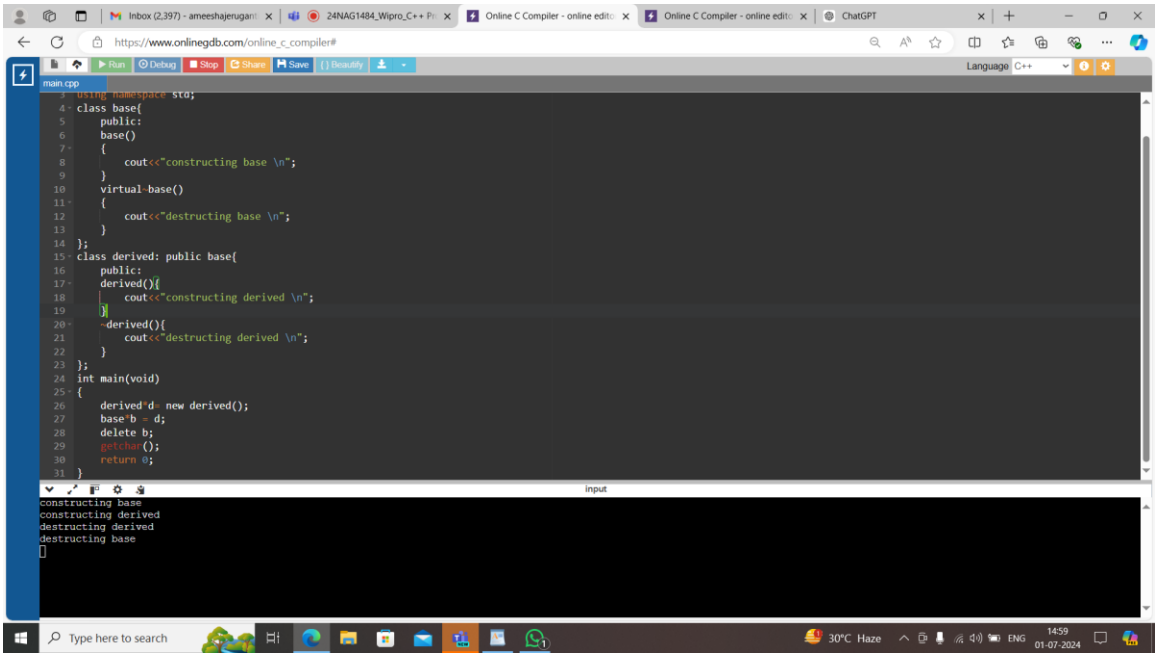


VIRTUAL CONSTRUCTOR :-

```
#include<iostream>
using namespace std;
class base{
	public:
	base()
	{
		cout<<"constructing base \n";
	}
	virtual~base()
	{
		cout<<"destructing base \n";
	}
};
class derived: public base{
	public:
	derived(){
		cout<<"constructing derived \n";
	}
	~derived(){
		cout<<"destructing derived \n";
	}
};
int main(void)
{
	derived*d= new derived();
	base*b = d;
	delete b;
```

```
    getchar();
    return 0;
}
```

OUTPUT :-

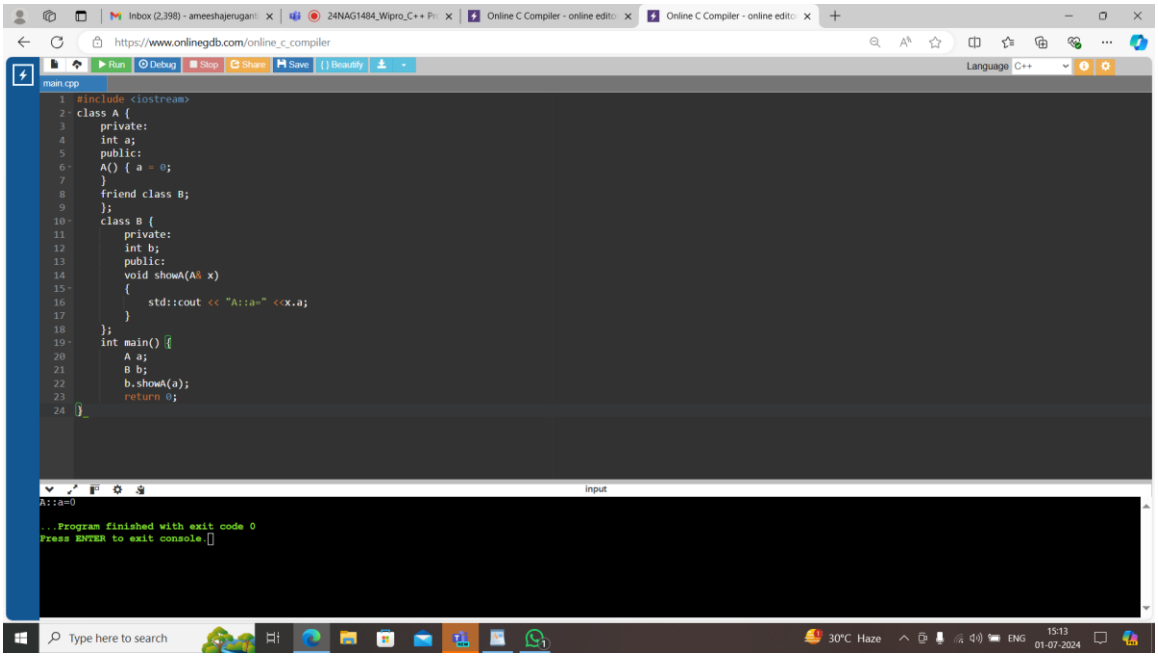


FRIEND CLASS AND FUNCTION :-

```
#include <iostream>
class A {
    private:
    int a;
    public:
    A() { a = 0;
    }
    friend class B;
    };
    class B {
        private:
```

```
        int b;
        public:
        void showA(A& x)
        {
            std::cout << "A::a=" <<x.a;
        }
    };
    int main() {
        A a;
        B b;
        b.showA(a);
        return 0;
}
```

OUTPUT :-

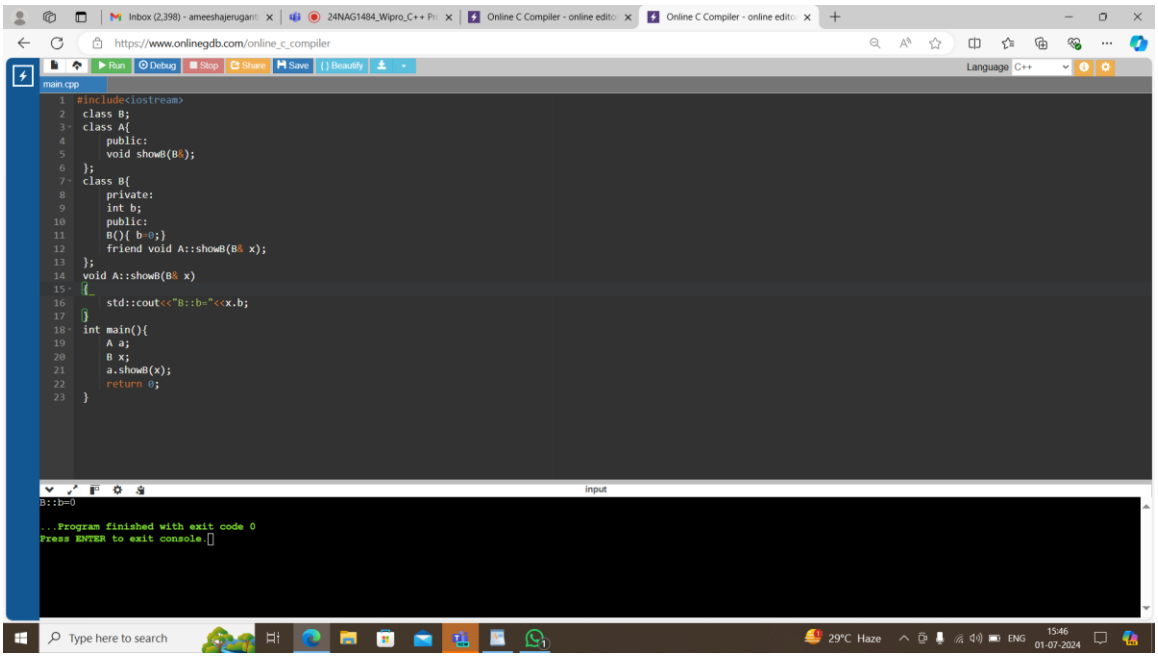


FRIEND CLASS AND FUNCTION :-

#include<iostream>

```
class B;
class A{
        public:
        void showB(B&);
};
class B{
        private:
        int b;
        public:
        B(){ b=0;}
        friend void A::showB(B& x);
};
void A::showB(B& x)
{
        std::cout<<"B::b="<<x.b;
}
int main(){
        A a;
        B x;
        a.showB(x);
        return 0;
}
```

OUTPUT :-

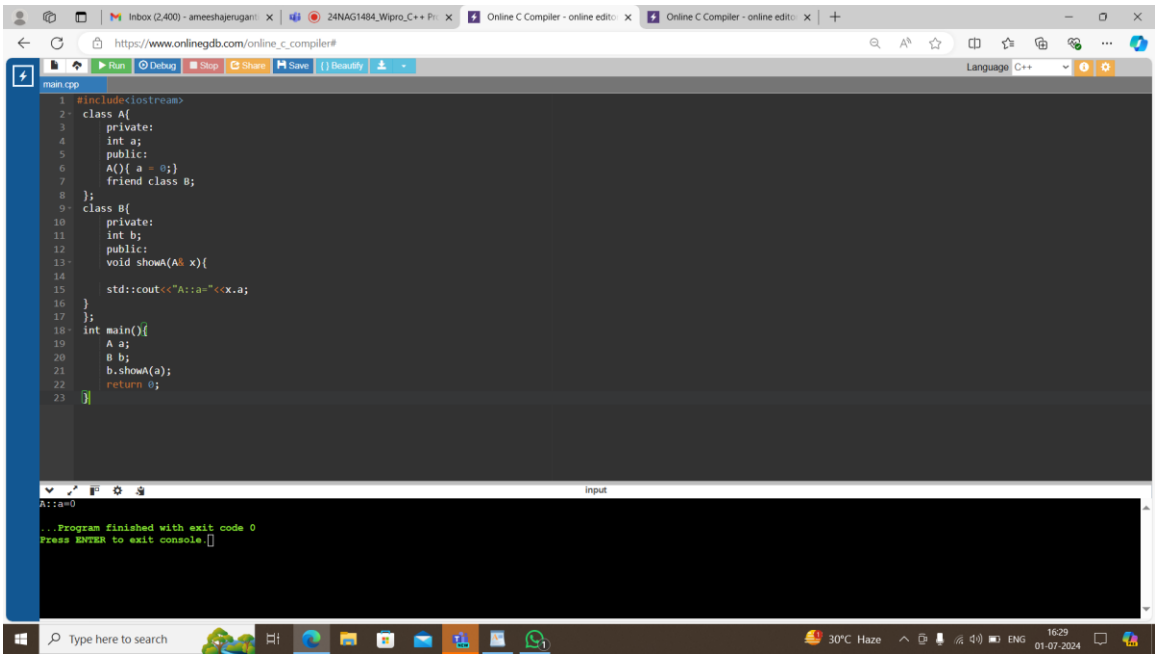


DEMONSTRATE FRIEND CLASS :-

```
#include<iostream>
 class A{
        private:
        int a;
        public:
        A(){ a = 0;}
        friend class B;
 };
 class B{
        private:
        int b;
        public:
        void showA(A& x){

        std::cout<<"A::a="<<x.a;
```

```
}
};
int main(){
    A a;
    B b;
    b.showA(a);
    return 0;
}
```

OUTPUT :-



You have a TemperatureSensor class that measures temperature in Celsius. You want a separate DisplayTemperature function to print the temperature in Fahrenheit. However, the conversion formula requires accessing the private celsius member.

Create a TemperatureSensor class with a private celsius member and a public constructor.

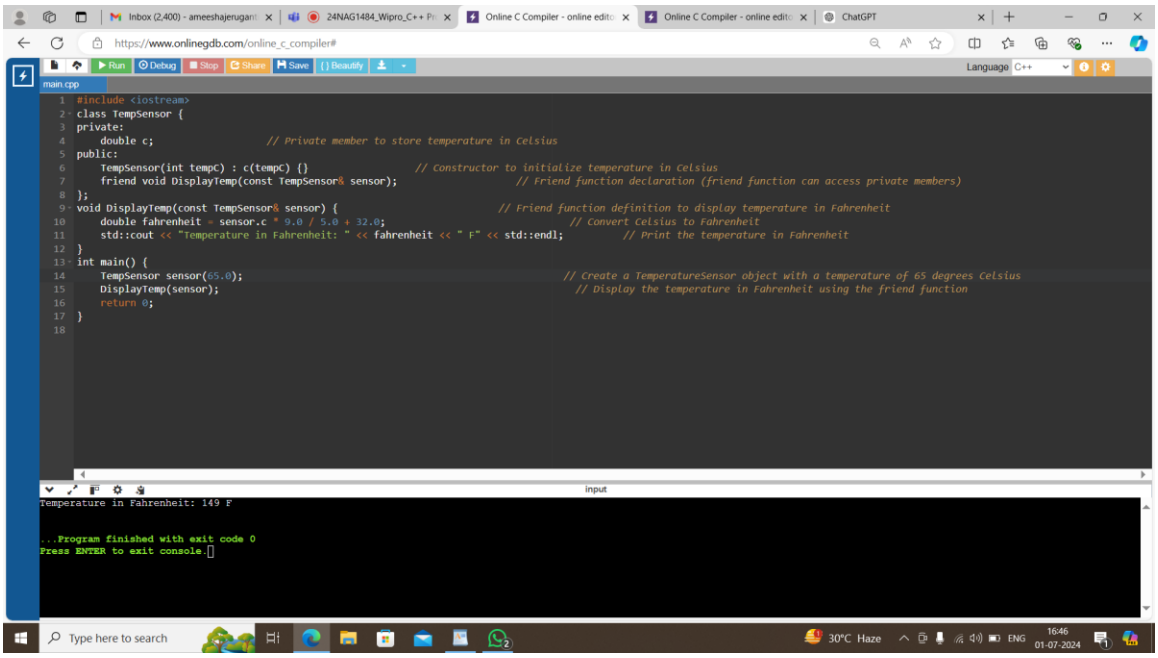
Implement a friend function DisplayTemperature that takes a TemperatureSensor object and prints the temperature in Fahrenheit (conversion formula provided).

Write a main function to demonstrate how to use the classes.

```
#include <iostream>
```

```
class TempSensor {

private:

    double c;                           // Private member to store temperature in Celsius

public:

    TempSensor(int tempC) : c(tempC) {}                         // Constructor to initialize temperature
in Celsius

    friend void DisplayTemp(const TempSensor& sensor);              // Friend function
declaration (friend function can access private members)

};

void DisplayTemp(const TempSensor& sensor) {                        // Friend function
definition to display temperature in Fahrenheit

    double fahrenheit = sensor.c * 9.0 / 5.0 + 32.0;            // Convert Celsius to Fahrenheit

    std::cout << "Temperature in Fahrenheit: " << fahrenheit << " F" << std::endl;              //
Print the temperature in Fahrenheit

}

int main() {

    TempSensor sensor(65.0);                        // Create a TemperatureSensor object with a
temperature                                     of 65 degrees Celsius

    DisplayTemp(sensor);                    // Display the temperature in Fahrenheit using the
friend    function

    return 0;

}
```

OUTPUT :-



Friend Class for Stream Insertion:

Scenario: You have a Point class with private members for x and y coordinates. You want to define a way to easily print Point objects to output streams like cout.

Create a Point class with private x and y members and a public constructor.

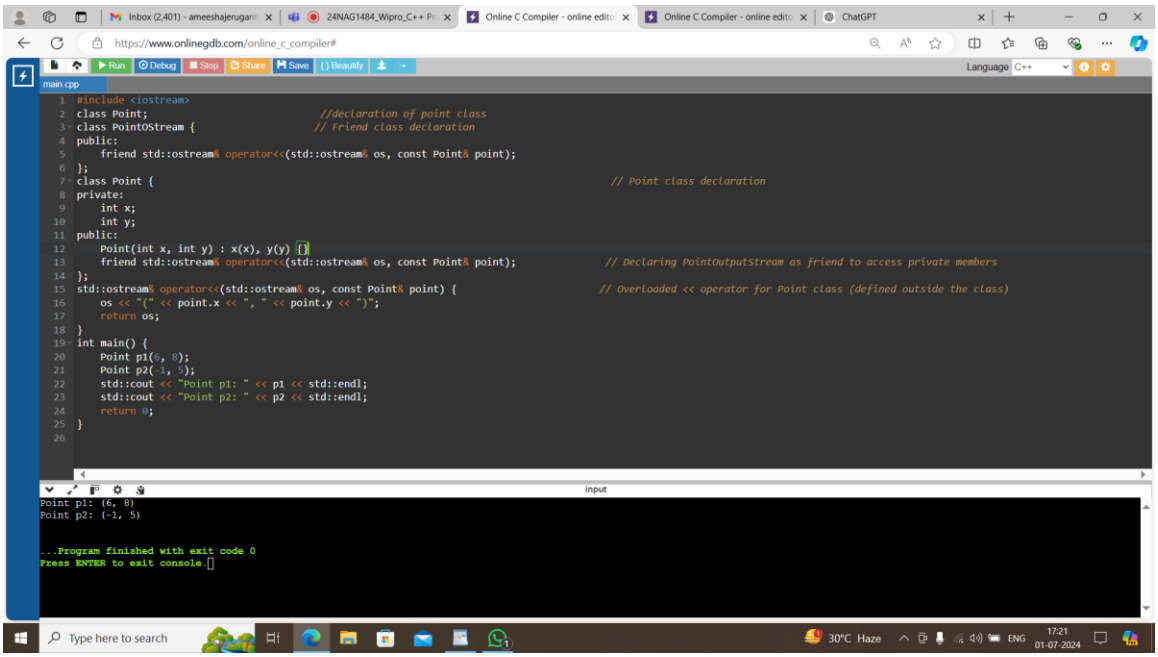
Design a friend class PointOutputStream that has an overloaded << operator to format and insert Point objects into output streams.

In main, demonstrate creating Point objects and printing them using cout.

```
#include <iostream>

class Point;                                    //declaration of point class

class PointOStream {                            // Friend class declaration

public:

    friend std::ostream& operator<<(std::ostream& os, const Point& point);

};

class Point {
// Point class declaration

private:

    int x;
```

```
    int y;
public:
    Point(int x, int y) : x(x), y(y) {}
    friend std::ostream& operator<<(std::ostream& os, const Point& point);                    //
Declaring PointOutputStream as friend to access private members
};
std::ostream& operator<<(std::ostream& os, const Point& point) {                    //
Overloaded << operator for Point class (defined outside the class)
    os << "(" << point.x << ", " << point.y << ")";
    return os;
}
int main() {
    Point p1(6, 8);
    Point p2(-1, 5);
    std::cout << "Point p1: " << p1 << std::endl;
    std::cout << "Point p2: " << p2 << std::endl;
    return 0;
}
```

OUTPUT :-


Online C Compiler - online editor
https://www.onlinegdb.com/online_c_compiler#
Run
Debug
Stop
Share
Save
{} Beautify
Language C++
main.cpp
#include <iostream>
class Point; //declaration of point class
class PointOStream { // Friend class declaration
public:
friend std::ostream& operator<<(std::ostream& os, const Point& point);
};
class Point { // Point class declaration
private:
int x;
int y;
public:
Point(int x, int y) : x(x), y(y) {}
friend std::ostream& operator<<(std::ostream& os, const Point& point); // Declaring PointOutputStream as friend to access private members
};
std::ostream& operator<<(std::ostream& os, const Point& point) { // Overloaded << operator for Point class (defined outside the class)
os << "(" << point.x << ", " << point.y << ")";
return os;
}
int main() {
Point p1(6, 8);
Point p2(-1, 5);
std::cout << "Point p1: " << p1 << std::endl;
std::cout << "Point p2: " << p2 << std::endl;
return 0;
}
input
Point p1: (6, 8)
Point p2: (-1, 5)
...Program finished with exit code 0
Press ENTER to exit console.
Type here to search
30°C Haze
ENG
17:21
01-07-2024