Object-Oriented Programming

Using C++

CIC-257

Submitted By: Satyam Kumar Jha

Faculty : Ms Chesta

Group:3C11 .

Roll No: 01996402723 .



Maharaja Agrasen Institute of Technology, PSP area, Sector – 22, Rohini, New Delhi – 110085 (Affiliated to Guru Gobind Singh Indraprastha University,New Delhi)

**Program 1**

**Aim :**

Write a program to perform matrix multiplication using OOPs Concepts.

**Theory :**

 **Class Structure**: The class groups matrix operations (like creating, displaying, and multiplying matrices) together, making the code organized and easy to reuse.

 **Matrix Multiplication**: The multiplication function takes two matrix objects and checks if their sizes are correct for multiplication before doing any calculations.

 **Using Loops**: The function uses loops to go through the rows and columns of the matrices, multiplying and adding the numbers to get the result.

**Source Code :**

#include<iostream>

using namespace std;

class *Matrix* {

private:

    int rows, cols;

    int mat[100][100];

public:

*Matrix*(int *r*, int *c*) {

        rows = *r*;

        cols = *c*;

    }

    void *inputMatrix*() {

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

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

                cout *<<* "Enter the element at position " *<<* i *<<* ", " *<<* j *<<* ": ";

                cin *>>* mat[i][j];

            }

        }

    }

    void *displayMatrix*() const {

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

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

                cout *<<* mat[i][j] *<<* " ";

            }

            cout *<<* *endl*;

        }

    }

*Matrix* *multiply*(const *Matrix* &*other*) {

        if (cols != *other*.cols) {

            cout *<<* "Matrix multiplication not possible!" *<<* *endl*;

*exit*(0);

        }

*Matrix* *result*(rows, *other*.cols);

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

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

                result.mat[i][j] = 0;

            }

        }

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

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

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

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

                }

            }

        }

        return result;

    }

};

int *main*() {

    int n1, m1, n2, m2;

    cout *<<* "Enter the number of rows and columns for the first matrix: ";

    cin *>>* n1 *>>* m1;

    cout *<<* "Enter the number of rows and columns for the second matrix: ";

    cin *>>* n2 *>>* m2;

    if (m1 != n2) {

        cout *<<* "Matrix multiplication not possible! Number of columns in the first matrix must equal the number of rows in the second matrix.\n";

        return 1;

    }

*Matrix* *mat1*(n1, m1);

*Matrix* *mat2*(n2, m2);

    cout *<<* "Enter elements of the first matrix:\n";

    mat1.*inputMatrix*();

    cout *<<* "Enter elements of the second matrix:\n";

    mat2.*inputMatrix*();

*Matrix* result = mat1.*multiply*(mat2);

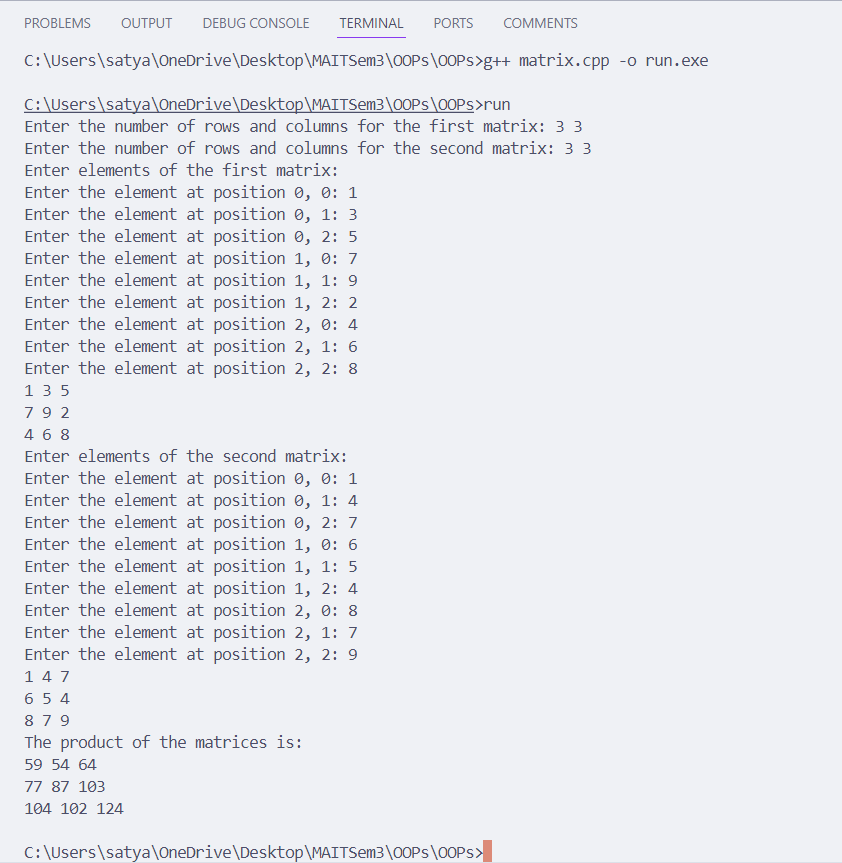
    cout *<<* "The product of the matrices is:\n";

    result.*displayMatrix*();

    return 0;

}

**Output**



**Program 2**

**Aim :**

Create a class Student, which have data members as name, branch, roll no., age, sex, five subjects. Display the name and student & his percentage who has more than 70%. Use array of objects.

**Theory :**

 **Class Structure**: The Student class encapsulates essential data members, including name, branch, roll number, age, sex, and an array to store marks across five subjects. This design follows the principles of encapsulation and data hiding.

 **Functionality**: The class includes public member functions for:

* Inputting student details,
* Calculating the percentage of marks obtained,
* Displaying details of students who achieve more than 70% in their subjects.

 **Efficient Data Management**: By utilizing an array of Student objects, the program effectively manages multiple student records, demonstrating modularity and the ability to handle complex data structures seamlessly.

 **OOP Principles**: The implementation showcases core Object-Oriented Programming principles, including encapsulation, modularity, and code reusability, thereby enhancing maintainability and scalability of the code.

**Code :**

#include<iostream>

using namespace std;

class *Student* {

private:

*string* name;

*string* branch;

    int roll;

    int age;

    char sex;

    float marks[5];

public:

    void *inputStudent*() {

        cout *<<* "Enter the name of the student: ";

        cin *>>* name;

        cout *<<* "Enter the branch of the student: ";

        cin *>>* branch;

        cout *<<* "Enter the roll number of the student: ";

        cin *>>* roll;

        cout *<<* "Enter the age of the student: ";

        cin *>>* age;

        cout*<<* "Enter the sex(M/F) of the student: ";

        cin*>>*sex;

        for (int i = 0; i < 5; i++) {

            cout *<<* "Enter the marks of subject " *<<* i + 1 *<<* ": ";

            cin *>>* marks[i];

        }

    }

    float *percentage*() const {

        float total = 0;

        for (int i = 0; i < 5; i++) {

            total += marks[i];

        }

        return total / 5;

    }

    void *displayStudent*() const {

        cout *<<* "Name: " *<<* name *<<* *endl*;

        cout *<<* "Branch: " *<<* branch *<<* *endl*;

        cout *<<* "Roll number: " *<<* roll *<<* *endl*;

        cout *<<* "Age: " *<<* age *<<* *endl*

        cout << "Sex: " << sex << endl;

        cout *<<* "Percentage: " *<<* *percentage*() *<<* *endl*;

    }

};

int *main*() {

    int n;

    cout *<<* "Enter the number of students (up to 100): ";

    cin *>>* n;

*Student* students[n];

    for (int i = 0; i < n; i++) {

        cout *<<* "Enter the details of student " *<<* i + 1 *<<* *endl*;

        students[i].*inputStudent*();

    }

    cout *<<* "Following students have percentage greater than 70%:" *<<* *endl*;

    for (int i = 0; i < n; i++) {

        if (students[i].*percentage*() > 70) {

            students[i].*displayStudent*();

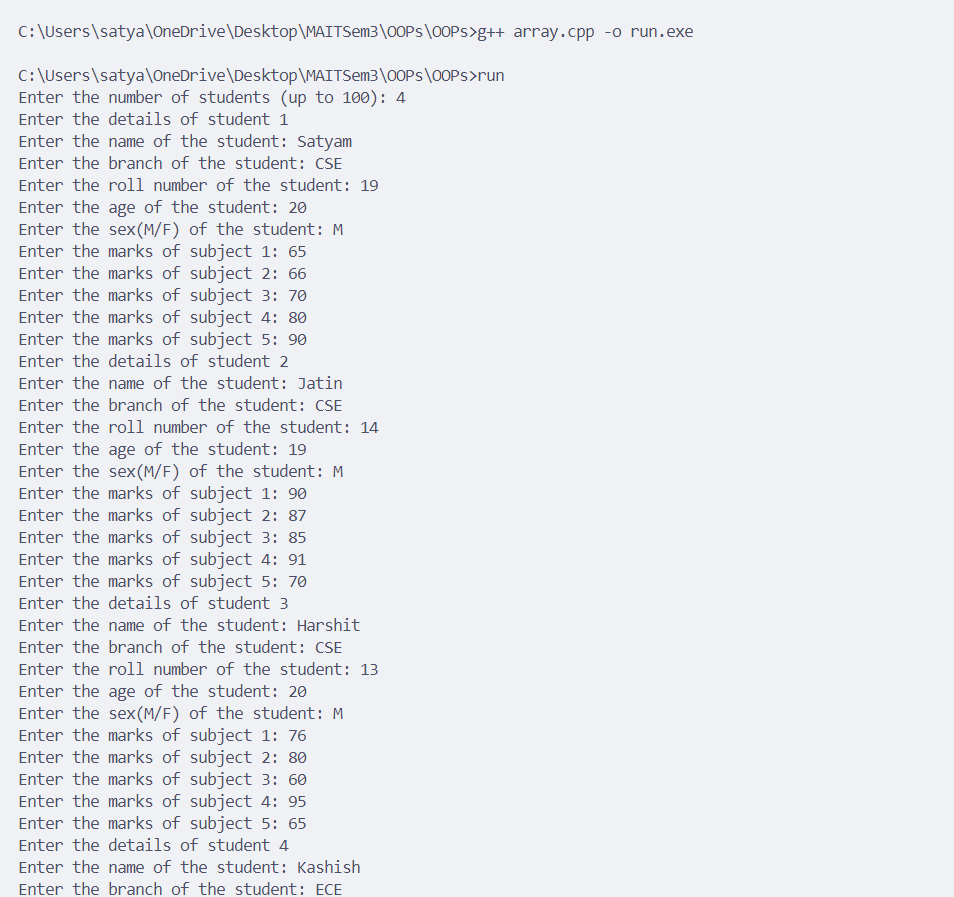
        }

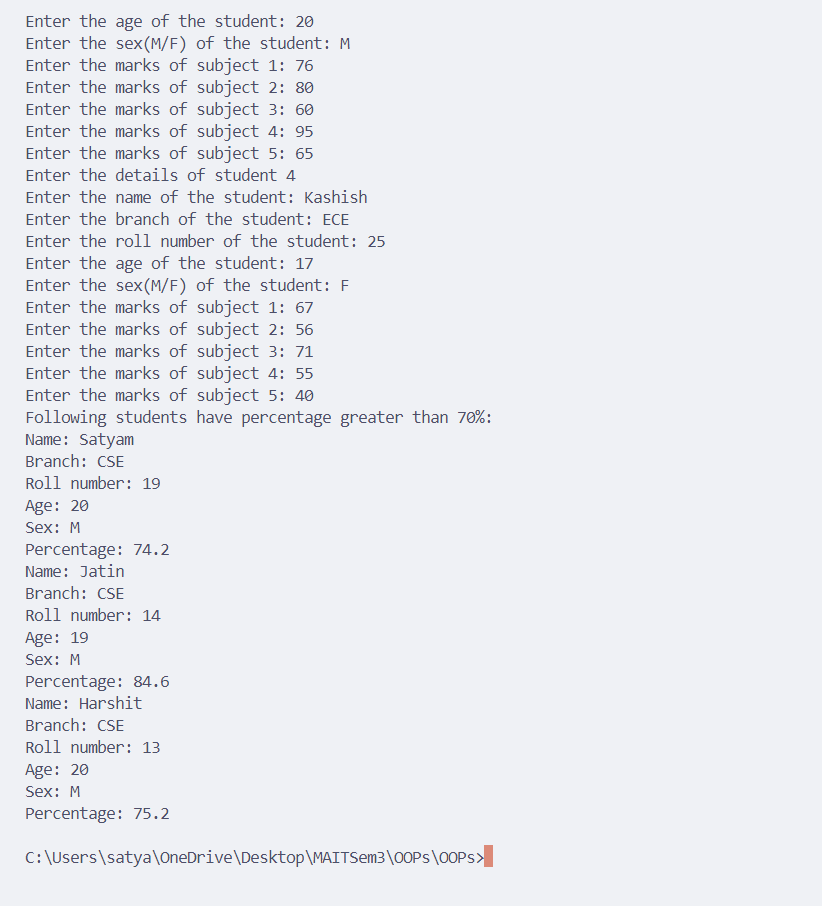
    }

    return 0;

}

**Output**

****

****

**Program 3**

**Aim :**

Write a program for Function Overloading to write functions for calculating the area of a triangle, circle, and rectangle.

**Theory :**

 **Definition and Purpose**: Function overloading in C++ allows multiple functions to share the same name while differing in parameter lists (type, number, or both). This feature enhances code flexibility, readability, and maintainability by enabling a unified function name for various operations.

 **Intuitive Function Selection**: The correct overloaded version of a function is invoked based on the provided arguments, making the code more intuitive. This reduces the likelihood of errors associated with misnamed functions, improving overall code quality.

 **Versatile Area Calculations**: For instance, a single area function can be overloaded to handle different shapes:

* **Triangle**: Accepts base and height as parameters, calculating area using the formula Area=12×base×height\text{Area} = \frac{1}{2} \times \text{base} \times \text{height}Area=21​×base×height.
* **Circle**: Requires just the radius, applying the formula Area=π×radius2\text{Area} = \pi \times \text{radius}^2Area=π×radius2.
* **Rectangle**: Takes length and width as inputs, computing area with Area=length×width\text{Area} = \text{length} \times \text{width}Area=length×width.

 **Enhanced Code Organization**: By employing function overloading, developers can write cleaner, more organized code that effectively manages multiple operations, thereby facilitating easier debugging and future code extensions.

**Code :**

#include<iostream>

using namespace std;

double *area*(double *r*){

    return 3.14\**r*\**r*;

}

double *area*(double *l*, double *w*){

    return *l*\**w*;

}

double *area*(int *b*, int *h*){

    return 0.5\**b*\**h*;

}

int *main*(){

    int r, l, b, h, w;

    cout *<<* "Enter the radius of the circle: ";

    cin *>>* r;

    cout *<<* "The area of the circle is: " *<<* *area*(r) *<<* *endl*;

    cout *<<* "Enter the length and breadth of the rectangle: ";

    cin *>>* l *>>* w;

    cout *<<* "The area of the rectangle is: " *<<* *area*(l, w) *<<* *endl*;

    cout *<<* "Enter the base and height of the triangle: ";

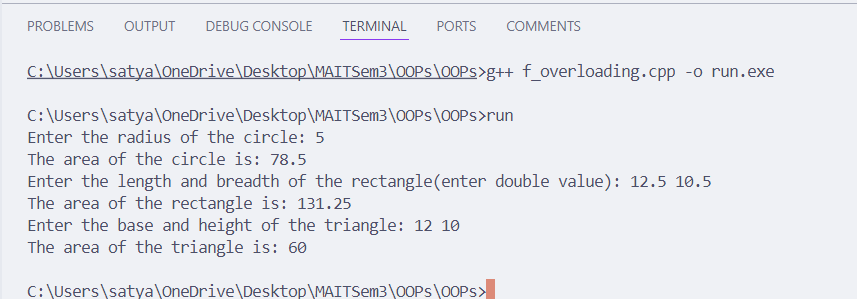
    cin *>>* b *>>* h;

    cout *<<* "The area of the triangle is: " *<<* *area*(b, h) *<<* *endl*;

    return 0;

}

**Output**

****

**Program 4**

**Aim :**

Write a program to create a class Time with members hours, minutes, and seconds, and implement functionality to take input, add two Time objects, pass objects to a function, and display the result.

**Theory :**

 **Class Definition**: The program defines a class named Time, designed to encapsulate time representation with data members for hours, minutes, and seconds. This structure emphasizes the principles of encapsulation and data abstraction.

 **Input Method**: The Time class includes a method to input the time components (hours, minutes, and seconds), allowing users to easily create Time objects with specific values.

 **Time Addition**: The class features a method for adding two Time objects. This method performs arithmetic operations while effectively managing overflow conditions, such as converting excess seconds into minutes and excess minutes into hours, ensuring accurate time calculations.

 **Result Display**: A dedicated method is provided to display the resulting time after addition, enhancing user experience by presenting data in a clear and readable format. This approach simplifies managing and operating on time data within an object-oriented structure, promoting code organization and reusability.

**Code :**

#include<iostream>

using namespace std;

class *Time* {

private:

    int hrs, min, sec;

public:

    void *inputTime*() {

        cout *<<* "Enter the time in hours, minutes and seconds: ";

        cin *>>* hrs *>>* min *>>* sec;

    }

    void *displayTime*() const {

        cout *<<* hrs *<<* " hours, " *<<* min *<<* " minutes and " *<<* sec *<<* " seconds" *<<* *endl*;

    }

*Time* *addTime*(const *Time* &*t*) {

*Time* at;

        at.sec = sec + *t*.sec;

        at.min = at.sec / 60;

        at.sec %= 60;

        at.min += min + *t*.min;

        at.hrs = at.min / 60;

        at.min %= 60;

        at.hrs += hrs + *t*.hrs;

        return at;

    }

};

int *main*(){

*Time* t1, t2, ft;

    t1.*inputTime*();

    t2.*inputTime*();

    ft *=* t1.*addTime*(t2);

    cout *<<* "The first time is: ";

    t1.*displayTime*();

    cout *<<* "The second time is: ";

    t2.*displayTime*();

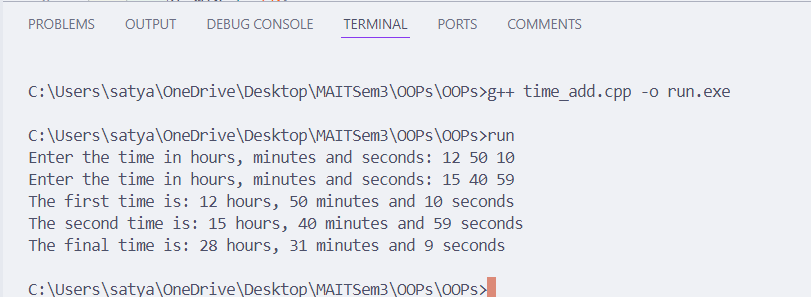
    cout *<<* "The final time is: ";

    ft.*displayTime*();

    return 0;

}

**Output**

****

**Program 5**

**Aim :**

Write a program to perform addition of two complex numbers using constructor overloading. The first constructor which takes no argument is used to create objects which are not initialised, second which takes one argument is used to initialise real and imag parts to equal values and third which takes two arguments is used to initialise real and imag to two different values.

**Theory :**

 **Concept of Constructor Overloading**: Constructor overloading is a key feature in object-oriented programming that allows a class to have multiple constructors, each with different parameter lists. This flexibility facilitates varied object initialization based on the specific needs of the application.

 **Complex Class Definition**: The program defines a Complex class to represent complex numbers, which consist of a real part and an imaginary part. This encapsulation adheres to the principles of data abstraction and encapsulation in object-oriented design.

 **Multiple Constructors**:

* **Default Constructor**: Initializes complex numbers to zero, enabling the creation of Complex objects without providing initial values, which simplifies instantiation.
* **Single Argument Constructor**: Initializes both the real and imaginary parts to the same value, which is particularly useful for creating complex numbers where both parts are intended to be equal (e.g., a+aia + aia+ai).
* **Two Argument Constructor**: Allows for the independent initialization of the real and imaginary parts with different values, offering maximum flexibility in object creation.

 **Addition Method**: The Complex class includes a method to add two complex numbers. This method performs addition by separately summing the real parts and the imaginary parts, effectively demonstrating how to manipulate objects and perform operations on them.

 **Enhanced Code Organization**: By implementing constructor overloading and encapsulating functionality within the Complex class, the program promotes cleaner, more organized code. This design approach enhances readability, maintainability, and reusability of the code while effectively managing complex number operations.

**Code :**

#include <iostream>

using namespace std;

class Complex {

private:

double real;

double imag;

public:

Complex() : real(0), imag(0) {}

Complex(double value) : real(value), imag(value) {}

Complex(double r, double i) : real(r), imag(i) {}

Complex add(const Complex& c) const {

return Complex(real + c.real, imag + c.imag);

}

void display() const {

cout << real << " + " << imag << "i" << endl;

}

};

int main() {

Complex c1, c2(5), c3(3, 4), result;

cout << "First complex number: ";

c1.display();

cout << "Second complex number: ";

c2.display();

cout << "Third complex number: ";

c3.display();

result = c2.add(c3);

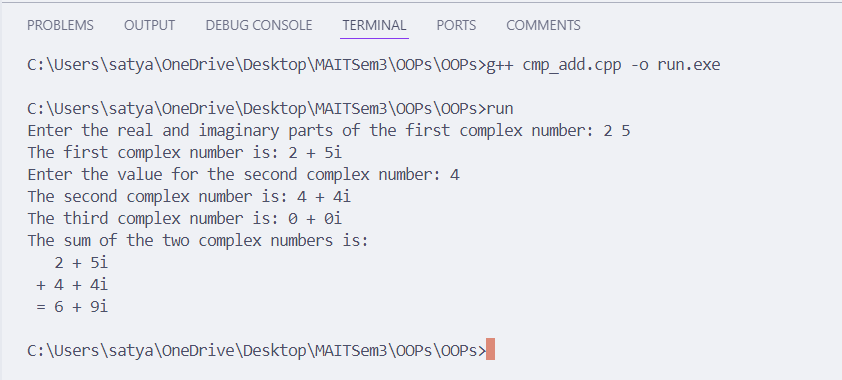
cout << "Result of addition: ";

result.display();

return 0;

}

**Output**

****

**Program 6**

**Aim :**

Write a function power to raise a number m to power n. The function takes a double value for m and int value for n. Use default value for n to make the function calculate squares when this argument is omitted.

**Theory :**

 **Function Definition**: The program features a function named power() designed to raise a number mmm to the power of n. The function accepts a double value for mmm and an integer value for n, which has a default value of 2. This design allows for versatile use of the function in various scenarios.

 **Default Parameter**: By setting a default value of 2 for the parameter n, the function can effortlessly compute the square of mmm when the second argument is omitted. This enhances usability and simplifies calls to the function.

 **Calculation Logic**: The function employs a loop to multiply the base mmm by itself n times, effectively computing n^m. This iterative approach ensures accurate power calculations, accommodating both positive and negative values of n.

 **Demonstration of Functionality**: In the main function, the program showcases the power() function by calling it with two arguments to compute 3^2 and with one argument to calculate the square of 4. This demonstrates the flexibility of the function, including the effective use of default parameters in practical scenarios.

**Code :**

#include <iostream>

using namespace std;

class *PowerCalc* {

public:

    double *Power*(double *m*) {

        int n = 2;

        double pow = 1.0;

        for (int i = 0; i < n; i++) {

            pow = pow \* *m*;

        }

        return pow;

    }

    double *Power*(double *m*, int *n*) {

        double pow = 1.0;

        for (int i = 0; i < *n*; i++) {

            pow = pow \* *m*;

        }

        return pow;

    }

};

int *main*() {

    double m;

    int n,c;

    double result;

    cout *<<* "Enter a double value for m: ";

    cin *>>* m;

    cout*<<*"Do you want to enter n(0 for No and 1 for Yes):";

    cin*>>*c;

    if(c==1){

        cout *<<* "Enter an integer value for n: ";

        cin *>>* n;

*PowerCalc* p;

        result = p.*Power*(m, n);

    }else{

*PowerCalc* p;

        result = p.*Power*(m);

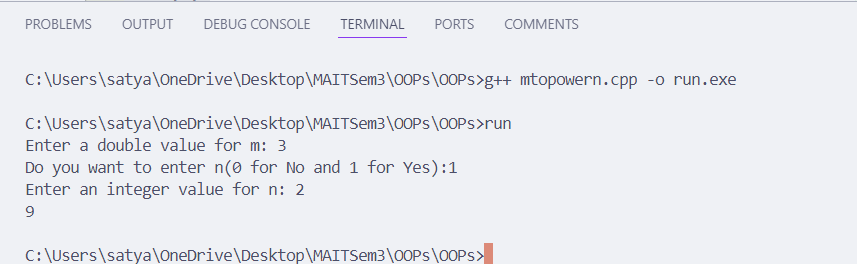
    }

    cout *<<* "The result is: " *<<* result *<<* *endl*;

    return 0;

}

**Output**

****

**Program 7**

**Aim :**

Write a program to enter any number and find its factorial using a constructor.

**Theory :**

 **Class Design for Factorial Calculation**: The program defines a class specifically designed to compute the factorial of a given number. This encapsulation of functionality promotes a clear structure and adheres to object-oriented programming principles.

 **Constructor Functionality**: The constructor of the class accepts an integer as an argument, representing the number for which the factorial is to be calculated. Upon initialization, the constructor automatically computes the factorial by multiplying the integer by all positive integers less than itself, ensuring efficient calculation during object instantiation.

 **Factorial Computation Logic**: The factorial is calculated using an iterative or recursive approach within the constructor, which encapsulates the logic needed to perform the computation. This approach not only promotes code reuse but also enhances readability.

 **Output Method**: The class includes a method to display the computed factorial, providing a clear and user-friendly output. This method ensures that users can easily access and understand the result of their factorial calculation, enhancing the overall usability of the program.

**Code :**

#include<iostream>

using namespace std;

class *Factorial* {

private:

    int n;

    int fact;

public:

*Factorial*(int *num*){

        n = *num*;

        fact = 1;

    }

    void *calculate*(){

        for (int i = 1; i <= n; i++){

            fact \*= i;

        }

    }

    void *display*(){

        cout *<<* "The factorial of " *<<* n *<<* " is: " *<<* fact *<<* *endl*;

    }

};

int *main*(){

    int num;

    cout *<<* "Enter the number to find the factorial: ";

    cin *>>* num;

*Factorial* *f*(num);

    f.*calculate*();

    f.*display*();

    return 0;

}

**Output**

****

**Program 8**

**Aim :**

Write a program to generate a Fibonacci series using Copy Constructor.

**Theory :**

 **Concept of Copy Constructor**: In object-oriented programming, a copy constructor is a specialized constructor used to create a new object as a duplicate of an existing one. This ensures that the copying process is handled correctly, preserving the integrity of the object's data.

 **Fibonacci Series Definition**: The Fibonacci series is a mathematical sequence where each number is the sum of the two preceding ones, typically starting with 0 and 1. This sequence is widely used in various applications, making it a suitable focus for object-oriented design.

 **Fibonacci Class Implementation**: The program defines a class named Fibonacci that generates and manages the Fibonacci series. The primary constructor initializes the series up to a specified number of terms, allowing users to easily specify how many Fibonacci numbers they wish to generate.

 **Utilization of the Copy Constructor**: The copy constructor is employed to create a new Fibonacci object as a copy of an existing one. This ensures that the series generated by the original object is accurately duplicated in the new object, maintaining data consistency and facilitating easy manipulation of Fibonacci series instances.

**Code :**

#include<iostream>

using namespace std;

class Fibonacci {

long int a, b;

public:

Fibonacci() {

a = -1;

b = 1;

}

void fibSeries(int n) {

int next;

cout << "\nResultant Fibonacci series\n\n";

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

next = a + b;

cout << next << endl;

a = b;

b = next;

}

}

};

int main() {

Fibonacci f;

int n;

cout << "\nFibonacci series\n";

cout << "\nEnter the range: ";

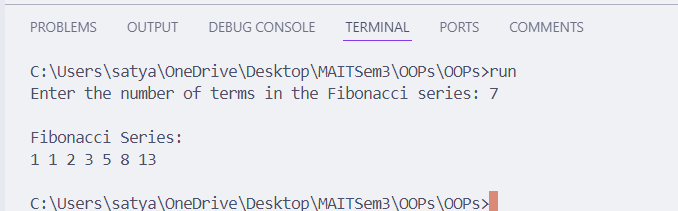
cin >> n;

f.fibSeries(n);

return 0;

}

**Output**

****

**Program 9**

**Aim :**

Create a class which keeps track of the number of instances. Use static data members, constructors, and destructors to maintain updated information about active objects.

**Theory :**

 **Static Data Members**: The program illustrates the use of static data members in C++ to maintain class-level information. The smartClass class contains a static data member called active, which is used to count the number of active instances of the class.

 **Constructor and Destructor Functionality**:

* **Constructor**: Each time a new smartClass object is created, the constructor increments the static active counter. This ensures that the count reflects the number of currently active objects in real-time.
* **Destructor**: When an object is destroyed, the destructor decrements the active counter, allowing for accurate tracking of active instances throughout the program’s execution.

 **Dynamic Count Display**: The class includes a member function called showActive, which displays the current number of active objects. This functionality provides a clear insight into how the number of active instances changes as objects are created and destroyed.

 **Demonstration in Main Function**: In the main function, multiple smartClass objects are created and destroyed within nested scopes. The program displays the number of active objects at various points, effectively illustrating how the active counter dynamically updates, showcasing the management of object lifecycle and the impact of static members on class behavior.

 **Practical Application**: This approach demonstrates the utility of static data members for maintaining shared state across all instances of a class, reinforcing the understanding of object-oriented concepts such as constructors, destructors, and static variables in C++.

**Code :**

#include<iostream>

using namespace std;

class *Student* {

private:

*string* name;

    int roll;

    static int count;

public:

*Student*() {

        count++;

    }

*~Student*() {

        count--;

    }

    void *inputStudent*() {

        cout *<<* "Enter the name of the student: ";

        cin *>>* name;

        cout *<<* "Enter the roll number of the student: ";

        cin *>>* roll;

    }

    void *displayStudent*() const {

        cout *<<* "Name: " *<<* name *<<* *endl*;

        cout *<<* "Roll number: " *<<* roll *<<* *endl*;

    }

    static void *displayCount*() {

        cout *<<* "The total number of students is: " *<<* count *<<* *endl*;

    }

};

int *Student*::count = 0;

int *main*() {

    int n;

    cout *<<* "Enter the number of students: ";

    cin *>>* n;

*Student*\* S = **new** *Student*[n];

    for (int i = 0; i < n; i++) {

        cout *<<* "Enter the details of student " *<<* i + 1 *<<* *endl*;

        S[i].*inputStudent*();

    }

    cout *<<* "The details of the students are: " *<<* *endl*;

    for (int i = 0; i < n; i++) {

        S[i].*displayStudent*();

    }

*Student*::*displayCount*();

    {

        cout*<<*"Entering a new block\n";

*Student* s1;

*Student*::*displayCount*();

    }

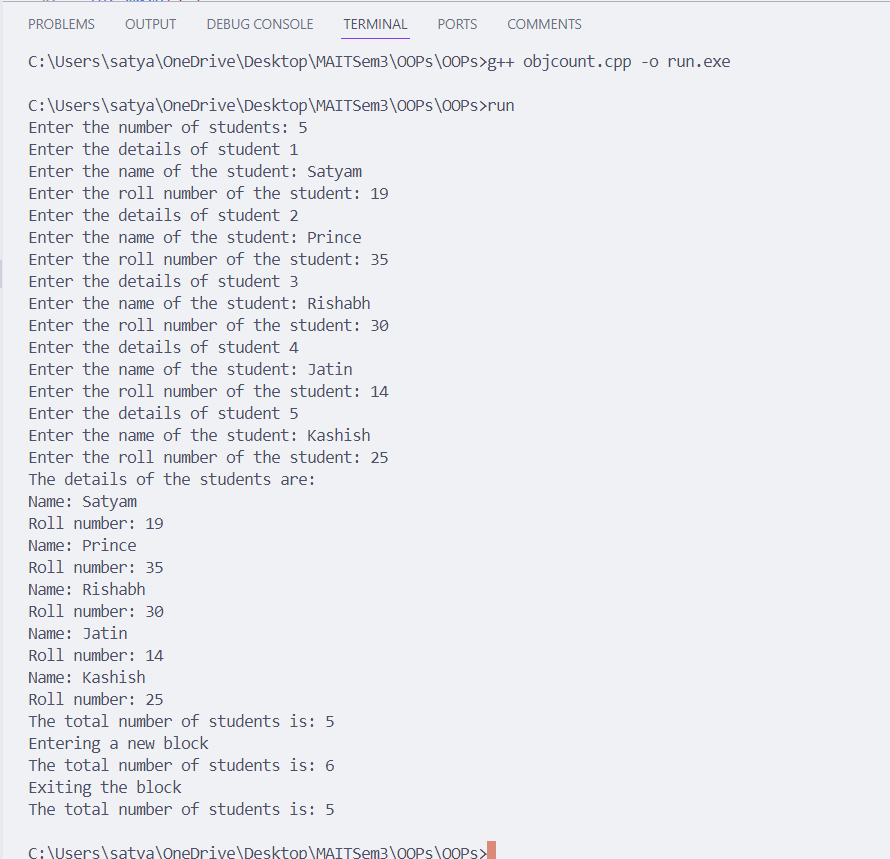
    cout*<<*"Exiting the block\n";

*Student*::*displayCount*();

    return 0;

}

**Output**

****

**Program 10**

**Aim :**

Write a program to find the greatest of two given numbers in two different classes using a friend function.

**Theory :**

 **Friend Function Concept**: The program demonstrates the use of a **friend function** in C++ to access and compare private data members of two different classes, allowing controlled access without breaking encapsulation.

 **Class Design**: **ClassA** and **ClassB** each contain a private data member, with the friend function findGreatest declared in both. This function compares the numbers stored in the objects of these classes to determine the greatest.

 **Functionality in Main**: In the main function, objects of **ClassA** and **ClassB** are created, and the friend function is called to find and display the greater number, showcasing how friend functions enable cross-class comparisons.

**Code :**

#include <iostream>

using namespace std;

class *Num2*;

class *Num1* {

private:

    double n;

public:

*Num1*(double *num1*) : *n*(*num1*) {}

    friend double *Greatest*(*Num1* &*n*, *Num2* &*m*);

};

class *Num2* {

private:

    double n;

public:

*Num2*(double *num2*) : *n*(*num2*) {}

    friend double *Greatest*(*Num1* &*n*, *Num2* &*m*);

};

double *Greatest*(*Num1* &*n*, *Num2* &*m*) {

    double greatest;

    if (*n*.n >= *m*.n) {

        greatest = *n*.n;

    } else {

        greatest = *m*.n;

    }

    return greatest;

}

int *main*() {

    double num1, num2;

    cout *<<* "Enter the first number: ";

    cin *>>* num1;

    cout *<<* "Enter the second number: ";

    cin *>>* num2;

*Num1* *n1*(num1);

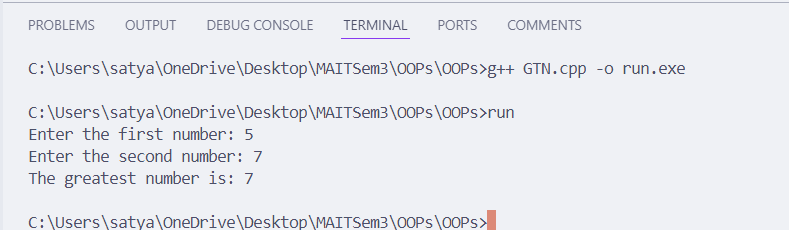
*Num2* *n2*(num2);

    cout *<<* "The greatest number is: " *<<* *Greatest*(n1, n2) *<<* *endl*;

    return 0;

}

**Output**

****