PROGRAMS

DATE:15/03/24

**15 programs**

**1-Define a C++ class named Car with a constructor that initializes a member variable model to "Unknown". Write a program to create an object of the Car class and display its model**

#include <iostream>

#include <string>

using namespace std;

class Car {

private:

string model;

public:

Car(string m = "Unknown") : model(m) {}

string getModel() {

return model;

}

};

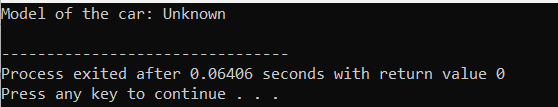
int main() {

Car myCar;

cout << "Model of the car: " << myCar.getModel() << endl;

return 0;

}



2-Extend the **Car** class from the previous question to have a parameterized constructor that sets the model based on user input. Write a program to create an object of the **Car** class with a specific model and display it.

#include <iostream>

#include <string>

using namespace std;

class Car {

private:

string model;

public:

Car(string m) : model(m) {}

string getModel() {

return model;

}

};

int main() {

string userEnteredModel;

cout << "Enter the model of the car: ";

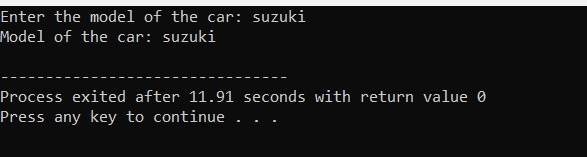
getline(cin, userEnteredModel);

Car myCar(userEnteredModel);

cout << "Model of the car: " << myCar.getModel() << endl;

return 0;

}



3-Define a C++ class named **Rectangle** with two constructors - one with default values for length and width, and another with parameters to set the length and width. Write a program to create objects of the **Rectangle** class using both constructors and display their dimensions.

#include <iostream>

using namespace std;

class Rectangle {

private:

double length;

double width;

public:

Rectangle(double len = 0.0, double wid = 0.0) : length(len), width(wid) {}

double getLength() {

return length;

}

double getWidth() {

return width;

}

};

int main() {

Rectangle defaultRect;

Rectangle customRect(5.0, 3.0);

cout << "Rectangle created using default constructor:" << endl;

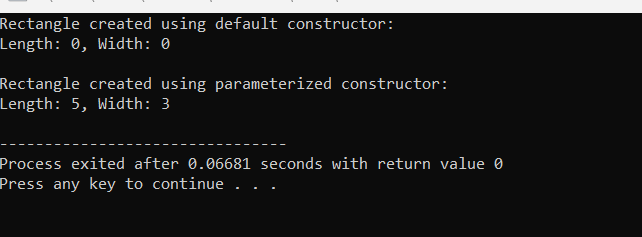
cout << "Length: " << defaultRect.getLength() << ", Width: " << defaultRect.getWidth() << endl;

cout << "\nRectangle created using parameterized constructor:" << endl;

cout << "Length: " << customRect.getLength() << ", Width: " << customRect.getWidth() << endl;

return 0;

}



4-. Create a C++ class named **Person** with member variables **name** and **age**. Implement a copy constructor to initialize a new object with the same values as an existing object. Write a program to demonstrate the use of the copy constructor.

#include <iostream>

#include <string>

using namespace std;

class Person {

private:

string name;

int age;

public:

Person(string n, int a) : name(n), age(a) {}

Person(const Person &p) : name(p.name), age(p.age) {}

void display() {

cout << "Name: " << name << ", Age: " << age << endl;

}

};

int main() {

Person person1("Alice", 30);

Person person2 = person1;

cout << "Details of person1: ";

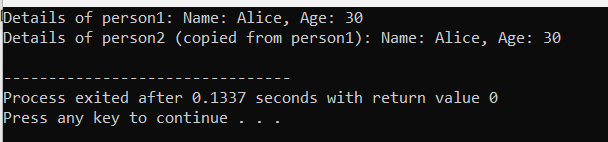
person1.display();

cout << "Details of person2 (copied from person1): ";

person2.display();

return 0;

}



5-. Define a C++ class named **DynamicArray** with a constructor that dynamically allocates an array of integers of a specified size. Implement a destructor to deallocate the memory. Write a program to create an object of the **DynamicArray** class and display its elements

#include <iostream>

using namespace std;

class DynamicArray {

private:

int \*arr;

int size;

public:

DynamicArray(int s) : size(s) {

arr = new int[size];

}

~DynamicArray() {

delete[] arr;

}

void display() {

cout << "Elements of the dynamic array:" << endl;

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

cout << arr[i] << " ";

}

cout << endl;

}

};

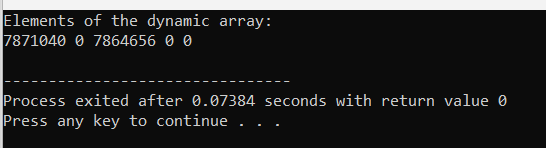
int main() {

DynamicArray dynArr(5);

dynArr.display();

return 0;

}



6-. Extend the **Person** class from question 4 with a destructor to display a goodbye message when an object is destroyed. Write a program to demonstrate the destruction of **Person** objects.

#include <iostream>

#include <string>

using namespace std;

class Person {

private:

string name;

int age;

public:

Person(string n, int a) : name(n), age(a) {}

~Person() {

cout << "Goodbye, " << name << "!" << endl;

}

void display() {

cout << "Name: " << name << ", Age: " << age << endl;

}

};

int main() {

Person person1("Alice", 30);

Person person2("Bob", 25);

cout << "Details of person1: ";

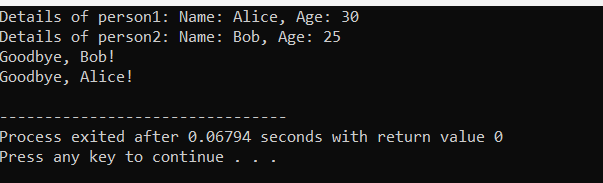
person1.display();

cout << "Details of person2: ";

person2.display();

return 0;

}



7-. Define a C++ class named **Point** representing a 2D point with member variables **x** and **y**. Implement operator overloading for the addition operator (+) to add two **Point** objects together. Write a program to demonstrate the operator overloading.

#include <iostream>

using namespace std;

class Point {

private:

int x;

int y;

public:

Point(int xVal = 0, int yVal = 0) : x(xVal), y(yVal) {}

Point operator+(const Point& other) {

Point result;

result.x = this->x + other.x;

result.y = this->y + other.y;

return result;

}

void display() {

cout << "(" << x << ", " << y << ")";

}

};

int main() {

Point p1(3, 4);

Point p2(1, 2);

Point sum = p1 + p2;

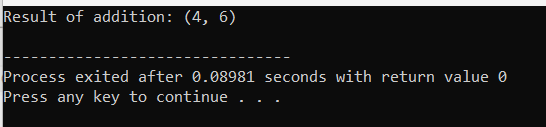
cout << "Result of addition: ";

sum.display();

cout << endl;

return 0;

}



8-. Extend the **Rectangle** class from question 3 to overload the addition operator (+) to add the areas of two rectangles together. Write a program to demonstrate the operator overloading.

#include <iostream>

using namespace std;

class Rectangle {

private:

double length;

double width;

public:

Rectangle(double len = 0.0, double wid = 0.0) : length(len), width(wid) {}

Rectangle operator+(const Rectangle& other) {

double totalLength = this->length + other.length;

double totalWidth = this->width + other.width;

return Rectangle(totalLength, totalWidth);

}

double calculateArea() {

return length \* width;

}

};

int main() {

Rectangle rect1(5.0, 3.0);

Rectangle rect2(4.0, 6.0);

Rectangle totalRect = rect1 + rect2;

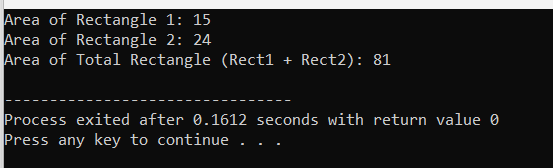
cout << "Area of Rectangle 1: " << rect1.calculateArea() << endl;

cout << "Area of Rectangle 2: " << rect2.calculateArea() << endl;

cout << "Area of Total Rectangle (Rect1 + Rect2): " << totalRect.calculateArea() << endl;

return 0;

}



9-. Define a C++ class named **Complex** representing a complex number with real and imaginary parts. Implement operator overloading for the addition, subtraction, and multiplication operators (+, -, \*) to perform arithmetic operations on **Complex** objects. Write a program to demonstrate the operator overloading.

#include <iostream>

using namespace std;

class Complex {

private:

double real;

double imag;

public:

Complex(double r = 0.0, double i = 0.0) : real(r), imag(i) {}

Complex operator+(const Complex& other) {

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

}

Complex operator-(const Complex& other) {

return Complex(real - other.real, imag - other.imag);

}

Complex operator\*(const Complex& other) {

double realPart = real \* other.real - imag \* other.imag;

double imagPart = real \* other.imag + imag \* other.real;

return Complex(realPart, imagPart);

}

void display() {

if (imag < 0)

cout << real << " - " << -imag << "i";

else

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

}

};

int main() {

// Creating two Complex objects

Complex c1(2.0, 3.0);

Complex c2(1.0, -2.0);

// Performing arithmetic operations on Complex objects using operator overloading

Complex addition = c1 + c2;

Complex subtraction = c1 - c2;

Complex multiplication = c1 \* c2;

// Displaying the results of arithmetic operations

cout << "Result of addition: ";

addition.display();

cout << endl;

cout << "Result of subtraction: ";

subtraction.display();

cout << endl;

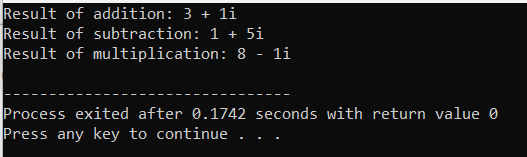
cout << "Result of multiplication: ";

multiplication.display();

cout << endl;

return 0;

}



10-Define a C++ class called **Number** with a single integer member variable **value**. Overload the unary minus (-) operator to negate the value of a **Number** object. Write a program to demonstrate the unary minus operator overloading.

#include <iostream>

using namespace std;

class Number {

private:

int value;

public:

Number(int val) : value(val) {}

Number operator-() {

return Number(-value);

}

int getValue() {

return value;

}

};

int main() {

Number num(10);

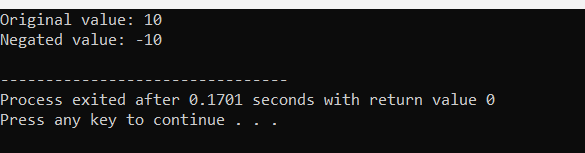
Number negatedNum = -num;

cout << "Original value: " << num.getValue() << endl;

cout << "Negated value: " << negatedNum.getValue() << endl;

return 0;

}



11-Create a class **Rectangle** with private attributes **length** and **width**. Implement a parameterized constructor to initialize these attributes. Also, include a function **calculateArea()** to compute and return the area of the rectangle.

#include <iostream>

using namespace std;

class Rectangle {

private:

double length;

double width;

public:

Rectangle(double len, double wid) : length(len), width(wid) {}

double calculateArea() {

return length \* width;

}

};

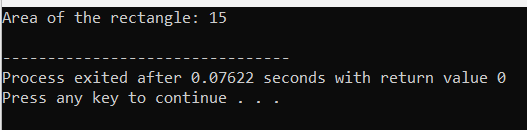
int main() {

Rectangle rect(5.0, 3.0);

cout << "Area of the rectangle: " << rect.calculateArea() << endl;

return 0;

}



1. 12-Define a class **Student** with private attributes **name** and **age**. Implement a default constructor that sets the name to "Unknown" and age to 0. Also, overload the constructor to accept values for name and age. Write a member function to display the details of the student.

#include <iostream>

#include <string>

using namespace std;

class Student {

private:

string name;

int age;

public:

Student() : name("Unknown"), age(0) {}

Student(string n, int a) : name(n), age(a) {}

void displayDetails() {

cout << "Name: " << name << ", Age: " << age << endl;

}

};

int main() {

Student student1;

Student student2("Alice", 20);

cout << "Details of student1: ";

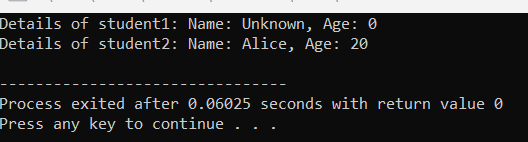
student1.displayDetails();

cout << "Details of student2: ";

student2.displayDetails();

return 0;

}



1. 13-Create a class **Complex** to represent complex numbers. Implement a parameterized constructor to initialize real and imaginary parts. Overload the addition operator (+) to add two complex numbers.

#include <iostream>

using namespace std;

class Complex {

private:

double real;

double imag;

public:

Complex(double r = 0.0, double i = 0.0) : real(r), imag(i) {}

Complex operator+(const Complex& other) {

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

}

void display() {

cout << "(" << real << " + " << imag << "i)";

}

};

int main() {

Complex c1(2.0, 3.0);

Complex c2(1.0, -2.0);

Complex sum = c1 + c2;

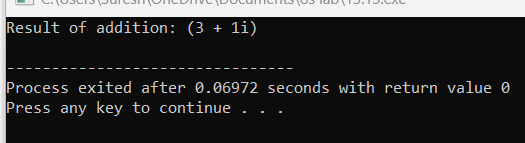
cout << "Result of addition: ";

sum.display();

cout << endl;

return 0;

}



14-Write a class **String** to represent a string of characters. Implement a copy constructor to perform a deep copy of the string

#include <iostream>

#include <cstring>

using namespace std;

class String {

private:

char \*str;

public:

String(const char \*s) {

str = new char[strlen(s) + 1];

strcpy(str, s);

}

String(const String &other) {

str = new char[strlen(other.str) + 1];

strcpy(str, other.str);

}

~String() {

delete[] str;

}

void display() {

cout << "String: " << str << endl;

}

};

int main() {

String str1("Hello");

String str2 = str1;

cout << "String 1: ";

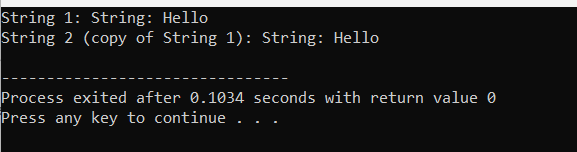
str1.display();

cout << "String 2 (copy of String 1): ";

str2.display();

return 0;

}



15-Develop a class **DynamicArray** that acts as a dynamically resizable array. Implement a parameterized constructor that initializes the array with a specified size. Also, include a destructor to release the memory allocated for the array.

#include <iostream>

using namespace std;

class DynamicArray {

private:

int \*arr;

int size;

public:

DynamicArray(int s) : size(s) {

arr = new int[size];

}

~DynamicArray() {

delete[] arr;

}

void display() {

cout << "Dynamic Array elements:" << endl;

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

cout << arr[i] << " ";

}

cout << endl;

}

};

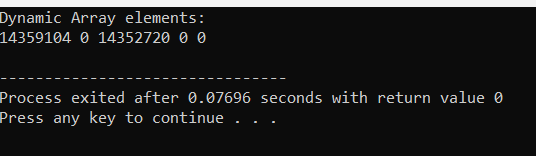
int main() {

DynamicArray dynArr(5);

dynArr.display();

return 0;

}



16-Implement a class **Vector2D** to represent 2-dimensional vectors. Overload the addition operator (+) to add two vectors component-wise.

#include <iostream>

using namespace std;

class Vector2D {

private:

double x;

double y;

public:

Vector2D(double xVal = 0.0, double yVal = 0.0) : x(xVal), y(yVal) {}

Vector2D operator+(const Vector2D& other) {

return Vector2D(x + other.x, y + other.y);

}

void display() {

cout << "(" << x << ", " << y << ")";

}

};

int main() {

Vector2D v1(2.0, 3.0);

Vector2D v2(1.0, -2.0);

Vector2D sum = v1 + v2;

// Displaying the result

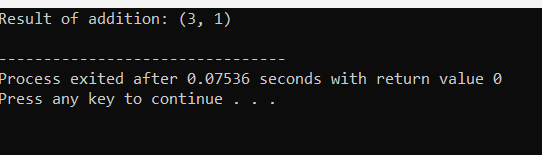
cout << "Result of addition: ";

sum.display();

cout << endl;

return 0;

}



17-Create a class **Book** with private attributes **title**, **author**, and **pages**. Implement a parameterized constructor to initialize these attributes. Overload the comparison operators (== and !=) to compare two books based on their titles.

#include <iostream>

#include <string>

using namespace std;

class Book {

private:

string title;

string author;

int pages;

public:

Book(string t, string a, int p) : title(t), author(a), pages(p) {}

bool operator==(const Book& other) {

return title == other.title;

}

bool operator!=(const Book& other) {

return !(\*this == other);

}

void display() {

cout << "Title: " << title << ", Author: " << author << ", Pages: " << pages << endl;

}

};

int main() {

Book book1("Title1", "Author1", 200);

Book book2("Title2", "Author2", 300);

Book book3("Title1", "Author3", 400);

cout << "Comparing book1 and book2: ";

if (book1 == book2)

cout << "Equal" << endl;

else

cout << "Not equal" << endl;

cout << "Comparing book1 and book3: ";

if (book1 != book3)

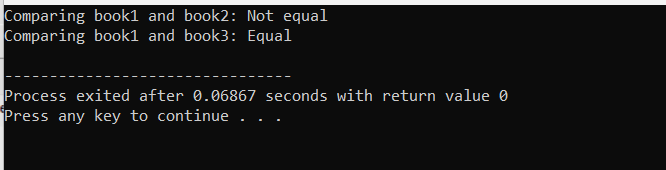
cout << "Not equal" << endl;

else

cout << "Equal" << endl;

return 0;

}



18-Define a class **Counter** to represent a simple counter. Include an integer attribute **count** and implement a default constructor that sets **count** to 0. Overload the prefix increment operator (++) to increment the counter.

#include <iostream>

using namespace std;

class Counter {

private:

int count;

public:

Counter() : count(0) {}

Counter& operator++() {

++count;

return \*this;

}

int getCount() const {

return count;

}

};

int main() {

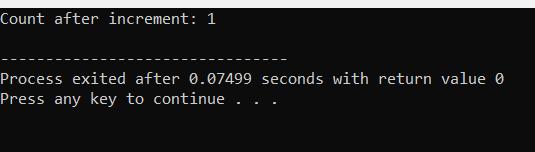
Counter counter;

++counter;

cout << "Count after increment: " << counter.getCount() << endl;

return 0;

}



19-Write a class **Matrix** to represent a 2x2 matrix. Implement a parameterized constructor to initialize the matrix elements. Overload the multiplication operator (\*) to perform matrix multiplication.

#include <iostream>

using namespace std;

class Matrix {

private:

int mat[2][2];

public:

Matrix(int a, int b, int c, int d) {

mat[0][0] = a;

mat[0][1] = b;

mat[1][0] = c;

mat[1][1] = d;

}

Matrix operator\*(const Matrix& other) {

Matrix result(0, 0, 0, 0);

for (int i = 0; i < 2; ++i) {

for (int j = 0; j < 2; ++j) {

for (int k = 0; k < 2; ++k) {

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

}

}

}

return result;

}

void display() {

cout << "Matrix:" << endl;

for (int i = 0; i < 2; ++i) {

for (int j = 0; j < 2; ++j) {

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

}

cout << endl;

}

}

};

int main() {

Matrix mat1(1, 2, 3, 4);

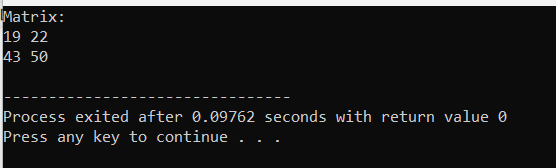
Matrix mat2(5, 6, 7, 8);

Matrix result = mat1 \* mat2;

result.display();

return 0;

}



20-Develop a class **Date** to represent dates (day, month, year). Implement a default constructor that sets the date to January 1, 2000. Overload the stream insertion (<<) and extraction (>>) operators to read and write dates from/to standard input/output.

#include <iostream>

using namespace std;

class Date {

private:

int day;

int month;

int year;

public:

Date() : day(1), month(1), year(2000) {}

friend ostream& operator<<(ostream& os, const Date& date) {

os << date.day << "/" << date.month << "/" << date.year;

return os;

}

friend istream& operator>>(istream& is, Date& date) {

char slash;

is >> date.day >> slash >> date.month >> slash >> date.year;

return is;

}

};

int main() {

Date myDate;

cout << "Default Date: " << myDate << endl;

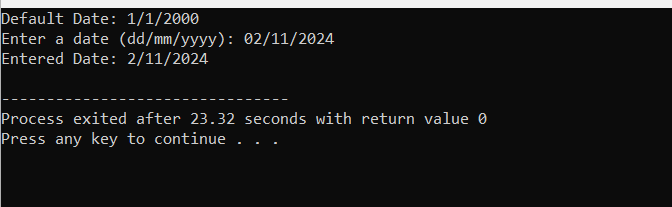
cout << "Enter a date (dd/mm/yyyy): ";

cin >> myDate;

cout << "Entered Date: " << myDate << endl;

return 0;

}



1.Implement a class **Matrix** to represent an arbitrary-sized matrix. Provide a parameterized constructor to initialize the matrix with the specified number of rows and columns. Implement a copy constructor to perform a deep copy of the matrix.

#include <iostream>

#include <vector>

class Matrix {

private:

int rows;

int cols;

std::vector<std::vector<int>> data;

public:

Matrix(int rows, int cols) : rows(rows), cols(cols), data(rows, std::vector<int>(cols, 0)) {}

Matrix(const Matrix &other) : rows(other.rows), cols(other.cols), data(other.data) {}

void setValue(int row, int col, int value) {

if (row >= 0 && row < rows && col >= 0 && col < cols)

data[row][col] = value;

else

std::cerr << "Error: Index out of bounds\n";

}

int getValue(int row, int col) const {

if (row >= 0 && row < rows && col >= 0 && col < cols)

return data[row][col];

else {

std::cerr << "Error: Index out of bounds\n";

return -1;

}

}

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;

}

}

};

int main() {

Matrix m1(3, 3);

m1.setValue(0, 0, 1);

m1.setValue(1, 1, 2);

m1.setValue(2, 2, 3);

std::cout << "Matrix m1:" << std::endl;

m1.display();

Matrix m2 = m1;

m2.setValue(1, 1, 9);

std::cout << "\nModified matrix m2:" << std::endl;

m2.display();

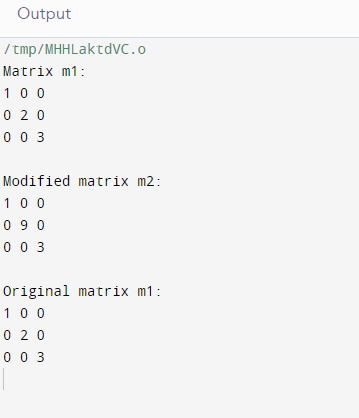
std::cout << "\nOriginal matrix m1:" << std::endl;

m1.display();

return 0;

}

OUTPUT:



1. 22. Create a class **LinkedList** to represent a linked list data structure. Implement a parameterized constructor to initialize the linked list with an array of elements. Include a destructor to free the memory allocated for the nodes.

#include <iostream>

class Node {

public:

int data;

Node\* next;

Node(int value) : data(value), next(nullptr) {}

};

class LinkedList {

private:

Node\* head;

public:

LinkedList(int arr[], int size) : head(nullptr) {

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

append(arr[i]);

}

}

~LinkedList() {

Node\* current = head;

while (current != nullptr) {

Node\* temp = current;

current = current->next;

delete temp;

}

head = nullptr;

}

void append(int value) {

Node\* newNode = new Node(value);

if (!head) {

head = newNode;

} else {

Node\* current = head;

while (current->next) {

current = current->next;

}

current->next = newNode;

}

}

void display() {

Node\* current = head;

while (current) {

std::cout << current->data << " ";

current = current->next;

}

std::cout << std::endl;

}

};

int main() {

int arr[] = {1, 2, 3, 4, 5};

int size = sizeof(arr) / sizeof(arr[0]);

LinkedList list(arr, size);

std::cout << "Initial linked list:" << std::endl;

list.display();

return 0;

}

OUTPUT:



1. 23. Develop a class **Polynomial** to represent polynomials. Implement a parameterized constructor to initialize the polynomial coefficients. Overload the addition operator (+) to add two polynomials.

#include <iostream>

#include <vector>

class Polynomial {

private:

std::vector<int> coefficients;

public:

Polynomial(const std::vector<int>& coeffs) : coefficients(coeffs) {}

Polynomial operator+(const Polynomial& other) const {

Polynomial result(std::vector<int>(std::max(coefficients.size(), other.coefficients.size()), 0));

for (size\_t i = 0; i < coefficients.size(); ++i) {

result.coefficients[i] += coefficients[i];

}

for (size\_t i = 0; i < other.coefficients.size(); ++i) {

result.coefficients[i] += other.coefficients[i];

}

return result;

}

void display() const {

for (int i = coefficients.size() - 1; i >= 0; --i) {

if (coefficients[i] != 0) {

if (i != coefficients.size() - 1)

std::cout << " + ";

if (coefficients[i] != 1 || i == 0)

std::cout << coefficients[i];

if (i != 0)

std::cout << "x^" << i;

}

}

std::cout << std::endl;

}

};

int main() {

std::vector<int> coeffs1 = {1, 2, 3};

std::vector<int> coeffs2 = {2, 3};

Polynomial poly1(coeffs1);

Polynomial poly2(coeffs2);

std::cout << "Polynomial 1: ";

poly1.display();

std::cout << "Polynomial 2: ";

poly2.display();

Polynomial sum = poly1 + poly2;

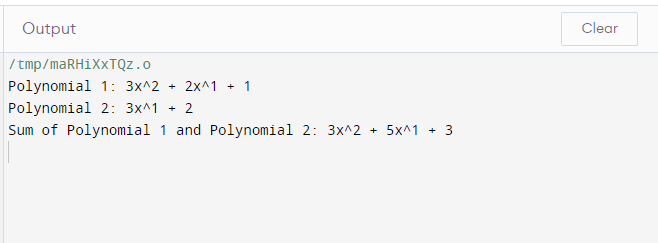
std::cout << "Sum of Polynomial 1 and Polynomial 2: ";

sum.display();

return 0;

}

OUTPUT

24. Write a class **Employee** to represent employees in a company. Implement a parameterized constructor to initialize employee details such as name, age, and salary. Overload the comparison operators (<, >) to compare employees based on their salaries.

#include <windows.h>

#include <iostream>

using namespace std;

class employee

{

int emp\_number;

char emp\_name[20];

float emp\_basic;

float emp\_da;

float emp\_it;

float emp\_net\_sal;

public:

void get\_emp\_details();

float find\_net\_salary(float basic, float da, float it);

void show\_emp\_details();

};

void employee :: get\_emp\_details()

{

cout<<"\nEnter employee number: ";

cin>>emp\_number;

cout<<"\nEnter employee name: ";

cin>>emp\_name;

cout<<"\nEnter employee basic: ";

cin>>emp\_basic;

cout<<"\nEnter employee DA: ";

cin>>emp\_da;

cout<<"\nEnter employee IT: ";

cin>>emp\_it;

}

float employee :: find\_net\_salary(float basic, float da, float it)

{

return (basic+da)-it;

}

void employee :: show\_emp\_details()

{

cout<<"\n\n\*\*\*\* Details of Employee \*\*\*\*";

cout<<"\nEmployee Name : "<<emp\_name;

cout<<"\nEmployee number : "<<emp\_number;

cout<<"\nBasic salary : "<<emp\_basic;

cout<<"\nEmployee DA : "<<emp\_da;

cout<<"\nIncome Tax : "<<emp\_it;

cout<<"\nNet Salary : "<<find\_net\_salary(emp\_basic, emp\_da, emp\_it);

cout<<"\n-------------------------------\n\n";

}

int main()

{

employee emp;

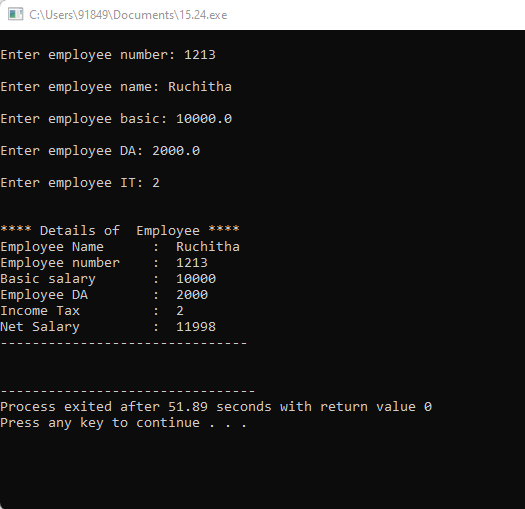
emp.get\_emp\_details();

emp.show\_emp\_details();

return 0;

}

OUTPUT:



25. Define a class **BinarySearchTree** to represent a binary search tree. Implement a copy constructor to create a deep copy of the tree. Ensure proper memory management and handling of dynamically allocated nodes

26.

#include <iostream>

class Fraction {

private:

int numerator;

int denominator;

public:

Fraction(int num, int denom) : numerator(num), denominator(denom) {}

Fraction operator/(const Fraction& other) const {

int num = numerator \* other.denominator;

int denom = denominator \* other.numerator;

return Fraction(num, denom);

}

void display() const {

std::cout << numerator << "/" << denominator;

}

};

int main() {

Fraction frac1(3, 4);

Fraction frac2(2, 5);

Fraction result = frac1 / frac2;

std::cout << "Result of division: ";

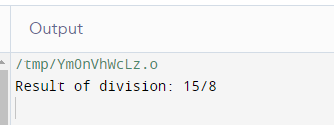
result.display();

std::cout << std::endl;

return 0;

}

OUTPUT:



27. Implement a class **String** to represent strings. Include a copy constructor to perform a deep copy of the string. Overload the concatenation operator (+) to concatenate two strings.

#include <iostream>

#include <cstring>

class String {

private:

char\* str;

public:

String(const char\* s = "") {

str = new char[strlen(s) + 1];

strcpy(str, s);

}

String(const String& other) {

str = new char[strlen(other.str) + 1];

strcpy(str, other.str);

}

~String() {

delete[] str;

}

String operator+(const String& other) const {

char\* temp = new char[strlen(str) + strlen(other.str) + 1];

strcpy(temp, str);

strcat(temp, other.str);

String result(temp);

delete[] temp;

return result;

}

void display() const {

std::cout << str;

}

};

int main() {

String str1("Hello, ");

String str2("world!");

String concatenated = str1 + str2;

std::cout << "Concatenated string: ";

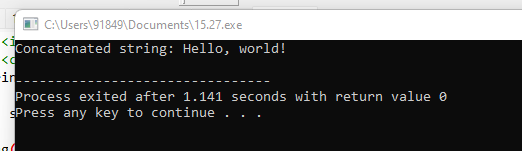
concatenated.display();

std::cout << std::endl;

return 0;

}

OUTPUT:



28.

#include <iostream>

#include <vector>

class Polynomial {

private:

std::vector<int> coefficients;

public:

Polynomial(const std::vector<int>& coeffs) : coefficients(coeffs) {}

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

std::vector<int> resultCoeffs(coefficients.size() + other.coefficients.size() - 1, 0);

for (size\_t i = 0; i < coefficients.size(); ++i) {

for (size\_t j = 0; j < other.coefficients.size(); ++j) {

resultCoeffs[i + j] += coefficients[i] \* other.coefficients[j];

}

}

return Polynomial(resultCoeffs);

}

void display() const {

for (int i = coefficients.size() - 1; i >= 0; --i) {

if (coefficients[i] != 0) {

if (i != coefficients.size() - 1)

std::cout << " + ";

if (coefficients[i] != 1 || i == 0)

std::cout << coefficients[i];

if (i != 0)

std::cout << "x^" << i;

}

}

std::cout << std::endl;

}

};

int main() {

std::vector<int> coeffs1 = {1, 2, 3};

std::vector<int> coeffs2 = {2, 3};

Polynomial poly1(coeffs1);

Polynomial poly2(coeffs2);

Polynomial result = poly1 \* poly2;

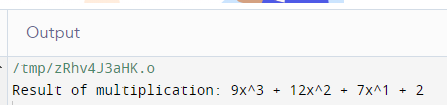
std::cout << "Result of multiplication: ";

result.display();

return 0;

}

OUTPUT:



29. Write a class **BigInt** to represent arbitrarily large integers. Implement a parameterized constructor to initialize the integer with a string representation. Overload the addition operator (+) to add two BigIntegers.

#include <iostream>

#include <string>

#include <algorithm>

#include <vector>

class BigInt {

private:

std::vector<int> digits;

public:

BigInt(const std::string& num) {

for (int i = num.size() - 1; i >= 0; --i) {

digits.push\_back(num[i] - '0');

}

}

BigInt operator+(const BigInt& other) const {

const std::vector<int>& num1 = digits.size() > other.digits.size() ? digits : other.digits;

const std::vector<int>& num2 = digits.size() > other.digits.size() ? other.digits : digits;

std::vector<int> result;

int carry = 0;

for (size\_t i = 0; i < num1.size(); ++i) {

int sum = num1[i] + (i < num2.size() ? num2[i] : 0) + carry;

result.push\_back(sum % 10);

carry = sum / 10;

}

if (carry > 0) {

result.push\_back(carry);

}

return BigInt(toString(result));

}

std::string toString(const std::vector<int>& nums) const {

std::string result;

for (int i = nums.size() - 1; i >= 0; --i) {

result += std::to\_string(nums[i]);

}

return result;

}

void display() const {

std::string result = toString(digits);

std::cout << result << std::endl;

}

};

int main() {

BigInt num1("12345678901234567890");

BigInt num2("98765432109876543210");

BigInt sum = num1 + num2;

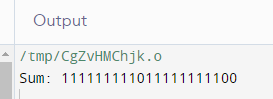
std::cout << "Sum: ";

sum.display();

return 0;

}

OUTPUT:



30. Create a class **Stack** to represent a stack data structure. Implement a copy constructor to create a deep copy of the stack. Ensure proper memory management and handling of dynamically allocated nodes.

#include <iostream>

#include <string>

using namespace std;

#define SIZE 5

template <class T> class Stack {

public:

Stack();

void push(T k);

T pop();

T topElement();

bool isFull();

bool isEmpty();

private:

int top;

T st[SIZE];

};

template <class T> Stack<T>::Stack() { top = -1; }

template <class T> void Stack<T>::push(T k)

{

if (isFull()) {

cout << "Stack is full\n";

}

cout << "Inserted element " << k << endl;

top = top + 1;

st[top] = k;

}

template <class T> bool Stack<T>::isEmpty()

{

if (top == -1)

return 1;

else

return 0;

}

template <class T> bool Stack<T>::isFull()

{

if (top == (SIZE - 1))

return 1;

else

return 0;

}

template <class T> T Stack<T>::pop()

{

T popped\_element = st[top];

top--;

return popped\_element;

}

template <class T> T Stack<T>::topElement()

{

T top\_element = st[top];

return top\_element;

}

int main()

{

Stack<int> integer\_stack;

Stack<string> string\_stack;

integer\_stack.push(2);

integer\_stack.push(54);

integer\_stack.push(255);

string\_stack.push("Welcome");

string\_stack.push("to");

string\_stack.push("GeeksforGeeks");

cout << integer\_stack.pop() << " is removed from stack"

<< endl;

cout << string\_stack.pop() << " is removed from stack "

<< endl;

cout << "Top element is " << integer\_stack.topElement()

<< endl;

cout << "Top element is " << string\_stack.topElement()

<< endl;

return 0;

}

OUTPUT:

