**CS213: OOP**

**Assignment 1 – Task 3**

**1# Omar Mostafa Azzam – 20230267**

**Introduction:**

* We have used chat GPT and preplexity to generate the code for the polynomial header file.
* We tried to use gemini but its code contained error and was hard to fix
* We used chat gpt to generate the test cases and test code also.

**Chat GPT implementation:**

* We used Chat GPT-4o for generating the code (in Appendix A) for polynomial class.
* We just gave it the polynomial header file and asked it for an implementation, Successfully It generated working and correct code from the first time except for some bugs I will mention.

**Chat GPT bugs:**

* if you created a polynomial object using coefficients constructor with extra zeros in the right Chat GPT doesn’t handle this and this causes logic errors in the code.

**Example:**

auto a = Polynomial({1, 0, 0});

auto b = Polynomial({1});

cout << a << endl;

cout << b << endl;

cout << (a == b) << endl;

**Output:**

**+ 1 // Error: bad extra plus in the start**

**1 // this has no problem**

**0 // Error: they should be equal**

Preplexity has automatically added an extra private Polynomial::trim method that trims the extra zeros from right.

Until now I have founded no errors in Chat GPT code except this edge case.

**Preplexity Implementation:**

* We worked with Preplexity for some time to generate the working and correct code (in Appendix B).
* We gave it the same input as Chat GPT (the Polynomial header file) and asked it for an implementation.
* Preplexity redfined the class again which we have not wanted, so we mentioned that and it produced the definition only for methods and operators after that.
* Preplexity added an extra Polynomial::trim() method as we said earlier so we added it to the header file in the private section.
* Perplexity tried to use the minus operator (which has not been defined for polynomial class) so we mentioned that and it haven’t used it again.

**Preplexity Bugs:**

* just one bug in the derivative method which, when using this method for an polynomial of degree zero it returns 1 instead of 0

**Example:**

auto a = Polynomial({5}); // function 5

cout << a.derivative() << endl;

**Output:**

**1 // wrong it should be zero**

The cause for this is that Preplexity used the default constructor which produces 1 not zero so we fixed this by using Polynomial({0}) instead.

**Test Cases:**

* Test cases are hard-coded at a .cpp file.
* We asked Chat GPT-4o mini to generate the test cases and the code that runs them which ouputs all the details including time measurements.
* some test cases failed but due to errors in test code not in the actual implementation so we fixed it.
* The bug in Preplexity that we said earlier made a test case actually fail
* You can find the test cases and code for them in Appendix C, and you can find the results of test cases for each AI in Appendix D.

**Comparsion:**

* Each of AIs contains a wrong point or two
* We can say that Preplexity code is a bit more efficient according to time measurements.
* We can say that Chat GPT is more responsive.
* It is hard to say that one AI is better than the other in everything
* The code they generated is simillar to a certain point.

**Appendix A**

**Chat GPT Implementation:**

#include "../polynomial.h"

// Constructors

Polynomial::Polynomial() : coeffs(1, 0.0) {} // Default constructor (constant 0)

Polynomial::Polynomial(const vector<double> &coefficients) : coeffs(coefficients) {} // Initialize with coefficients

Polynomial::Polynomial(const Polynomial &other) : coeffs(other.coeffs) {} // Copy constructor

// Destructor

Polynomial::~Polynomial() {}

// Assignment operator

Polynomial &Polynomial::operator=(const Polynomial &other)

{

if (this != &other)

{

coeffs = other.coeffs;

}

return \*this;

}

// Arithmetic operators

Polynomial Polynomial::operator+(const Polynomial &other) const

{

vector<double> result(max(coeffs.size(), other.coeffs.size()), 0.0);

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

{

if (i < coeffs.size())

result[i] += coeffs[i];

if (i < other.coeffs.size())

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

}

return Polynomial(result);

}

Polynomial Polynomial::operator-(const Polynomial &other) const

{

vector<double> result(max(coeffs.size(), other.coeffs.size()), 0.0);

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

{

if (i < coeffs.size())

result[i] += coeffs[i];

if (i < other.coeffs.size())

result[i] -= other.coeffs[i];

}

return Polynomial(result);

}

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

{

vector<double> result(coeffs.size() + other.coeffs.size() - 1, 0.0);

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

{

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

{

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

}

}

return Polynomial(result);

}

// Equality operator

bool Polynomial::operator==(const Polynomial &other) const

{

return coeffs == other.coeffs;

}

// Output operator (friend)

ostream &operator<<(ostream &out, const Polynomial &poly)

{

for (int i = poly.coeffs.size() - 1; i >= 0; --i)

{

if (poly.coeffs[i] != 0)

{

if (i != poly.coeffs.size() - 1 && poly.coeffs[i] > 0)

out << " + ";

else if (poly.coeffs[i] < 0)

out << " - ";

out << abs(poly.coeffs[i]);

if (i > 0)

out << "x";

if (i > 1)

out << "^" << i;

}

}

return out;

}

// Utility functions

int Polynomial::degree() const

{

return coeffs.size() - 1;

}

double Polynomial::evaluate(double x) const

{

double result = 0.0;

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

{

result = result \* x + coeffs[i];

}

return result;

}

Polynomial Polynomial::compose(const Polynomial &q) const

{

Polynomial result;

Polynomial temp({0.0}); // To store intermediate powers of q

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

{

result = result \* q + Polynomial({coeffs[i]});

}

return result;

}

Polynomial Polynomial::derivative() const

{

if (coeffs.size() == 1)

return Polynomial({0.0});

vector<double> result(coeffs.size() - 1);

for (size\_t i = 1; i < coeffs.size(); ++i)

{

result[i - 1] = coeffs[i] \* i;

}

return Polynomial(result);

}

Polynomial Polynomial::integral() const

{

vector<double> result(coeffs.size() + 1, 0.0);

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

{

result[i + 1] = coeffs[i] / (i + 1);

}

return Polynomial(result);

}

double Polynomial::integral(double x1, double x2) const

{

Polynomial integralPoly = integral();

return integralPoly.evaluate(x2) - integralPoly.evaluate(x1);

}

double Polynomial::getRoot(double guess, double tolerance, int maxIter)

{

Polynomial deriv = derivative();

double x = guess;

for (int i = 0; i < maxIter; ++i)

{

double fx = evaluate(x);

double dfx = deriv.evaluate(x);

if (abs(fx) < tolerance)

return x;

if (dfx == 0)

break; // Avoid division by zero

x -= fx / dfx;

}

return x; // Return the result after maxIter iterations

}

// Set and Get Coefficients

void Polynomial::setCoefficients(const vector<double> &coefficients)

{

coeffs = coefficients;

}

double Polynomial::getCoefficient(int degree) const

{

if (degree < 0 || degree >= coeffs.size())

return 0.0;

return coeffs[degree];

}

**Appendix B**

**Preplexity Implementation:**

#include <iostream>

#include <vector>

#include <stdexcept>

#include <cmath>

#include "../polynomial.h"

using namespace std;

void Polynomial::trim()

{

while (!coeffs.empty() && coeffs.back() == 0)

{

coeffs.pop\_back();

}

}

Polynomial::Polynomial() : coeffs(1, 0.0) {}

Polynomial::Polynomial(const vector<double> &coefficients) : coeffs(coefficients)

{

trim();

}

Polynomial::Polynomial(const Polynomial &other) : coeffs(other.coeffs) {}

Polynomial::~Polynomial() {}

Polynomial &Polynomial::operator=(const Polynomial &other)

{

if (this != &other)

{

coeffs = other.coeffs;

trim();

}

return \*this;

}

Polynomial Polynomial::operator+(const Polynomial &other) const

{

vector<double> result(max(coeffs.size(), other.coeffs.size()), 0);

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

{

if (i < coeffs.size())

result[i] += coeffs[i];

if (i < other.coeffs.size())

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

}

return Polynomial(result);

}

Polynomial Polynomial::operator-(const Polynomial &other) const

{

vector<double> result(max(coeffs.size(), other.coeffs.size()), 0);

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

{

if (i < coeffs.size())

result[i] += coeffs[i];

if (i < other.coeffs.size())

result[i] -= other.coeffs[i];

}

return Polynomial(result);

}

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

{

vector<double> result(coeffs.size() + other.coeffs.size() - 1, 0);

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

{

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

{

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

}

}

return Polynomial(result);

}

bool Polynomial::operator==(const Polynomial &other) const

{

return coeffs == other.coeffs;

}

ostream &operator<<(ostream &out, const Polynomial &poly)

{

if (poly.coeffs.empty())

{

out << "0"; // Handle zero polynomial

return out;

}

bool firstTerm = true; // To handle the sign of the first term

for (int i = poly.degree(); i >= 0; --i)

{

if (poly.coeffs[i] != 0)

{

// Skip the constant term (x^0)

if (i == 0)

continue;

// Print the sign for the first term

if (firstTerm)

{

out << poly.coeffs[i] << "x^" << i;

firstTerm = false; // Set to false after first term

}

else

{

out << (poly.coeffs[i] > 0 ? "+" : "")

<< poly.coeffs[i] << "x^" << i;

}

}

}

// Handle the constant term separately if it exists

if (poly.coeffs[0] != 0)

{

out << (firstTerm ? "" : "+") << poly.coeffs[0];

}

return out;

}

int Polynomial::degree() const

{

return coeffs.empty() ? -1 : static\_cast<int>(coeffs.size()) - 1;

}

double Polynomial::evaluate(double x) const

{

double result = 0;

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

{

result += coeffs[i] \* pow(x, i);

}

return result;

}

Polynomial Polynomial::compose(const Polynomial &q) const

{

Polynomial result({0}); // Start with zero polynomial

for (int i = degree(); i >= 0; --i)

{

result = result \* q + Polynomial({coeffs[i]});

}

return result;

}

Polynomial Polynomial::derivative() const

{

if (coeffs.size() <= 1)

return Polynomial();

vector<double> derivCoeffs(coeffs.size() - 1);

for (size\_t i = 1; i < coeffs.size(); ++i)

{

derivCoeffs[i - 1] = coeffs[i] \* i;

}

return Polynomial(derivCoeffs);

}

Polynomial Polynomial::integral() const

{

vector<double> intCoeffs(coeffs.size() + 1);

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

{

intCoeffs[i + 1] = coeffs[i] / (i + 1);

}

return Polynomial(intCoeffs);

}

double Polynomial::integral(double x1, double x2) const

{

// Get the antiderivative

Polynomial antiderivative = this->integral();

// Evaluate at x2 and x1

return antiderivative.evaluate(x2) - antiderivative.evaluate(x1);

}

double Polynomial::getRoot(double guess, double tolerance, int maxIter)

{

double x = guess;

for (int iter = 0; iter < maxIter; ++iter)

{

double f\_x = evaluate(x);

double f\_prime\_x = derivative().evaluate(x);

if (fabs(f\_prime\_x) < tolerance)

break; // Avoid division by zero

x -= f\_x / f\_prime\_x;

if (fabs(f\_x) < tolerance)

return x; // Found root

}

throw runtime\_error("Root not found within the maximum iterations.");

}

void Polynomial::setCoefficients(const vector<double> &coefficients)

{

coeffs = coefficients;

trim();

}

double Polynomial::getCoefficient(int degree) const

{

if (degree < 0 || degree >= static\_cast<int>(coeffs.size()))

throw out\_of\_range("Degree out of range.");

return coeffs[degree];

}

**Appendix C**

**Test cases and code:**

#include <iostream>

#include <vector>

#include <chrono>

#include <functional> // Include for std::function

#include "../polynomial.h" // Assuming this is the name of your header file

using namespace std;

using namespace std::chrono;

// Function to measure and print the execution time

void measureTime(const string &testName, const std::function<void()> &testFunction)

{

auto start = high\_resolution\_clock::now();

testFunction();

auto stop = high\_resolution\_clock::now();

auto duration = duration\_cast<microseconds>(stop - start);

cout << "Execution time: " << duration.count() << " microseconds." << endl;

}

// Test cases for the Polynomial class

void testPolynomial()

{

// Test Cases for Constructors

cout << "=== Test Case 1: Default Constructor ===" << endl;

measureTime("Default Constructor", []()

{

Polynomial p;

cout << "Polynomial: " << p << endl; // Print polynomial

// Check if the polynomial has no coefficients (should be empty)

if (p.degree() == 0) {

cout << "Passed: Default constructor creates a polynomial with zero degree." << endl;

} else {

cout << "Failed: Default constructor did not create a zero degree polynomial." << endl;

} });

cout << endl; // New line for clarity

cout << "=== Test Case 2: Constructor with Coefficients ===" << endl;

measureTime("Constructor with Coefficients", []()

{

Polynomial p({3, 2, 1}); // Represents 1 + 2x + 3x^2

cout << "Polynomial: " << p << endl; // Print polynomial

if (p.degree() == 2) {

cout << "Passed: Created polynomial with correct degree." << endl;

} else {

cout << "Failed: Degree was " << p.degree() << endl;

} });

cout << endl; // New line for clarity

cout << "=== Test Case 3: Copy Constructor ===" << endl;

measureTime("Copy Constructor", []()

{

Polynomial p1({4, 0, -2});

Polynomial p2(p1);

cout << "Polynomial 1: " << p1 << endl; // Print polynomial

cout << "Polynomial 2: " << p2 << endl; // Print polynomial

if (p1 == p2) {

cout << "Passed: Copy constructor creates an equal polynomial." << endl;

} else {

cout << "Failed: Copy constructor did not create an equal polynomial." << endl;

} });

cout << endl; // New line for clarity

cout << "=== Test Case 4: Assignment Operator ===" << endl;

measureTime("Assignment Operator", []()

{

Polynomial p1({1, 2, 3});

Polynomial p2;

p2 = p1; // Assign p1 to p2

cout << "Polynomial 1: " << p1 << endl; // Print polynomial

cout << "Polynomial 2 (after assignment): " << p2 << endl; // Print polynomial

if (p1 == p2) {

cout << "Passed: Assignment operator copies polynomial correctly." << endl;

} else {

cout << "Failed: Assignment operator did not copy polynomial correctly." << endl;

} });

cout << endl; // New line for clarity

cout << "=== Test Case 5: Degree Function ===" << endl;

measureTime("Degree Function", []()

{

Polynomial p({1, 0, -4}); // Represents -4 + x^2

cout << "Polynomial: " << p << endl; // Print polynomial

if (p.degree() == 2) {

cout << "Passed: Degree function returns the correct degree." << endl;

} else {

cout << "Failed: Degree function returned " << p.degree() << endl;

} });

cout << endl; // New line for clarity

// Test Cases for Evaluate Function

cout << "=== Test Case 6: Evaluate Function (x=2) ===" << endl;

measureTime("Evaluate Function (x=2)", []()

{

Polynomial p({1, 2, 1}); // Represents 1 + 2x + x^2

cout << "Polynomial: " << p << endl; // Print polynomial

double result = p.evaluate(2);

cout << "Expected Output: 9, Actual Output: " << result << endl; // Print expected and actual output

if (result == 9) {

cout << "Passed: Evaluate function returns the correct value for x=2." << endl;

} else {

cout << "Failed: Evaluate function returned " << result << " for x=2." << endl;

} });

cout << endl; // New line for clarity

cout << "=== Test Case 7: Evaluate Function (x=0) ===" << endl;

measureTime("Evaluate Function (x=0)", []()

{

Polynomial p({3, 0, 1}); // Represents 1 + 3

cout << "Polynomial: " << p << endl; // Print polynomial

double result = p.evaluate(0);

cout << "Expected Output: 3, Actual Output: " << result << endl; // Print expected and actual output

if (result == 3) {

cout << "Passed: Evaluate function returns the correct value for x=0." << endl;

} else {

cout << "Failed: Evaluate function returned " << result << " for x=0." << endl;

} });

cout << endl; // New line for clarity

cout << "=== Test Case 8: Evaluate Function (x=-1) ===" << endl;

measureTime("Evaluate Function (x=-1)", []()

{

Polynomial p({1, -1}); // Represents 1 - x

cout << "Polynomial: " << p << endl; // Print polynomial

double result = p.evaluate(-1);

cout << "Expected Output: 2, Actual Output: " << result << endl; // Print expected and actual output

if (result == 2) {

cout << "Passed: Evaluate function returns the correct value for x=-1." << endl;

} else {

cout << "Failed: Evaluate function returned " << result << " for x=-1." << endl;

} });

cout << endl; // New line for clarity

cout << "=== Test Case 9: Evaluate Function (x=1) ===" << endl;

measureTime("Evaluate Function (x=1)", []()

{

Polynomial p({1, -2, 1}); // Represents 1 - 2x + x^2

cout << "Polynomial: " << p << endl; // Print polynomial

double result = p.evaluate(1);

cout << "Expected Output: 0, Actual Output: " << result << endl; // Print expected and actual output

if (result == 0) {

cout << "Passed: Evaluate function returns the correct value for x=1." << endl;

} else {

cout << "Failed: Evaluate function returned " << result << " for x=1." << endl;

} });

cout << endl; // New line for clarity

cout << "=== Test Case 10: Evaluate Function (x=3) ===" << endl;

measureTime("Evaluate Function (x=3)", []()

{

Polynomial p({1, 3, 2}); // Represents 2x^2 + 3x + 1

cout << "Polynomial: " << p << endl; // Print polynomial

double result = p.evaluate(3);

cout << "Expected Output: 28, Actual Output: " << result << endl; // Print expected and actual output

if (result == 28) {

cout << "Passed: Evaluate function returns the correct value for x=3." << endl;

} else {

cout << "Failed: Evaluate function returned " << result << " for x=3." << endl;

} });

cout << endl; // New line for clarity

// Test Cases for Arithmetic Operators

cout << "=== Test Case 11: Addition Operator ===" << endl;

measureTime("Addition Operator", []()

{

Polynomial p1({1, 1}); // Represents 1 + x

Polynomial p2({2, 2}); // Represents 2 + 2x

Polynomial result = p1 + p2; // Should be 3 + 3x

Polynomial expected({3, 3});

cout << "Polynomial 1: " << p1 << endl; // Print polynomial

cout << "Polynomial 2: " << p2 << endl; // Print polynomial

cout << "Expected Output: " << expected << ", Actual Output: " << result << endl; // Print expected and actual output

if (result == expected) {

cout << "Passed: Addition operator works correctly." << endl;

} else {

cout << "Failed: Addition operator did not return the expected polynomial." << endl;

} });

cout << endl; // New line for clarity

cout

<< "=== Test Case 12: Subtraction Operator ===" << endl;

measureTime("Subtraction Operator", []()

{

Polynomial p1({5, 2}); // Represents 2x + 5

Polynomial p2({3, 1}); // Represents x + 3

Polynomial result = p1 - p2; // Should be 2 + x

Polynomial expected({2, 1});

cout << "Polynomial 1: " << p1 << endl; // Print polynomial

cout << "Polynomial 2: " << p2 << endl; // Print polynomial

cout << "Expected Output: " << expected << ", Actual Output: " << result << endl; // Print expected and actual output

if (result == expected) {

cout << "Passed: Subtraction operator works correctly." << endl;

} else {

cout << "Failed: Subtraction operator did not return the expected polynomial." << endl;

} });

cout << endl; // New line for clarity

cout << "=== Test Case 13: Multiplication Operator ===" << endl;

measureTime("Multiplication Operator", []()

{

Polynomial p1({1, 1}); // Represents 1 + x

Polynomial p2({1, 1}); // Represents 1 + x

Polynomial result = p1 \* p2; // Should be 1 + 2x + x^2

Polynomial expected({1, 2, 1});

cout << "Polynomial 1: " << p1 << endl; // Print polynomial

cout << "Polynomial 2: " << p2 << endl; // Print polynomial

cout << "Expected Output: " << expected << ", Actual Output: " << result << endl; // Print expected and actual output

if (result == expected) {

cout << "Passed: Multiplication operator works correctly." << endl;

} else {

cout << "Failed: Multiplication operator did not return the expected polynomial." << endl;

} });

cout << endl; // New line for clarity

cout << "=== Test Case 14: Equality Operator (Equal Polynomials) ===" << endl;

measureTime("Equality Operator (Equal Polynomials)", []()

{

Polynomial p1({3, 2, 1}); // Represents 1 + 2x + 3x^2

Polynomial p2({3, 2, 1}); // Represents the same polynomial

cout << "Polynomial 1: " << p1 << endl; // Print polynomial

cout << "Polynomial 2: " << p2 << endl; // Print polynomial

if (p1 == p2) {

cout << "Passed: Equality operator identifies equal polynomials." << endl;

} else {

cout << "Failed: Equality operator did not identify equal polynomials." << endl;

} });

cout << endl; // New line for clarity

cout << "=== Test Case 15: Equality Operator (Different Polynomials) ===" << endl;

measureTime("Equality Operator (Different Polynomials)", []()

{

Polynomial p1({1, 2}); // Represents 1 + 2x

Polynomial p2({1, 2, 3}); // Represents 1 + 2x + 3x^2

cout << "Polynomial 1: " << p1 << endl; // Print polynomial

cout << "Polynomial 2: " << p2 << endl; // Print polynomial

if (!(p1 == p2)) {

cout << "Passed: Equality operator identifies different polynomials." << endl;

} else {

cout << "Failed: Equality operator did not identify different polynomials." << endl;

} });

cout << endl; // New line for clarity

// Test Cases for Utility Functions

cout << "=== Test Case 16: Derivative Function ===" << endl;

measureTime("Derivative Function", []()

{

Polynomial p({2, 3}); // Represents 3x + 2

Polynomial derivative = p.derivative(); // Should be 3

Polynomial expected({3});

cout << "Polynomial: " << p << endl; // Print polynomial

cout << "Expected Output: " << expected << ", Actual Output: " << derivative << endl; // Print expected and actual output

if (derivative == expected) {

cout << "Passed: Derivative function works correctly." << endl;

} else {

cout << "Failed: Derivative function did not return the expected polynomial." << endl;

} });

cout << endl; // New line for clarity

cout << "=== Test Case 17: Integral Function ===" << endl;

measureTime("Integral Function", []()

{

Polynomial p({1, 2}); // Represents 2x + 1

Polynomial integral = p.integral(); // Should be x^2 + 2x

Polynomial expected({0, 1, 1}); // Constants are ignored for the integral representation

cout << "Polynomial: " << p << endl; // Print polynomial

cout << "Expected Output: " << expected << ", Actual Output: " << integral << endl; // Print expected and actual output

if (integral == expected) {

cout << "Passed: Integral function works correctly." << endl;

} else {

cout << "Failed: Integral function did not return the expected polynomial." << endl;

} });

cout << endl; // New line for clarity

cout << "=== Test Case 18: Derivative of a Constant Polynomial ===" << endl;

measureTime("Derivative of a Constant Polynomial", []()

{

Polynomial p({5}); // Represents 5

Polynomial derivative = p.derivative(); // Should be 0

Polynomial expected({0});

cout << "Polynomial: " << p << endl; // Print polynomial

cout << "Expected Output: " << expected << ", Actual Output: " << derivative << endl; // Print expected and actual output

if (derivative == expected) {

cout << "Passed: Derivative of a constant polynomial is zero." << endl;

} else {

cout << "Failed: Derivative returned a non-zero polynomial." << endl;

} });

cout << endl; // New line for clarity

cout << "=== Test Case 19: Integral from 0 to 1 ===" << endl;

measureTime("Integral from 0 to 1", []()

{

Polynomial p({0, 0, 1}); // Represents x^2

double result = p.integral(0, 1); // Should return 1/3

cout << "Polynomial: " << p << endl; // Print polynomial

cout << "Expected Output: " << 1.0 / 3.0 << ", Actual Output: " << result << endl; // Print expected and actual output

if (fabs(result - (1.0 / 3.0)) < 1e-6) {

cout << "Passed: Integral from 0 to 1 is correct." << endl;

} else {

cout << "Failed: Integral returned " << result << endl;

} });

cout << endl; // New line for clarity

cout << "=== Test Case 20: Getting a Root ===" << endl;

measureTime("Getting a Root", []()

{

Polynomial p({1, -3, 2}); // Represents x^2 - 3x + 2

double root = p.getRoot(); // Should return a root, for example, 1 or 2

cout << "Polynomial: " << p << endl; // Print polynomial

cout << "Expected Output: A root (1 or 2), Actual Output: " << root << endl; // Print expected and actual output

if (fabs(p.evaluate(root)) < 1e-6) {

cout << "Passed: Found a root of the polynomial." << endl;

} else {

cout << "Failed: The found root did not satisfy the polynomial." << endl;

} });

cout << endl; // New line for clarity

}

int main()

{

testPolynomial();

return 0;

}

**Appendix D**

**I) Chat GPT results**

=== Test Case 1: Default Constructor ===

Polynomial:

Passed: Default constructor creates a polynomial with zero degree.

Execution time: 12 microseconds.

=== Test Case 2: Constructor with Coefficients ===

Polynomial: 1x^2 + 2x + 3

Passed: Created polynomial with correct degree.

Execution time: 42 microseconds.

=== Test Case 3: Copy Constructor ===

Polynomial 1: - 2x^2 + 4

Polynomial 2: - 2x^2 + 4

Passed: Copy constructor creates an equal polynomial.

Execution time: 18 microseconds.

=== Test Case 4: Assignment Operator ===

Polynomial 1: 3x^2 + 2x + 1

Polynomial 2 (after assignment): 3x^2 + 2x + 1

Passed: Assignment operator copies polynomial correctly.

Execution time: 12 microseconds.

=== Test Case 5: Degree Function ===

Polynomial: - 4x^2 + 1

Passed: Degree function returns the correct degree.

Execution time: 21 microseconds.

=== Test Case 6: Evaluate Function (x=2) ===

Polynomial: 1x^2 + 2x + 1

Expected Output: 9, Actual Output: 9

Passed: Evaluate function returns the correct value for x=2.

Execution time: 8 microseconds.

=== Test Case 7: Evaluate Function (x=0) ===

Polynomial: 1x^2 + 3

Expected Output: 3, Actual Output: 3

Passed: Evaluate function returns the correct value for x=0.

Execution time: 29 microseconds.

=== Test Case 8: Evaluate Function (x=-1) ===

Polynomial: - 1x + 1

Expected Output: 2, Actual Output: 2

Passed: Evaluate function returns the correct value for x=-1.

Execution time: 6 microseconds.

=== Test Case 9: Evaluate Function (x=1) ===

Polynomial: 1x^2 - 2x + 1

Expected Output: 0, Actual Output: 0

Passed: Evaluate function returns the correct value for x=1.

Execution time: 8 microseconds.

=== Test Case 10: Evaluate Function (x=3) ===

Polynomial: 2x^2 + 3x + 1

Expected Output: 28, Actual Output: 28

Passed: Evaluate function returns the correct value for x=3.

Execution time: 22 microseconds.

=== Test Case 11: Addition Operator ===

Polynomial 1: 1x + 1

Polynomial 2: 2x + 2

Expected Output: 3x + 3, Actual Output: 3x + 3

Passed: Addition operator works correctly.

Execution time: 18 microseconds.

=== Test Case 12: Subtraction Operator ===

Polynomial 1: 2x + 5

Polynomial 2: 1x + 3

Expected Output: 1x + 2, Actual Output: 1x + 2

Passed: Subtraction operator works correctly.

Execution time: 20 microseconds.

=== Test Case 13: Multiplication Operator ===

Polynomial 1: 1x + 1

Polynomial 2: 1x + 1

Expected Output: 1x^2 + 2x + 1, Actual Output: 1x^2 + 2x + 1

Passed: Multiplication operator works correctly.

Execution time: 17 microseconds.

=== Test Case 14: Equality Operator (Equal Polynomials) ===

Polynomial 1: 1x^2 + 2x + 3

Polynomial 2: 1x^2 + 2x + 3

Passed: Equality operator identifies equal polynomials.

Execution time: 27 microseconds.

=== Test Case 15: Equality Operator (Different Polynomials) ===

Polynomial 1: 2x + 1

Polynomial 2: 3x^2 + 2x + 1

Passed: Equality operator identifies different polynomials.

Execution time: 71 microseconds.

=== Test Case 16: Derivative Function ===

Polynomial: 3x + 2

Expected Output: 3, Actual Output: 3

Passed: Derivative function works correctly.

Execution time: 10 microseconds.

=== Test Case 17: Integral Function ===

Polynomial: 2x + 1

Expected Output: 1x^2 + 1x, Actual Output: 1x^2 + 1x

Passed: Integral function works correctly.

Execution time: 10 microseconds.

=== Test Case 18: Derivative of a Constant Polynomial ===

Polynomial: 5

Expected Output: , Actual Output:

Passed: Derivative of a constant polynomial is zero.

Execution time: 8 microseconds.

=== Test Case 19: Integral from 0 to 1 ===

Polynomial: 1x^2

Expected Output: 0.333333, Actual Output: 0.333333

Passed: Integral from 0 to 1 is correct.

Execution time: 19 microseconds.

=== Test Case 20: Getting a Root ===

Polynomial: 2x^2 - 3x + 1

Expected Output: A root (1 or 2), Actual Output: 1

Passed: Found a root of the polynomial.

Execution time: 10 microseconds.

**II) Preplexity results**

=== Test Case 1: Default Constructor ===

Polynomial:

Passed: Default constructor creates a polynomial with zero degree.

Execution time: 12 microseconds.

=== Test Case 2: Constructor with Coefficients ===

Polynomial: 1x^2+2x^1+3

Passed: Created polynomial with correct degree.

Execution time: 32 microseconds.

=== Test Case 3: Copy Constructor ===

Polynomial 1: -2x^2+4

Polynomial 2: -2x^2+4

Passed: Copy constructor creates an equal polynomial.

Execution time: 9 microseconds.

=== Test Case 4: Assignment Operator ===

Polynomial 1: 3x^2+2x^1+1

Polynomial 2 (after assignment): 3x^2+2x^1+1

Passed: Assignment operator copies polynomial correctly.

Execution time: 12 microseconds.

=== Test Case 5: Degree Function ===

Polynomial: -4x^2+1

Passed: Degree function returns the correct degree.

Execution time: 14 microseconds.

=== Test Case 6: Evaluate Function (x=2) ===

Polynomial: 1x^2+2x^1+1

Expected Output: 9, Actual Output: 9

Passed: Evaluate function returns the correct value for x=2.

Execution time: 17 microseconds.

=== Test Case 7: Evaluate Function (x=0) ===

Polynomial: 1x^2+3

Expected Output: 3, Actual Output: 3

Passed: Evaluate function returns the correct value for x=0.

Execution time: 11 microseconds.

=== Test Case 8: Evaluate Function (x=-1) ===

Polynomial: -1x^1+1

Expected Output: 2, Actual Output: 2

Passed: Evaluate function returns the correct value for x=-1.

Execution time: 7 microseconds.

=== Test Case 9: Evaluate Function (x=1) ===

Polynomial: 1x^2-2x^1+1

Expected Output: 0, Actual Output: 0

Passed: Evaluate function returns the correct value for x=1.

Execution time: 9 microseconds.

=== Test Case 10: Evaluate Function (x=3) ===

Polynomial: 2x^2+3x^1+1

Expected Output: 28, Actual Output: 28

Passed: Evaluate function returns the correct value for x=3.

Execution time: 7 microseconds.

=== Test Case 11: Addition Operator ===

Polynomial 1: 1x^1+1

Polynomial 2: 2x^1+2

Expected Output: 3x^1+3, Actual Output: 3x^1+3

Passed: Addition operator works correctly.

Execution time: 16 microseconds.

=== Test Case 12: Subtraction Operator ===

Polynomial 1: 2x^1+5

Polynomial 2: 1x^1+3

Expected Output: 1x^1+2, Actual Output: 1x^1+2

Passed: Subtraction operator works correctly.

Execution time: 18 microseconds.

=== Test Case 13: Multiplication Operator ===

Polynomial 1: 1x^1+1

Polynomial 2: 1x^1+1

Expected Output: 1x^2+2x^1+1, Actual Output: 1x^2+2x^1+1

Passed: Multiplication operator works correctly.

Execution time: 15 microseconds.

=== Test Case 14: Equality Operator (Equal Polynomials) ===

Polynomial 1: 1x^2+2x^1+3

Polynomial 2: 1x^2+2x^1+3

Passed: Equality operator identifies equal polynomials.

Execution time: 31 microseconds.

=== Test Case 15: Equality Operator (Different Polynomials) ===

Polynomial 1: 2x^1+1

Polynomial 2: 3x^2+2x^1+1

Passed: Equality operator identifies different polynomials.

Execution time: 8 microseconds.

=== Test Case 16: Derivative Function ===

Polynomial: 3x^1+2

Expected Output: 3, Actual Output: 3

Passed: Derivative function works correctly.

Execution time: 8 microseconds.

=== Test Case 17: Integral Function ===

Polynomial: 2x^1+1

Expected Output: 1x^2+1x^1, Actual Output: 1x^2+1x^1

Passed: Integral function works correctly.

Execution time: 10 microseconds.

=== Test Case 18: Derivative of a Constant Polynomial ===

Polynomial: 5

Expected Output: 0, Actual Output: 1

Failed: The found root did not satisfy the polynomial.

Execution time: 6 microseconds.

=== Test Case 19: Integral from 0 to 1 ===

Polynomial: 1x^2

Expected Output: 0.333333, Actual Output: 0.333333

Passed: Integral from 0 to 1 is correct.

Execution time: 14 microseconds.

=== Test Case 20: Getting a Root ===

Polynomial: 2x^2-3x^1+1

Expected Output: A root (1 or 2), Actual Output: 1

Passed: Found a root of the polynomial.

Execution time: 8 microseconds.