**📄 README — Matrix Operation Library (C++)**

**✅ Project Title:**

**Matrix Operation Library using Inheritance and Polymorphism in C++**

**📌 Assigned Task Description**

The project requires implementing a C++ program that performs operations on dynamically allocated 2D integer matrices using **object-oriented programming** principles, including:

* **Dynamic memory allocation** for 2D arrays (int\*\*)
* A **base abstract class** MatrixOp that defines a pure virtual function for matrix operations.
* Derived classes that override this operation:
  + AddMatrixOp for matrix addition.
  + MulMatrixOp for matrix multiplication.
  + InverseMatrixOp for matrix inversion using Gaussian elimination.
* Use of **pointer arithmetic** to perform matrix element access.
* Matrix input/output using cin and cout.
* **Polymorphic dispatching**: storing MatrixOp\* in an array to call operations based on user menu.

**🛠️ How It Was Completed**

1. **Matrix Memory Management:**
   * allocateMatrix() dynamically creates 2D integer arrays.
   * freeMatrix() deallocates them to prevent memory leaks.
2. **Object-Oriented Design:**
   * An abstract class MatrixOp defines the common interface.
   * Derived classes (AddMatrixOp, MulMatrixOp, InverseMatrixOp) implement specific matrix operations.
   * Polymorphism allows unified access to different operations using base class pointers.
3. **Gaussian Elimination**:
   * Used in InverseMatrixOp to invert a square matrix.
   * Matrix inversion is done using double-precision to reduce rounding errors.
4. **Menu-Driven Interface:**
   * The program offers options for addition, multiplication, inversion, and exit.
   * Loop continues until the user chooses to exit.
5. **Exception Handling:**
   * Invalid operations (e.g., wrong matrix dimensions) are caught with try-catch.

**📎 Code Annotation and Explanation**

Here's a condensed and annotated explanation of the code:

#include <iostream> // // For input/output

#include <iomanip> // For nicely formatted output

#include <stdexcept> // To throw/catch exceptions

#include <cmath> // For math functions like fabs, round

using namespace std; //Avoids prefixing with std::

// Function: Dynamically allocates a matrix of integers

int\*\* allocateMatrix(int r, int c) {

int\*\* mat = new int\*[r]; //Allocate array of int pointers (rows)

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

mat[i] = new int[c]; Allocate array for each row (columns)

return mat;

}

// Function: Deallocates a matrix of integers

void freeMatrix(int\*\* mat, int r) {

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

delete[] mat[i]; //Deallocate each row

delete[] mat; // Deallocate the array of row pointers

}

// Function: Displays a matrix

void printMatrix(int\*\* mat, int r, int c) {

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

for (int j = 0; j < c; ++j)

cout << setw(6) << \*(\*(mat + i) + j)

cout << endl;

}// Print each element formatted

}

// Abstract base class for matrix operations

class MatrixOp {

public:

virtual int\*\* apply(int\*\*, int\*\*, int, int, int, int, int&, int&) = 0;

virtual ~MatrixOp() {}

};

// Derived class for matrix addition

class AddMatrixOp : public MatrixOp {

public:

int\*\* apply(int\*\* A, int\*\* B, int r1, int c1, int r2, int c2, int& rr, int& rc) override {

if (r1 != r2 || c1 != c2)

throw invalid\_argument("Addition requires same dimensions.");

rr = r1; rc = c1;

int\*\* result = allocateMatrix(rr, rc);

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

for (int j = 0; j < rc; j++)

\*(\*(result + i) + j) = \*(\*(A + i) + j) + \*(\*(B + i) + j);

return result;

}

};

// Derived class for matrix multiplication

class MulMatrixOp : public MatrixOp {

public:

int\*\* apply(int\*\* A, int\*\* B, int r1, int c1, int r2, int c2, int& rr, int& rc) override {

if (c1 != r2)

throw invalid\_argument("Multiplication requires A columns = B rows.");

rr = r1; rc = c2;

int\*\* result = allocateMatrix(rr, rc);

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

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

\*(\*(result + i) + j) = 0;

for (int k = 0; k < c1; k++)

\*(\*(result + i) + j) += \*(\*(A + i) + k) \* \*(\*(B + k) + j);

}

return result;

}

};

// Derived class for matrix inversion using Gaussian elimination

class InverseMatrixOp : public MatrixOp {

public:

int\*\* apply(int\*\* A, int\*\*, int r1, int c1, int, int, int& rr, int& rc) override {

if (r1 != c1)

throw invalid\_argument("Inversion requires square matrix.");

int n = r1;

rr = rc = n;

➡ We assign the number of rows (r1) of matrix A to n (because a square matrix has the same rows and columns). We also assign rr and rc the value n to store result dimensions.

double\*\* mat = new double\*[n];

double\*\* inv = new double\*[n];

➡ We dynamically allocate two double matrices:

mat: stores a copy of matrix A

inv: will become the inverse of A (starting as identity matrix)

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

mat[i] = new double[n];

inv[i] = new double[n];

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

mat[i][j] = A[i][j];

inv[i][j] = (i == j) ? 1 : 0;

}

}

➡ We:

Initialize each row of mat and inv.

Copy values from A into mat.

Initialize inv to an identity matrix (1s on the diagonal, 0 elsewhere).

📐 Gaussian Elimination Process

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

double pivot = mat[i][i];

if (fabs(pivot) < 1e-9)

throw runtime\_error("Matrix is singular or non-invertible.");

➡ For each row i:

Get the pivot element mat[i][i].

If it’s too close to zero, the matrix can’t be inverted, so throw an error.

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

mat[i][j] /= pivot;

inv[i][j] /= pivot;

}

➡ Normalize row i by dividing it by the pivot, both in mat and inv.

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

if (k == i) continue;

double factor = mat[k][i];

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

mat[k][j] -= factor \* mat[i][j];

inv[k][j] -= factor \* inv[i][j];

}

}

}

➡ For each other row k (not equal to i):

Subtract a multiple of the pivot row to eliminate mat[k][i] (make it 0).

Apply same operations to inv.

🔄 Convert Inverse to Integer Matrix

int\*\* result = allocateMatrix(n, n);

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

for (int j = 0; j < n; ++j)

result[i][j] = round(inv[i][j]);

➡ Convert the inverse matrix from double to int by rounding each value.

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

delete[] mat[i];

delete[] inv[i];

}

delete[] mat;

delete[] inv;

return result;

➡ Clean up the memory used for temporary matrices mat and inv, then return the integer matrix result.

🧾 Function to Input Matrix from User

int\*\* inputMatrix(int& rows, int& cols, const string& name) {

➡ Function to let the user input a matrix. It takes references to row and column counts and the matrix name (A or B).

cout << "Enter number of rows for " << name << ": ";

cin >> rows;

cout << "Enter number of columns for " << name << ": ";

cin >> cols;

➡ Ask the user to enter the number of rows and columns.

int\*\* mat = allocateMatrix(rows, cols);

cout << "Enter elements of " << name << ":\n";

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

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

cout << name << "[" << i << "][" << j << "] = ";

cin >> mat[i][j];

}

return mat;

}

➡ Allocate the matrix and let the user enter each element. Then return the filled matrix.

🏁 Main Function and Menu

int r1, c1, r2, c2;

int\*\* A = inputMatrix(r1, c1, "A");

int\*\* B = inputMatrix(r2, c2, "B");

➡ Declare dimensions for two matrices, A and B. Input their values from the user.

MatrixOp\* ops[3] = {new AddMatrixOp(), new MulMatrixOp(), new InverseMatrixOp()};

➡ Create an array of matrix operation pointers:

ops[0]: addition

ops[1]: multiplication

ops[2]: inversion

int choice;

int\*\* result = nullptr;

int rr = 0, rc = 0;

➡ Variables to store the menu choice, the result matrix, and its dimensions.

📜 Menu Loop

do {

cout << "\nMenu:\n";

cout << "1. Add A + B\n";

cout << "2. Multiply A \* B\n";

cout << "3. Invert Matrix A\n";

cout << "4. Exit\n";

cout << "Enter choice: ";

cin >> choice;

➡ Show options to the user and let them choose what operation to perform.

✅ Switch Statement – Perform Action

switch (choice) {

case 1:

result = ops[0]->apply(A, B, r1, c1, r2, c2, rr, rc);

cout << "Result of A + B:\n";

printMatrix(result, rr, rc);

freeMatrix(result, rr);

break;

➡ If user chose 1: Perform addition of A and B, print result, and free the memory.

case 2:

result = ops[1]->apply(A, B, r1, c1, r2, c2, rr, rc);

cout << "Result of A \* B:\n";

printMatrix(result, rr, rc);

freeMatrix(result, rr);

break;

➡ If 2: Do multiplication.

case 3:

result = ops[2]->apply(A, nullptr, r1, c1, 0, 0, rr, rc);

cout << "Inverse of A (rounded to int):\n";

printMatrix(result, rr, rc);

freeMatrix(result, rr);

break;

➡ If 3: Do inversion of A (B is not used).

case 4:

cout << "Exiting...\n";

break;

default:

cout << "Invalid choice.\n";

}

➡ If 4: Exit. If another number: Show invalid choice.

} catch (exception& e) {

cout << "Error: " << e.what() << endl;

}

➡ Catch any errors (e.g., incompatible matrix dimensions or non-invertible matrix) and display the error message.

} while (choice != 4);

freeMatrix(A, r1);

freeMatrix(B, r2);

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

delete ops[i];

➡ Continue the loop until user chooses 4 (exit), then:

Free matrices A and B.

Delete each matrix operation object to free memory.

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

delete ops[i];

return 0;

}

**✅ Conclusion**

This project demonstrates a clean application of:

* **Polymorphism**
* **Dynamic memory allocation**
* **Matrix algebra operations**
* **Structured menu-based programs in C++**