Student: Krum Trifonov

Faculty Number: F108883

Task:

Implement a class Matrix that represents a fully-functioning matrix arithmetic. The class must be able to execute all basic matrix operations: sum, subtraction, multiplication, division, determinant calculation. Intrinsically, the matrix is represented by a dynamic two-dimensional array of floating-point numbers. The class must have the following members:

- default constructor;
- constructor with parameters;
- copy constructor;
- destructor;
- overloaded assignment operator;
- overloaded arithmetical operators: +, -, *, /;
- overloaded stream extraction >> and insertion operators <<;
- determinant

Solution:

To begin with, we will split the program into three different parts:

1. We will use Matrix.h to **define the class and the member functions**.
2. We will use Matrix.cpp to **implement the member functions.**
3. We will use main.cpp to **execute the functions**.

Let's make one thing clear from the start- there is no such thing as "matrix division".

The closest we can get to dividing two matrices is getting the inverted matrix of one of them and then multiplying the other matrix by that same inverted matrix.

For example, assume we have matrix A and matrix B. The way we "divide" them, as in A / B, is:

1. Get the inverted matrix of B. So, we need to get B^-1.
2. Multiply matrix A by B^-1.
3. The result will be matrix C which is defined as: C = A * B^-1 .

This is the exact way matrix division is handled within this program.

Now that we've clarified the way division is handled, we can proceed with the rest of the code, which is fairly simple, with a few exceptions:

What's contained within Matrix.h:

Firstly, we need to define our data structures and variables. Given that the only operations we will be executing will be performed on matrices, the only variables we need are:

1. A variable of type double holding the number of **rows**.
2. A variable of type double holding the number of **columns**.
3. A **two-dimensional vector** which will represent the matrices.

Secondly, we need to initialize:

1. A default constructor.
2. A constructor with parameters.
3. Copy constructor.
4. Destructor.
5. Overloaded member functions for the operators: = , + , - , * , % (this one is used for the inverse of the matrix).

And finally, we need to initialize a function that will take care of the determinant.

What's contained within Matrix.cpp:

At the beginning of the file we have all the constructors and the destructor. As those are very simple, we will skip the in-depth explanation of them.

Just know that for the copy constructor we're using "other" as the name of the matrix that will be input. (therefore the variables in the copy constructor are other.rows, other.columns etc..)

As a brief summary as to how the overloaded functions are designed:

Every overloaded function overloads the said operator, which is noted in the title of the function, for the Matrix class.

Each overloaded function also takes a constant reference to another Matrix object (other) as a parameter, which will be the matrix to multiply with the current matrix (*this).

And finally, each overloaded function returns a new Matrix object that represents the result of the operation it is meant to perform.

What comes next is the **input overload function:**

Inside of it we take the user input for the rows and columns and store it in the respective variables. After that the user is asked to input every element of the matrix.

After that we have the **output overload function**:

This one is used simply used for printing out the matrices by row and column.

After that we have the **overloaded assignment operator function**:

Here we first check if the current object (this) is the same as the object being assigned (other). It ensures that self-assignment is handled correctly. If the two objects are the same, there's no need to perform any assignment, so the function simply returns the current object.

If the objects are not the same, the code proceeds to copy the data from the other object to the current object. It assigns the rows and columns values of other to the corresponding member variables of the current object. It also assigns the data pointer of (other) to the data pointer of the current object. Note that this is a shallow copy, meaning the two objects will end up pointing to the same memory location.

Finally, the function returns a reference to the current object (*this). This allows chaining of assignment operations.

After that we have the **overloaded sum operator function**:

We first create a new Matrix object called result with the same number of rows and columns as the current matrix. It initializes the result matrix with the same dimensions.

Then we check if the dimensions of the current matrix and the other matrix are different. If they are not the same, it means the matrices cannot be added together. An error message is printed to the console, and the result matrix (initialized earlier) is returned. The returned

matrix will have the same dimensions as the current matrix, but its data will remain uninitialized. If the dimensions of the matrices match, this line resizes the result.data vector to match the number of rows and columns. It ensures that the result matrix has enough space to store the summed values.

Then we have two nested loops which iterate over each element of the matrices. For each element at position (i, j), it adds the corresponding elements from the current matrix (data[i][j]) and the other matrix (other.data[i][j]) and stores the sum in the corresponding position of the result matrix (result.data[i][j]).

And finally, the result matrix, which contains the summed values, is returned.

After that we have the **overloaded subtract operator function**:

This one is almost the same as the sum function, however instead of adding each element at position (i, j) of the current matrix (data[i][j]) and the other matrix (other.data[i][j]), it subtracts them.

After that we have the **overloaded product operator function**:

To start with, the function checks if the number of columns in the current matrix is not equal to the number of rows in the other matrix. If they are not equal, it means matrix multiplication is not possible. An error message is displayed, and the result matrix (initialized earlier) is returned. The returned matrix will have the same number of rows as the current matrix and the same number of columns as the other matrix, but its data will remain uninitialized.

If the matrices' dimensions allow multiplication, this line resizes the result.data vector to match the number of rows of the current matrix and the number of columns of the other matrix. It ensures that the result matrix has enough space to store the multiplied values.

After that we implement three nested loops which perform the actual matrix multiplication. The outer loop iterates over the rows of the current matrix, the middle loop iterates over the columns of the other matrix, and the innermost loop iterates over the columns of the current matrix (or the rows of the other matrix, they are equivalent in this context). It calculates the dot product of the i-th row of the current matrix (data[i][k]) and the k-th column of the other matrix (other.data[k][j]). The dot product is accumulated in the corresponding position of the result matrix (result.data[i][j]).

And finally, the result matrix, which contains the multiplied values, is returned.

After that we have the **overloaded modular division function, which is used for inversion**:

First there is a condition that checks if the matrix is square by comparing the number of rows and columns. If they are not equal, it means the matrix is not square, and an exception is thrown with a runtime error indicating that a square matrix is required to calculate its inverse.

Then a new Matrix object called identity is created, with the same number of rows and columns as the original matrix. It will be used to store the identity matrix, which is initialized later.

After that the nested loops iterate over the elements of the identity matrix and set the value at position (i, j) to 1.0 if i is equal to j, representing the main diagonal. Otherwise, it sets the value to 0.0.

Then, a copy of the original matrix (other) is initialized to perform the operations. The copy is stored in the matrix variable.

The code then proceeds with Gaussian elimination and back substitution to calculate the inverse of the matrix using the provided **algorithm** (*a thorough explanation of the algorithm is provided in a separate file called* ***gaussian_algorithm.pdf***).

Finally, the inverse matrix (identity) is returned.

After that we have the **function that calculates the determinant of a matrix**:

Here, once again, the Gaussian elimination method is used to get the determinant.

The function first checks if the matrix is square by comparing the number of rows and columns. If they are not equal, it means the matrix is not square, and a determinant cannot be calculated. In this case, a message is printed, and the function returns 0.0.

The code clones the vector data holding the values of the matrix into a temporary vector tempData. This is done to perform the Gaussian elimination operations on the cloned data while preserving the original matrix.

The variable n is assigned the number of rows (or columns) of the matrix, representing its size.

The variable det is initialized as 1.0. This variable will hold the determinant value and is initially set to the identity value for multiplication.

The outer loop iterates over the rows of the matrix (i represents the current row).

Within each iteration, the algorithm finds the row with the maximum absolute value for the current column (i). It starts from the current row (i) and searches for a row with a larger absolute value at the current column (j = i + 1).

If a row with a larger absolute value is found (abs(tempData[j][i]) > abs(tempData[maxRow][i])), the maxRow index is updated.

If the maximum absolute value row (maxRow) is different from the current row (i), it means a row swap is required to bring the maximum value to the current row. The algorithm swaps the rows in the tempData vector and updates the determinant by multiplying it by -1.0 (swapping rows changes the sign of the determinant).

If the diagonal element (tempData[i][i]) is zero, it means the matrix is singular, and the determinant is zero. In this case, the function returns 0.0 immediately.

The determinant is updated by multiplying it with the diagonal element (tempData[i][i]).

The algorithm performs row operations to eliminate the elements below the diagonal in the current column. It iterates over each row below the current row (j = i + 1) and calculates a factor (factor) by dividing the value of the element in the current row (tempData[j][i]) by the diagonal element (tempData[i][i]). It then subtracts a scaled multiple of the current row from the subsequent rows, making the elements below the diagonal zero.

After completing the Gaussian elimination process, the determinant value is returned.

What's contained within main.cpp:

The main function is fairly short and simple.

It begins with initializing the two matrices the program will use to perform all of the operations.

- Matrix A
- Matrix B

Then the code calls for the user input using "cin>>" for both matrices, which is handled by the overloaded function.

Once the input is complete a total of eight new matrices are initialized:

- Matrix C which is the sum of A and B
- Matrix D which is the subtraction of A and B
- Matrix E which is the product of A and B
- Matrix J which is the product of B and A
- Matrix F which is the inverse matrix of A
- Matrix G which is the inverse matrix of B
- Matrix H which is the "division" of A and B
- Matrix I which is the "division" of B and A

All of the matrices use the already mentioned and discussed overloaded operators = , + , - , * , % .

P.S.

If you can see this capybara, you're a really good and likable teacher. (this statement holds true even if you don't see it)