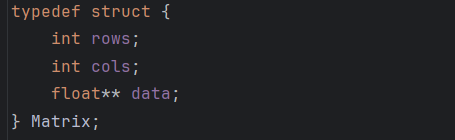
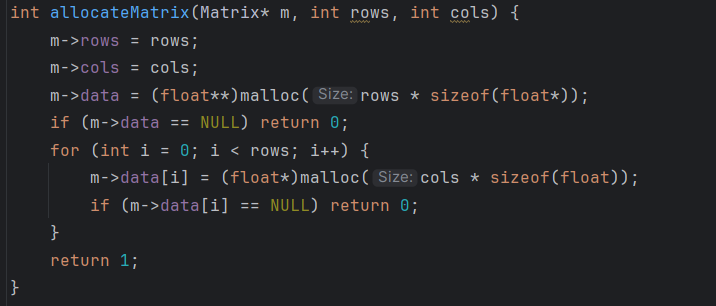
**Code Documentation**

**Matrix Structure Definition**



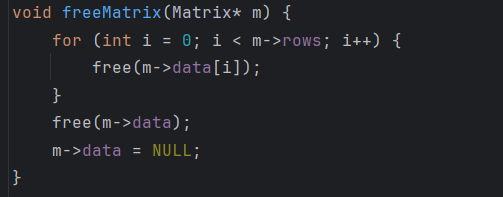
The Matrix struct defines a dynamic matrix by storing its number of rows, number of columns, and a pointer to a 2D array of floating-point elements.

**Dynamic Memory Allocation**



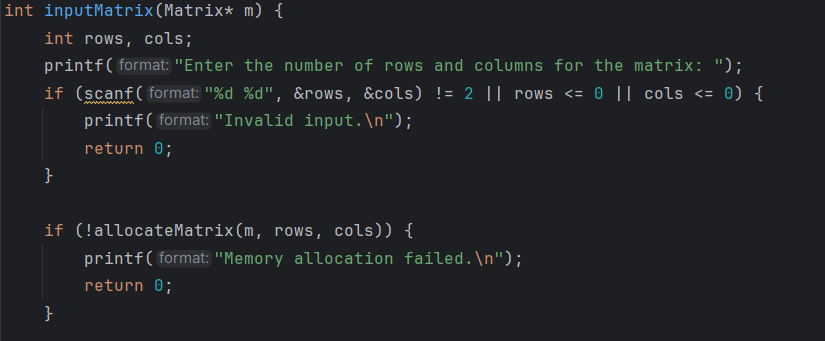
The “allocateMatrix” function creates space in memory for a matrix of floating-point numbers. It takes the number of rows and columns and saves them in the matrix structure. Then, it allocates memory for each row and for the elements in each row. If any memory allocation fails, the function returns 0 to show an error. If everything works, it returns 1 to show success.

**Freeing the Memory**

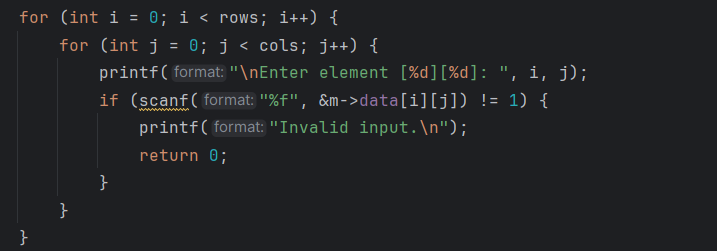


Here we free the memory we allocated earlier. We free memory to avoid leaks.

**Getting the Input**

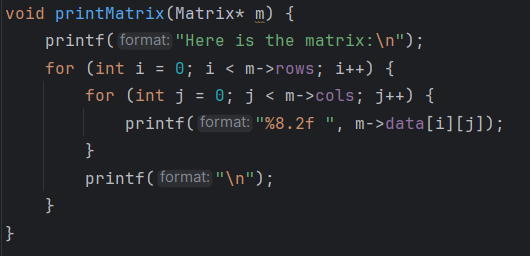


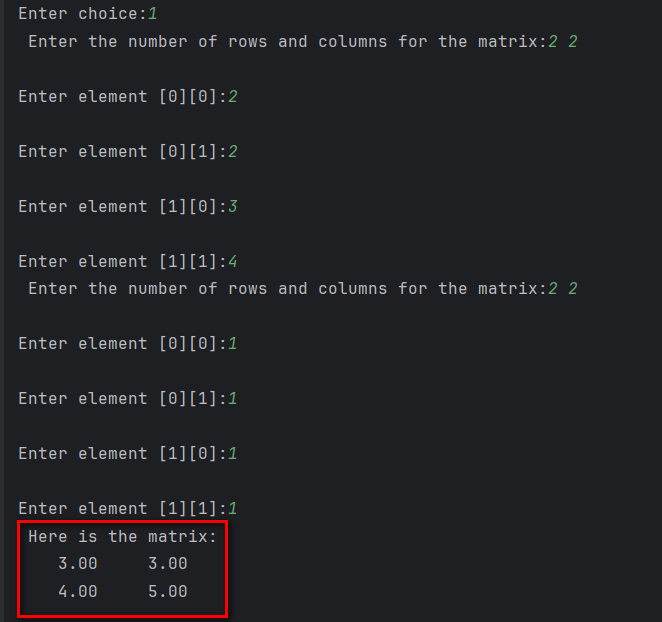
This takes a pointer to a Matrix structure as input. It prompts the user to enter the number of rows and columns for the matrix, reads these values, and validates the input to ensure exactly two positive integers are provided.



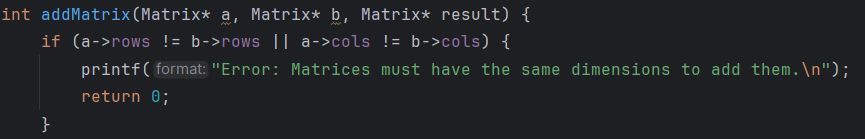
User enters the number of rows and columns, and proceeds asking for the elements of the matrix

**Printing the Matrix**



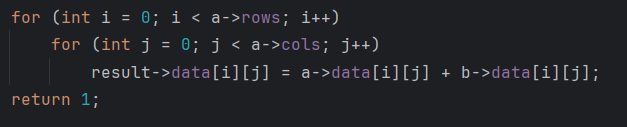
Prints each matrix element formatted to two decimal places.

**Addition of Matrices**

  
If the rows of the “a” matrix is not equal to the rows of the “b” matrix, or their columns aren’t equal to each other, the addition would fail.

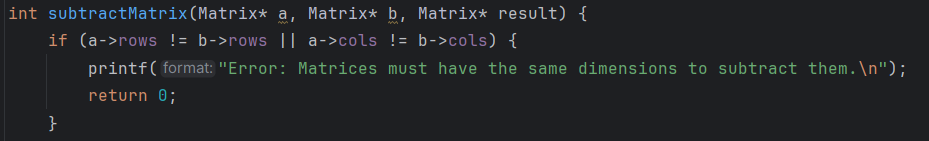


This line checks if there was memory allocated, if no it returns 0.



Adds matrices of the same size to each other and stores it in a matrix.

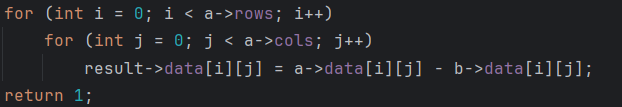
**Subtraction of Matrices**



If the rows of the “a” matrix is not equal to the rows of the “b” matrix, or their columns aren’t equal to each other, the subtraction would fail.

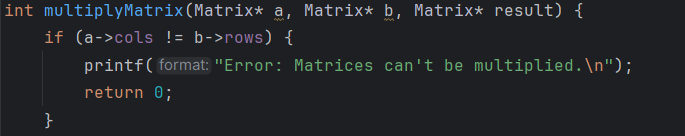


This line checks if there was memory allocated, if no it returns 0.



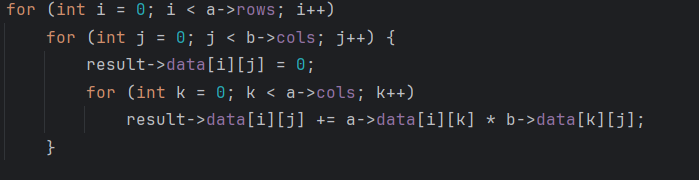
Subtracts matrices of equal dimensions and stores result in another matrix.

**Matrix Multiplication**

If the columns of matrix “a” is not equal to the rows of matrix “b”, the multiplication would fail.

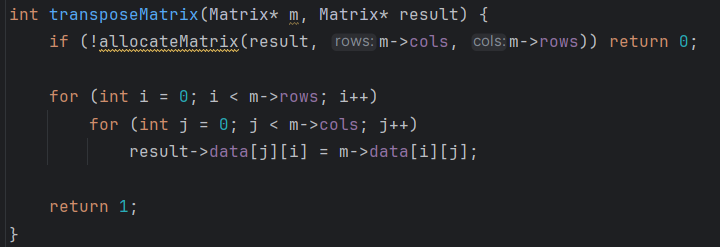


This line checks if there was memory allocated, if no it returns 0.



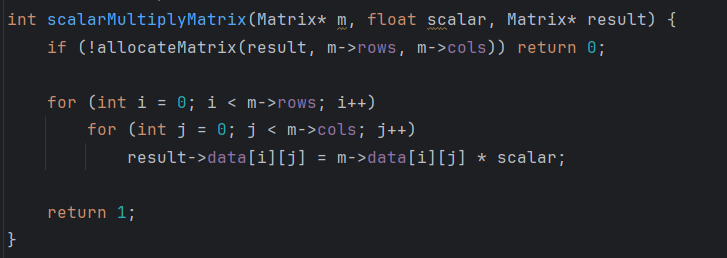
This loop performs matrix multiplication by computing each element of the result matrix as the dot product of a row from matrix “a” and a column from matrix “b”. It goes through each row of “a” and each column of “b”, initializing the result cell to zero, then accumulates the sum of products between corresponding elements using a third loop, following the standard formula for matrix multiplication.

**Transposing the Matrix**



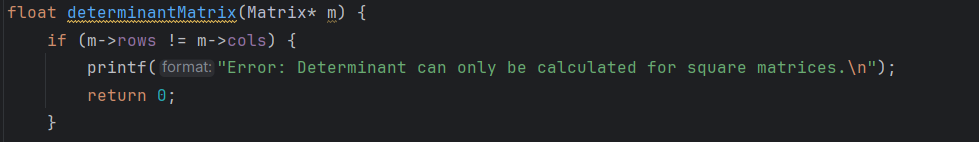
First line checks if there was memory allocated, if no it returns 0. The Other lines swap rows and columns of the original matrix. The resulting matrix will have the columns as the rows of the original matrix, and the rows as the columns of the original matrix (due to them being swapped for the matrix to be transposed).

**Scalar Multiplication**

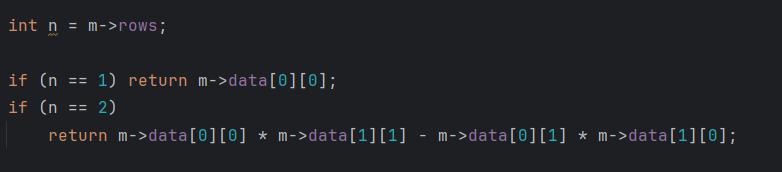


First line checks if there was memory allocated, if no it returns 0. The other lines multiplies every matrix element by a scalar value.

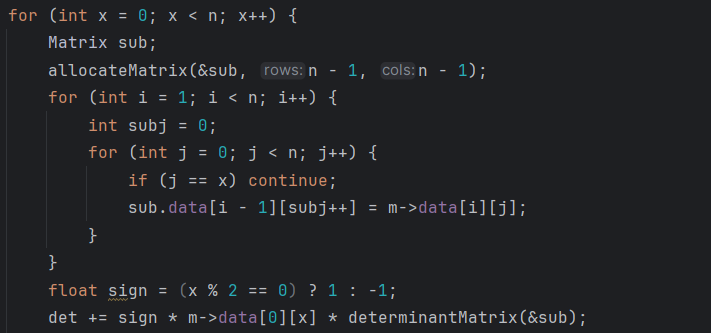
**Determinant Calculation**



If rows and columns are equal, meaning it is not a square matrix, the calculation of the determinant would fail.



If the number of rows is 1, meaning there is only one element in the matrix, the determinant would be that element. If the number of rows is 2, meaning 2 rows and 2 columns, the determinant would be first element of the first row multiplied by the second element of the second row subtracted the second element of the first row multiplied by the first element of the second row.



This part constructs a submatrix by excluding the first row and the current column x, which is used to recursively compute the determinant of an n by n matrix. For each column x in the first row, it creates a smaller (n-1) by (n-1) matrix called sub, which skips column x and the first row, storing the remaining elements row by row into sub.data to be used in the recursive call. 

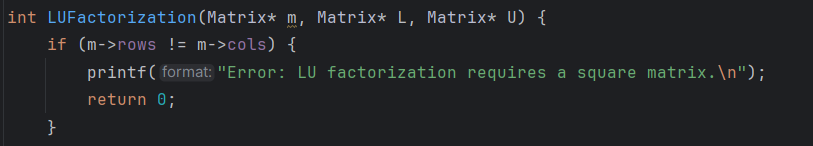
This code calculates the determinant of a matrix using cofactor expansion along the first row. It determines the sign based on whether the column index is even or odd, alternating between +1 and -1. It then multiplies this sign by the corresponding matrix element in the first row and the determinant of the submatrix formed by removing the current row and column. The result is added to the running total of the determinant.

This method is called Laplace Expansion.

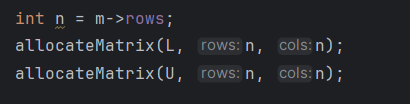


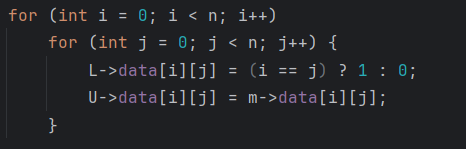
After calculations, we free the memory allocated for the “sub” matrix

**LU Factorization**

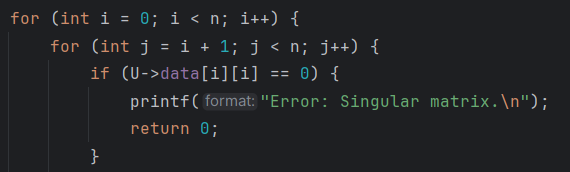


If rows and columns are equal, meaning it is not a square matrix, the calculation of the LU Factorization would fail.

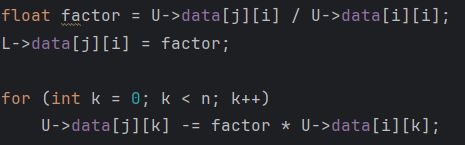
  
Here we allocate memory for the L (Lower triangular) and U (Upper triangular) Matrices.



Here we make the “U” matrix equal to our original. And If the element is in the diagonal of the L matrix, it must be 1, if not diagonal 0.

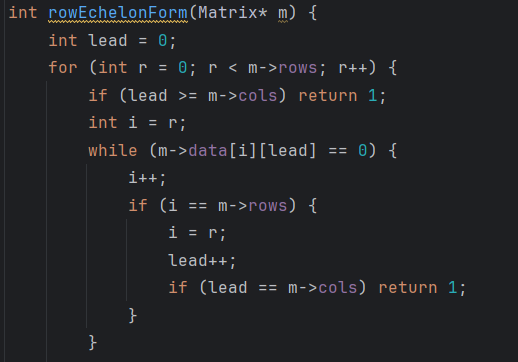


Here we check if the matrix is singular or not (singular means having 0 as a determinant). As LU Factorization can be applied only to matrices which are not singular.

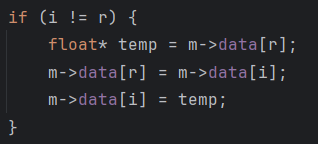


This part iteratively eliminates elements below the pivot in the U matrix and stores the corresponding multipliers in the L matrix. If a pivot element is zero, the matrix is singular and the process stops with an error message.

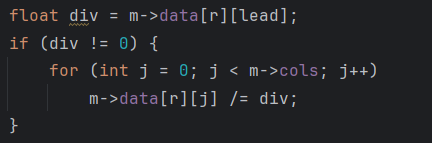
**Row Echelon Form**



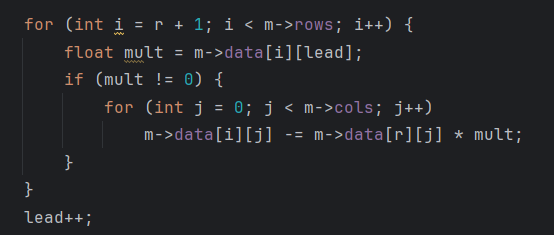
This function converts a matrix to row echelon form by finding pivot elements in each column. If the current pivot is zero, it searches rows below to swap. If a column has no pivot, it moves to the next. The process stops when all rows are processed or columns are exhausted.



This block swaps row r with row i if they are different. It iterates through each column and exchanges corresponding elements to bring a non-zero pivot row to the current working row position.

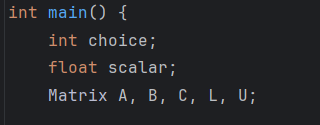


This part divides all the elements of the row by the leading pivot value, ensuring the pivot becomes 1.

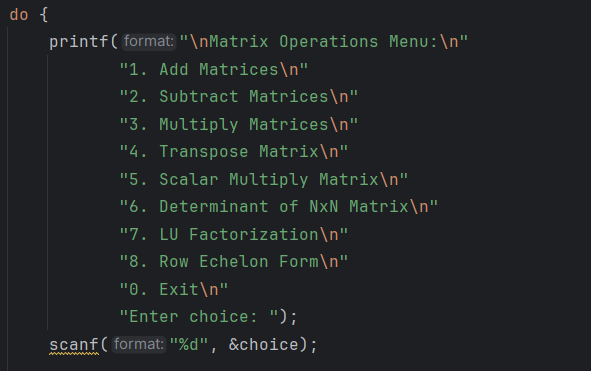


It eliminates elements below the pivot in a column by subtracting multiples of the pivot row from the rows below, getting the row echelon form.

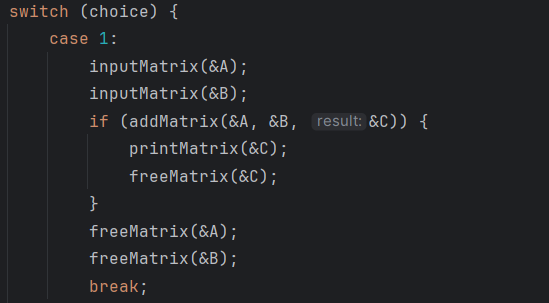
**Main**



Here we declare the variables and Matrices needed for the functions.



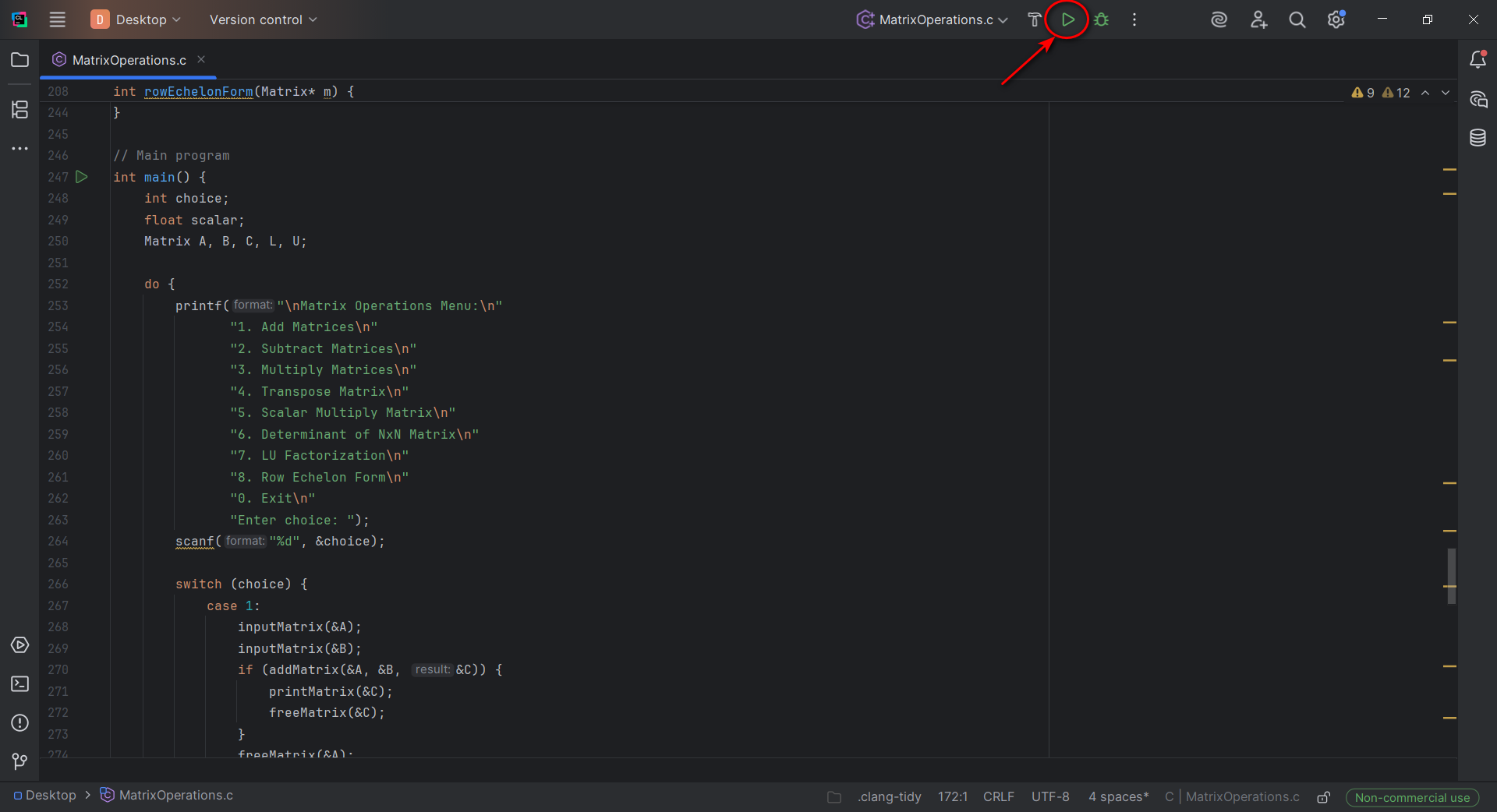
Here we print the operations and based on the number user inputs, the code will perform the corresponding operation. The program will constantly ask the user for inputs unless 0 is entered.



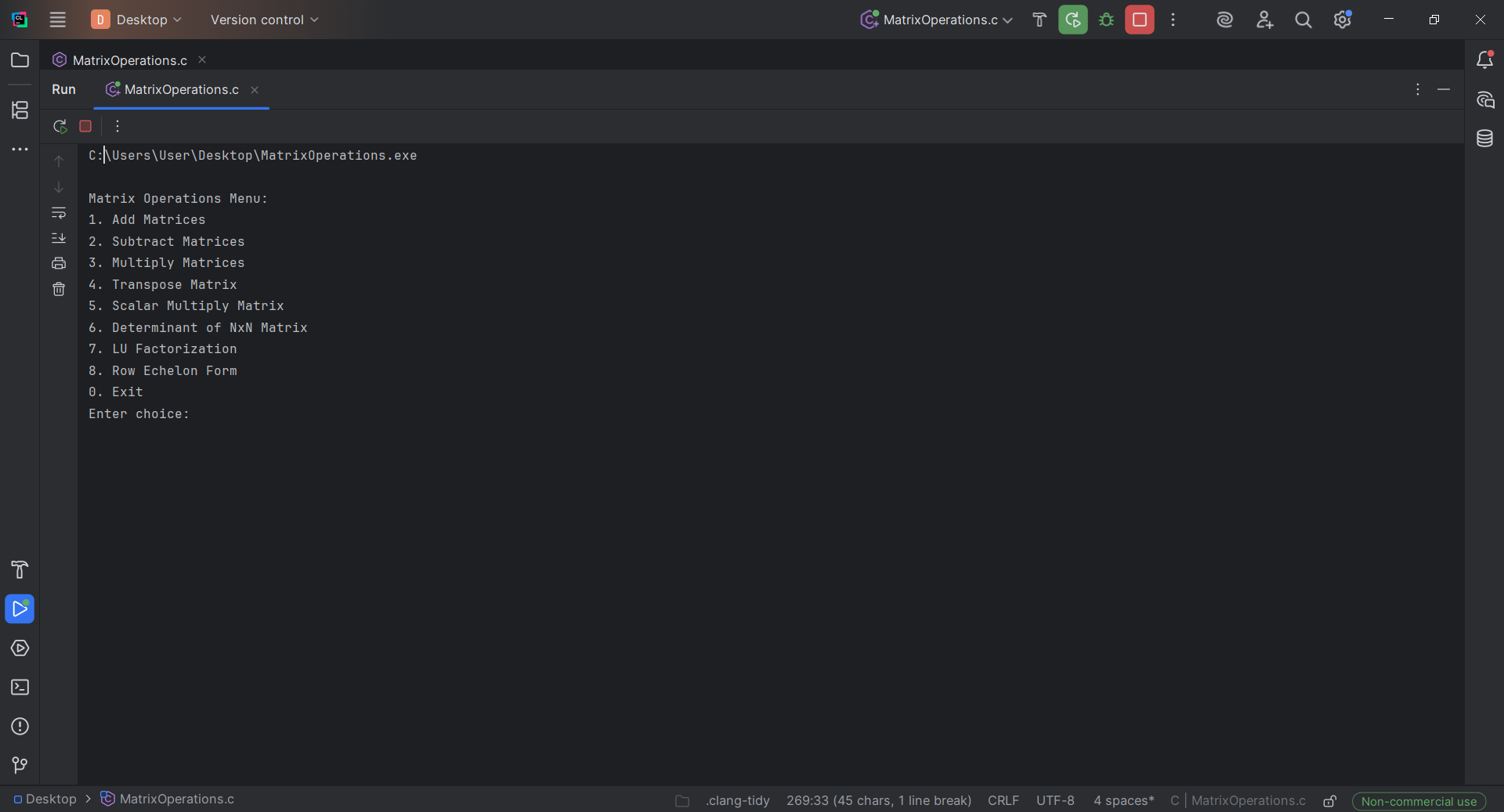
This is the function for addition, however the rest have more or less the same structure. After the printing of the result, we free the memory we allocated earlier.

**Instructions**

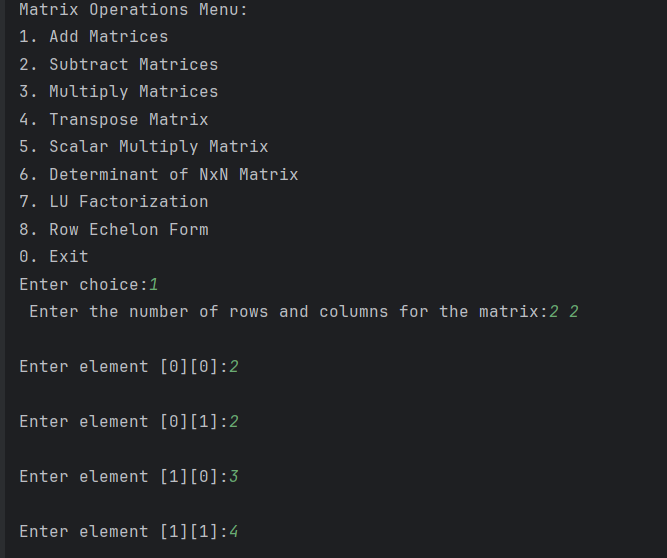
Here I will talk about how to use the code.



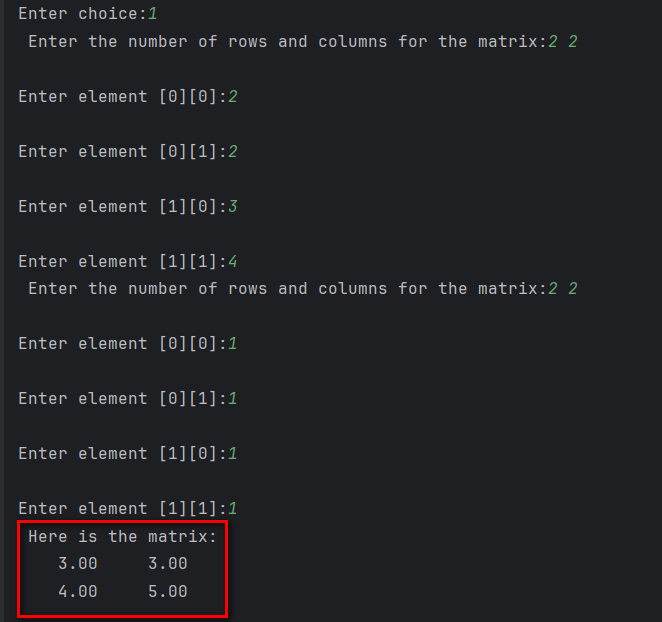
When opening the program, press the Run button.



Here you can see the list of Operation available. Enter the number corresponding to the operation you want to do.



After entering the number, you must input the dimensions of the matrix and after that the elements of that matrix. In case of addition, subtraction and matrix multiplication, you would be asked to enter another matrix (same steps: entering dimensions and elements). In case of scalar multiplication, you would be asked to enter a scalar. For the rest of operations, the first matrix is enough.



As we enter everything the operation needs, after the message “Here is the matrix:” the result of the operation is printed.