# Assignment 1-Matrix Operations

Problem Statement:

Write C/C++ program for storing matrix. Write functions for a) Check whether given matrix is upper triangular or not

1. Compute summation of diagonal elements
2. Compute transpose of matrix
3. Add, subtract and multiply two matrices

## Concepts Used

* Memory Allocation

Static

Dynamic

* Creating a 2-D array

## Theory of Concepts Used

### Memory Allocation

### Staic Memory Allocation

The first type of memory allocation is known as a static memory allocation, which corresponds to file scope variables and local static variables. The addresses and sizes of these allocations are fixed at the time of compilation1 and so they can be placed in a fixed-sized data area which then corresponds to a section within the final linked executable file. Such memory allocations are called static because they do not vary in location or size during the lifetime of the program.

There can be many types of data sections within an executable file; the three most common are normal data, BSS data and read-only data. BSS data contains variables and arrays which are to be initialised to zero at run-time and so is treated as a special case, since the actual contents of the section need not be stored in the executable file. Read-only data consists of constant variables and arrays whose contents are guaranteed not to change when a program is being run.

Static Allocation means, that the memory for your variables is allocated when the program starts. The size is fixed when the program is created. It applies to global variables, file scope variables, and variables qualified with static defined inside functions.

### Dynamic Memory Allocation

Dynamic memory allocation is a bit different. You now control the exact size and the lifetime of these memory locations. If you don’t free it, you’ll run into memory leaks, which may cause your application to crash, since it, at some point cannot allocation more memory.

*int* ∗ *func*() *int* ∗ *mem* = *malloc*(1024); *returnmem*;

*int* ∗ *mem* = *func*();*/* ∗ *stillaccessible* ∗ */*

In the upper example, the allocated memory is still valid and accessible, even though the function terminated. When you are done with the memory, you have to free it:

*free*(*mem*);

### Creating a 2-D Array

An array keeps track of multiple pieces of information in linear order, a onedimensional list. However, the data associated with certain systems (a digital image, a board game, etc.) lives in two dimensions. To visualize this data, we need a multi-dimensional data structure, that is, a multi-dimensional array. A two-dimensional array is really nothing more than an array of arrays (a threedimensional array is an array of arrays of arrays).

Multidimensional arrays are not limited to two indices (i.e., two dimensions). They can contain as many indices as needed. Although be careful: the amount of memory needed for an array increases exponentially with each dimension.

For example:

char century [100][365][24][60][60]; declares an array with an element of type char for each second in a century. This amounts to more than 3 billion char! So this declaration would consume more than 3 gigabytes of memory!

At the end, multidimensional arrays are just an abstraction for programmers, since the same results can be achieved with a simple array, by multiplying its indices:

int jimmy [3][5]; // is equivalent to int jimmy [15]; // (3 \* 5 = 15)

## Algorithm

(Matrix Addition) MATADD (iMatrix1,iMatrix2, M, P)

Let iMatrix1 be an M\*P matrix array, and let iMatrix2 be a M\*P matrix array.

This algorithm stores the addition of iMatrix1 and iMatrix2 in an M\*P matrix array iMatrix3.

Step 1: Repeat steps 2 and amp: 3 for I=1 to M:

Step 2: Repeat Step 3 for J=1 to P:

Step 3: iMatrix3 [I, J]:= iMatrix1 [I, J]+ iMatrix2 [I, J]

[End of Step 2 inner loop.]

[End of Step 1 outer loop.]

Step 5: Exit

(Matrix Multiplication) MATMUL (iMatrix1,iMatrix2, M, P, N)

Let iMatrix1 be an M\*P matrix array, and let iMatrix2 be a P\*N matrix array.

This algorithm stores the product of iMatrix1 and iMatrix2 in an M\*N matrix array iMatrix3.

Step 1: Repeat steps 2 to 4 for I=1 to M:

Step 2: Repeat Steps 3 and 4 for J=1 to N:

Step 3: Set iMatrix3 [I, J]: =0. [Initializes iMatrix3 [I, J].] Step 4 Repeat for K=1 to P:

iMatrix3 [I, J]:= iMatrix3 [I, J]+ iMatrix1[I, K]\*iMatrix2 [K, J]

[End of inner loop.]

[End of Step 2 middle loop.]

[End of Step 1 outer loop.]

Step 5: Exit

(Matrix Transpose) MATTRANS (iMatrix1, M, P)

Let iMatrix1 be an M\*P matrix array. This algorithm stores the transpose of iMatrix1 in P\*M array iMatrix.

Step 1: Repeat steps 2 and amp:3 for I=1 to M: Step 2: Repeat Step 3 for J=I to P:

Step 3: iMatrix2 [I, J]:= iMatrix1 [J, I]

[End of Step 2 inner loop.]

[End of Step 1 outer loop.]

## Flowchart

Output:

