

## Introduction to Computer Programming and Data Structures

### Assignment 04

Maximum Marks: **100**

Submission Deadline: **2022-Sep-09**

Bonus: **20** – for well indentation, variable name, programming style and following the check-points

**Topic:** Multi-dimensional arrays, matrix operations and command line arguments.

#### Assignment problem # AP0401

- Matrix Library: Consider you are building a library for matrix operations. Let the file be “myMatrix.c”. The file should have the following functions.

1. `void ** my_malloc_2D( int n, int m, int unit_size)`: It takes dimension of any 2D matrices  $A_{n \times m}$  and allocates memory for that matrix and returns the pointer to the allocated memory as `void **`. It returns NULL in case of failure.

Here, `unit_size` is the no of bytes in each memory. E.g., for `int`, `unit_size` = 4 and for `char`, `unit_size` = 1. Hint: after a matrix is allocated, it can be typecast as `A= (int **)my_malloc_2D(n,m, sizeof(int))`. Note that this 2D Memory allocation can be applied to any structure too.

2. `void *** my_malloc_3D( int m, int n, int p, int unit_size)`: It takes dimension of any 3D matrices  $A_{m \times n \times p}$  and allocates memory for that matrix and returns the pointer to the allocated memory as `void ***`. It returns NULL in case of failure.

3. `void show_2D_matrix(void **A, int n, int m, char type)`: It takes a 2D array pointer and its dimension and type of variable indicator `type`. It prints the elements of the matrix  $A_{n \times m}$ .

Here, consider the following types.

`a` → `char`, `b` → `int`, `c` → `unsigned int`, `d` → `long`, `e` → `unsigned long`, `f` → `float`, `g` → `double`, `h` → `long double`.

4. `void show_3D_matrix(void ***A, int m, int n, int p, char type)`: It takes a 3D array pointer and its dimension and type of variable indicator `type`. It prints the elements of the matrix  $A_{m \times n \times p}$ .

5. `scan_2D_matrix_from_opened_file(void **A, int n, int m, char type, FILE * inp_file_ptr)`: It takes a 2D array pointer, its dimension and type of variable indicator `type`. It scans the elements of the matrix  $A_{n \times m}$  from the file opened already in `inp_file_ptr`. It returns 0 in case of failure.
6. `scan_3D_matrix_from_opened_file(void **A, int m, int n, int p, char type, FILE * inp_file_ptr)`: It takes a 3D array pointer, its dimension and type of variable indicator `type`. It scans the elements of the matrix  $A_{m \times n \times p}$  from the file opened already in `inp_file_ptr`. It returns 0 in case of failure.
7. `scan_2D_matrix_from_unopened_file(void **A, char type, char inp_file_name[])`: It takes a 2D array pointer and type of variable indicator `type`. First, it scans the dimensions  $m, n$  of the matrix, then it scans the elements of the matrix  $A_{n \times m}$  from the file `inp_file_name`. It returns 0 in case of failure.
8. `scan_3D_matrix_from_unopened_file(void **A, char type, char inp_file_name[])`: It takes a 3D array pointer, and type of variable indicator `type`. First, it scans the dimensions  $m, n, p$  of the matrix, then it scans the elements of the matrix  $A_{m \times n \times p}$  from the file `inp_file_name`. It returns 0 in case of failure.
9. `void ** matrix_mult(A, B, m, n ,p, type)`: This takes 2 matrices and outputs its products in a new matrix. Note that typecasting is mandatory after receiving the result.
10. Write your own main function that tests matrix multiplication. This can be done as follows.
  - (a) Suppose each input matrix is kept in a separate file. For example, let “input\_matrix\_a.txt” and “input\_matrix\_b.txt” be two files, each of which is kept a 2D matrix.
  - (b) Suppose the program file `myMatrix.c` is compiled. Then run as  
`./a.out input_matrix_a.txt input_matrix_b.txt b`.  
 The program should output the product of the two matrices kept in “input\_matrix\_a.txt” and “input\_matrix\_b.txt”. Thus, for multiplication, pass the name of the matrix files from the command line. The third argument indicates the type of variables of the matrix entries.
  - (c) The main function first reads the first file, scans dimensions of the matrix, allocates memory for that matrix, scan the matrix entries say in  $A$ . Then it does the same for the next file, say in  $B$ .
  - (d) Then it calls `void ** matrix_mult(A, B, m, n ,p, type)` to multiply the matrices and prints the result matrix in the terminal.
  - (e) Try to return an error message in case of any failure.

[100]

## Submission checkpoints

1. All functions must be declared before defining any.
2. Any repeating work must be done in a function. Main function is only for testing the other defined functions.
3. Check indentation
4. Throw error and exit if any of `scanf` or `fscanf` is failed.
5. Take inputs from files only
6. submit your `input_file` along with each problem. If AP0104 is the problem ID, then the input file must be “`input_AP0104_stName`”. `stName` indicates student name as usual.
7. make `free` all dynamically allocated memories.

8. Any 2D input matrix should be kept in file as follows

$$\begin{matrix} n & m \\ a_{11} & a_{12} & \dots & a_{1m} \\ a_{21} & a_{22} & \dots & a_{1m} \\ \vdots & & & \\ a_{n1} & a_{n2} & \dots, & a_{nm} \end{matrix}$$

9. Any 3D input matrix should be kept in file as following

$$\begin{matrix} n & m & p \\ a_{111} & a_{121} & \dots & a_{1m1} \\ a_{211} & a_{221} & \dots & a_{1m1} \\ \vdots & & & \\ a_{n11} & a_{n21} & \dots, & a_{nm1} \\ \\ a_{112} & a_{122} & \dots & a_{1m2} \\ a_{212} & a_{222} & \dots & a_{1m2} \\ \vdots & & & \\ a_{n12} & a_{n22} & \dots, & a_{nm2} \\ \\ a_{11p} & a_{12p} & \dots & a_{1mp} \\ a_{21p} & a_{22p} & \dots & a_{1mp} \\ \vdots & & & \\ a_{n1p} & a_{n2p} & \dots, & a_{nmp} \end{matrix}$$

10. A single input file may contain multiple matrices. However, the dimensions must be kept before matrix entries.

[40]