C SC 230 – Summer 2014 – Assignment 1 part 2 (Programming)

Due Monday, June 2, by 5:00 p.m.

Total Marks = 50 (30 for execution, 20 code design/documentation)

1. The problem to be solved and implemented with a C program

Given a 2-dimensional matrix of integers, check whether it is symmetric, skew-symmetric or orthogonal.

You are asked to do computation on 2-dimensional matrices. You need to read the input from a file containing integer data which represent the elements of matrices. For each matrix you must compute its transpose and check whether the matrix is symmetric, skew-symmetric or othogonal (or none of them). A set of matrices are to be read from an input file until end-of-file is reached.

2. Background definitions

The following definitions from linear algebra are useful to understand the domain of your application.

2.1. Transpose of a matrix

Given a 2-dimensional matrix A of integers, of size $m \times n$, (where m is the number of rows and n is the number of columns), the matrix A^T is called the ***transpose*** of A, where the rows of A are the columns of A^T. An example is given in Figure 1. More formally, the transpose of an $m \times n$ matrix A is the $n \times m$ matrix A^T where:

$$A_{ij}{}^{T} = A_{ji} \text{ for } (1 \le i \le n),(1 \le j \le m)$$

The matrix M1 (*3 x 4*):

$$M1_{3\times 4} = \begin{bmatrix} 1 & 2 & 3 & 4 \\ -5 & -6 & 7 & 8 \\ 9 & 10 & -11 & 12 \end{bmatrix}$$

and its transpose $M1^T$ (*4 x 3*):

$$M1^{T}{}_{4\times 3} = \begin{bmatrix} 1 & -5 & 9 \\ 2 & -6 & 10 \\ 3 & 7 & -11 \\ 4 & 8 & 12 \end{bmatrix}$$

Figure 1: An example matrix "M1" of size (3x4) and its transpose $M1^T$ of size (4x3)

2.2. Symmetric matrix

A *square* matrix *A* is defined to be ***symmetric*** if it is equal to its transpose, that is: $A = A^T$ as shown in the example of Figure 2.

$$M2_{3\times 3} = \begin{bmatrix} 3 & 2 & -1 \\ 2 & -5 & 17 \\ -1 & 17 & 4 \end{bmatrix} \qquad M2^T{}_{3\times 3} = \begin{bmatrix} 3 & 2 & -1 \\ 2 & -5 & 17 \\ -1 & 17 & 4 \end{bmatrix}$$

Figure 2: The square matrix M2 (*3 x 3*) is symmetric, since M2 = $M2^T$

2.3. Skew-symmetric matrix

A *square* matrix *A* is defined to be ***skew-symmetric*** if its transpose is equal to its negative matrix, that is: $-A = A^T$ as shown in the example of Figure 3.

$$M3_{3\times 3} = \begin{bmatrix} 0 & 6 & 4 \\ -6 & 0 & 5 \\ -4 & -5 & 0 \end{bmatrix} \qquad M3^T{}_{3\times 3} = \begin{bmatrix} 0 & -6 & -4 \\ 6 & 0 & -5 \\ 4 & 5 & 0 \end{bmatrix}$$

Figure 3: The square matrix M3 (*3 x 3*) is skew-symmetric, since M3 = -$M3^T$

2.4. Matrix multiplication

Matrix multiplication is defined between two matrices only if the number of columns of the first matrix is the same as the number of rows of the second matrix. If *A* is an $m \times n$ matrix and *B* is an $n \times p$ matrix, then their product is an $m \times p$ matrix denoted by *AB* (or sometimes $A \cdot B$). If $C = AB$, and $c_{i,j}$ denotes the entry in *C* at position (*i*,*j*), then

$$c_{i,j} = \sum_{r=0}^{n-1} a_{i,r} b_{r,j} = a_{i,0} b_{0,j} + a_{i,1} b_{1,j} + \ldots + a_{i,n-1} b_{n-1,j}$$

An example of matrix multiplication is shown in Figure 4.

If A (*3 x 4*) is:

$$A_{3\times 4} = \begin{bmatrix} 11 & 12 & -13 & 14 \\ -21 & 22 & 23 & 24 \\ 31 & 32 & -33 & -34 \end{bmatrix}$$

and B (*4 x 3*) is:

$$B_{4\times 3} = \begin{bmatrix} 0 & 1 & 2 \\ 10 & 11 & 12 \\ 20 & 21 & 22 \\ 30 & 31 & 32 \end{bmatrix}$$

then C = A x B is:(*3 x 3*)

$$C_{3\times 3} = \begin{bmatrix} 280 & 304 & 328 \\ 1400 & 1448 & 1496 \\ -1360 & -1364 & -1368 \end{bmatrix}$$

Figure 4: Example 1 of matrix multiplication

In case you are not familiar with matrix multiplication, you should look it up. Here is only a short summary of how it is done for the small example above, by considering how each element c[i,j] of the matrix C is computed. The rows are numbered from *0* to *n-1* and the columns from *0* to *m-1*, given that *n* is the number of rows and *m* the number of columns (following the indexing in C arrays).
The element c[0,0] is the result of multiplying each element of row 0 of matrix A with each element of column 0 of matrix B and summing the result. Thus we have:

$$c[0,0] = (a[0,0] \times b[0,0]) + (a[0,1] \times b[1,0]) + (a[0,2] \times b[2,0]) + (a[0,3] \times b[3,0])$$

that is:

$$280 = c[0,0] = (11 \times 0) + (12 \times 10) + (-13 \times 20) + (14 \times 30)$$

The element c[0,1] is the result of multiplying each element of row 0 of matrix A with each element of column 1 of matrix B and summing the result. Thus we have:

$$c[0,1] = (a[0,0] \times b[0,1]) + (a[0,1] \times b[1,1]) + (a[0,2] \times b[2,1]) + (a[0,3] \times b[3,1])$$

that is:

$$304 = c[0,1] = (11 \times 1) + (12 \times 11) + (-13 \times 21) + (14 \times 31)$$

Continuing with another example, the element c[2,1] is the result of multiplying each element of row 2 of matrix A with each element of column 1 of matrix B and summing the result. Thus we have:

$$c[2,1] = (a[2,0] \times b[0,1]) + (a[2,1] \times b[1,1]) + (a[2,2] \times b[2,1]) + (a[2,3] \times b[3,1])$$

that is:

$$(-1364) = c[2,1] = (31 \times 1) + (32 \times 11) + (-33 \times 21) + (-34 \times 31)$$

Make sure to check the other elements and understand how to do it before programming. In order to help you practice manually and to test your code, here are some examples in Figures 5, 6, 7 and 8.

If A (*3 x 4*) is: and B (*4 x 3*) is: then C = A x B is:(*3 x 3*)

$$A_{3\times4} = \begin{bmatrix} 11 & 12 & -13 & 14 \\ -21 & 22 & 23 & 24 \\ 31 & 32 & -33 & -34 \end{bmatrix} \quad B_{4\times3} = \begin{bmatrix} 0 & 1 & 2 \\ 10 & 11 & 12 \\ 20 & 21 & 22 \\ 30 & 31 & 32 \end{bmatrix} \quad C_{3\times3} = \begin{bmatrix} 280 & 304 & 328 \\ 1400 & 1448 & 1496 \\ -1360 & -1364 & -1368 \end{bmatrix}$$

Figure 5: Example 2 of matrix multiplication

If A (*3 x 4*) is: and B (*4 x 3*) is: then C = A x B is:(*3 x 3*)

$$A_{3\times4} = \begin{bmatrix} 11 & 12 & -13 & 14 \\ -21 & 22 & 23 & 24 \\ 31 & 32 & -33 & -34 \end{bmatrix} \quad B_{4\times3} = \begin{bmatrix} 1 & -5 & 9 \\ 2 & -6 & 10 \\ 3 & 7 & -11 \\ 4 & 8 & 12 \end{bmatrix} \quad C_{3\times3} = \begin{bmatrix} 52 & -106 & 530 \\ 188 & 326 & 66 \\ -140 & -850 & 554 \end{bmatrix}$$

Figure 6: Example 3 of matrix multiplication

If A (*3 x 3*) is: and B (*3 x 3*) is: then C = A x B is:(*3 x 3*)

$$A_{3\times 3} = \begin{bmatrix} 0 & 6 & 4 \\ -6 & 0 & 5 \\ -4 & -5 & 0 \end{bmatrix} \quad B_{3\times 3} = \begin{bmatrix} 0 & -6 & -4 \\ 6 & 0 & -5 \\ 4 & 5 & 0 \end{bmatrix} \quad C_{3\times 3} = \begin{bmatrix} 52 & 20 & -30 \\ 20 & 61 & 24 \\ -30 & 24 & 41 \end{bmatrix}$$

Figure 7: Example 4 of matrix multiplication

If A (*3 x 3*) is: and B (*3 x 3*) is: then C = A x B is:(*3 x 3*)

$$A_{3\times 3} = \begin{bmatrix} -1 & 0 & 0 \\ 0 & -1 & 0 \\ 0 & 0 & -1 \end{bmatrix} \quad B_{3\times 3} = \begin{bmatrix} -1 & 0 & 0 \\ 0 & -1 & 0 \\ 0 & 0 & -1 \end{bmatrix} \quad C_{3\times 3} = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

Figure 8: Example 5 of matrix multiplication

2.5. Orthogonal matrix

A *square* matrix *A* is defined to be ***orthogonal*** if its transpose is also its inverse, that is: $A \times A^T = A^T \times A = I$, where I is the identity matrix of appropriate order. Remember that multiplying a matrix by its inverse gives the identity, as in: $A \times A^{-1} = A^{-1} \times A = I$ .See the examples shown in Figures 9 and 10.

$$M4 = \begin{bmatrix} -1 & 0 & 0 \\ 0 & -1 & 0 \\ 0 & 0 & -1 \end{bmatrix}$$

Figure 9: The square matrix M4 (*3 x 3*) is symmetric and orthogonal.

The I (*3 x 3*) identity is:

$$I_{3\times 3} = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

Figure 10: Identity matrices

In general the I (*n x n*) identity is:

$$I_{n\times n} = \begin{bmatrix} 1 & 0 & 0 & \dots & 0 \\ 0 & 1 & 0 & \dots & 0 \\ 0 & 0 & 1 & \dots & 0 \\ \dots & \dots & \dots & \dots & \dots \\ 0 & 0 & 0 & 0 & 1 \end{bmatrix}$$

3. The expected processing of the program to be implemented

The tasks to be performed on the data from the input file are as follows, in this order:

1. Print opening messages.
2. Open the input file. If there is an error opening, print a message and exit gracefully.
3. Read the sizes of a matrix, namely 2 integers, *r* for number of rows and *c* for number of columns. No error checking is necessary.
4. Read the elements of a matrix (*r x c* integers) into the appropriate data structure (a 2-dimensional array).
5. Print heading messages and print the matrix to the screen.
6. Compute the transpose of the matrix.
7. Print heading messages and print the transpose to the screen.
8. Check if the matrix is symmetric and print a message with the result.
9. Check if the matrix is skew-symmetric and print a message with the result.
10. Check if the matrix is orthogonal and print a message with the result.
11. Repeat the process from step 3, until end of file is reached.
12. Close the input file.
13. Print a final message.
14. Exit the program.

The easiest way to formalize the flow of your application is to outline the overall functionality with pseudo-code or with a flowchart. Both are given in Figures 11 and 12. More detailed specifications are also given below.

```
Print initial messages including identification
Open the input file
    if problems, print message and exit program
Call "RdRowSize" to read the number of rows of the matrix -> returns flag for possible end of file
While it is not end of file {
    Read the number of columns of the matrix
    Call "RdMatrix" to read the integer elements of matrix and store in 2-dimensional array
        data structure
    Print the matrix headings
    Call "PrMat" to print the matrix
    Call "Transpose" to compute the transpose of the matrix
    Print the transpose headings
    Call "PrMat" to print the transpose
    If the matrix is square
        Call "Symm" to test if symmetric
        Print message about symmetric
        Call "SkewSymm" to test if skew-symmetric
        Print message about skew-symmetric
        Call "Ortho" to test if orthogonal
        Print message about orthogonal
    Call "RdRowSize" to read the number of rows of the matrix  -> returns flag for
        possible end of file
}
Exit program: close the input file, print a closing message, exit.
```

Figure 11: Pseudo Code for the overall program

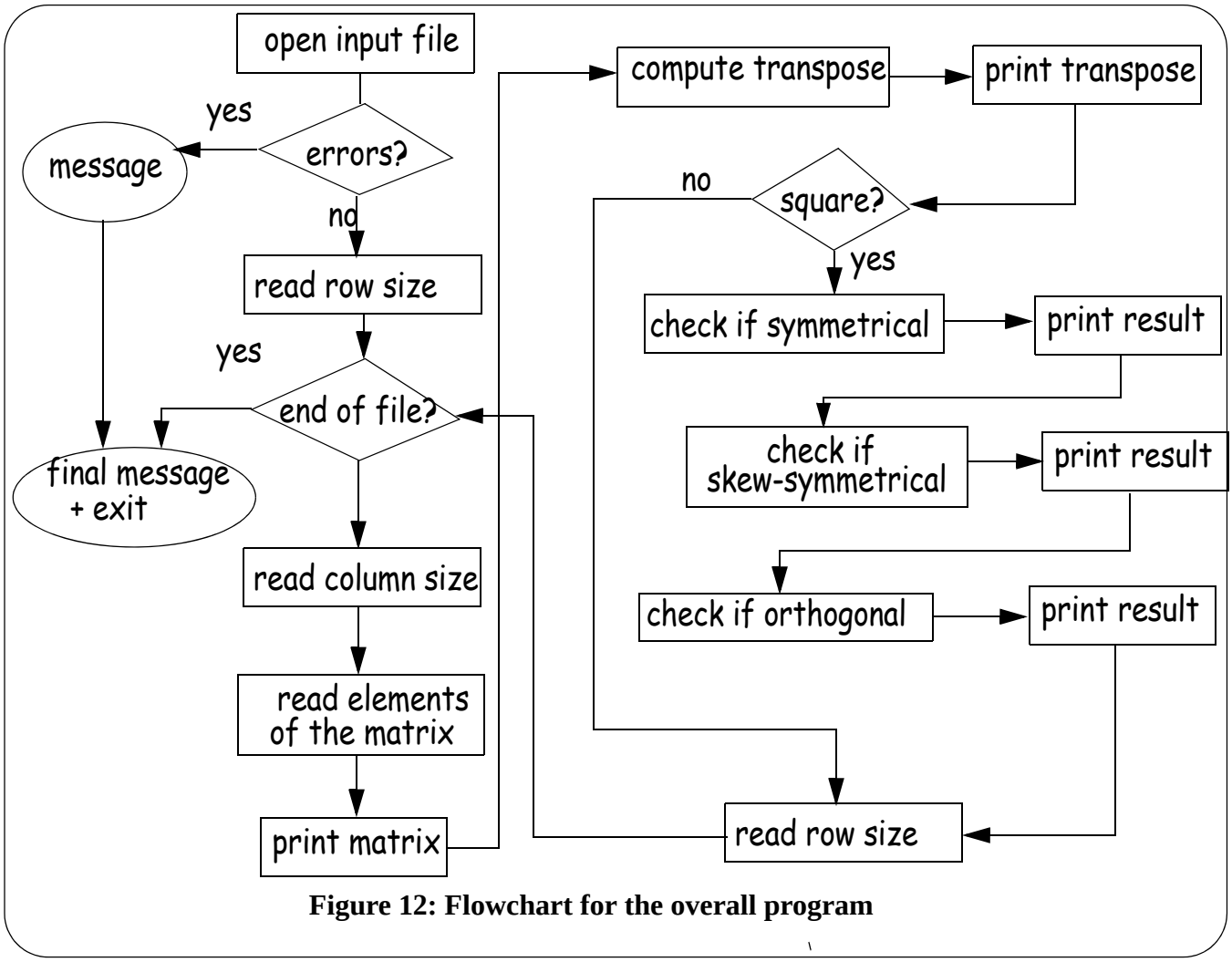

Figure 12: Flowchart for the overall program

4. The input file

The data in the input text file is a group of integers, where each group represents a matrix. In each group the first 2 integers represent the number of rows *r* and the number of columns *c*, while the following $r \times c$ integers represent the entries of the matrix in row-major order. For example, the text file for the matrices M1, M2, M3 and M4 shown above would be (the integers could be in different lines or not):

3 4 1 2 3 4 -5 -6 7 8 9 10 -11 12 3 3 3 2 -1 2 -5 17 -1 17 4
3 3 0 6 4 -6 0 5 -4 -5 0 3 3 -1 0 0 0 -1 0 0 0 -1

Each integer is separated by one or more blank spaces and the number of lines is irrelevant. The calls to "fscanf" take care of end-of-line characters as well. In this case there are 4 groups of integers representing 4 matrices and the program must read the input file until end of file is reached. The input file ***must*** be called: "`INA1.txt`". You ***must*** use "fscanf" to read input.

4.1. Assumptions on the input data

- The maximum number of rows of a matrix is $r = 10$, and $c = 10$ for the number of columns.
- Given the dimensions *r* and *c* of a matrix, it is guaranteed that exactly $r \times c$ integers follow, in the input file, thus there is no need to do any validation on the data.
- All entries are guaranteed to be integers.

5. Sample input and output

The sample input file shown above to be used for testing purposes is posted on the web pages. However be aware that the final evaluation will be using a different set of matrices. Add your own test cases as the ones provided are not a complete test set (e.g. how about a matrix of size 1 x 1).

Figures 13 and 14 show a sample type of output for the example matrices used above in the sample testing file. Note the details of the expectations for the format, including name, student number, headings, etc. While your output does not have to be exactly the same, it must be similar enough and look neat, clear and professional. (To save space, the sample output is shown in 4 panels which would appear in a single consecutive output on the screen).

```
Matrix testing program starts
Captain Picard V00123456
Summer 2012 - CSC 230 Assignment 1

*** MATRIX  1 ***  Size =  3 x  4
         1      2      3      4
        -5     -6      7      8
         9     10    -11     12

    Transpose:  Size =  4 x  3
         1     -5      9
         2     -6     10
         3      7    -11
         4      8     12

    ==>Not Square - no testing
```

```
*** MATRIX  2 ***  Size =  3 x  3
         3      2     -1
         2     -5     17
        -1     17      4

    Transpose:  Size =  3 x  3
         3      2     -1
         2     -5     17
        -1     17      4

    ==>Symmetric
    ==>Not Skew-Symmetric
    ==>Not Orthogonal
```

Figure 13: Sample output (part 1 and 2)

```
*** MATRIX  3 ***  Size =  3 x  3
         0      6      4
        -6      0      5
        -4     -5      0

    Transpose:  Size =  3 x  3
         0     -6     -4
         6      0     -5
         4      5      0

    ==>Not Symmetric
    ==>Skew-Symmetric
    ==>Not Orthogonal
```

Figure 14: Sample output (part 3 and 4)

```
*** MATRIX  4 ***  Size =  3 x  3
        -1      0      0
         0     -1      0
         0      0     -1

    Transpose:  Size =  3 x  3
        -1      0      0
         0     -1      0
         0      0     -1

    ==>Symmetric
    ==>Not Skew-Symmetric
    ==>Orthogonal

All done - Bye
```

6. The specifications: storage and I/O

6.1. Allocating the storage for the data structures.

Given that one reads the dimensions of a matrix from an input file, it may seem logical to declare storage in memory for a 2-dimensional array with exactly those sizes. While correct, the implementation

becomes a little more complex using some C features not covered in this course (i.e. dynamic allocation). Thus here the storage must be allocated statically for the maximum possible dimensions defined, that is, 10 x 10 arrays. The dimensions read from the input file simply give the boundaries of how many elements in the rows and columns are actually used from the available 10 x 10 space as declared for a particular instance in the computation.

Your declaration must include:

```
#define MAXSIZE 10       /*max size for rows and columns */
```

and in "main" there must be:

```
int MatMain[MAXSIZE][MAXSIZE];     /*the initial matrix*/
int RrowsM, CcolsM;      /*number of rows and number of columns*/
int MatTransp[MAXSIZE][MAXSIZE];   /*the transpose*/
int RrowsTr,CcolsTr;     /*transpose number of rows and number of
                         columns*/
```

For consistency you must use the above names as given as a requirement. You may use any other name you wish within each function and for other variables.

6.2. Opening the input file.

Instructions on how to open and close files are available in the documents posted in the Software Resources web pages. All this can be done within the main routine, without any extra functions. You only need to check for errors in opening the file, but not for errors in finding an early end of file. That is, you are guaranteed that once you start reading successfully, at least two integers for the dimension of a 2-dimensional array and all the elements for the whole 2-dimensional array are available correctly.

6.3. Reading in the dimensions of a matrix (number of rows and number of columns).

The dimensions are given as two positive integers, no bigger than 10. Store them in two variables called "`RrowsM`" and "`CcolsM`". You are not expected to do any input validation. This implies that you do not need to check that these numbers are positive and within range. This can be an assumption because normally, before any lengthy computation, one would put a set of input data in a file through a preprocessing stage to clean up any invalid data as a separate step.

Use a function to read the row size as in:

```
/*==== Function RdRowSize: read the row size of a matrix from a file */
/*Input parameters:
     FILE *fp       pointer to input file
     int *Nrows     pointer to row size to be returned
  Output parameter:
     1 if okay, -1 if end of file found*/
int RdRowSize(FILE *fp, int *Nrows) {
     /* your code here */
}
```

The function returns "1" if the number of rows has been read in correctly and it returns "-1" if the end of file has been detected. The first parameter is a pointer to the input file (read the I/O documentation). You do not need a separate function to read the column size, but you can have one if you wish.

6.4. Reading in the matrix.

The matrix is given as a set of integers and you are guaranteed that the correct number is present, that is, there are exactly `RrowsM X CcolsM` of them. Use a procedure as in:

```
/*==== Procedure RdMatrix: read the elements of a matrix from a file */
/*Input parameters:
     FILE *fp                       pointer to input file
     int Mat[MAXSIZE][MAXSIZE]      2D array for matrix
     int R,int C                    number of rows and columns
  Output parameters:
     None */
void RdMatrix(FILE *fp,int Mat[MAXSIZE][MAXSIZE],int R,int C) {
     /* your code here */
}
```

The first parameter is a pointer to the input file (read the I/O documentation).

6.5. Printing the matrix.

The matrix should be printed row by row as integers, lined up properly (use at least "`%3d`" in the formatting). Make sure that you read the documentation for printing "*characters*", printing "*strings*" and printing "*integers*", as the semantics are quite different.

Use a procedure:

```
/*===== Procedure PrMat: print a 2-D matrix of integers row by row*/
/*Input parameters:
     int Mat[MAXSIZE][MAXSIZE]      the matrix to be printed
     int R, int C                   number of rows and columns
 Output parameter: None*/
void PrMat (int Mat[MAXSIZE][MAXSIZE],int R,int C) {
     /* your code here */
}
```

6.6. Names for the input file, and for the program file

- The input file *__must__* be called: "`INA1.txt`"
- The program file containing the source code *__must__* be called: "`A1csc230.c`"

7. The specifications: implementing the tasks

Each task must be implemented as a separate function (subroutine) and here is a more detailed explanation of requirements and specifications. Note the interface shown which should give you an idea of the expectations for good documentation in *any* programming language and absolutely so in this assignment. There are 3 essential elements in *any* function interface documentation:

- the parameters are listed;
- a summary of what a function is supposed to do is stated clearly and succinctly;
- when necessary, a summary of how the work is supposed to be done (the algorithm).

7.1. Compute the transpose of a matrix

```
/*===== Procedure Transpose: construct the transpose of a matrix*/
/*Input parameters:
     int Mat[MAXSIZE][MAXSIZE]     the original matrix
     int Transp[MAXSIZE][MAXSIZE]  the transpose to be built
     int RM,int CM                 original number of rows and columns
     int *RT,int *CT          transpose number of rows and columns
 Output parameter: None*/
/*Given a matrix Mat and its dimensions in RM and CM,
 construct its transpose in Transp with dimensions RT and CT as in:
 copy rows 0,1,...,CM-1 of Mat to cols 0,1,...,RT-1 of Transp */
void Transpose (    int Mat[MAXSIZE][MAXSIZE],
                    int Transp[MAXSIZE][MAXSIZE],
                    int RM,int CM,int *RT,int *CT) {
     /* your code here */
}
```

When calling the function `Transpose` from `main` with the matrix called `MatMain` of dimension `RrowsM x CcolsM` to obtain a second matrix called `MatTransp` of size `RrowsTr x CcolsTr`, it might be logical to assign the dimensions of `MatTransp` before the function call directly from the known dimensions of `MatMain`. However this program is also a technical exercise in C programming and some features of C must be shown explicitly. In this case, the call to the function `Transpose` must have the dimensions of the resulting matrix `MatTransp` *unassigned* at call time and passed by reference. The values to `RrowsTr` and `CcolsTr` must be assigned within `Transpose` appropriately. Review the lecture notes on pointers and parameter passing (value and reference) Marks will be lost if the implementation does not use passing by reference as shown above.

7.2. Check for a symmetrical matrix

```
/*===== Function Symm: check for symmetric matrix*/
/*Input parameters:
     int Mat[MAXSIZE][MAXSIZE]     the matrix
     int Transp[MAXSIZE][MAXSIZE]  its transpose
     int Size                      dimensions
 Output parameter:
     0 for yes or -1 for no */
/*Given a square matrix, check if it is symmetric
 by comparing if Mat = Transp*/
int Symm (int Mat[MAXSIZE][MAXSIZE],
          int Transp[MAXSIZE][MAXSIZE],int Size) {
     /* your code here */
}
```

7.3. Check for a skew-symmetrical matrix

```
/*===== Function SkewSymm: check for symmetric matrix*/
/*Input parameters:
```

```
     int Mat[MAXSIZE][MAXSIZE]     the matrix
     int Transp[MAXSIZE][MAXSIZE]  its transpose
     int Size                      dimensions
 Output parameter:
     0 for yes or -1 for no */
/*Given a square matrix, check if it is skew-symmetric
 by comparing if Mat = - Transp*/
int SkewSymm (int Mat[MAXSIZE][MAXSIZE],
               int Transp[MAXSIZE][MAXSIZE],int Size) {
     /* your code here */
}
```

7.4. Check for an orthogonal matrix

```
/*===== Function Ortho: check for orthogonal matrix*/
/*Input parameters:
     int Mat[MAXSIZE][MAXSIZE]     matrix
     int Transp[MAXSIZE][MAXSIZE]  its transpose
     int Size                      dimensions
 Output parameter:
    0 for yes or -1 for no*/
 /*Given a square matrix, its dimensions in Size,
 and its transpose in Transp, check if Mat is
 orthogonal by comparing if Mat x Transp = Identity */
/*It also calls the function:
 MatMult(Mat,Transp,Prod,MR,MC,TR,TC,&PR,&PC)
 to multiply the two matrices before comparing the result to I*/
int Ortho (int Mat[MAXSIZE][MAXSIZE],
               int Transp[MAXSIZE][MAXSIZE],int Size) {
     /* your code here */
}
```

7.5. Multiply two matrices

```
/*===== Function MatMult: multiply 2 matrices*/
/*Input parameters:
     int MatA[MAXSIZE][MAXSIZE]    matrix 1
     int MatB[MAXSIZE][MAXSIZE]    matrix 2
     int MatP[MAXSIZE][MAXSIZE]    resulting matrix
     int RowA,int ColA             dimensions matrix 1
     int RowB,int ColB             dimensions matrix 2
     int *RowP, int *ColP          dimensions result
 Output parameter:
    0 if okay, or -1 if incompatible sizes*/
int MatMult (int MatA[MAXSIZE][MAXSIZE],int MatB[MAXSIZE][MAXSIZE],
               int MatP[MAXSIZE][MAXSIZE],int RowA,int ColA,
               int RowB, int ColB, int *RowP, int *ColP) {
     /* your code here */
}
```

The matrix multiplication in this program is only applied to square matrices, the only ones who can possibly be orthogonal. However the implementation must be for *general* matrix multiplication, not just for square matrices. Marks will be lost if the implementation is only for square matrices.
One might be tempted to surf the web to find code segments for matrix multiplication. Indeed C code is easy to find. However, at least two of the examples which come up quickly are not suitable. One is incorrect, the second uses complex C constructs and it is easy to see that it is not based on the knowledge acquired in this course. Most of all, copying code from the web may seem expedient, but it is completely useless as a learning experience for you and it is a breach of academic integrity.

If you are having trouble with the programming, *come and ask questions* and make sure to learn. This is what you are here for, not for copying code from others. Finally, there will be no web to support you when you are asked to write code segments in a test.

Matrix multiplication can be tricky to program at first. It is strongly suggested that you write some temporary code just to test the correctness of the matrix multiplication before you embed it in the call from another function. To help you in this, some results from multiplying the matrices in the above examples are given in the note in the appendix, so you can check your code. Make sure though that you add a few more tests, produced manually by you!

8. The specifications: everything else

8.1. The required functions and naming conventions

The sample code segments shows most of the modularity expected and the naming conventions. Study it carefully, observe its details and keep the consistency in your own code. In a work environment you would never be given a blank page. You would be expected to insert your program in a framework and remain precise and consistent. There is space for creativity in programming, but perhaps not yet in this first attempt.

8.2. Extra functions

For a better implementation, it is useful sometimes to write extra routines to do common little jobs, separately. You will learn more about modularity and similar design issues in your software engineering classes as well. Feel free to add extra functions or procedures if you believe that it would make a better design and document your choices. For example, in my solution, I wrote an extra function "`CheckId`" which, given a matrix, checks whether it is equal to the appropriate identity.

8.3. Global variables.

In summary, NO global variables should be used. The exception might be the necessary file handles for opening, closing files and for reading and writing to and from files.[1] Otherwise *all* other information needed by a subroutine should be passed through its parameters only.

8.4. Printing characters and strings.

A *character* is a 1-byte entity. A *string* is defined as a sequence of characters. A string of *n* characters is implemented in C as an array of *n+1* bytes with the rightmost byte containing the "0x00" NULL byte to denote the end of the sequence (reminder: the "0x" denotes an hexadecimal quantity following).

1.Only if you truly do not understand how to pass the file pointer as a parameter.

8.5. Input and Output Examples.

Look at the web pages for the file posted for examples of input to aid you in testing your program, but do not hesitate to build up your own. The sample output above as produced by an implementation should give you an idea of the expectations.

8.6. Documentation.

Do not forget about good documentation throughout, that is, meaningful and logical comments, not simply stating the sometimes obvious semantics of each line of code. *Every* subroutine must have a few lines in the headers stating what it does, what are the input and output parameters, and how it does it (that is, show the pseudo code of the algorithm used if appropriate).

8.7. Messages

All printing of messages is done by `main` (the manager of your program). Note, for example, how the function "*Symm*" returns a flag for the answer - it is up to the caller routine to print a message interpreting the answer received. Also all other printing is controlled by `main`, never called by a subroutine.

9. A helpful systematic approach

Here are some strong suggestions to acquire a good and a systematic approach, a s a basis for good habits.

1. Start with the file given to you, namely "`A1Frame.c`". Given that you are probably new to I/O in C and almost certainly new to I/O from files, the helpful initial code includes the basic I/O processing as in:
   - open an input file
   - read the row size and check for end of file
   - read the column size and read the matrix
   - print the matrix
   - repeat until end of file
   - close the input file

   Study this code carefully! It is a gift to you as it implements a lot of the processing already. Your learning experience is to understand it well and use it correctly. After spending a while analyzing the code, you should feel less overwhelmed by the whole assignment (at least, that is my goal). However all the above code resides in the main routine. Thus your first task is to understand the given code and modularize it such that you implement the `RdRowSize`, `RdMatrix` and `PrMat` in functions or procedures as required by the specifications. Call your *new* program `A1csc230V1.c` (5 execution marks if this is all you manage to hand in at the end).

2. Design, implement and test the code for the transpose. Call your *new* program `A1csc230V2.c.` Take a break, you have earned it! (5 more execution marks if this is all you manage to hand in at the end).

3. Design implement and test the function required for symmetric checking. Learn well from the experience (now go and have a rest break!). Call your *new* program `A1csc230V3.c.` (5 execution marks).

4. Continue by designing, implementing and testing the function for skew-symmetric checking (which should be easy now). Call your *new* program `A1csc230V4.c.`(5 execution marks).

5. Finally tackle doing the matrix multiplication, which is the trickiest computation here. Design and test your code by writing a separate program which only multiplies two matrices. It is worth the time

(and it is a good learning experience for you). Just re- use the initial framework (which you now know well) to get your input and output, and write only the code in a subroutine to multiply 2 matrices, initialized as hard-coded at the beginning (if you want to avoid the I/O). Change the matrices to test various ones, as given in the examples in Figures 4, 5, 6, 7 and 8.

6. Insert your tested function for matrix multiplication into your main program and add the code for checking orthogonality. This can be tricky as well (how do you check for an identity matrix?). (10 execution marks).

7. Finally check all the output style and the documentation and call your program `A1csc230.c`, ready to be submitted (hopefully for full marks). It is important to make a separate version of your program file without overwriting your result whenever you start a new task. That way, should you not be able to finish, you will still be able to submit a correct partial product, which will garner you a better evaluation than a full program which has many errors. It is also much easier to backtrack to a tested working version when debugging. (20 marks overall for code design, architecture, following specifications, documentation).

The systematic approach above appears tedious and you all probably think you can do better than that (and faster) using your own brilliant methodology. Trust the instructions above, no matter how pedantic they look - they work as they divide the work into manageable and easily testable modules, and they give an ordering, possibly non-intuitive, to the tasks you need to do. Moreover, should you be able to do only a subset of what is expected, you can still hand in something that works and will give you partial marks as stated. More instructions will also be given in a lecture.

10. What to hand in by June 2, Monday, at 5 p.m.

1. Your working implementation in a file named "`A1csc230.c`" by electronic submission through the Assignment menu item on the course Connex web page.
2. Your source code file must start with: name and student number, file name, assignment number, course and the purpose of the program overall.

11. How your program will be tested and evaluated

1. Your program will be compiled in the lab machine environment using the gcc compiler with the following options:

```
gcc -Wall -ansi A1csc230.c
```

2. You are free to develop your program on any platform you wish and with any C compiler of your choice. However it **must** work in the environment above or you will lose marks.
3. **No** warnings from compilation are acceptable. If the program compiles with any warnings (but eventually executes) you will be given 0 marks by the tester who will no further evaluate. You will have to come to me to have it tested (changing the offending lines) and you will lose a minimum of 10%. If the program has compilation errors, you will receive 0 marks.
4. 20 of the 50 marks are for code design, architecture, following specifications, documentation.
5. 30 of the 50 marks are for correct execution on a test file which may be different from the sample one given to you.
6. Questions? Doubts? Problems? Come to office hours and do not wait too late!