CSCI103L Programming Midterm – Spring 2018

Overview:

The goal of the programming midterm is to give student an opportunity to show that they have obtained a basic level of competency with C++ programming through a short programming exercise. The exam is structured to test basic syntax including variables and looping, reading data from a file and using dynamic memory.

Spring 2018 Introduction:

The assignment for the Spring 2018 programming midterm is to develop a program to perform 3 mathematical operations on 2D matrices. Matrix add, subtract and multiply are foundational operations in linear algebra that have applications across computer science from physics simulations to computer graphics to machine learning. Your task is to develop a program that will do the following:

- Support allocating and deallocating 2D arrays of type 'double' to hold matrix data.
- Support adding, subtracting or multiplying 2 matrices together
    - Including checking that the supplied matrix dimensions are appropriate to the operation.
- Support reading matrix data from a file
- Analyze the command line arguments given to your program to determine which operation should be performed.
- Error checking and reporting of failure conditions. Some examples:
    - The input file can not be opened.
    - The matrix dimensions do not support the requested operation
    - The command line arguments are incorrect

To get started download the skeleton code here: http://bytes.usc.edu/files/cs103/pm-sp18.zip

In order to make testing your code easier and to allow for partial credit opportunities the development for this exam is split into two pieces: matrix_ops.cpp and matrix.cpp.

We give you a Makefile to build the program:

- Use `make matrix_ops.o` if you want to check to see if your matrix_ops.cpp code compiles. If so, upload it to the autochecker to test the functions.

- Use `make matrix` if you want to build matrix.cpp and matrix_ops.cpp. You can then test your main() using the examples at the end of this document.

matrix_ops.cpp

matrix_ops: The operations for matrix_allocate(), matrix_delete(), matrix_add(), matrix_subtract() and matrix_multiply() are implemented in matrix_ops.cpp. We give you matrix_ops.h as a starting point and an empty matrix_ops.cpp.

For full credit on matrix_ops.cpp implement the following. I **strongly** suggest you do them in the listed order.

- double** matrix_allocate(int rows, int cols)
    - This function uses new to allocate a 2D array of doubles to store a matrix of size rows x cols. The outer dimension should be an array of double* and the inner dimension of type double. Return a pointer to this array.

- void matrix_delete(double** M, int rows)
    - This function deletes a matrix allocated by matrix_allocate().

- double** matrix_add(double** A, double** B, int rows, int cols)
    - Perform R = A + B.
    - Use matrix_allocate() to allocate an array of appropriate size to hold R. Use a loop(s) to calculate:
        - R[i][j] = A[i][j] + B[i][j] i = 0…(rows-1), j = 0…(cols-1)
    - Return the pointer to R.

- double** matrix_subtract(double** A, double** B, int rows, int cols)
    - Perform R = A – B
    - Use matrix_allocate() to allocate an array of appropriate size to hold R.
    - Use a loop(s) to calculate:
        - R[i][j] = A[i][j] – B[i][j] i = 0…(rows-1), j = 0…(cols-1)
    - Return the pointer to R.

- double** matrix_multiply(double** A, int rows_A, int cols_A, double** B,
  int rows_B, int cols_B)
    - Perform R = A*B
    - Use matrix_allocate() to allocate an array of appropriate size to hold R.
    - Use a loop(s) to implement the matrix multiplication algorithm. See notes below.
    - Return the pointer to R.

Notes for matrix_ops.cpp:

- Each of the functions in matrix_ops.cpp can be tested incrementally with the autochecker. If a function returns a pointer and you have not implemented that function yet, just return the NULL pointer. You will obviously fail that test, but your other, correctly implemented functions will pass.

- The dimensions of two matrices must match ($NR_A == NR_B$, $NC_A == NC_B$) for adding or subtracting. Here NR_A is number of rows in A, NC_A is number of columns in A, NR_B is number of rows in B and NC_B is number of columns in B.

The algorithm for matrix multiply is usually shown as:

If $\mathbf{A}$ is an $n \times m$ matrix and $\mathbf{B}$ is an $m \times p$ matrix,

$$\mathbf{A} = \begin{pmatrix} a_{11} & a_{12} & \cdots & a_{1m} \\ a_{21} & a_{22} & \cdots & a_{2m} \\ \vdots & \vdots & \ddots & \vdots \\ a_{n1} & a_{n2} & \cdots & a_{nm} \end{pmatrix}, \quad \mathbf{B} = \begin{pmatrix} b_{11} & b_{12} & \cdots & b_{1p} \\ b_{21} & b_{22} & \cdots & b_{2p} \\ \vdots & \vdots & \ddots & \vdots \\ b_{m1} & b_{m2} & \cdots & b_{mp} \end{pmatrix}$$

the *matrix product* $\mathbf{AB}$ (denoted without multiplication signs or dots) is defined to be the $n \times p$ matrix

$$\mathbf{C} = \begin{pmatrix} c_{11} & c_{12} & \cdots & c_{1p} \\ c_{21} & c_{22} & \cdots & c_{2p} \\ \vdots & \vdots & \ddots & \vdots \\ c_{n1} & c_{n2} & \cdots & c_{np} \end{pmatrix}$$

such that

$$c_{ij} = a_{i1}b_{1j} + \cdots + a_{im}b_{mj} = \sum_{k=1}^{m} a_{ik}b_{kj},$$

for $i = 1, \ldots, n$ and $j = 1, \ldots, p$.

- This can be translated into the following pseudo-code

```
o A := matrix of size $NR_A$ x $NC_A$
o B := matrix of size $NR_B$ x $NC_B$
o R := matrix to hold the result of size $NR_A$ x $NC_B$, all elements = 0.0
o for i = 0 … $NR_A$-1
    ▪ for j = 0 … $NC_B$-1
        • for x = 0 … $NC_A$-1
            o R[i][j] += A[i][x] * B[x][j]
```

- In order to multiply two matrices together $NC_A == NR_B$

matrix.cpp

The main() for your program is in matrix.cpp. For full credit on matrix.cpp implement the following: (note, do not copy-paste the error messages. This will cause problems)

- Create an ifstream object and attempt to open the file given as argv[1]
  - If this file can not be opened, you **must** print the following error message and return 1 (i.e. quit the program)
    - "Unable to open file: <filename>" where <filename> is the file name given as argv[1]
- Create two integer variables to hold the rows and columns for matrix A. Read in these two values from the input file
- Create a variable of type double** and use matrix_allocate() to allocate a 2D matrix to hold matrix A
- Read in the values for matrix A into the 2D array (you do not need getline())
- Create two integer variables to hold the rows and columns for matrix B. Read in these two values from the input file
- Create a variable of type double** and use matrix_allocate() to allocate a 2D matrix to hold matrix B
- Read in the values for matrix B into the 2D array (you do not need getline())
- Write some code to determine if the matrix dimensions are compatible with the requested operation. If the dimensions are not compatible print an error and return 1. If the operation as specified as argv[2] is not add, subtract or multiply print an error and return 1.
  - If the operation is add or subtract, then the matrix dimensions must match $NR_A == NR_B$ and $NC_A == NC_B$. If this is not the case print the following message:
    - "Number of rows and number of columns must match for add or subtract."
  - If the operation is multiply, then $NC_A == NR_B$. If this is not the case print the following error message:
    - Number of columns in A must match number of rows in B for multiply."
  - If the operation is not add, subtract or multiply print the following message and return 1:
    - "Unknown Operation: <operation>
    - Valid operations: add, subtract, multiply" where <operation> is what was given as argv[2]
- Based on the operation given as argv[2] call the appropriate function to compute the result. Store the dimensions of result in the given variables.
- Use matrix_delete() to de-allocate your dynamic memory.

Input File Format

The input file for this exam defines two matrices, A and B in the following way:

```
<int number rows A> <int number of columns A>
<double> <double> <double> … <double>
<double> <double> <double> … <double>
…
<double> <double> <double> … <double>
<int number of rows B> <int number of columns B>
<double> <double> <double> … <double>
<double> <double> <double> … <double>
…
<double> <double> <double> … <double>
```

So the 1st line has two ints, specifying the number of rows of A and the number of columns of A. That line is followed by NR_A lines, each with NC_A doubles. i.e a RxC grid of doubles. Then there is a line with two ints specifying the number of rows for B and the number of columns for B. That line is then followed by NR_B lines, each with NC_B doubles. All values are separated by spaces and/or newlines (ie. whitespace).

For example, the following file specifies a matrix A of size 2x2 and a B of size 2x3:

```
2 2
15.0 100
-5.7 78.3
2 3
1.0 12.5 99.1
-8.0 67.2 -55.1
```

Do not overthink the file format details. It is as straightforward as it seems.

Rubric

In order to allow for as much partial credit as possible, the points will be assigned as follows

matrix_ops.cpp
Each function will be worth 3 points, for a total of 15 points.

matrix.cpp
This file will total 10 points, one point for each bullet item on the requirements page.

The total for the exam will be 20 points, you may use partial credit from each portion to add up to 20 points. If you exceed 20 points your score will be clamped at 20 points (sorry, no extra credit)

Autochecker

You will submit your code to http://bit.ly/2pMIkuC where there is an autochecker. The autochecker is your friend. It attempts to check as many of the individual rubric items individually so you can build your solution incrementally and earn partial credit. Submit early and often. Do not leave the exam with code that doesn't compile, this will make your exam hard to grade and likely cost you partial credit.

Usage Examples

We include two files, t1.txt and t2.txt that can be used to test your code as shown below:

```
shell$ ./matrix t1.txt add
   6.00   8.00
  10.00  12.00
shell$ ./matrix t1.txt subtract
  -4.00  -4.00
  -4.00  -4.00
shell$ ./matrix t2.txt multiply
   7.00  10.00
  15.00  22.00
  23.00  34.00
  31.00  46.00
```

The error cases look like:

```
shell$ ./matrix notafile.txt add
Unable to open file: notafile.txt

shell$ ./matrix bad_add_sub.txt add
Number of rows and number of columns must match for add or subtract.

shell$ ./matrix bad_add_sub.txt subtract
Number of rows and number of columns must match for add or subtract.

shell$ ./matrix bad_mult.txt multiply
Number of columns in A must match number of rows in B for multiply.

shell$ ./matrix t1.txt badcmd
Unknown operation: badcmd
Valid operations: add, subtract, multiply
```