Lab 4 Activities [30 Marks]

An Autograded Evaluation

Do not use `printf()/cout` or `scanf()/cin` for autograded submissions.

IMPORTANT: **Your submission must compile without syntactic errors to receive grades. Non-compilable solutions will not be graded.**

IMPORTANT: **Penalties will be applied for poor memory management practices. Refer to marking schemes for details to watch out for.**

About Lab 4

- Use the following link to accept the lab invitation: https://classroom.github.com/a/PxW9lgsU
- Lab 4 is due on **Friday, November 22, 2024, at 11:59 pm** on Github.

- You are permitted to use the C++ std::string library. Reference provided in the manual.

- **Global Variables are NOT PERMITTED.**
  Including them will result in zero for the question.

- **Memory Profiler tools will be used to check for any memory leakage originating from within your implementation. Marks will be deducted for poor memory management practices.**

- **Note: You must provide your student information (Section, Mac ID, and Student ID) in the README file of the Lab 4 code package. Failure to do so will result in a 5% penalty on the entire Lab 4 grade.**

- **Lab 4 comes with launch.json to enable the VSCode debugger.**
  You need to set the workspace to your Lab 4 folder to start the debugging session.

Important Notes about Memory Leakage Evaluation

Given that memory leakages may take place within the installed toolchains and third-party libraries, you are not expected to eliminate all memory leakages captured by the memory profiler tool on your system. You are, however, required to **analyze the report and ensure that none of the leakages originate from `malloc()` / `new` calls within your code implementation**.

REVIEW: How to Perform Memory Usage Profiling

Windows

In VSCode terminal, under your Lab 4 folder containing the compiled Lab4.exe, type in:

```
drmemory ./Lab4.exe
```

The memory leakage report will be generated.

If Dr. Memory doesn't work on your system due to OS memory security block, you should first open a Windows *command prompt* as an *administrator* and then run the memory profiler through it. If the problem persists ask the TAs for help.

MacOS

In terminal, navigate under Lab 4 folder with the compiled Lab4 executable, do the following:

1. Enter: `export MallocStackLogging=1`
2. Enter: `leaks --atExit --list -- ./Lab4`

The memory leakage report will be generated.

Linux / Chrome

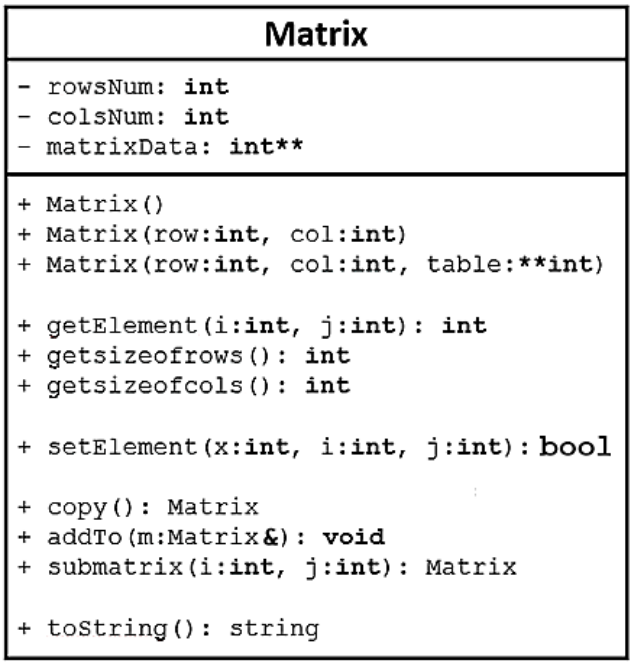
In terminal, navigate under Lab 4 folder with the compiled Lab4 executable, type in:

```
valgrind –leak-check=yes ./Lab4
```

The memory leakage report will be generated.

Lab Question – Matrix Class [30 marks]

This is the only question in lab 4, helping you get familiar with the general structure of a C++ class, and how to implement the behaviour of the class by completing the code implementation for every member function defined in the skeleton code. The UML diagram of the Matrix class is provided on the right for your reference.



WARNING!! Respect the Rule of Six / Minimum Four

There are some missing special member functions – without them, your Matrix object will likely suffer memory leak and heap error. Add them as you see required (follow what we've discussed in class). Remember, DEEP COPY ONLY!!!

You may optionally include any private helper functions in your implementation. It is not necessary to have them to complete the lab question, however.

- **Private Data Members**
  - `int rowsNum`
    - Storing the number of rows in the matrix. Self explanatory.
  - `int colsNum`
    - Storing the number of cols in the matrix. Self explanatory.
  - `int** matrixData`
    - Double integer pointer intended to store the starting address of a 2D array on the heap.
    - The actual allocation should happen within the constructors.

- **Public Member Functions (Interface)**
  - `Matrix()`
    - Default constructor, already implemented as your learning reference.
  - `Matrix(int row, int col)`
    - Additional constructor.
    - Creates a matrix on the heap with its dimensions specified by row and col.
    - If row or col input values are invalid (i.e. smaller than or equal to zero), set them to the default size of row = 3 and col = 3, then initialize the matrix accordingly.
  - `Matrix(int row, int col, int** table)`
    - Additional constructor.
    - Creates a matrix on the heap with its dimensions specified by row and col, and then copy all the values in the 2D integer array "table" into its corresponding elements in the matrix.
    - Assume that the input integer array size has identical dimensions as specified by row and col.
  - **IMPORTANT:** To ensure you can run the test cases correctly, implement `toString()` method first after completing the constructor design.
  - **IMPORTANT: Respect the Rule of Six / Minimum Four. Add the missing special member functions!!**

- `int getElement(int i, int j)`
  - Getter method. Returns the value of the matrix element at index [i, j].
  - If `i` and/or `j` are out of range, you should throw a C++ exception as specified in the skeleton code comments. (Using other error handling methods will result in failed test case.)

- `bool setElement(int x, int i, int j)`
  - Setter method. Set the value of the matrix element at index [i, j] with x.
  - This method uses an alternative error handling approach. If i or j are out of range such that the value-setting operation failed, you should return FALSE to indicate the operation had failed. Otherwise, return TRUE after the value is properly set in the designated element.

- `int getsizeofrows()`
  - Getter method. Return the row dimension of the matrix. Rather straightforward.

- `int getsizeofcols()`
  - Getter method. Return the col dimension of the matrix. Rather straightforward.

- `Matrix copy()`
  - Copy Operation method (NOT copy constructor)
  - This method creates a new Matrix instance, copies all the element values in the current matrix into the copied matrix, then returns the copied Matrix itself (NOT its reference)
  - For this method, a faulty implementation is provided in the comments with two semantic bugs. Uncomment the code and perform VSCode IDE debugging sessions OR debugging message outputs (cout statements) to fix the bugs. Produce a simple bug report to validate your debugging process.
  - **[Not Graded]** For your self learning purpose, think about this statement: **"Returning the instance of an object is highly inefficient."** This method clearly falls under the inefficient category. Can you think about why, and how you would fix it?

- `void addTo(Matrix &m)`
  - Matrix Addition method (for experienced C++ programmers, it's NOT the operator+ overload)
  - This method adds matrix *m* to the current matrix. The result will be stored in the current matrix.
  - Obviously, the two matrices must have the same dimensions for addition to be valid. Therefore, you should check whether the two matrices have identical dimensions. If not, you should throw a C++ exception as specified in the skeleton code comments. (Using other error handling methods will result in failed test case.)
  - **[Not Graded]** For your self leaning purpose, think about whether (and how) you can deploy the alternative error handling approach – the Boolean return approach - in setElement() method?

- `Matrix subMatrix(int i, int j)`
  - A getter method in disguise.
  - This method extracts the submatrix defined between [0, 0] and [i, j] of the current matrix, saves the submatrix in a new Matrix object, and returns the submatrix object.
  - For example:
    - If $m = \begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{bmatrix}$, then `m.submatrix(1, 1)` will return: $\begin{bmatrix} 1 & 2 \\ 4 & 5 \end{bmatrix}$
  - Think about what conditions will result in invalid submatrix extraction operations and handle these conditions by throwing the exception specified in the skeleton code comments. (Using other error handling methods will result in failed test case.)

- `string toString()`
  - A string conversion method. Common in OOD for outputting object contents in string format.
  - This method should output the matrix contents in the following format:
    - If $m = \begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{bmatrix}$, `m.toString()` returns `"1 2 3 \n4 5 6 \n7 8 9 "`
    - Pay close attention to the spaces between characters!!
  - Tip 1: You are allowed to use C++ <string> library. Use this reference: https://cplusplus.com/reference/string/string/
  - Tip 2: You can use the `to_string()` utility method from <string> library to convert any non-string primitive data types into its string-equivalent representation.
    - For example, given `int a = 4`, and `string str = "2SH"`, then
    - `str += to_string(a);` will make `str = "2SH4"`

- **IMPORTANT: Respect the Rule of Six / Minimum Four. Add the missing special member functions!!**
- **WARNING!!!** Among the missing special member functions, there is one acting as the critical safeguard to prevent memory leak. You will run into memory leakage if you merely implement the methods as they are presented in the skeleton code. Add the special member function to resolve the memory leakage issue. **Think about what it is!**

Marking Scheme

- Code Implementation
  - **[1 marks]** Correct implementation of the `Matrix(int row, int col)` constructor
  - **[2 marks]** Correct implementation of the `Matrix(int row, int col, int** table)` constructor
  - **[2 marks]** Correct implementation of the `toString()` method
  - **[2 marks]** Correct implementation of the `getElement()` method with correct exception thrown
  - **[1 mark]** Correct implementation of the `getsizeofrows()` method
  - **[1 mark]** Correct implementation of the `getsizeofcols()` method
  - **[2 marks]** Correct implementation of the `setElement()` method
  - **[4 marks]** Fixing the implementation of the faulty `copy()` method and generate a debugging report.
    - **Warning!** Without a debugging report for effort validation, all 4 marks will be deducted
  - **[3 marks]** Correct implementation of the `addTo()` method with correct exception thrown
  - **[4 marks]** Correct implementation of the `subMatrix()` method with correct exception thrown
  - **[2 marks]** Implementing the required special member function to prevent memory leakage.
    - **Warning!** If this function is not implemented, you will lose BOTH 2 marks from this item, AND the additional 4 marks for memory leakage.

- Code Behaviour, Analysis, and Test Plans
  - **[6 mark]** Correct Program Behaviour – Passing all default test cases.

- Memory Management Penalties
  - **Segmentation Failure or Heap Error will lead to additional 2 marks deducted**
  - **Any detected memory leakage from within your class design will lead to additional 4 marks deducted. There may be minor leakages (up to 8 bytes) from the exception throwing statement in getElement() test case – we will not hold you accountable for it as it is out of scope of the course.**