CS11 Intro C++ Lab 6: Integer Matrix

This assignment is your first foray into heap memory allocation in C++. As discussed in class, you should generally avoid having to explicitly manage heap memory allocations yourself; you should usually prefer to rely on classes that take care of this responsibility for you, unless it made a lot of sense to do it yourself. But, in order to learn more of the nitty gritty details of C++, we will write a C++ `Matrix` class that implements a 2D integer matrix, dynamically allocating a 1D array of memory from the heap to store the matrix data. Your `Matrix` should be declared in `matrix.h`, and any definitions can go into `matrix.cpp`. You should feel free to define functions in the class declaration, partly because it facilitates inlining of your functions, and partly because it makes your code more streamlined.

Note that the elements are integers, not floats or some other type. We are keeping it simple this week!

Provide these constructors:

- `Matrix{int rows, int cols}` initializes a *rows* x *cols* matrix where all elements are set to 0. The *rows* and *cols* values must be positive, or an `invalid_argument` exception should be thrown.
- `Matrix{}` initializes a 0x0 matrix.

Since your `Matrix` class dynamically allocates memory, you need to provide non-default implementations of these operations:

- `Matrix` copy-constructor
- `Matrix` copy-assignment operator
- `Matrix` destructor

This is an example of the Rule of Three: **If your class requires a user-defined destructor, a user-defined copy-constructor, or a user-defined copy-assignment operator, it almost certainly requires all three.**

When you implement the above operations, you should think carefully about avoiding code duplication. Most of the operations that would be duplicated are pretty short, so it isn't the most serious problem, but a good programmer always tries to minimize code duplication.

Other Operations

You should additionally provide these operations. *Note that we are not specifying where you should use the `const` keyword and pass-by-reference semantics, because you should be learning these rules yourself.* When we review your submission, we

will point out places in your code where corrections need to be made to follow best practices.

- `int numRows()` returns the total number of rows
- `int numCols()` returns the total number of columns
- `int get(int r, int c)` returns the value stored at the specified row and column. This member function should throw an `invalid_argument` exception if the row or column index is out of bounds
- `int set(int r, int c, int value)` sets the value stored at the specified row and column. Again, this member function should throw an `invalid_argument` exception if the row or column index is out of bounds

Finally, we would like to be able to compare whether two matrix objects have the same values or not, so you should implement the == and `!=` operators for your `Matrix` class. These operators can be implemented as either member operator overloads or nonmember operator overloads; this week you should implement them as member operator overloads. The reason is that it's shorter and likely more efficient to iterate over the internal 1D arrays directly, which you will be able to do from inside the member functions. If you implement nonmember operator overloads, you will have to call the above accessors, and it will be more complicated.

- `bool operator==(Matrix m)` should return `true` iff the specified matrix is "equal to" this matrix; i.e. the total number of rows and columns are identical, and all of the values are also identical

  (Don't forget that you must use `const` and pass-by-reference where appropriate!)

- `bool operator!=(Matrix m)` should do the opposite of `operator==()`. *(Hint, hint!)*

Coding Style

Comment your class thoroughly in the Doxygen style, including a class-level description, and a comment for every data-member and member-function. (Comments may be brief if the function's purpose is obvious, but you must still comment all of these things.)

You do not need to create a `Doxyfile` or run Doxygen this time, as long as you follow the Doxygen format properly.

Make sure to always identify what arguments are illegal. Additionally, always document any exceptions that may be thrown, and the circumstances that would

cause them to be thrown.

Testing

A test suite for your class is provided here; make sure to fix any test failures you encounter.

Build Automation

As before, write a `Makefile` for your project that builds and tests your `Matrix` code.

- The `all` target should build the test binary, but not run it.
- The `test` target should run the test binary.
- The `clean` target should delete `.o` files and binaries. **As always, make sure to back up your code before testing your `clean` target for the first time!**
- Make sure all files are compiled with the `-Wall` and `-Werror` arguments (these will go in your `CXXFLAGS` variable) so that any malformed but otherwise legal code is identified by the compiler.

Submitting Your Work

Once you have completed the above tasks, submit your work through csman. Make sure to submit these files:

- `Makefile`
- `matrix.h`
- `matrix.cpp` (if present)

You do not need to submit the test code; we will test your program with a fresh copy of these files.

Assignment Feedback Survey

Please also complete and submit a feedback survey with your submission, telling us about your experience with this assignment. Doing so will help us to improve CS11 Intro C++ in the future.

Generated from cpp-lab6.md.