Project #02: mymatrix<T>

Complete By: **Bonus:** Thursday, Jan. 30th @ 11:59pm (+10%)
On-time: Saturday, Feb. 1st @ 11:59pm
Late: Sunday, Feb. 2nd @ 11:59pm (-10%)

Assignment: C++ .h file to implement mymatrix<T> class

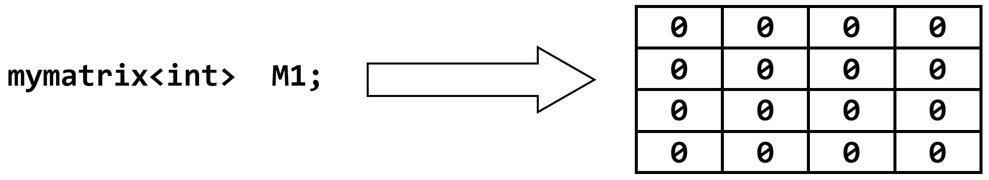
Policy: Individual work only, late work *is* accepted

Submission: "mymatrix.h" and "main.cpp" files via Gradescope

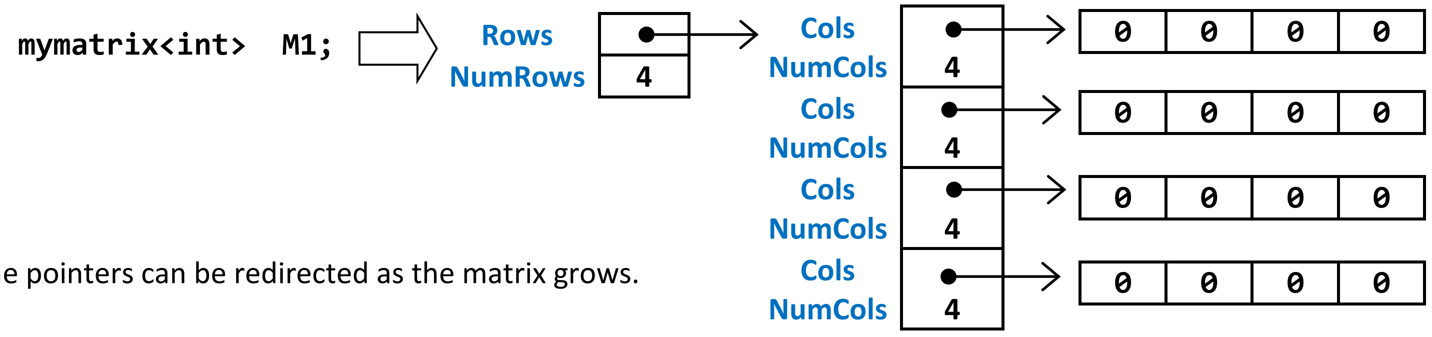
Assignment

In C++, **std::vector<T>** is one of the most commonly used data structures. It's efficient, and grows dynamically as needed. However, it's a one-dimensional data structure. While it's possible to use vector<T> to define two-dimensional structures --- e.g. vector<vector<int>> --- it's awkward because each row is initially empty, requiring you to add columns on a row-by-row basis. There are ways around this, but non-obvious.

The goal of this project is to define a class **mymatrix<T>** explicitly designed to support a 2D data structure. Like vector<T>, it can grow dynamically in terms of rows and columns. Unlike vector<T>, the use of push_back is not required to add elements. Instead, a matrix is defined to have a given number of rows and columns, and the resulting elements are initialized to C++'s natural default value. For example, the default is a 4x4 matrix:



Keep in mind this is an abstraction, the actual implementation of the matrix is quite different. To allow the matrix to dynamically grow, pointers to C-style arrays are used:



The pointers can be redirected as the matrix grows.

Assignment details

The assignment consists of implementing a set of member functions for class **mymatrix<T>**. The first requirement is that you *must* implement mymatrix as discussed on the previous page, and defined in the provided “mymatrix.h” header file. There are lots of possible implementations, but we require that you use this approach. Here are the relevant declarations from “mymatrix.h”:

```
template<typename T>
class mymatrix
{
private:
  struct ROW
  {
    T*  Cols;     // dynamic array of column elements
    int NumCols;  // total # of columns (0..NumCols-1)
  };

  ROW* Rows;     // dynamic array of ROWs
  int  NumRows;  // total # of rows (0..NumRows-1)
```

You are free to add additional member variables to improve the implementation, as well as private helper functions. But you cannot change the overall implementation of a matrix: it must remain a pointer to an array of ROW structures, where each ROW contains a pointer to an array of elements of type T. You cannot switch to a vector-based implementation, nor other data structures.

By default, a matrix is 4x4. Here’s the code for the default constructor that creates this 4x4 matrix. You’ll want to match this up with the diagram shown on the bottom of page 1:

```
public:
  //
  // default constructor:
  //
  // Called automatically by C++ to construct a 4x4 matrix.  All
  // elements are initialized to the default value of T.
  //
  mymatrix()
  {
    Rows = new ROW[4];  // an array with 4 ROW structs:
    NumRows = 4;

    for (int r = 0; r < NumRows; ++r) // initialize each row to have 4 columns:
    {
      Rows[r].Cols = new T[4];  // an array with 4 elements of type T:
      Rows[r].NumCols = 4;

      for (int c = 0; c < Rows[r].NumCols; ++c) // initialize to default value:
        Rows[r].Cols[c] = T{};  // default value for type T:
    }
  }
```

For completeness, the Appendix at the end of this document provides the "mymatrix.h" file that you have to implement. This file is also available on the course dropbox for project #02. Note that there is no associated "mymatrix.cpp" file --- since the class is templated, all functions must be "inline" and placed within the .h file.

The parts you need to complete are marked with TODO. You'll note that some member functions are "throwing" exceptions to denote erroneous conditions; this is the standard approach in modern programming languages. We'll discuss this in class at some point, but for now just know that throwing an exception terminates the program, identifying errors quickly.

Testing

An interesting component of this assignment is testing: how do you test your **mymatrix<T>** class? In general, when building software, how do you test it? Normally you run and look at the output, but this is tedious, error-prone, and difficult to repeat. If anything, your eyes get tired. What you want to think about is ways to automate the testing, much like submissions on zybooks and Gradescope --- push a button and get an answer. As you might expect, we are going to delay the release of our Gradescope submission site so you'll need to do some testing yourself. We are also going to collect and evaluate your testing code as part of your final grade for the project.

In Java, the **JUnit** testing framework is very popular. In the world of C++, there is no single framework that everyone uses. But the C++ **Catch** framework is one we'll eventually use; it's free, platform-neutral, and header-only (i.e. all you need is the .h file). Regardless of framework, the idea is easy enough: write code that makes calls to your class, and checks the return values. For example, suppose you want to test to make sure your **size()** function is working. You could write a test function in "main.cpp" as follows:

```
bool size_test1()
{
  mymatrix<int>  M;  // creates 4x4 matrix

  if (M.size() == 16)
    return true;
  else
    return false;
}

int main()
{
   int passed = 0;
   int failed = 0;

   if (size_test1())
     passed++;
   else {
     cout << "size_test1 failed" << endl;
     failed++;
   }
```

```
      .
      .
      .

      cout << “Tests passed: “ << passed << endl;
      cout << “Tests failed: “ << failed << endl;

      return 0;
   }
```

If this seems like a lot of work, it is. The general rule of thumb is that it takes as much time to test a piece of software as it does to create it. But once you write the tests, the advantage is huge: at any moment you can run the tests and get a sense of how well the software is working. [*Have you heard of Test-Driven Development (TDD)? This is where you write the tests first, and the software second. This is a popular software development approach.*]

Be aware that the results are only as good as the tests you write. If the tests are not very extensive, then the likelihood of the software being correct is low. When you think about testing, you want to think about the following:

1. Have I tested every public function?
2. Since the class is templated, have I tested different types (int, double, string)?
3. Have I tested boundary conditions, e.g. jagged rows? Accessing the first and last row? Accessing the first and last column? Growing a matrix where the new sizes are smaller?

Testing is a topic we’ll discuss further in class, and lab.

Programming Environment

You are free to program on whatever platform you want, using whatever compiler / programming environment you want. The provided “mymatrix.h” file is available in the course dropbox under Projects, project02-files. If you do not have a C++ programming environment that you like, we recommend **Codio**; a project has been provided named “**cs251-project02-mymatrix**”. You can join Codio via the following link.

Be aware that we are using **Gradescope** as our grading platform, and it’s common for C++ programs to “work on my platform” but fail on Gradescope. This is due to logic errors in *your* program, not an error with Gradescope. The most common mistake is a memory-related error, e.g. using an uninitialized variable or accessing memory outside the bounds of an array or via an invalid pointer. These errors are hard to find; the tools **valgrind** and **cppcheck** (available on Codio) can help; note that valgrind will also report memory leaks, ignore those messages (we don’t care about memory leaks in this project).

Gradescope is running on Ubuntu Linux, and we are compiling via **g++** with **-std=C++11**. Do not ask us to change the C++ version; we are compiling against C++ 11.

Requirements

1. The implementation of a matrix must follow the diagram shown at the bottom of page 1, and as defined in the provided "mymatrix.h" file. In other words, a matrix must remain a pointer to an array of ROW structures, where each ROW contains a pointer to an array of elements of type T. You cannot switch to a vector-based implementation, nor other data structures.
2. Feel free to define additional variables or functions. However, define them as private member variables and functions. No global or static variables.
3. Your "mymatrix.h" program file must have a header comment with your name and a class overview. Much of this has already been written for you. Likewise, each function must have a header comment above the function, explaining the function's purpose, parameters, and return value (if any). Inline comments should be supplied as appropriate; comments like "*declares variable*" or "*increments counter*" are useless. Comments that explain non-obvious assumptions or behavior *are* appropriate.
4. You need to put some effort into testing by writing a "main.cpp" file and submitting that for evaluation along with your "mymatrix.h" file. You can expect your testing code to be evaluated roughly as "bad", "okay", "good", or "excellent". Excellent means you tested every member function in a non-trivial way.
5. Do not worry about freeing memory, or writing a destructor. Ignore this for now, since freeing memory often triggers hard-to-find pointer errors. We'll worry about freeing memory in future projects; for now, enjoy leaking memory without worry :-)

Grading, electronic submission, and Gradescope

Submission: "mymatrix.h" and "main.cpp". The latter is your testing code.

Your score on this project is based on two factors: (1) correctness as determined by your Gradescope submission, and (2) manual review by the TAs for commenting, style, and approach (e.g. quality of testing code). The entire project is worth 150 points: 100 points for correctness, and 50 points for commenting, style, and approach. In this class we expect all submissions to compile, run, and pass at least some of the test cases; do not expect partial credit with regards to correctness. Example: if you submit to Gradescope and your score is a reported as a 0, then that's your correctness score. The only way to raise your correctness score is to re-submit and obtain a higher score by passing one or more test cases. You have unlimited submissions on this project assignment.

Note that the TAs will also review for adherence to requirements; breaking a requirement can result in a final score of 0 out of 150. We take all requirements seriously.

A bonus of 10% is earned for submitting by the "Bonus" deadline on page 1. Bonus points are accumulated and can be applied to a future project. To earn bonus points, your submission must post before the Bonus deadline, earn a correctness score of 100, *and* meet all project requirements. For example, you cannot submit a correct solution without testing code and expect to earn bonus points. Bonus points are reserved for early and well-written submissions.

To submit to Gradescope, you must first create an account; check your UIC email for an invitation to Gradescope. If you cannot find this invitation email, post privately on Piazza and we will send another invite; you cannot register yourself. Gradescope is running on Ubuntu Linux, and we are using **g++** with **-std=C++11**. Do not ask us to change the C++ version; we are compiling against C++ 11. You can submit a set of files or a .zip; we'll extract the files we need: "mymatrix.h" and "main.cpp".

By default, we grade your **last** submission. Gradescope keeps a complete submission history, so you can **activate** an earlier submission if you want us to grade a different one; this must be done before the due date. We assume *every* submission on your Gradescope account is your own work; do not submit someone else's work for any reason, otherwise it will be considered academic misconduct.

Policy

Late work *is* accepted. You may submit as late as 24 hours after the deadline for a penalty of 10%. After 24 hours, no submissions will be accepted.

All work submitted for grading *must* be done individually. While we encourage you to talk to your peers and learn from them (e.g. your "iClicker teammates"), this interaction must be superficial with regards to all work submitted for grading. This means you *cannot* work in teams, you cannot work side-by-side, you cannot submit someone else's work (partial or complete) as your own. The University's policy is available here:

https://dos.uic.edu/conductforstudents.shtml .

In particular, note that you are guilty of academic dishonesty if you extend or receive any kind of unauthorized assistance. Absolutely no transfer of program code between students is permitted (paper or electronic), and you may not solicit code from family, friends, or online forums. Other examples of academic dishonesty include emailing your program to another student, copying-pasting code from the internet, working in a group on a homework assignment, and allowing a tutor, TA, or another individual to write an answer for you. It is also considered academic dishonesty if you click someone else's iClicker with the intent of answering for that student, whether for a quiz, exam, or class participation. Academic dishonesty is unacceptable, and penalties range from a letter grade drop to expulsion from the university; cases are handled via the official student conduct process described at https://dos.uic.edu/conductforstudents.shtml .

Appendix: “mymatrix.h”

```
/*mymatrix.h*/

//
// mymatrix
//
// The mymatrix class provides a matrix (2D array) abstraction.
// The size can grow dynamically in both directions (rows and
// cols).  Also, rows can be "jagged" --- i.e. rows can have
// different column sizes, and thus the matrix is not necessarily
// rectangular.  All elements are initialized to the default value
// for the given type T.  Example:
//
//   mymatrix<int>  M;  // 4x4 matrix, initialized to 0
//
//   M(0, 0) = 123;
//   M(1, 1) = 456;
//   M(2, 2) = 789;
//   M(3, 3) = -99;
//
//   M.growcols(1, 8);  // increase # of cols in row 1 to 8
//
//   for (int r = 0; r < M.numrows(); ++r)
//   {
//      for (int c = 0; c < M.numcols(r); ++c)
//         cout << M(r, c) << " ";
//      cout << endl;
//   }
//
// Output:
//   123 0 0 0
//   0 456 0 0 0 0 0 0
//   0 0 789 0
//   0 0 0 -99
//

#pragma once

#include <iostream>
#include <exception>
#include <stdexcept>

using namespace std;

template<typename T>
class mymatrix
{
private:
  struct ROW
  {
    T*  Cols;     // dynamic array of column elements
    int NumCols;  // total # of columns (0..NumCols-1)
  };

  ROW* Rows;     // dynamic array of ROWs
  int  NumRows;  // total # of rows (0..NumRows-1)
```

```
public:
  //
  // default constructor:
  //
  // Called automatically by C++ to construct a 4x4 matrix.  All
  // elements are initialized to the default value of T.
  //
  mymatrix()
  {
    Rows = new ROW[4];  // an array with 4 ROW structs
    NumRows = 4;

    // initialize each row to have 4 columns:
    for (int r = 0; r < NumRows; ++r)
    {
      Rows[r].Cols = new T[4];  // an array with 4 elements of type T:
      Rows[r].NumCols = 4;

      // initialize the elements to their default value:
      for (int c = 0; c < Rows[r].NumCols; ++c)
      {
        Rows[r].Cols[c] = T{};  // default value for type T:
      }
    }
  }

  //
  // parameterized constructor:
  //
  // Called automatically by C++ to construct a matrix with R rows,
  // where each row has C columns. All elements are initialized to
  // the default value of T.
  //
  mymatrix(int R, int C)
  {
    if (R < 1)
      throw invalid_argument("mymatrix::constructor: # of rows");
    if (C < 1)
      throw invalid_argument("mymatrix::constructor: # of cols");

    //
    // TODO
    //
  }


  //
  // copy constructor:
  //
  // Called automatically by C++ to construct a matrix that contains a
  // copy of an existing matrix.  Example: this occurs when passing
  // mymatrix as a parameter by value
  //
  //   void somefunction(mymatrix<int> M2)  <--- M2 is a copy:
  //   { ... }
  //
  mymatrix(const mymatrix<T>& other)
  {
    //
    // TODO
    //
  }


  //
  // numrows
  //
  // Returns the # of rows in the matrix.  The indices for these rows
  // are 0..numrows-1.
  //
  int numrows() const
  {
    //
    // TODO
    //

    return -1;
  }


  //
  // numcols
  //
  // Returns the # of columns in row r.  The indices for these columns
  // are 0..numcols-1.  Note that the # of columns can be different
  // row-by-row since "jagged" rows are supported --- matrices are not
  // necessarily rectangular.
  //
  int numcols(int r) const
  {
    if (r < 0 || r >= NumRows)
      throw invalid_argument("mymatrix::numcols: row");

    //
    // TODO
    //

    return -1;
  }


  //
  // growcols
  //
  // Grows the # of columns in row r to at least C.  If row r contains
  // fewer than C columns, then columns are added; the existing elements
  // are retained and new locations are initialized to the default value
  // for T.  If row r has C or more columns, then all existing columns
  // are retained -- we never reduce the # of columns.
  //
  // Jagged rows are supported, i.e. different rows may have different
  // column capacities -- matrices are not necessarily rectangular.
  //
  void growcols(int r, int C)
  {
    if (r < 0 || r >= NumRows)
      throw invalid_argument("mymatrix::growcols: row");
    if (C < 1)
```

```
      throw invalid_argument("mymatrix::growcols: columns");

    //
    // TODO:
    //
  }


  //
  // grow
  //
  // Grows the size of the matrix so that it contains at least R rows,
  // and every row contains at least C columns.
  //
  // If the matrix contains fewer than R rows, then rows are added
  // to the matrix; each new row will have C columns initialized to
  // the default value of T.  If R <= numrows(), then all existing
  // rows are retained -- we never reduce the # of rows.
  //
  // If any row contains fewer than C columns, then columns are added
  // to increase the # of columns to C; existing values are retained
  // and additional columns are initialized to the default value of T.
  // If C <= numcols(r) for any row r, then all existing columns are
  // retained -- we never reduce the # of columns.
  //
  void grow(int R, int C)
  {
    if (R < 1)
      throw invalid_argument("mymatrix::grow: # of rows");
    if (C < 1)
      throw invalid_argument("mymatrix::grow: # of cols");

    //
    // TODO:
    //
  }


  //
  // size
  //
  // Returns the total # of elements in the matrix.
  //
  int size() const
  {
    //
    // TODO
    //

    return -1;
  }


  //
  // at
  //
  // Returns a reference to the element at location (r, c); this
  // allows you to access the element or change it:
  //
```

```
//    M.at(r, c) = ...
//    cout << M.at(r, c) << endl;
//
T& at(int r, int c)
{
  if (r < 0 || r >= NumRows)
    throw invalid_argument("mymatrix::at: row");
  if (c < 0 || c >= Rows[r].NumCols)
    throw invalid_argument("mymatrix::at: col");

  //
  // TODO
  //

  T temp = {};  // we need to return something, so a temp for now:

  return temp;
}


//
// ()
//
// Returns a reference to the element at location (r, c); this
// allows you to access the element or change it:
//
//    M(r, c) = ...
//    cout << M(r, c) << endl;
//
T& operator()(int r, int c)
{
  if (r < 0 || r >= NumRows)
    throw invalid_argument("mymatrix::(): row");
  if (c < 0 || c >= Rows[r].NumCols)
    throw invalid_argument("mymatrix::(): col");

  //
  // TODO
  //

  T temp = {};  // we need to return something, so a temp for now:

  return temp;
}

//
// scalar multiplication
//
// Multiplies every element of this matrix by the given scalar value,
// producing a new matrix that is returned.  "This" matrix is not
// changed.
//
// Example:  M2 = M1 * 2;
//
mymatrix<T> operator*(T scalar)
{
  mymatrix<T> result;

  //
```

```
    // TODO
    //

    return result;
  }


  //
  // matrix multiplication
  //
  // Performs matrix multiplication M1 * M2, where M1 is "this" matrix and
  // M2 is the "other" matrix.  This produces a new matrix, which is returned.
  // "This" matrix is not changed, and neither is the "other" matrix.
  //
  // Example:  M3 = M1 * M2;
  //
  // NOTE: M1 and M2 must be rectangular, if not an exception is thrown.  In
  // addition, the sizes of M1 and M2 must be compatible in the following sense:
  // M1 must be of size RxN and M2 must be of size NxC.  In this case, matrix
  // multiplication can be performed, and the resulting matrix is of size RxC.
  //
  mymatrix<T> operator*(const mymatrix<T>& other)
  {
    mymatrix<T> result;

    //
    // both matrices must be rectangular for this to work:
    //

    //
    // TODO
    //
    // if (this matrix is not rectangular)
    //   throw runtime_error("mymatrix::*: this not rectangular");
    //
    // if (other matrix is not rectangular)
    //   throw runtime_error("mymatrix::*: other not rectangular");

    //
    // Okay, both matrices are rectangular.  Can we multiply?  Only
    // if M1 is R1xN and M2 is NxC2.  This yields a result that is
    // R1xC2.
    //
    // Example: 3x4 * 4x2 => 3x2
    //

    //
    // TODO
    //
    // if (this matrix's # of columns != other matrix's # of rows)
    //   throw runtime_error("mymatrix::*: size mismatch");

    //
    // Okay, we can multiply:
    //

    //
    // TODO
    //
    return result;
  }


  //
  // _output
  //
  // Outputs the contents of the matrix; for debugging purposes.
  //
  void _output()
  {
    for (int r = 0; r < this->NumRows; ++r)
    {
      for (int c = 0; c < this->Rows[r].NumCols; ++c)
      {
        cout << this->Rows[r].Cols[c] << " ";
      }
      cout << endl;
    }
  }

};
```