# NIU CSCI 340 GRADE-O-MATIC ASSIGNMENT

## WRITING A 2D GRID CLASS

### INTRODUCTION

There are a lot of situations that come along where it'd be really nice to be able to store data in higher than one dimension. You'll come across several of them later in the course, but you're ready to start working with one of them now.

For this assignment, you'll be implementing a class that provides resizeable 2D grid functionality. It allows you to access its contents via coordinates, whether they're (x,y) or (row, column). It'll also provide access to the individual rows and columns via iterators.

### YOUR TASK

- I've given you the declaration of a class called `grid_row_major` in the file `grids.decl.h`. You need to implement all of the methods that have not already been provided, putting that code into `grids.h`. The autograder will always use *its* version of the `grids.decl.h` so don't write your code in such a way that it depends on that file being changed.
- Develop a class to implement the column iterators properly. I've given you the declaration of something that can be made to work, but you are not required to use it, if you'd rather solve it in a different way. The methods that provide these iterators have `auto`-deduced return types so you can return anything that would work as an appropriate iterator in them.
- Implement the `matrix_multiply(lhs,rhs,result)` function.

### NOTES

You should do all of your work in `grids.h`. Changes made in other files will not be used by the autograder.

You should feel free to create whatever files you want to test things locally, including writing your own simple programs to test small parts on their own. I actually *encourage* you to write unit test programs for yourself, but they should not be a part of your submission. Only `grids.h` should be submitted.

### OUR GRID CLASS: `grid_row_major`

You can find the declaration for `grid_row_major` in `grids.decl.h`. Before we get into a discussion of the methods and what you need to do to implement them, let's talk about how the data will actually be stored.

Each instance of our class will have some random access container actually holding the data for the grid. To provide access to that data as a 2D grid, we'll need to map our coordinates to the cells in that array-like object. One really common way to do this is called "row major order", where all of the elements of one row are listed next to each other, and one row starts after all the elements of the last one have been listed. If we were using a 25-element `vector` as a $5 \times 5$ grid, it would be organized like this:

| | | | | |
|---|---|---|---|---|
| 0 | 1 | 2 | 3 | 4 |
| 5 | 6 | 7 | 8 | 9 |
| 10 | 11 | 12 | 13 | 14 |
| 15 | 16 | 17 | 18 | 19 |
| 20 | 21 | 22 | 23 | 24 |

Taking a look at the numbers, the number goes up by one as we move from left to right across rows, and it goes up by the width of the table whenever we move down the columns. It can also be noted that the total number of cells is always the product of the width times the height.

### MEMBERS (DATA/METHODS) OF `grid_row_major`

#### Data Member `data`

- This will be an instance of whatever container you chose to use for your `RA_CONTAINER` template parameter. This will default to a `std::vector` containing the type of element contained. Any container that supports random access iterators should work, in general, but we will mainly use `std::vector` and `std::deque` in practice.

- If you try to instantiate a grid with an `RA_CONTAINER` with a type that isn't a container or with a container type that doesn't support random access, then compilation is not expected to succeed.

**Data Members `_height` and `_width`**

- These keep track of the dimensions of the grid, and you'll need them in order to calculate the index where the data for a particular set of coordinates resides in.

**Constructor `( width, height )`**

- Make the grid width × height, without taking the trouble to change the data values. They'll remain at their default until you change them with something else.

**Constructor `( width, height, begin, end )`**

- This is the STL-style cross-container-type iterator-based copy constructor. The grid is resized to width $× height and values are copied from the iterator range into the grid directly, interpreted, once again, as being in row major format.

**Copy Constructor**

- Construct a new grid by copying the contents of another grid. The new grid gets its width, height, and data from the existing grid.

**Assignment Instructor `operator =`**

- Overwrite this grid by copying in the contents of the other grid. Replace this grid's width, height, and data, copying them from the other grid.

**Dimensions Checking `width()` and `height()`**

- These methods return the width, or the height, respectively, of the grid.

**`empty()`**

- This method should return `true` if, and only if, there are no elements present in the grid.

**Bounds Checking: `boundscheckxy` and `boundscheckrc`**

- These methods take coordinates as parameters – the `xy` version takes (x,y) coordinates and the `rc` version takes (row, column) coordinates. If the coordinates fit within the boundaries of the grid, they will return true, otherwise false. Notice that both sets of coordinates start with 0 as the lowest available.

**Grid Element Access `atxy, atrc`** These functions are used to access (by reference) the location in the grid corresponding to the coordinates provided. These are analogous to the `at` method of the STL sequence containers, but adapted to take two-dimensional grid coordinates instead of a one-dimensional index. Just like `at`, you should perform a bounds check in these and throw a `std::out_of_range` exception if the user tries to go out of bounds.

- `atrc(row, col)`: takes row, column coordinates – row 0 is the first row, and row (height-1) is the last row. Column 0 is the first, leftmost column. These coordinates are typically used by mathematicians (they may decide to start the numbering at 1 instead of 0) and you'll see them used a lot when a program involves linear algebra (like matrix multiplication).
- `atxy(x,y)`: takes screen x,y coordinates – (x=0,y=0) is the top left element, and (x=width-1, y=height-1) is the bottom right. These coordinates are commonly used when doing raster graphics on a computer screen.

**Row Iterator Access `rowbegin`, `rowend`**

- `rowbegin(i)` provides an iterator that points to the first element in row `i` (numbering from 0). `rowend(i)` returns an iterator that points just past the last element on row `i`. These are simple to provide in a row major grid.

### Column Iterator Access `colbegin`, `colend`

- `colbegin(i)` will return a *column* iterator that points to the first element in column `i` (numbering from 0)
- `colend(i)` returns a *column* iterator that points to the position that you'd get to after iterating one time past the last valid element in the column.
- You will need to do some extra work to implement the iterators used here compared to the ones for the rows, as a simple iterator's ++ operation would be moving within a row, and these require you to move through a column. I've provided the declaration for a class that could be used to implement this, but if there's another way that you prefer to handle the problem, feel free to use your own solution, as long as it was you who wrote it and it works.

### `resize()`

- Destructively resize the grid. Do not worry about keeping the data that was previously present intact. Make the `data` container big enough to fit a width × height grid, and change the `_height` and `_width` parameters to the new size.

### `clear()`

- Get rid of all of the data that was being stored, and change the stored height and width accordingly.

### `load_from_file( filename )`

- This method will try to replace the contents of this grid with the data from the specified file. To do this, it will open the file with a `std::fstream` and ask for data with the abstraction operator.
  - Before you try to interpret any of the data, read two integers in from the file. The first one will be the width, and the second will be the height.
  - From that point on, you read values in, extracting the data into whatever type our container holds, and putting them into our backing container *in row order*.
  - If there's too much data in the file, stop reading when you've filled the grid. If there's too little data in the file, print an error message to standard error when you've run out early.

## COLUMN ITERATORS `col_iterator`

- I've provided an outline of a possible class that you could use to implement the column iterators. You are not forced to use the `col_iterator` class if you do not want to, but the code *will* need to work with whatever solution you choose.
- Remember that the point of the column iterators is to ensure that, when you increment the iterator, it points to the next element in the *column*. Our grid is row major, so just incrementing the position by one would leave the current column. You'll need to overload the ++ operator to handle the *correct* movement.
- Unlike the iterators you implemented for your linked list, the iterators provided by your grid's backing container provide random access. It would be useful to implement things like operators `+`, `-`, and `[]`, but this is not required.

## MATRIX MULTIPLICATION

Matrix multiplication is a very useful operation that can be performed on grids that you will likely see used in a lot of application (3D graphics use them extensively). Your function will treat the input grids as matrices and perform that calculation, overwriting the data in the result grid with the result.

In matrix multiplication, each cell (row `i`, col `j`) will contain the "dot product" of row `i` in the left hand side matrix with column `j` in the right hand side matrix. A dot product is the sum of the multiples of corresponding elements.

- $[a_1, a_2, a_3, a_4] \cdot [b_1, b_2, b_3, b_4] = (a_1 \times b_1) + (a_2 \times b_2) + (a_3 \times b_3) + (a_4 \times b_4)$

Details on how to perform matrix multiplication can be found here:

- `https://en.wikipedia.org/wiki/Matrix_multiplication`

If you don't like the Wikipedia article, or if it has changed since the writing of this document to a state that isn't useful, there are many results on the web that you can search for.

As a reminder, you need to write the code based on an explanation of the process, **not** find and use someone else's code.

### `matrix_multiply( lhs, rhs, result )`

**Template Parameters** The `GRID1`, `GRID2`, and `GRID3` template parameters are each (potentially different) classes that implement our grid functionality. What is important is that the `lhs` and `rhs` grids contain elements of types that can be multipled with each other and added to a `double` in a way that makes sense. The `result` grid needs to be able to have `doubles` assigned to it. You won't need to specify these types explicitly to call this function; the compiler will be able to deduce them automatically.

**Parameters**

- `GRID1 lhs` - The grid to be used as the left hand side of the matrix multiplication
- `GRID2 lhs` - The grid to be used as the right hand side of the matrix multiplication
- `GRID3 result` - The grid that will be used to store the result.

**Return Code** This function returns `true` if the grids supplied for `lhs` and `rhs` can be matrix multiplied. If they can't, it should return `false`, and `result` should not be touched.

## HOW TO SUBMIT

Submit your working `grids.h`, and no other files, to the autograder.

## GRADING CONSIDERATIONS

- Does it compile? Does it run? All of the tests should compile and run on turing/hopper/autograder with the `Makefile` provided, and points will be deducted for each test that will not compile.
- Does the output match for all of the tests? Your program's output will be compared to reference output for each of the test programs, so you can compare your program's output to what is expected.
- Did you submit the right file(s) and *only* the right file(s)?
- Indentation aids in the readability of source code, and if you're not indenting your code blocks. The grader evaluating your code for documentation will legitimately dislike you for it. I'm instructing them to mark you off if you subject them to reading that.
- Did you document your code?
  - You need a docbox at the top of every one of the files you're required to change including:
    - Your name
    - Your zid
    - Your course section
    - A description of what the program does
  - You should add a docbox for every function that you implement, explaining what it does and what each parameter is for.
  - Add other comments inside your code blocks describing what you're doing and why.
  - The use of `doxygen` style comments is encouraged, but not required.