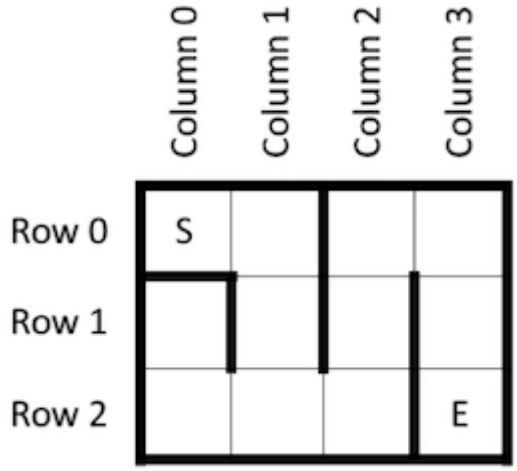
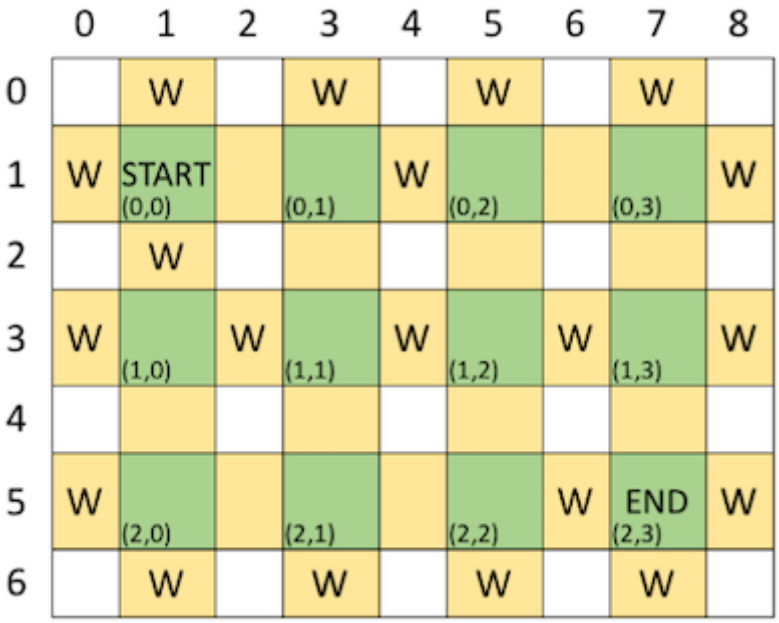
Mazes!

This week you will create a class to represent mazes. Mazes are a bit tricky because the most important things - the walls - sit between the cells. Let's say we have the following maze:



The cells form a 2D grid, which we shall reference with (*row*, *column*) coordinates. Cell (0, 0) is the starting point, and cell (2, 3) is the ending point.

We can represent the maze with an array, but we also need to be able to store the walls somewhere. One approach for representing mazes is to add extra rows and columns between cells, where the walls themselves will be represented. Therefore, the above maze would be stored like this:



The green boxes are the cells themselves, and the yellow boxes represent the walls. The white boxes are unused in this representation, which may seem a bit wasteful, but we won't be creating giant mazes, so we'll pick a clear and slightly wasteful representation, rather than a confusing and extremely efficient one.

You can see that for a maze of size $N_{rows} \times N_{cols}$, our representation requires a total of $(2N_{rows} + 1) \times (2N_{cols} + 1)$ elements. For our above example, with 3 rows and 4 columns, we would need (2 * 3 + 1) × (2 * 4 + 1) cells, or 7×9 cells.

C++ and Multidimensional Arrays

As mentioned in class, C++ doesn't have very good support for dynamically allocated multidimensional arrays; languages like Java are much better at this kind of thing. In C++, we typically allocate a 1-dimensional array with enough space for the total number of elements, and then map our 2D (or 3D, or whatever) coordinates into our 1D array. For example, given a 2D array with N_{rows} rows and N_{cols} columns:

- The program can allocate a 1D array with $N_{rows} \times N_{cols}$ elements
- An element at 2D position (*row*, *col*) can be mapped to $row \times N_{cols} + col$

This mapping should make sense: for each row we are going to skip, we must move past N_{cols} cells to get to the next row. Of course, this is not the only way we can map 2D coordinates into the 1D array, but this is the typical approach for C and C++. It is called a *row-major order*. (A few other languages, like Matlab and Fortran, use a *column-major order* for how they represent multidimensional arrays.)

Accessing Maze Cells and Walls

In order to provide a simple abstraction for people to use, we will implement a `Maze` class that uses *cell coordinates* to reference various aspects of the maze, but internally we will map cell coordinates into the "expanded representation" we use internally. We will call the coordinates in the expanded representation *expanded coordinates*.

You should be able to see how to map cell coordinates into expanded coordinates very easily. Given a maze with $N_{cell\text{-}rows} \times N_{cell\text{-}cols}$ cells, we can map the cell coordinates (r_{cell}, c_{cell}) into expanded coordinates (r_{exp}, c_{exp}) as follows:

- $r_{exp} = 2\, r_{cell} + 1$
- $c_{exp} = 2\, c_{cell} + 1$

Given a particular cell, we will want to know if there is a wall on a particular side of the cell; we will use the cardinal directions "North", "East", "South" or "West" of the cell. Then, we can ask questions like:

- Is there a wall to the west (left) of cell (1, 2) in the original maze? **Yes.**
- Is there a wall to the north of cell (2, 3) in the maze? **No.**

Accessing the walls between cells is similarly straightforward:

- To access the wall north of a given cell, compute the cell's expanded coordinates and subtract 1 from the row value.
- To access the wall south of a given cell, compute the cell's expanded coordinates and add 1 to the row value.
- To access the wall east of a given cell, compute the cell's expanded coordinates and add 1 to the column value.
- To access the wall west of a given cell, compute the cell's expanded coordinates and subtract 1 from the column value.

Your Tasks

This week you will need to complete the implementation of the `Maze` class. There are quite a few member functions to implement, but you should not worry about this since the vast majority of them will end up being only a few lines long, at most. Only a few of these functions will actually require careful thought.

To help you know what you need to do, we are providing you with an initial version of `maze.hh`, which you can download and then rename to `maze.hh`. This file contains the following:

- A declaration of the `Maze` class, which includes all operations you must support. Feel free to add member functions if you wish (in particular, private helper functions), but don't change the public interface that the class specifies!
- A `MazeCell` enumeration that specifies all the values that cells in the expanded representation of the maze can take on. You should not need to change this.
- A `Direction` enumeration that specifies the cardinal directions that are used by the maze code. You should not need to change this.
- A `Location` class that can be used to represent (row, column) locations. You shouldn't need to make any changes to this class.

The reason for the `Location` class is that C++ functions cannot return more than one value, but it is very helpful to be able to return a (row,col) value from various functions. Thus, we can package the row and column value into a `Location` object, and return that object from a function.

The required operations in the maze class are explained in the comments in the `maze.hh` header file. Feel free to ask for any clarifications.

Suggested Approach

You will need to dynamically allocate an array for the expanded representation of the maze using the heap. Since you are dynamically allocating memory, you will need to write a destructor to free the memory, and a copy-constructor and an assignment operator to handle copying and assignment correctly. (That is, you will need to perform a deep copy, not a shallow copy.) This is the **Rule of Three** - if you implement a copy-constructor, assignment operator, or destructor for your class, you really should implement all three.

If you create simple abstractions for your class, you will find that your implementation goes much easier. For example, you might want to create these private helper functions (i.e. declare them in the `private` section of your class):

```
// Take 2D expanded coordinates and compute the corresponding 1D array index
int getArrayIndex(const Location &loc) const;

// Returns the expanded coordinates of the specified cell coordinates
Location getCellArrayCoord(int cellRow, int cellCol) const;

// Returns the expanded coordinates of the wall on a specific side of
// a cell given in cell coordinates
Location getWallArrayCoord(int cellRow, int cellCol,
                           Direction direction) const;
```

Then, your other functions can use these helpers to perform their tasks. By taking this approach, you minimize the number of places where you will have to write the tricky array-indexing operations that are so easy to get wrong. If you have bugs, you will only have to fix them in one place.

Use assertions liberally! Here are some ideas; if you see other places to use assertions, put them there too!

- Check every set of incoming coordinates to be sure they are in the proper range.
- Check row coordinates and column coordinates in separate assertions.
- As indicated in the comments for `getNeighborCell()`, trip an assertion if someone tries to get a neighbor cell that is invalid.

Probably the hardest function to write will be the `print()` function. You can use the << stream-output operator on the `ostream` object, just as if it were `cout`. You may want to follow a structure like the following:

```
Output the number of cell-rows and cell-columns

For each row r from 0 to numRows - 1:
    Output the walls above row r
    Output the cells in row r, along with the walls in between those cells

Finally, output the walls below the last row
```

Testing

You can download this program to exercise your maze code. It performs a pretty exhaustive battery of tests, so by the time your program passes all the tests, you should have a pretty high confidence that it works. You can compile and run the test program like this:

```
g++ -Wall -std=c++14 maze.cc test-maze.cc testbase.cc -o test-maze
./test-maze
```

All Done?

When you have finished your maze program, you should submit your completed `maze.hh` and `maze.cc` files on csman.