# CSCI-1200 Data Structures — Spring 2022
# Homework 3 — Connect Four

In this assignment you will build a custom class named *Board*, which will be used to run a game similar to Connect Four: https://en.wikipedia.org/wiki/Connect_Four

Building this data structure will give you practice with pointers, dynamic array allocation and deallocation, 2D pointers, and class design. The implementation of this data structure will involve writing one new class. You are not allowed to use any of the STL container classes in your implementation or use any additional classes or structs besides `Board` and STL `string`. You will need to make use of the `new` and `delete` keywords. You can use array indexing (`[]`). Please read the entire handout (**there are 4 pages**) before beginning your implementation. Review the Lecture 8 notes and our implementation of the `Vec` class mimicking STL's `vector`. You may also want to review our discussion of 2D dynamic arrays from Lecture 6.

## Gameplay

In our version of the game, the board consists of a grid of at least 5 rows and 4 columns (written as a 5x4 board). The game is played by two players which will each have tokens denoted by a specific string read from standard input. We will count the leftmost column as column 0, and the bottom-most row as row 0.

Three strings will be passed to our board's constructor (details later in the assignment): Player 1's token string, Player 2's token string, and the blank string. For example if the first player will play by placing `R` tokens on the board, the second will use `Y` tokens, and blank spaces will be denoted by `.` then the board would initially be printed as:

```
. . . .
. . . .
. . . .
. . . .
. . . .
```

In a real-world game, gravity would force tokens to fall to the bottom of the board. In our game we will simulate this by saying that when a player chooses a column, their token will fall to the "bottom" of the grid. For example, suppose Player 1 places a piece in column 0. This is accomplished by standard input using `p1 0` ( The board would then look like:

```
. . . .
. . . .
. . . .
. . . .
R . . .
```

Players take turns placing a token in a column of their choosing. If Player 2 wants to place a token in column 1, standard input would contain `p2 1`. You do not need to verify that players are actually taking turns, for example, if after Player 2 placed their token in column 1, Player 1 placed tokens in columns 2 and 3 `p1 2 p1 3` then the board would look like:

```
. . . .
. . . .
. . . .
. . . .
R Y R R
```

Our board is unusual in two ways. The first way is that if a token is placed in a column that is already "full", the board will simply grow larger, allowing the new token to be placed at the top of the column. For example, if Player 1 and Player 2 alternate placing tokens in column 1 twice, and then Player 1 places another token in column 1, the board will look like:

```
. R . .
. Y . .
. R . .
. Y . .
. R . .
R Y R R
```

The second unusual thing about the board is that if a player tries to place a token in a positive column that doesn't exist yet, the board will grow to the right (column 0 is still the left-most column) to fit. For example, if Player 1 places a token in column 4, the board will then look like:

```
. R . . .
. Y . . .
. R . . .
. Y . . .
. R . . .
R Y R R R
```

The final concept to introduce before moving on to implementation details is the idea of "connecting four", which is the winning condition of the game. If four tokens belonging to the same player are in a vertical or a horizontal line, they have "connected four" and win the game. Note that unlike the actual Connect Four game, we do **not** count diagonal lines for connecting four.

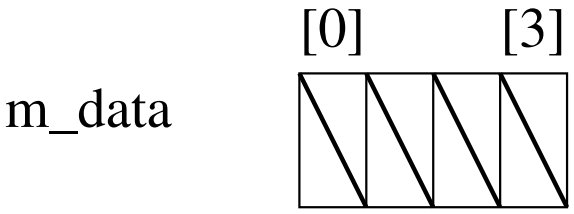
### Board Representation

The board itself is part of a larger Board class, and is represented by a member variable, *m_data*. Before we get into the Board class, first consider the actual board itself, which should be implemented as a 2-D string array. Below are diagrams for the initial board, and the last board that was shown above. Keep in mind that individual columns should never be larger than they need to be, which may result in column pointers that are `NULL`. Your implementation should follow this memory layout as closely as possible to receive full credit.

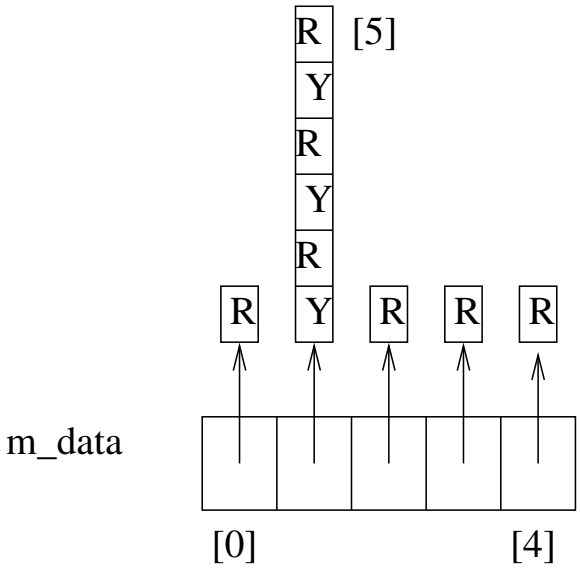
In the memory diagrams on the following page, *m_data* points to an array of column pointers, with index 0 corresponding to the left-most column of the board (column 0). Within each column array, index 0 corresponds to row 0, index 1 corresponds to row 1, and so on.

**IMPORTANT:** You must always make sure to only use as much dynamic memory as is needed. In other words, the "Initial layout" example on the next page would only have an array of 4 null pointers, while the "last example in Gameplay section" would have an array of 5 pointers leading to (from left to right) an array of 1 string, an array of 6 strings, and finally three more arrays each holding one string. This is to make sure you get lots of dynamic memory and pointer practice!

Initial layout, with a 5x4 board that is all empty.




Board corresponding to the last example in the Gamplay section.




### The `Board` Class

You must write the `Board` class. It should have a constructor that takes in three three strings: Player 1's token string, Player 2's token string, and the blank string in that order. The Board class must work with std::cout (by using `operator<<`), and should print boards formatted like the ones shown earlier in this handout. Your class should also have a *clear()* function that restores the Board to its initial state.

*numRows()* and *numColumns()* return the number of rows and columns in the board respectively. *numTokensInRow()* should take a single `int`, the row number, and should return how many tokens are in that row (or -1 if it doesn't exist).

*numTokensInColumn()* also takes a single `int`, the column number, and returns how many tokens are in that column (or -1 if it doesn't exist).

Finally, your class should have a function called *insert()* that takes two arguments, an `int` column number, and a `bool` which is true if player 1 is placing a token, and false if player 2 is placing a token. *insert()* should place the token (you can assume we will only give non-negative column numbers) and then check if there is any connected four on the board. It should return a string which is the blank spaces string (`.` in our example) if there is no connected four, otherwise it should return the player's token string corresponding to which player won (`R` or `Y` in our example). Assume that we will not call *insert()* on a board that already has a winner.

To check for a connected four, you must write an iterative (not recursive) algorithm. You are free to get creative with how you check, but the simplest solution is to start at a corner of the board and check if that position has 3 adjacent tokens of the same type, and if it doesn't, repeat this step for another starting position, until you've tried starting from all locations on the board. *This is not necessarily the best way to do the search, but for this assignment we are concerned more with representing the data structure and navigating it as opposed to the most efficient algorithm design.*

Beyond the required methods listed above and shown in the starter code (see below), you are welcome to add any additional member functions and member variables that you want to the class.

## Testing and Debugging

We provide a `connect_four_main.cpp` file with a variety of tests of the `Board` class. We recommend that you get your class working on the basic tests, and then write your own tests as you proceed. Study the provided test cases to understand the arguments and return values.
*Note: Do not edit the provided* `connect_four_main.cpp` *file, except to add your own test cases where specified.*

The `assert()` function is used throughout the test code. This is a function available in both C and C++ that will do nothing if the condition is true, and will cause an immediate crash if the condition is false. If the condition is false, your command line should show the assertion that failed immediate prior to the crash. See the additional materials on the course calendar under Lecture 4 for the example file, `assert_demo.cpp` .

We recommend using a memory debugging tool to find memory errors and memory leaks. Information on installation and use of the memory debuggers "Dr. Memory" (available for Linux/MacOSX/Windows) and "Valgrind" (available for Linux/OSX) is presented on the course webpage:

http://www.cs.rpi.edu/academics/courses/spring22/csci1200/memory_debugging.php

The homework submission server will also run your code with Dr. Memory to search for memory problems. Your program must be memory error free and memory leak free to receive full credit.

## Your Task & Provided Code

You must implement the `Board` class as described in this handout. Your class should be split between a `.cpp` and a `.h` file. You should also include some extra tests in the *StudentTest()* function in `connect_four_main.cpp`.

When implementing the class, pay particular attention to correctly implementing the copy constructor, assignment operator, and destructor. The `Vec<T>` class we looked at in lecture is templated, while the `Board` class is not. You may still find the lecture notes to be a useful starting point, but remember that in the `Board` class we should not be writing `template class<T>` anywhere. As you implement your classes, be careful with return types, the `const` keyword, and passing by reference.

If you have correctly implemented the `Board` class, then running the provided `connect_four_main.cpp` file with your class, should produce the output provided in `sample_output.txt`. Your whitespace must match the output exactly to receive full credit.

In the `README.txt` there are several complexity questions that you will also need to answer. These will be graded by the TAs.

We recommend you compile using the `-Wall -Wextra -Werror=vla` flags to help with debugging and avoiding accidentally not using dynamic memory. Submitty will also be compiling using these flags.

## Submission

You will need to submit your `connect_four_main.cpp`,`Board.cpp`,`Board.h`, and `README.txt` file. Be aware that Submitty will be using an instructor copy of `connect_four_main.cpp` for most of the tests, so you must make sure your `Board` implementation can compile given the provided file.

Be sure to write your own new test cases and don't forget to comment your code! Use the provided template `README.txt` file for notes you want the grader to read. Fill out the order notation section as well in the `README.txt` file. You must do this assignment on your own, as described in the "Collaboration Policy & Academic Integrity" handout. If you did discuss this assignment, problem solving techniques, or error messages, etc. with anyone, please list their names in your `README.txt` file.

**NOTE:** If you receive 9 points total on Test Cases 4, 7, and 8 by the end of Wednesday 11:59pm, you will earn a 1 day extension on HW3.