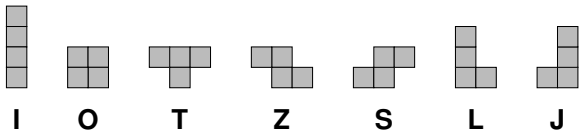
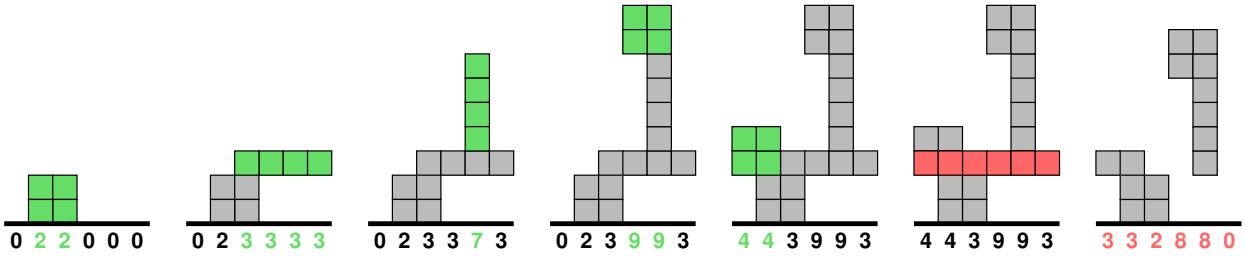
CSCI-1200 Data Structures — Fall 2021
Homework 3 — Dynamic Tetris Arrays

In this assignment you will use dynamically-allocated arrays to keep track of blocks on the 2D grid of the classic *Tetris* computer game. Follow these links to read about the history of the game and play an online interactive version: http://en.wikipedia.org/wiki/Tetris http://tetris.com/play-tetris/

There are 7 different piece shapes in the game of Tetris – each named by the letter that most closely matches its shape. Each piece is built from 4 small squares and can optionally be rotated *clockwise* by 90°, 180°, or 270°, before being dropped onto the board. The different pieces are drawn below in their default orientations (0° rotation).



In each step of the game, a piece will fall down vertically from the sky onto the board – stopping when it collides either with the bottom of the board, or with another piece. Below is a small example of game play.

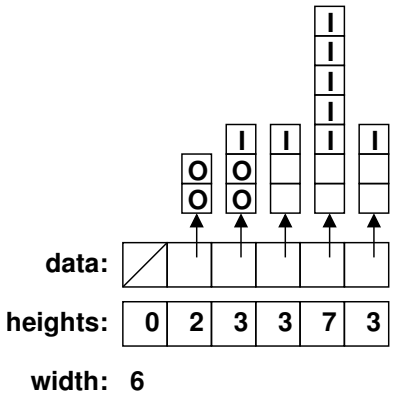


The sequence of commands on the right will create the above example. We start by constructing an empty board with width = 6. Then we drop 5 shapes onto the board using the `add_piece` member function. This function takes in 3 arguments: a single character naming the piece shape, an integer indicating piece rotation from default orientation, and an integer specifying the horizontal placement of the leftmost square of the piece. Each new piece is highlighted in green.

```
Tetris tetris(6);
tetris.add_piece('O', 0,1);
tetris.add_piece('I',90,2);
tetris.add_piece('I', 0,4);
tetris.add_piece('O', 0,3);
tetris.add_piece('O', 0,0);
```

The final two frames of the sequence above illustrate the scoring mechanism for Tetris. When a horizontal row stretching across the entire board is filled with blocks (no air gaps!), then that entire row (shown in red) is deleted. All squares above the deleted row are shifted down by 1 unit (or shifted n units if n rows were deleted). Note that we might be left with squares that are unsupported and floating. These unsupported squares *do not* fall after a score. The member function `remove_full_rows` should search the board for rows that are full and should be removed. This function returns the number of rows removed.

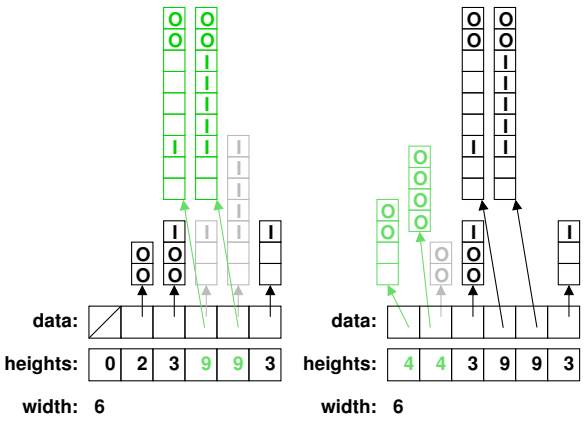
In the illustrations above we track the integer height of each column of the board. This is not the count of how many squares are in each column, but the height of the topmost square in that column.



The Data Representation

Your task is to implement a class named `Tetris` with a `tetris.h` header file and `tetris.cpp` implementation file. You are required to follow the specific memory layout diagrammed on the right (which corresponds to the 3rd step of the earlier example). The class has 3 member variables: the integer `width`, an array of integer column `heights`, and a two dimensional array structure holding the character column `data`. *NOTE: You may not use the STL* `vector` *class for the implementation of this assignment.*

As the game progresses and the board changes, the data structure must be updated. When we add the next piece we must first identify which columns increase in height and by how much. New column arrays of the appropriate size are dynamically allocated, data from the old column arrays are copied to the new arrays, and the new piece squares are filled in with the appropriate piece letter. The old arrays are deleted, pointers are shuffled, and the `heights` array is updated with the new column array sizes. Note that often pockets of air will be trapped under pieces. We will represent air with the space character.



Similarly, when a full row is identified during scoring, all of the columns will decrease in height. You are required to re-allocate new, smaller arrays for every column and copy the remaining squares of data to the new arrays. *The board representation should always use the smallest size arrays possible.* The data structure for this assignment (intentionally) involves a lot of memory allocation & deallocation. Certainly there are other implementation designs for the game of Tetris, and it is possible to revise this design for improved performance and efficiency. However, please implement the data structure *exactly* as presented. Our version of Tetris also allows us to increase or decrease the width of the board. Note that when the board is decreased in width, we may lose some squares. See the provided `main.cpp` for examples of these functions and the corresponding expected program output. Draw your own diagrams to understand which parts of the data structure must be reallocated and copied, and which parts can be reused just by changing pointers.

The `clear` function should clean up all the blocks on the board (and deallocate the arrays storing the blocks) leaving an empty board. You will also need to implement the destructor, which deallocates *all* of the dynamically allocated member variables for the class so you don't have any memory leaks. Finally, you should also implement the copy constructor and assignment operator for the `Tetris` data structure.

Most of the provided test cases use only the '`I`' and '`O`' shapes, so you are encouraged to limit your implementation to these shapes initially. Once you have all other parts of the assignment complete, you can generalize your implementation. A full implementation of the complete set of shapes will be worth extra credit.

Testing, Debugging, and Printing

We provide a `main.cpp` file with a variety of tests of the `Tetris` data structure. Some of these tests are initially commented out. We recommend that you get your class working on the basic tests, and then uncomment the additional tests as you implement and debug the remaining functionality. Study the provided test cases to understand the arguments and return values. We provide an ASCII art print function, and your board output should match the samples exactly to receive full marks on the submission server.
Note: Do not edit the provided `main.cpp` *file, except to uncomment the provided test cases as you work through your implementation and to add your own test cases where specified.*

You should use a memory debugging tool to find memory errors and memory leaks – e.g., "Dr. Memory" (available for Linux/MacOSX/WSL/Windows) or "Valgrind" (available for Linux/WSL).

http://www.cs.rpi.edu/academics/courses/fall21/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.

Submission

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. 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.