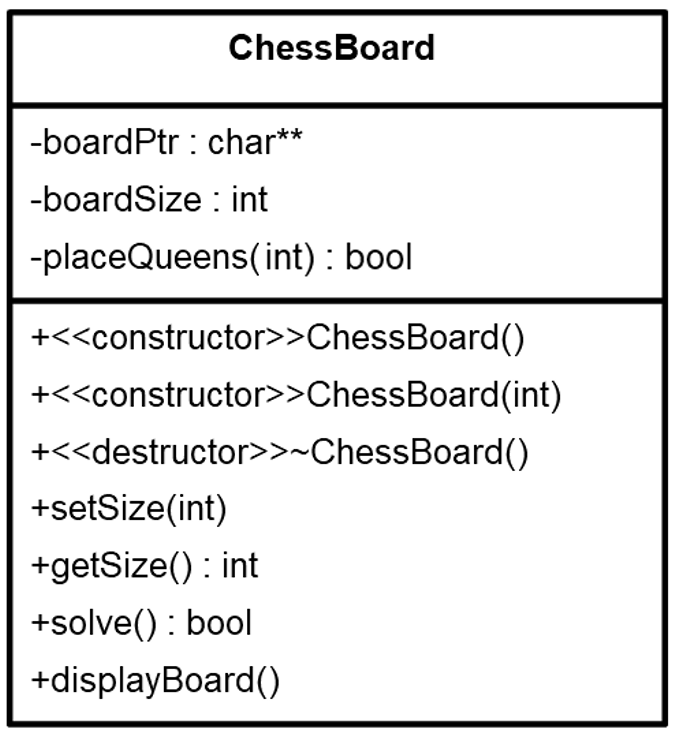
Programming Project: N-Queens

Prerequisites: Chapter 5

For this project you will create a class that solves a variation of the 8-Queens Problem, the N-Queens Problem. If you need a recap of the 8-Queens Problem [watch this video](https://www.youtube.com/watch?v=jPcBU0Z2Hj8).

The N-Queens Problem is identical to the 8-Queens Problem *except* we vary the size of the board. So, we can find the placement of queens on a 4x4 board (the 4-Queens Problem) or a 9x9 board (the 9-Queens Problem), etc.

Your solution to the N-Queens Problem will be implemented entirely in the class ChessBoard which will at minimum include:



You should break additional functionality down into private helper methods, but do not add additional public methods.

Specifications for the items in the UML diagram:

|  |
| --- |
| boardPtr - pointer to a 2-dimensional character array on the heap which stores the board. If you need a refresher on how 2-D arrays on the heap work [see here](https://www.tutorialcup.com/cplusplus/dynamic-memory.htm) and scroll down to the section titled “Dynamic memory allocation for two-dimensional arrays.” |
| boardSize – Tracks the size of the board. Valid sizes are 4 through 12. For example, if boardSize is 10 the board is 10 x 10. |
| placeQueens(int column) – Implements the provided recursive pseudocode (see the end of this document) to find placements for the queens on the board.  The call placeQueens(column) begins the search for a valid placement of a queen at the indicated column in row 0. For instance, placeQueens(4) begins checking to see if a queen can be placed in row index 0, column index 4 on the board.  placeQueens() returns true if a queen is successfully placed in column *and* all columns to the right of column. |
| ChessBoard() – Default constructor which sets the default board size to 8. |
| ChessBoard(int) – Constructor which sets the board size to the given int. If the int < 4 it sets the size to 4, if the int > 12 it sets the size to 12. |
| setSize(int) – Deletes the existing board and resets the board to be the size given.  For instance, if setSize(9) is called the ChessBoard object should now contain an 9 x 9 board. If the int < 4 it sets the size to 4, if the int > 12 it sets the size to 12. |
| getSize() – Returns the current size of the board.  For instance, if the board is 10 x 10, getSize() returns 10. |
| solve() – Non-recursive function that calls the recursive placeQueens(). Returns true if the board was solved successfully.  Once solve() is finished, if a solution was successfully found, the 2-dimensional array pointed to by boardPtr should contain the solution. |
| displayBoard() – Displays the board to the screen. Queens are displayed as a Q. Blank squares on the board should be displayed as a single \*. Individual squares on the board display with 1 space between them. So, a 4x4 board is displayed as:  \* \* Q \*  Q \* \* \*  \* \* \* Q  \* Q \* \* |

## Additional Requirements

* ChessBoard must be implemented as ChessBoard.h and ChessBoard.cpp.
* Your ChessBoard must work with the supplied main.cpp.
* You must implement placeQueens() recursively based on the algorithm given in the pseudocode at the end of this document. Iterative solutions or recursive solutions drastically different from the pseudocode will not be accepted.
* Your project must meet the requirements of the course style guide.

## Turn In

Upload your ChessBoard.h and ChessBoard.cpp prior to the due date.

## Algorithm

This is a modified version of the algorithm from the textbook.

placeQueens(int **column**)

{

int **row** = 0

if (**column** >= board size)

{

The board is filled, problem is solved. Return true.

} // end if

else

{

while (unconsidered rows exist in **column**)

{

if ([row][column] is unattacked)

{

Place a queen in the un-attacked square.

Do a recursive call to try and place queens

in subsequent columns:

if (!placeQueens(**column** + 1))

{

If we’re here, placement of the last queen resulted in a dead end; no solution could be found. Remove the last queen placed.

Move to next **row** so search can continue in next iteration.

} // end if

else

{

If we’re here, recursive calls were able to place queens in all columns to the right of **column,** the problem is solved. Return true.

} // end else

} // end if

else

{

Square is attacked, move to next **row**.

} // end else

} // end while

All rows have been considered in **column** without a successful queen placement. Backtrack by returning false.

} // end else

} // end placeQueens()

Jacob Kier

Johnson County Community College