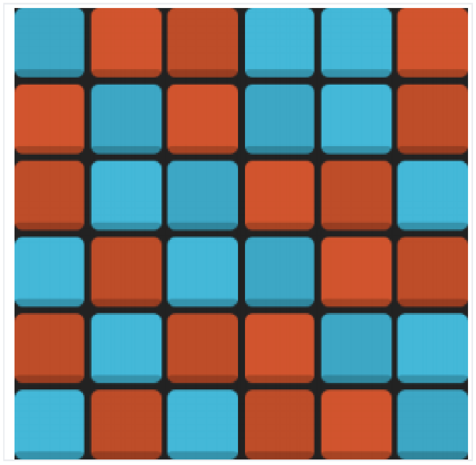
p3-0hh1

EECS 183 Project 3: 0hh1

Project Due Friday, October 21, 2022, 11:59 pm Eastern

Direct autograder link



In this project, you will develop a command-line application to read, check, solve, and play basic instances of 0h h1, a Sudoku-like puzzle game.

By completing this project, you will learn to:

- Write algorithms using 2-dimensional arrays
- Create test cases for complex functions
- Design programs with several layers of function calls
- Write functions with mixtures of pass-by-value, pass-by-reference, and array parameters
- Separate code into multiple source and header files

You will apply the following skills you learned in lecture:

- Lecture 8
  - Write algorithms with nested loops
- Lecture 9
  - Write functions with pass by reference parameters and call them from other functions
  - Identify and apply patterns used to return values out of functions that use pass by value and pass by reference
- Lecture 11
  - Declare arrays
  - Access elements in arrays
  - Pass arrays to functions

- Lecture 12
    - Access elements in multi-dimensional arrays
    - Traverse a 2-dimensional array by row and by column

Getting Started

Overview Video

Here is a project overview video that summarizes how to get started with the project:

Starter Files

Begin by heading over to 0hh1.com and playing the game to gain familiarity with the rules.

After downloading the distribution code at this link, you'll find these files:

File	Role	What you will do
`ohhi.cpp`	Contains the functions you will implement for this project.	**Implement stubs, submit to autograder**
`test.cpp`	Contains the test cases for your project.	**Implement test cases, submit to autograder**
`ohhi.h`	Contains the RMEs and declarations for the functions you will implement in ohhi.cpp.	Do not modify or submit
`utility.h`, `utility.cpp`	Contains helper functions you may call in your code.	Do not modify or submit
`driver.h`, `driver.cpp`, `main.cpp`, `color.h`,	Code that calls your functions to play the 0hh1 game.	Do not modify or submit

`color.cpp`, `main.cpp`		
`start.cpp`	Code that includes the main function. Use this to choose between playing 0hh1 or running your tests.	Do not modify or submit

We recommend you write one function at a time. Similar to P2, the best strategy is to write test cases in `test.cpp` for a function, then write the function in `ohhi.cpp` .

Submission and Grading

Submit your code to the autograder here. You receive 4 submits each day and your best overall submission counts as your score. *You will submit two files, which must be called `ohhi.cpp` and `test.cpp` .*

The deadline is Friday, October 21, 2022 at 11:59PM Eastern. If your last submission is on Wednesday, October 19 by 11:59PM, you will receive a 5% bonus. If your last submission is on Thursday, October 20 by 11:59PM, you will receive a 2.5% bonus.

Here is a grade breakdown:

- **60 points correctness.** Implement functions in `ohhi.cpp` . To what extent does your code implement the features required by our specification? To what extent is your code consistent with our specifications and free of bugs?
- **10 points testing.** Write a test suite in `test.cpp` that is able to expose a range of bugs for the functions you implement.
- **10 points style.** To what extent is your code written well? To what extent is your code readable? Check the Style rubric for the project and the EECS 183 Style Guide to make sure you are conforming to our style guidelines. *Only `ohhi.cpp` will be graded for style, NOT `test.cpp` .*

Working with a Partner

- For Projects 3 and 4, you may choose to work with one other student who is currently enrolled in EECS 183.
- Although you are welcome to work alone if you wish, we encourage you to consider partnering up for Project 3. If you would like a partner but don't know anyone in the class, we encourage you to use the Search for Teammates post on Piazza if you want to find someone! Please make sure to mark your search as *Done* once you've found a partner.
- As a further reminder, a partnership is defined as two people. You are encouraged to help each other and discuss the project in English (or in some other human language), but don't share project code with anyone but your partner.
- **To register a partnership on the autograder, go to the autograder link for the project and select "Send group invitation". Then, add your partner to the group by entering their email when prompted.** They will receive a confirmation after registration, and must accept the invitation before the partnership can submit. **You must choose whether or not to register for a group on the autograder before you can submit. If you select the option to work alone, you will not be able to work with a partner later in the project.** If a partnership needs to be changed after you register, you may submit an admin request.
- The partnership will be treated as one student for the purpose of the autograder, and you **will not** receive additional submits beyond the given 4 submits per day.

- If you decide to work with a partner, be sure to review the guidelines for working with a partner.
- If you choose to use late days and you are working in a partnership, review this document for how late days will be charged against each partner.

Multiple Files

- Most programs in the real world are written in more than just one file so as to break down the functionality into smaller parts and to keep the program's organization clean. As you start writing more complex and more involved (and more exciting!) programs, you too will work with multiple files.
- For Projects 1 and 2, you worked with just one file, such as `birthdays.cpp`. It had a `main()` function, where the execution began, and then some other functions that were called from `main()` or from other functions. But as a program gets more complicated, you can imagine that the file would get longer and longer and it would be difficult to keep it organized, let alone test the program.
- Therefore, a common practice is to put (at least some) functions into separate files. The functions' declarations (aka prototypes) go in what are known as **header files** that end in `.h`, and the functions' definitions (aka implementations) go in `.cpp` files. Each `.cpp` file will `#include` the `.h` files that contain the declarations for the functions it implements or calls.

  Then the program will contain one other `.cpp` file (without a header file) that will contain a `main()` function that drives the program. This `.cpp` file will `#include` any header files that declare any functions that it might need to run.
- In the Project 3 distribution code, you'll find `color.h`, `color.cpp`, `driver.h`, `driver.cpp`, `utility.h`, `utility.cpp`, `main.cpp`, and `start.cpp`. These files contain prototypes and implementations of the utility and driver functions for the project, such as reading input, printing the board (in color), and so on. You should not change any of these files. We recommend that you read the RMEs and code in `utility.h` and `utility.cpp`, but you do not need to understand the code in other files. Much of programming today is done by standing on other people's shoulders, which means relying on code that's already been written. Just like you didn't have worry about how the `floor()`, `ceil()`, `sqrt()`, and `getline()` functions are implemented in order to use them, you can rely on the code we've provided without having to fully understand how it works.

  The file `ohhi.h` contains prototypes of the functions that you will implement in `ohhi.cpp`. Finally, `test.cpp` contains functions that you will use to test your code and ensure correctness of your program. You are responsible for writing your own tests, and you will submit both `ohhi.cpp` and `test.cpp`.

  After you've finished writing your tests and/or completed your code, you'll be able to use `start.cpp`, which has a `main()` function, to run `main.cpp` to play and solve 0h h1 puzzles as well as run the tests you wrote in `test.cpp`.

Suggested Timeline

You will be approximately on track for this project if you follow this timeline:

- Thursday, October 6: Start search for partner or decide to work alone
- Friday, October 7: Starter code downloaded and set up in IDE, stubs written and submitted to autograder
- Tuesday, October 11: `count_unknown_squares()`, `row_has_no_threes_of_color()`, `col_has_no_threes_of_color()`, `board_has_no_threes()`, completed and tested

- Thursday, October 13: `solve_balance_row()` , `solve_balance_column()` , `rows_are_different()` , `cols_are_different()` , `board_has_no_duplicates()` completed and tested
- Friday, October 14: `solve_three_in_a_row()` , `solve_three_in_a_column()` , `board_is_solved()` , `check_valid_input()` , and `check_valid_move()` completed and tested
- Wednesday, October 19: Final submission made to autograder for 5% extra credit

Collaboration Policy

We want students to learn from and with each other, and we encourage you to collaborate. We also want to encourage you to reach out and get help when you need it. You are encouraged to:

- Give or receive help in understanding course concepts covered in lecture or lab.
- Practice and study with other students to prepare for assessments or exams.
- Consult with other students to better understand project specifications.
- Discuss general design principles or ideas as they relate to projects.
- Help others understand compiler errors or how to debug parts of their code.

To clarify the last item, you are permitted to look at another student's code to help them understand what is going on with their code. You are not allowed to tell them what to write for their code, and you are not allowed to copy their work to use in your own solution. If you are at all unsure whether your collaboration is allowed, please contact the course staff via the admin form before you do anything. We will help you determine if what you're thinking of doing is in the spirit of collaboration for EECS 183.

The following are considered Honor Code violations:

- Submitting others' work as your own.
- Copying or deriving portions of your code from others' solutions.
- Collaborating to write your code so that your solutions are identifiably similar.
- Sharing your code with others to use as a resource when writing their code.
- Receiving help from others to write your code.
- Sharing test cases with others if they are turned in as part of your solution.
- Sharing your code in any way, including making it publicly available in any form (e.g. a public GitHub repository or personal website).

The full collaboration policy can be found in the syllabus.

Solution Overview

Problem Statement

Like Sudoku, 0h h1 is a popular puzzle game in which the goal is to find a valid coloring of the game board without violating a particular set of constraints. Instead of filling in the numbers 1 through 9, in 0h h1 we simply use squares of either **RED** or **BLUE**, as shown above.

In 0h h1, there are three rules that define a valid board:

1. **Equal representation.** Each row and column must have the same number of red and blue squares.
2. **Runs of three or more are not allowed.** There may not be more than two consecutive squares of the same color either horizontally or vertically.
3. **No duplicates.** No two rows can be the same, nor two columns.

Note that rules 1 and 3 only apply to a completed board. For instance, it is possible for a partially completed, valid board to have two duplicate rows with unknown squares, since they could be filled in such a way that the final board has no duplicates. Similarly, a partially filled row does not have to have the same number of **RED** squares as **BLUE** squares.

The first sections of your program will check partially-completed 0h h1 boards to ensure that they adhere to these rules. You will then develop algorithms to solve 0h h1 puzzles. Finally, you will implement functions that allow a user to play 0h h1.

For this project, you will write a checker, a solver, and gameplay functions for 0h h1.

You do not need to write a driver for 0h h1. Since we want you to concentrate on the more interesting parts of the program, we have implemented the driver for you in `start.cpp` and `main.cpp` ; see Running with start.cpp below for the details. We strongly recommend that you hold off on using the driver to run 0h h1 until you have completed and thoroughly tested each function you have to write. Instead, write tests in `test.cpp` to test each function and use the driver to run those tests first.

Counter

The first function you write, `count_unknown_squares()` , will traverse and count the number of `UNKNOWN` squares in an 0h h1 board, stored in a 2-dimensional array of proportions `MAX_SIZE x MAX_SIZE` . This function will serve as a stepping stone to the more involved array functionality of the next two sections.

> **NOTE:** In `utility.h` , we have defined several global constants (`UNKNOWN` , `RED` , and `BLUE`) which represent the color of a square–please use them when checking or manipulating the contents of a board.

> **NOTE:** The size of the board is passed in as the `size` argument in every function you will write. The board is assumed to be square, so `size` is the size in both dimensions. `size` may be less than or equal, but no more than, `MAX_SIZE` .

Checker

In this section of the project, you will design algorithms to verify 2 of the 3 rules of 0h h1. (The first rule has already been written for you as an example: `board_is_balanced()` in `driver.cpp` .)

- `board_has_no_threes()` : This function will ensure that no row or column contains 3 consecutive squares of the same color. It uses the following functions:
    - `row_has_no_threes_of_color()` : This function ensures that a specific row does not contain 3 consecutive squares of the same color.
    - `col_has_no_threes_of_color()` : This function ensures that a specific column does not contain 3 consecutive squares of the same color.
- `board_has_no_duplicates()` : This function will verify that no two completed rows or columns are identical.

  - `rows_are_different()` : This function verifies that the two given rows are different if they are both complete.
  - `cols_are_different()` : This function verifies that the two given column are different if they are both complete.

Solver

So that you are able to focus on the most interesting portions of this section, we have implemented some of the "structural" pieces of the solver for you. The main portion of our pre-written code is contained in the `solve()` function, found in `driver.cpp` . This function reads in an 0h h1 puzzle and attempts to solve it by calling several key functions, which we have generously left blank for you to implement at your leisure. These functions are as follows:

`solve_three_in_a_row()` and `solve_three_in_a_column()` will identify certain instances of squares which cannot be assigned one of the colors and therefore must be assigned the opposite; for instance, a square with **BLUE** squares on either side cannot be assigned **BLUE** without violating the rule of no consecutive threes in a row or column, and therefore must be designated as **RED**.

`solve_balance_row()` and `solve_balance_column()` will look at the given row or column to determine if exactly half the squares are **RED** or exactly half the squares are **BLUE**. Since by the rules of 0h h1 a row or column must have equal parts **RED** and **BLUE** squares, these functions will complete such a row or column using the underrepresented color.

All solver functions take in an `announce` argument, which controls whether or not `mark_square_as()` prints a message when a square is changed. (The solver is used both in solving a board and in generating one, and the `announce` parameter allows the former to announce each move without requiring the latter to print out lots of extraneous output giving away the solution.) Make sure to pass on `announce` when you call `mark_square_as()` .

> **IMPORTANT:** When marking a square a particular color, you must use the `mark_square_as()` function, which will also print out the steps as your program goes along if the `announce` argument is true. In order for your project to pass the Autograder's tests, your functions must check from top to bottom within a column, and left to right within a row.

Gameplay

The final three functions you will write enable a user to play 0h h1 interactively. We have already written the gameplay driver for you in `play_board()` , found in `main.cpp` , and `make_move()` , found in `driver.cpp` , which read in user input for you. The three functions you are responsible for check the validity of user input, whether or not a move is legal, and whether the board is solved:

- `board_is_solved()` returns `true` if the board is solved and `false` otherwise. In order for the board to be solved, all squares must be assigned a color, and all validity checks must pass.
- `check_valid_input()` determines whether or not a user's input (i.e. attempted move) corresponds to a valid row, column, and color.
  - The row input that is passed to this function is the row number ranging from 1 to the size of the board inclusive, as in the printed format of a board.

- Similarly, the column input is a letter ranging from `A` to the letter representing the size of the board. However, your code should accept both upper and lower case letters; you may find the `toupper()` library function useful.

- Finally, the color must be one of the global constants `UNKNOWN_LETTER`, `RED_LETTER`, or `BLUE_LETTER`, but as with the column, your code should accept lower case letters as well.

- If the input is a valid row, column, and color, then the function should modify the pass-by-reference parameters `row` and `col` to be the corresponding row and column indices (remember that arrays are 0-indexed, meaning array indices start at 0, not 1). The function should then return `true`.

- If the input is not valid, the `row` and `col` parameters should not be modified. Instead, the function should return `false` after printing out the following: `Sorry, that's not a valid input.`

- `check_valid_move()` takes in the translated row and column indices (i.e. counting from 0) and the color and checks whether or not the given move is valid.
  - If the user attempts to change a square that is set in the original board, the function should print the following and return `false`: `Sorry, original squares cannot be changed.`
  - If the move results in an invalid board, then the function should return `false` after printing the following: `Sorry, that move violates a rule.`
  - Otherwise, the function should return `true`.

> **NOTE:** You may find it useful to make a copy of the board in order to check whether or not a move is valid.

> **NOTE:** `check_valid_move()` relies on your validity checks to be properly implemented, so we recommend writing it last.

Hints

Play the game here a couple times to make sure you understand the rules. If you get stuck, you can click on the eye at the bottom of the screen to get hints. We suggest playing several games on the 6×6 board, so you see a variety of positions and how the rules apply.

Most functions do not require nested loops, and none require more than two levels of nesting, which means you should think more carefully about what the function is doing if you find yourself writing a lot of nested loops. Examples of correct function outputs are in the RMEs, but here are some additional hints along with our suggested order for implementing the functions:

1. `count_unknown_squares()`
2. `row_has_no_threes_of_color()`, `col_has_no_threes_of_color()`
3. `board_has_no_threes()`
4. `solve_balance_row()`, `solve_balance_column()`
5. `rows_are_different()`, `cols_are_different()` - Two rows/columns can only be the same if neither of them have `UNKNOWN` squares.
6. `board_has_no_duplicates()`
7. `solve_three_in_a_row()`, `solve_three_in_a_column()` - Be sure to not only check the case where there are two consecutive squares, but also the case where two squares are separated by a blank one! Also watch out for

having two consecutive squares on the edge of the board. In order to satisfy the autograder, these two functions must only iterate over the given row or column once, from left to right or top to bottom. When considering a square, you must check all cases that may apply in these functions.

8. `board_is_solved()`
9. `check_valid_input()`
10. `check_valid_move()`

To help you understand this project, and to give you an idea of the code you'll be writing, `driver.cpp` implements the functions `row_is_balanced`, `col_is_balanced`, and `board_is_balanced`. Feel free to look at them for inspiration on how to write the remaining functions.

Additionally, these utility functions are written for you to use:

- `opposite_color()`
- `print_board()`
- `clear_board()`
- `copy_board()`
- `mark_square_as()`
- `board_is_valid()`
- `read_board_from_string()`

They are declared in `utility.h` and defined in `utility.cpp`.

Function Table

You will need to implement and test the following functions in `ohhi.cpp`. You do not need to implement or change any functions in files other than `ohhi.cpp` and `test.cpp`.

Function	Other functions it should call
`count_unknown_squares`	Does not utilize any other functions
`row_has_no_threes_of_color`	Does not utilize any other functions
`col_has_no_threes_of_color`	Does not utilize any other functions
`board_has_no_threes`	`row_has_no_threes_of_color`, `col_has_no_threes_of_color`
`rows_are_different`	Does not utilize any other functions
`cols_are_different`	Does not utilize any other functions
`board_has_no_duplicates`	`rows_are_different`, `cols_are_different`
`solve_three_in_a_row`	`mark_square_as`, `opposite_color`
`solve_three_in_a_column`	`mark_square_as`, `opposite_color`
`solve_balance_row`	`mark_square_as`, `opposite_color`
`solve_balance_column`	`mark_square_as`, `opposite_color`

`board_is_solved`	`count_unknown_squares`, `board_is_valid`
`check_valid_input`	`toupper`
`check_valid_move`	`copy_board`, `board_is_valid`

A note about `const` function parameters

You will notice that some functions have the `const` keyword in front of their parameters (for example, in `utility.h`, the parameter for the function `bool board_is_valid(const int board[MAX_SIZE][MAX_SIZE], int size)`. A `const` parameter means that the function promises not to modify the variable you pass as an argument. Thus, if you are getting an error about `const`, it is likely because you are trying to modify the value of a variable you do not need to change. The full details about `const` will be covered in EECS 280.

Testing

- As you write code, it's important to test it! Catching and fixing bugs early is much easier than later on; this will save you hours when you work. So you'll be required to create and submit a test suite for this project.

> **NOTE:** A very good practice is to write tests before even implementing functions.

- The distribution includes the file `test.cpp`. This is the test suite for `ohhi.cpp`, and you will be able to run your tests through `start.cpp`.

> **NOTE:** We will run your test suite against buggy code to determine whether or not your tests are effective at identifying bugs.

- For each function in `ohhi.cpp`, write a function in `test.cpp` to test it. Here is the starter example for `count_unknown_squares()` included in the distribution:

```
void test_count_unknown_squares() {
    int board[MAX_SIZE][MAX_SIZE];

    // test case 1
    string test_board_1[] = {"O-OX",
                             "OO--",
                             "X---",
                             "-O--"};
    int size_1 = 4;
    read_board_from_string(board, test_board_1, size_1);
    cout << count_unknown_squares(board, size_1) << endl;

    // add more tests here
}
```

As the example illustrates, you can create a testing board by using the `read_board_from_string()` utility function.

> **IMPORTANT:** Do not read from `cin` in your test code. The autograder will not provide any input to standard input, so you will fail the autograder if you attempt to read from `cin`.

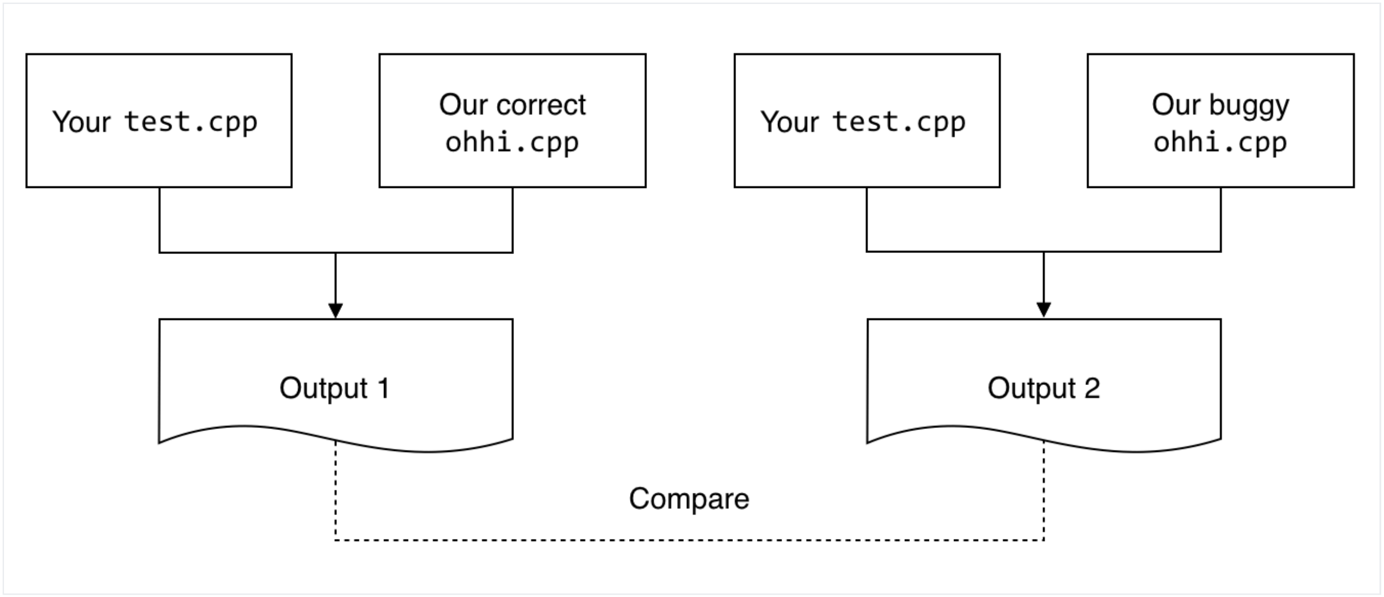
> **IMPORTANT:** Only write test cases in `test.cpp` for functions declared in `ohhi.h` . If you have helper functions you want to test, write test cases for those in a separate file (e.g. `test2.cpp`) with its own `main()` . We will be linking your `test.cpp` with our own `ohhi.cpp` , so it should only rely on functions declared in the distribution code or included in the standard C++ library.

- Once you write a test function, add a call to it from `startTests()` in `test.cpp` .
- Notice how the return value of `count_unknown_squares()` is printed. This is because `count_unknown_squares()` computes and returns a value, and it has no side effects of its own that you can observe just by calling it. You have to print out the return value so you have something to look at to check if it is correct.

  On the other hand, a function like `solve_three_in_a_row()` does not return anything but has the side effect of modifying the input board. In that case, you should make a simple function call (i.e. in its own statement) to `solve_three_in_a_row()` and then use `print_board()` to print out the modified board so you can see what it did.
- Carefully note the Requires clauses for all functions. They tell you what you may assume about the values of a function's arguments. Furthermore, you should not be calling it from the test suite with a value that violates the Requires clause. For example, don't call `count_unknown_squares()` with a `size` argument of `-1` .

> **WARNING:** If you submit a test case that violates the Requires clause, we will stop grading that submission and you will receive a very low score.

- As you work on the functions in `ohhi.cpp` , it's a good idea to write some test cases first, then write the implementation, and then run the tests to check if the implementation is correct.
- When you submit `test.cpp` , we will compile and run it with our correct implementation of `ohhi.cpp` and with our buggy implementation of `ohhi.cpp` , so as to generate two different outputs. We'll then compare two outputs. If there is any difference, you've successfully exposed a bug! The autograder does not go into the details of what the difference is, it only sees if there exists a difference.

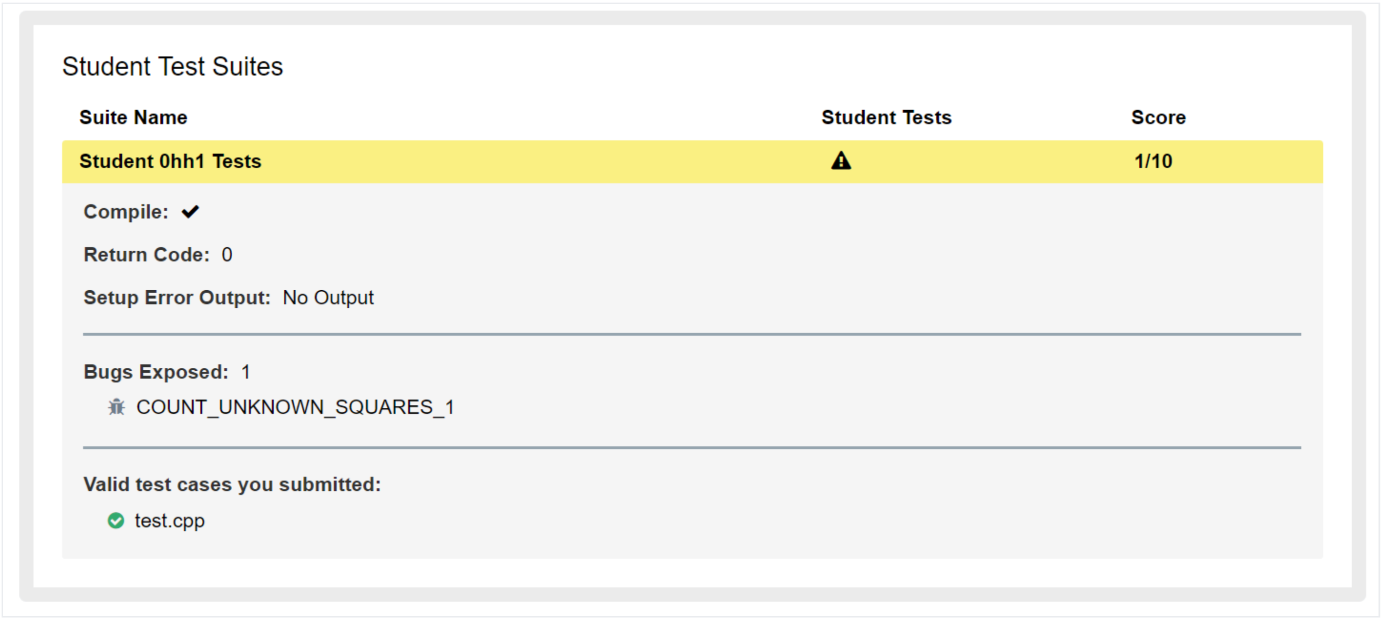


- Remember that some functions don't print anything on their own; we have to print their return value, as with the function `board_is_solved()` :

```
cout << board_is_solved(board_1, size_1) << endl;
cout << board_is_solved(board_2, size_2) << endl;
```

(Keep in mind that `bool` values are printed as `0` or `1` for `false` or `true`, respectively.)

- After you submit your test suite, you might see output that looks like this:



Student Test Suites

Suite Name	Student Tests	Score
Student 0hh1 Tests	⚠	1/10

Compile: ✓

Return Code: 0

Setup Error Output: No Output

Bugs Exposed: 1

COUNT_UNKNOWN_SQUARES_1

Valid test cases you submitted:

✓ test.cpp

That means that your test suite exposed 1 out of 11 bugs in the staff's "buggy" implementations of `ohhi.cpp` and your score for the test suite is 1 out of 10 points.

Bugs To Expose

There are a total of 11 unique bugs to find in our implementations. Your tests will need to expose 7 of the bugs to receive full points for test.cpp. The autograder will tell you the names of the bugs that you have exposed, from the following set:

- COUNT_UNKNOWN_SQUARES_1
- COUNT_UNKNOWN_SQUARES_2
- ROW_HAS_NO_THREES_OF_COLOR
- COLS_ARE_DIFFERENT
- BOARD_HAS_NO_DUPLICATES
- SOLVE_THREE_IN_A_ROW
- SOLVE_THREE_IN_A_COLUMN
- SOLVE_BALANCE_ROW
- CHECK_VALID_INPUT
- CHECK_VALID_MOVE_1
- CHECK_VALID_MOVE_2

Running with start.cpp

Once you have completed `test.cpp`, you can use the included `start.cpp` to run your tests. Once you have completed `ohhi.cpp`, you can use the included `start.cpp` to play or solve a game.

> **IMPORTANT:** Do NOT start by using `start.cpp` to play or solve a game by executing the `ohhi()` function in `main.cpp`. You will have to test your code to make sure it works, and you will need to eventually submit `test.cpp`. We strongly recommend that you write tests and test your code thoroughly by first using `start.cpp` to run your tests using the `startTests()` function in `test.cpp` before trying to run the game using the `ohhi()` function defined in `main.cpp`.

Here is an overview of the behavior of `start.cpp`, assuming a correct implementation of `ohhi.cpp`.

Menu

The initial menu allows you to choose between running `ohhi()` or running `startTests()`. Here is the initial menu:

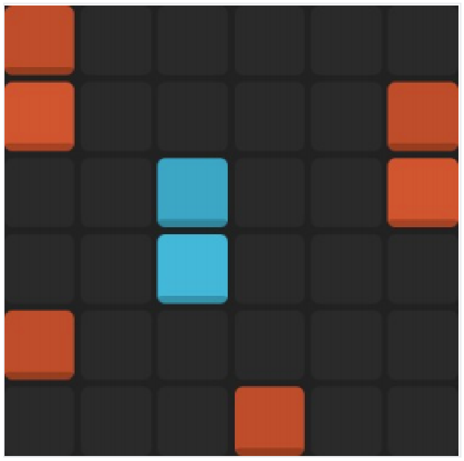
```
-------------------------------
EECS 183 Project 3 Menu Options
-------------------------------
1) Execute testing functions in test.cpp
2) Execute ohhi() function in main.cpp
Choice -->
```

ohhi Menu

Inputting `2` executes `ohhi()`, which displays the game menu. The game menu allows you to solve a custom game, play a random game, or play a custom game. Here is the game menu:

```
Menu Options
------------
1) Play a random game
2) Play a custom game
3) Solve a custom game
4) Play a random game with colors
5) Play a custom game with colors
```

Input



When solving or a playing a custom game, the program requires you to input the board. The distribution file contains code that reads in a board, with the following notation:

- `X` is a red square
- `O` is a blue square
- `-` is an unfilled square

Accordingly, the text notation below translates to the 0h h1 board above.

```
X-----
X----X
--O--X
--O---
X-----
---X--
```

Output

Given a board, the program will attempt to print out:

- The board that was given as input
- Whether the board is valid or invalid

When solving a game, it will also attempt to print:

- The steps to solving the puzzle
- The finished puzzle

When playing a game, the board will continue to print out the state of the board, ask for a move, check if the move is valid and legal, and update the board if it is. This continues until the board is solved.

Printing the board

In this example, the user runs the program, chooses to solve a custom game, and then enters the following via standard input (`cin`), pressing Enter after each line:

```
X-----
X----X
--O--X
--O---
X-----
---X--
```

The program will repeat the board it was given, properly formatted for readability.

```
Your board is:
   A B C D E F
  =============
1| X - - - - - |1
2| X - - - - X |2
3| - - O - - X |3
```

```
4| - - O - - - |4
5| X - - - - - |5
6| - - - X - - |6
  ============
   A B C D E F
```

Assessing validity

The program then calls each of your Checker functions in turn to decide whether this board is valid and can be solved. If the board is valid and your functions work correctly, the program will state the following:

```
This board is balanced.
This board does not contain threes-in-a-row/column.
This board does not contain duplicate rows/columns.
```

Solving the puzzle

It will then begin attempting to solve the 0h h1 puzzle by using your algorithms:

```
This board is valid; solving...
looking for threes-in-a-row/column...
marking (3, A) as O
marking (2, C) as X
marking (5, C) as X
marking (1, F) as O
marking (4, F) as O
marking (2, B) as O
marking (3, B) as X
marking (5, B) as O
looking for rows/columns with half X/O...
marking (2, D) as O
marking (2, E) as O
marking (4, A) as O
marking (6, A) as O
looking for threes-in-a-row/column...
....(continued)
```

Printing the result

If everything goes well, your functions will eventually generate a complete solution, which will be presented as output. The program will use your `count_unknown_squares()` to determine when the solution is complete.

```
Solved!
   A B C D E F
  ============
1| X X O O X O |1
2| X O X O O X |2
3| O X O X O X |3
4| O X O X X O |4
5| X O X O X O |5
```

```
6| O O X X O X |6
   ============
    A B C D E F
```

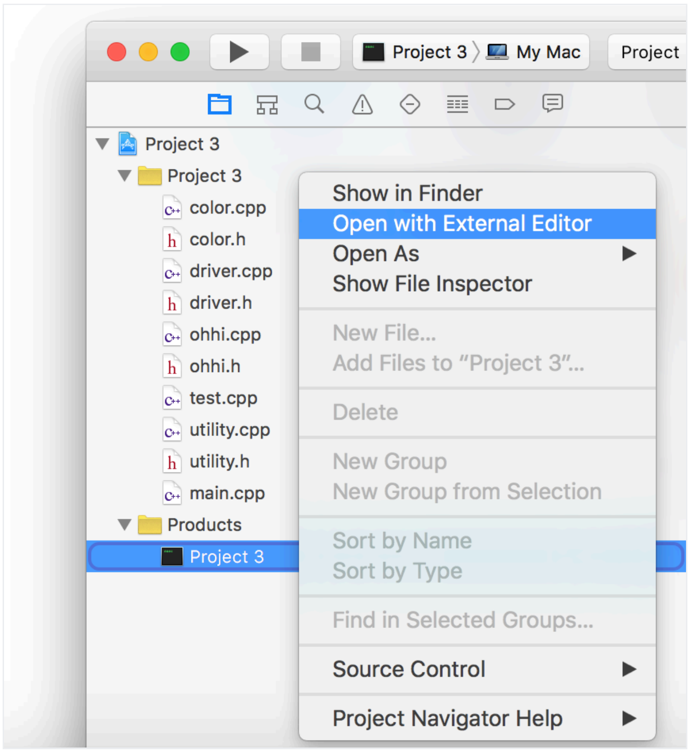
Playing the game

- Once you've implemented your counting, checking, solving, and gameplay functions, you will be able play a game of 0h h1. You can manually enter a board to play, or you can ask the program to generate a random board for you. The board generator, which we have provided for you, relies on your solver to ensure that the board is uniquely solvable using only the rules you've implemented.

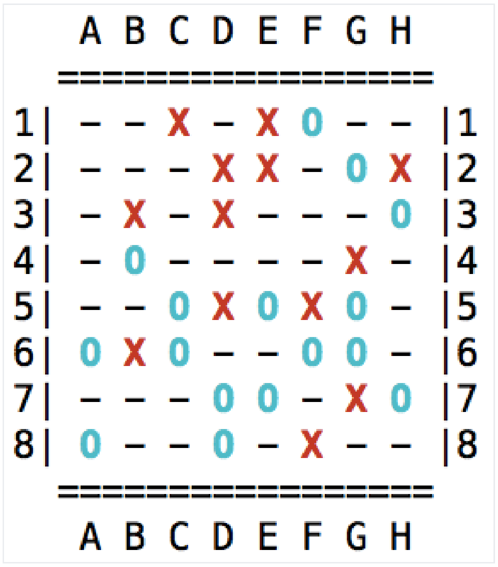
- You'll notice that the menu includes options to play in color:

```
Menu Options
------------
1) Play a random game
2) Play a custom game
3) Solve a custom game
4) Play a random game with colors
5) Play a custom game with colors
```

If you run the program through Visual Studio or a Terminal program, you can play in color without any issues. Xcode, on the other hand, does not support colors in its integrated console. However, you can open the program in a terminal by expanding **Products**, right-clicking on the name of the program and selecting *Open with External Editor*.

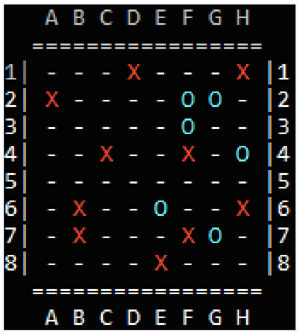


- On macOS, the board will look like the following when you play with colors:



```
   A B C D E F G H
  =================
1| - - X - X O - - |1
2| - - - X X - O X |2
3| - X - X - - - O |3
4| - O - - - - X - |4
5| - - O X O X O - |5
6| O X O - - O O - |6
7| - - - O O - X O |7
8| O - - O - X - - |8
  =================
   A B C D E F G H
```

- On Windows, it will look like this:



```
   A B C D E F G H
  =================
1| - - - X - - - X |1
2| X - - - - O O - |2
3| - - - - - O - - |3
4| - - X - - X - O |4
5| - - - - - - - - |5
6| - X - - O - - X |6
7| - X - - - X O - |7
8| - - - - X - - - |8
  =================
   A B C D E F G H
```

Sample Runs

Here are some sample runs of the game with `start.cpp` , assuming a correct implementation of `ohhi.cpp` . User's input is shown in red. You will want to use a diffchecker tool to compare your program with the sample outputs.

Sample Run #1

```
-------------------------------
EECS 183 Project 3 Menu Options
-------------------------------
1) Execute testing functions in test.cpp
2) Execute ohhi() function in main.cpp
Choice --> 2
Menu Options
------------
1) Play a random game
2) Play a custom game
3) Solve a custom game
4) Play a random game with colors
5) Play a custom game with colors

Choice --> 3
```

```
What board do you want to solve?
X-----
X----X
--O--X
--O---
X-----
---X--

Your board is:
   A B C D E F
  =============
1| X - - - - - |1
2| X - - - - X |2
3| - - O - - X |3
4| - - O - - - |4
5| X - - - - - |5
6| - - - X - - |6
  =============
   A B C D E F

This board is balanced.
This board does not contain threes-in-a-row/column.
This board does not contain duplicate rows/columns.

This board is valid; solving...
looking for threes-in-a-row/column...
marking (3, A) as O
marking (2, C) as X
marking (5, C) as X
marking (1, F) as O
marking (4, F) as O
marking (2, B) as O
marking (3, B) as X
marking (5, B) as O
looking for rows/columns with half X/O...
marking (2, D) as O
marking (2, E) as O
marking (4, A) as O
marking (6, A) as O
looking for threes-in-a-row/column...
marking (4, B) as X
looking for rows/columns with half X/O...
marking (4, D) as X
marking (4, E) as X
looking for threes-in-a-row/column...
marking (5, D) as O
looking for rows/columns with half X/O...
looking for potential duplicate rows/columns...
marking (5, E) as X
marking (5, F) as O
marking (1, C) as O
marking (6, C) as X
```

```
looking for threes-in-a-row/column...
marking (6, B) as O
marking (6, E) as O
marking (3, E) as O
marking (6, F) as X
marking (3, D) as X
marking (1, E) as X
looking for rows/columns with half X/O...
marking (1, B) as X
marking (1, D) as O

Solved!
   A B C D E F
  =============
1| X X O O X O |1
2| X O X O O X |2
3| O X O X O X |3
4| O X O X X O |4
5| X O X O X O |5
6| O O X X O X |6
  =============
   A B C D E F
```

Sample Run #2

```
-------------------------------
EECS 183 Project 3 Menu Options
-------------------------------
1) Execute testing functions in test.cpp
2) Execute ohhi() function in main.cpp
Choice --> 2
Menu Options
------------
1) Play a random game
2) Play a custom game
3) Solve a custom game
4) Play a random game with colors
5) Play a custom game with colors

Choice --> 3
What board do you want to solve?
O-XO-X-O
X---OO--
O-O--X-O
XOX---OX
--O--X--
XO---O--
XXOOX--O
O--O-X--

Your board is:
   A B C D E F G H
  ================
1| O - X O - X - O |1
2| X - - - O O - - |2
3| O - O - - X - O |3
4| X O X - - - O X |4
5| - - O - - X - - |5
6| X O - - - O - - |6
7| X X O O X - - O |7
8| O - - O - X - - |8
  ================
   A B C D E F G H

This board is balanced.
This board does not contain threes-in-a-row/column.
This board does not contain duplicate rows/columns.

This board is valid; solving...
looking for threes-in-a-row/column...
marking (2, D) as X
marking (2, G) as X
marking (3, B) as X
marking (5, A) as O
marking (5, B) as X
marking (6, C) as X
marking (6, D) as X
marking (4, F) as O
marking (2, H) as X
marking (4, E) as X
marking (6, E) as O
marking (6, G) as X
marking (4, D) as O
looking for rows/columns with half X/O...
marking (2, B) as O
marking (2, C) as O
marking (6, H) as O
marking (8, C) as X
marking (3, D) as X
marking (5, D) as X
marking (7, F) as O
marking (5, H) as X
marking (8, H) as X
looking for threes-in-a-row/column...
marking (3, E) as O
marking (5, E) as O
marking (5, G) as O
marking (7, G) as X
marking (8, G) as O
marking (1, E) as X
marking (3, G) as X
marking (1, G) as O
looking for rows/columns with half X/O...
```

```
marking (1, B) as X
marking (8, B) as O
marking (8, E) as X

Solved!
   A B C D E F G H
  =================
1| O X X O X X O O |1
2| X O O X O O X X |2
3| O X O X O X X O |3
4| X O X O X O O X |4
5| O X O X O X O X |5
6| X O X X O O X O |6
7| X X O O X O X O |7
8| O O X O X X O X |8
  =================
   A B C D E F G H
```

Sample Run #3

```
-------------------------------
EECS 183 Project 3 Menu Options
-------------------------------
1) Execute testing functions in test.cpp
2) Execute ohhi() function in main.cpp
Choice --> 2
Menu Options
------------
1) Play a random game
2) Play a custom game
3) Solve a custom game
4) Play a random game with colors
5) Play a custom game with colors

Choice --> 2
What board do you want to solve?
X-
--

Your board is:
   A B
  =====
1| X - |1
2| - - |2
  =====
   A B

This board is balanced.
This board does not contain threes-in-a-row/column.
This board does not contain duplicate rows/columns.

This board is valid; solving...
   A B
  =====
1| X - |1
2| - - |2
  =====
   A B

Please enter your move (ROW COLUMN COLOR): 0 0 x
Read: 0 0 x
Sorry, that's not a valid input.

Please enter your move (ROW COLUMN COLOR): 1 A o
Read: 1 A o
Sorry, original squares cannot be changed.

Please enter your move (ROW COLUMN COLOR): 1 B o
Read: 1 B o

   A B
  =====
1| X O |1
2| - - |2
  =====
   A B

Please enter your move (ROW COLUMN COLOR): 2 a o
Read: 2 a o

   A B
  =====
1| X O |1
2| O - |2
  =====
   A B

Please enter your move (ROW COLUMN COLOR): 2 b x
Read: 2 b x

   A B
  =====
1| X O |1
2| O X |2
  =====
   A B

Congratulations, you've won!
```

Sample Run #4

```
-------------------------------
```

```
EECS 183 Project 3 Menu Options
-------------------------------
1) Execute testing functions in test.cpp
2) Execute ohhi() function in main.cpp
Choice --> 2
Menu Options
------------
1) Play a random game
2) Play a custom game
3) Solve a custom game
4) Play a random game with colors
5) Play a custom game with colors

Choice --> 2
What board do you want to solve?
-X--
OOXX
XXOO
-O-X

Your board is:
   A B C D
  =========
1| - X - - |1
2| O O X X |2
3| X X O O |3
4| - O - X |4
  =========
   A B C D

This board is balanced.
This board does not contain threes-in-a-row/column.
This board does not contain duplicate rows/columns.

This board is valid; solving...
   A B C D
  =========
1| - X - - |1
2| O O X X |2
3| X X O O |3
4| - O - X |4
  =========
   A B C D

Please enter your move (ROW COLUMN COLOR): 4 a o
Read: 4 a o

   A B C D
  =========
1| - X - - |1
2| O O X X |2
3| X X O O |3
```

```
4| O O - X |4
  =========
   A B C D

Please enter your move (ROW COLUMN COLOR): 4 c x
Read: 4 c x
Sorry, that move violates a rule.

Please enter your move (ROW COLUMN COLOR): 4 c o
Read: 4 c o
Sorry, that move violates a rule.

Please enter your move (ROW COLUMN COLOR): 4 a -
Read: 4 a -

   A B C D
  =========
1| - X - - |1
2| O O X X |2
3| X X O O |3
4| - O - X |4
  =========
   A B C D

Please enter your move (ROW COLUMN COLOR): 4 a x
Read: 4 a x

   A B C D
  =========
1| - X - - |1
2| O O X X |2
3| X X O O |3
4| X O - X |4
  =========
   A B C D

Please enter your move (ROW COLUMN COLOR): 4 c o
Read: 4 c o

   A B C D
  =========
1| - X - - |1
2| O O X X |2
3| X X O O |3
4| X O O X |4
  =========
   A B C D

Please enter your move (ROW COLUMN COLOR): 1 a o
Read: 1 a o

   A B C D
  =========
1| O X - - |1
2| O O X X |2
3| X X O O |3
4| X O O X |4
  =========
   A B C D

Please enter your move (ROW COLUMN COLOR): 1 c x
Read: 1 c x

   A B C D
  =========
1| O X X - |1
2| O O X X |2
3| X X O O |3
4| X O O X |4
  =========
   A B C D

Please enter your move (ROW COLUMN COLOR): 1 d o
Read: 1 d o

   A B C D
  =========
1| O X X O |1
2| O O X X |2
3| X X O O |3
4| X O O X |4
  =========
   A B C D

Congratulations, you've won!
```

Style

Your code must follow the EECS 183 style guide.

Style Rubric

Top Comment

Must have name, uniqname, program name, and project description at the top of the file.

If all or part of the top comment is missing, take 1 point off.

Readability violations

-1 for each of the following:

Indentations

- Not using a consistent number of spaces for each level of code indentation
  - This includes using tabs on some lines and spaces on others
- Not indenting lines at all
- Failing to indent the blocks of code inside curly braces

Spacing

- Not putting a space around operators (e.g., `5*7` instead of `5 * 7` or `count=0;` instead of `count = 0;`)
  - Includes stream insertion (<<) and extraction (>>) operators
- Not putting a space between if, while, or for and the condition to be evaluated
- Putting a space between a function name and the opening parenthesis

Bracing

- Using a mix of Egyptian-style and hanging braces
  - Egyptian-style: '{' at the end of a statement
  - Hanging: '{' on its own line
- Braces should always be used for conditionals, loops, and functions
  - Examples:

```
// good
if (x == 1) {
   return false;
}
if (x == 2)
{
   return true;
}

// bad
if (x == 1) return false;
if (x == 2)
   return true;
```

Variables

- Variable names not meaningful
- Inconsistent variable naming style (`camelCase` vs. `snake_case`)
  - Excluding const variables, which are always `SNAKE_CASE`
- Not declaring const variables as `const`

- Not using all uppercase `SNAKE_CASE` for `const` variable names
- Using variable types that do not make sense in context

Line limit

- Going over 80 characters on a line
    - Includes lines of comments and lines of code

Statements

- More than one statement on a single line
    - A statement ends in a semicolon
    - Do not count off for multiple statements as part of a for loop declaration

Comments

- Commenting on the end of a line of code

```
// A comment should be placed before a line of code
int count = 0; // not on the same line as the code
```

- Insufficient comments or excessive comments
    - Code should be thoroughly commented such that lines' functionality is apparent from comments alone or from quickly glancing at code
    - Example of appropriate comment:

    ```
    // convert cups of flour to bags of flour
    int bagFlour = ceil((CUPS_FLOUR * numBatches) / CUPS_IN_LB_FLOUR);
    ```

    - Example of excessive comments:

    ```
    // declare variable
    int bagFlour;
    ```

    - Unneeded comments left in the code:

    ```
    // your code goes here
    // TODO: implement
    // this function doesn't work
    // FIXED
    ```

    - Commented out code:

    ```
    // int numBatches = people / 12;
    int numBatches = ceil(people / NUM_IN_BATCH);
    ```

RMEs

- Missing RMEs for any of the defined functions, except for main. This includes functions from the distribution code and any functions created by the student
- Having RMEs outside of header files

Coding quality

-2 for each of the following:

Global variables

- Global variables not declared as const

Magic numbers

- Using 0, 1, and 2 instead of UNKNOWN, RED, and BLUE without mentioning what they stand for
- Using the ASCII int value instead of the character

```
// bad code:
if (str[i] == 22)
// good code:
if (str[i] == ' ')
```

Egregious code

- Having redundant statements for RED and BLUE instead of using `opposite_color()`
- Logic that is clearly too involved or incorrect
    - e.g. instead of basing numbers on conversions, writing:

    ```
    if (year >= 1700 && year < 1800) {
       century = 17;
    } else if (year >= 1800 && year < 1900) {
       century = 18;
    }
    ```

    and so on

Function misuse

- Not calling helper functions where appropriate
    - Not calling `row_has_no_threes_of_color()` and `col_has_no_threes_of_color()` inside `board_has_no_threes()`
    - Not calling `rows_are_different()` and `cols_are_different()` inside `board_has_no_duplicates`
    - Not calling `board_is_valid()` inside `check_valid_move()`
    - Not calling `count_unknown_squares()` and `board_is_valid()` in `board_is_solved()`
- Having tester functions

bools

- **Only deduct 1 point for this category**
- Writing `<bool> == true` , `<bool> != true` , `<bool> == false` , or `<bool> != false`
    - Same for comparing bools to `0` and `1`
- Returning `0` and `1` instead of `true` and `false` for a bool functions