Learning C++

Battleship

Command Line Game

Nathan King

Contents

Analysis

Battleship – Rules

Battleship is a short turn-based game consisting of two players. The game contains a board for each player with 10x10 squares. Both players will secretly place their ships, then the first player will select one of the squares to 'attack', followed by the second player. This continues until a player has sunken all their opponent's ships.

Here is a list of basic rules for a player versus computer setup:

- There will be 4 ships for each player.
- Each ship contains a different length, 2 squares, 3 squares, 4 squares and 5 squares.
- The player can place their ships horizontally or vertically.
- The player's ships must not overlap but can be adjacent.
- The player will take the first turn.
- The player/computer will only get one 'attack' per turn.
- If the player sinks all the computer's ships, the player has won.
- If the computer sinks all the player's ships, the computer has won.

Approach

Starting with the concept of the game Battleship I will identify the key requirements of the user and conduct research for concepts I am new at regarding programming. Then I produce a list of my main objectives that will determine the success of the program. I will also identify and design the main functions/algorithms I will need through flow charts and pseudo code. After this I will create my main solution, followed by extensive testing that will be based on my objectives. Finally, I will evaluate the program, exploring possible improvements and identifying what was a success throughout the project.

Player – Requirements

The player must be able to interact will the game and not run into any errors. Therefore, I am tasked with checking every input.
Interacting with the game must be simple, which suggests having few select options to not overwhelm the player. Additionally, I will implement a random placement option in the instance that they do not want to manually input their ships at the start of the game.

- The player can enter their name, and the game will address them with what they input.
- The game will ask the user to reinput coordinates if they are not in the 10x10 grid.
- The game will ask the user to reinput coordinates if they have already selected that square.
- The game will ask the user to reinput coordinates if the ships does not fit inside the grid successfully.
- The game will inform the player if they have sunken a ship or one of their ships has sunk.
- The game will inform the player if they have won or lost.
- After the game is over, the player will be able go again.

Final Objectives

The program will:

1. Allow the user to play with the computer or with another player.
   - 1.1. They will be able to select their choice at the start of the game.
   - 1.2. If the user selected to play with a computer then the user will begin first, and they will not be able to see the computer's turns.
   - 1.3. If the user selected to play with another player then each player will take turns, and they will have to 'look away' when it is not their turn to avoid seeing the other board.
2. Use player names throughout the program.
   - 2.1. Players will be asked their name at the start of the game.
   - 2.2. A player name will appear to indicate when it is their turn to perform an action.
3. Generate player and guess boards.
   - 3.1. A player board will show the player where their ships are with updated information on hit locations.
   - 3.2. A guess board will have no ship locations but will show any hits or misses.
   - 3.3. A board will be formed by a 10x10 grid.
   - 3.4. The rows and columns will go from 0 to 9.
4. Allow the user to place their ships in selected location and in a desired orientation.
5. Allow the computer to place their ships in a random location and a random orientation.
6. Have error checks on all inputs to prevent crashes.
   - 6.1. If the user is asked for a number
   - 6.2. If the user is asked for a letter
   - 6.3. When the user is asked for their name it will take any input they desire.
7. Have validity checks on number inputs.
   - 7.1. If a position is already taken (another ship is found here) then it will ask (or generate) a new position and orientation.
   - 7.2. If the position chosen does not fit on the board it will ask (or generate) a new position and orientation.
8. Allow the users or computer to guess where the other ships are on the other board.
   - 8.1. Computer guesses will always be different random locations on the board.
   - 8.2. If the user or computer hits a ship, it will be indicated with a symbol and the user cannot select that position. It will also check if the ship has been sunk.
   - 8.3. If the user misses a ship, it will be indicated with a symbol and the user cannot select that position.
   - 8.4. If a ship has been sunk, it will inform the user, indicating which ship was sunk. It will also check if all the ships have been sunk.
   - 8.5. If all the ships have been sunk, then the game will inform the user if they have won or lost.
9. Allow the users to play again after the game has been won or lost.
   - 9.1. If the player selects yes, then the boards will be reset, and they will start a new game.
   - 9.2. If the player selects no, then the program will end.

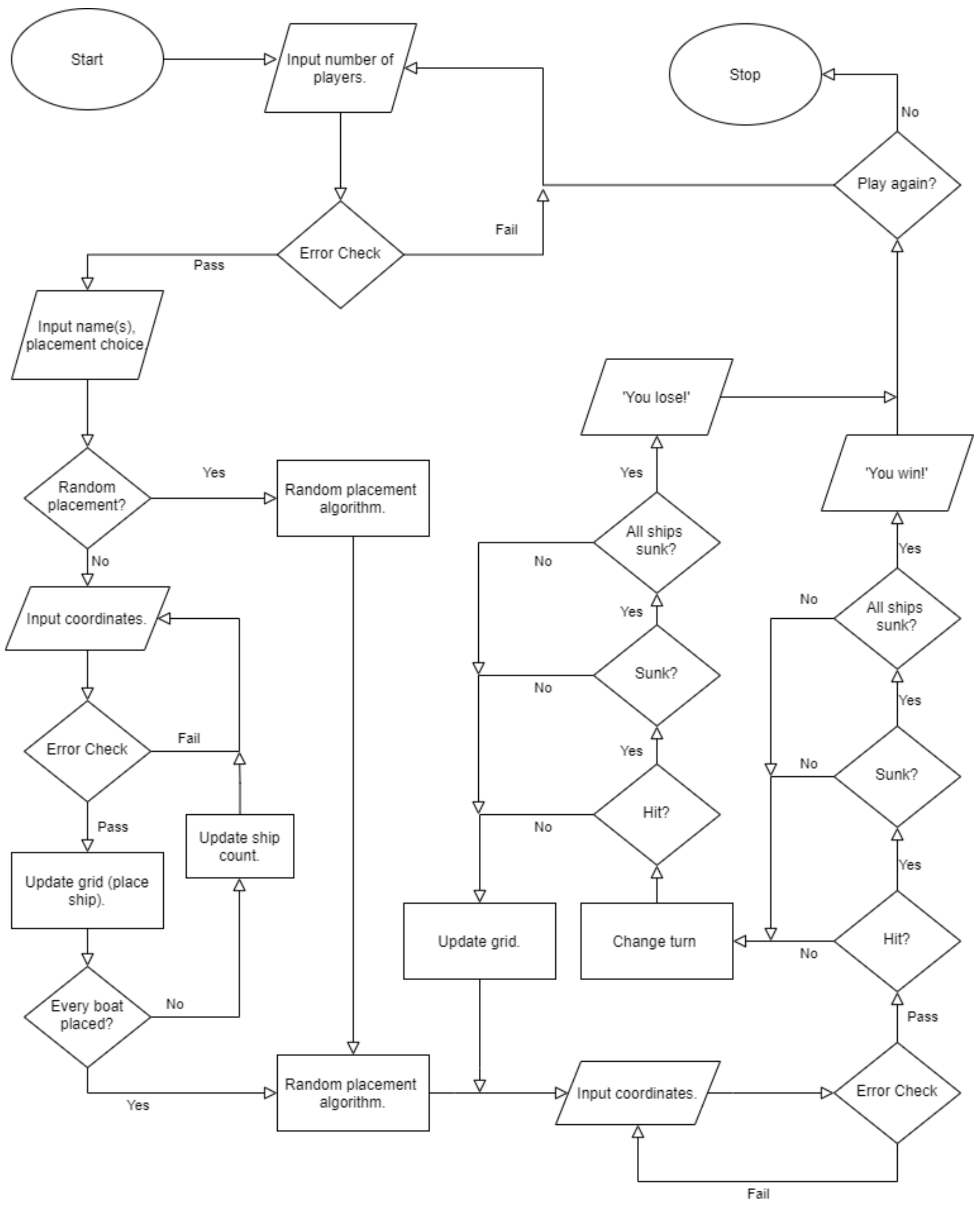
Design

General Overview

Designing the system will require planning and drafting of the key algorithms that will be essential to making my project a success. Having a detailed list of the possible functions will help me develop my final solution as I can produce each working function separately. I will also need to identify how the game boards will be displayed.

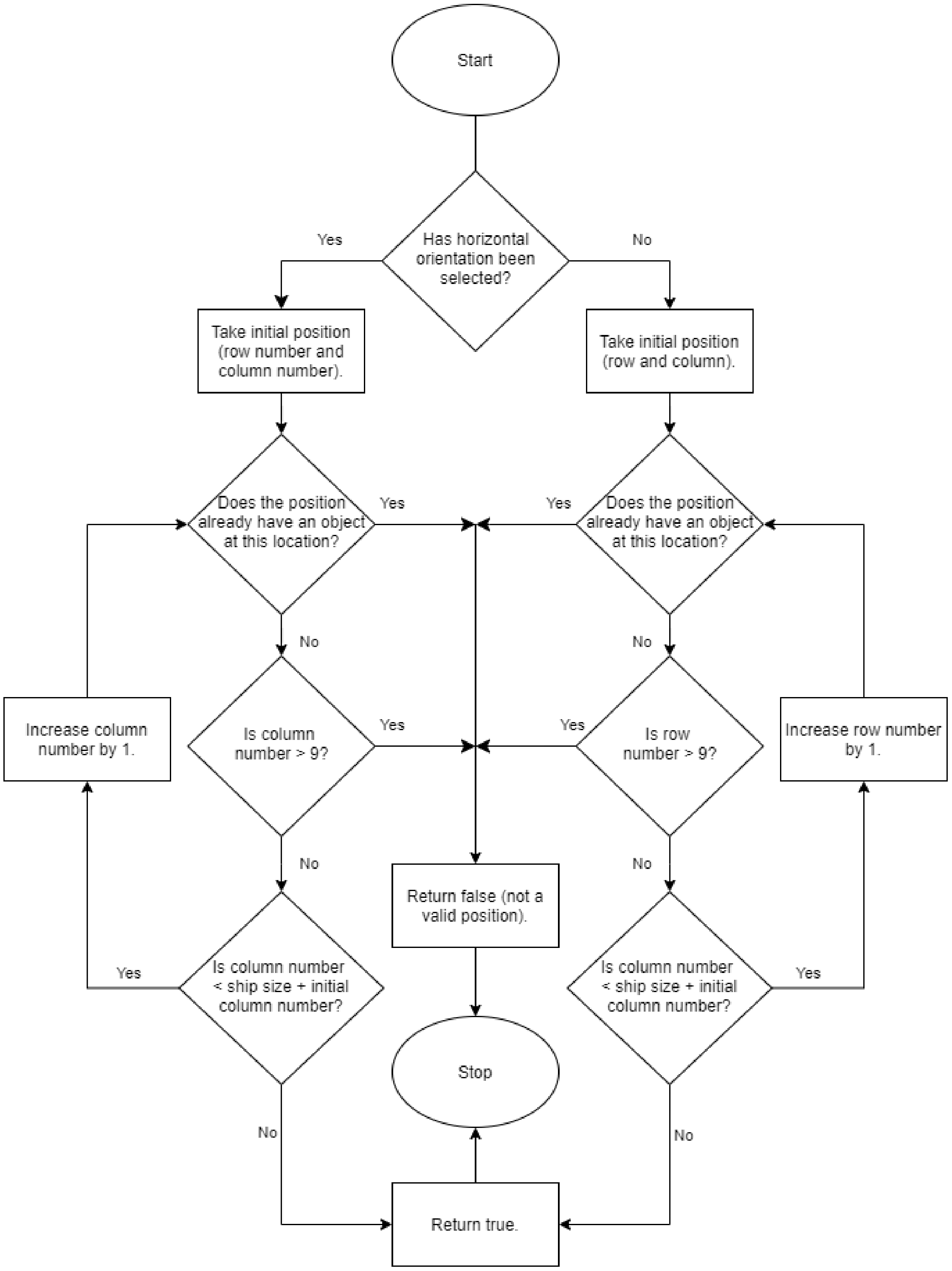
General Flowchart

The general flowchart details how the game will operate with limited details. From this, I will be able to tackle algorithms and expand on each component to formulate the general functions required.



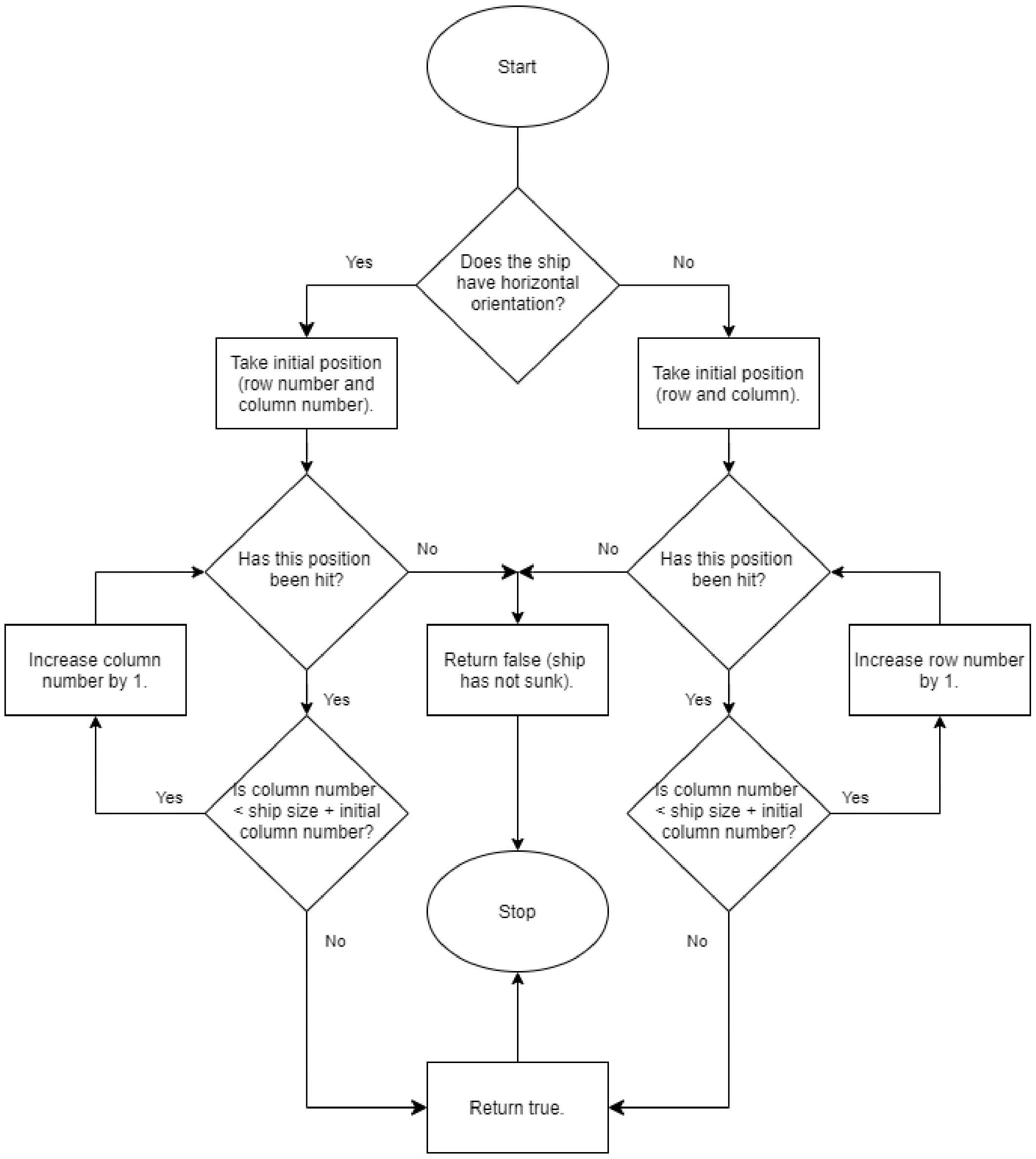
Valid Placement Algorithm

This flowchart represents how the algorithm that checks whether a boat placement is valid. The criteria for a valid placement are that it does not overlap with a previously placed ship and that it completely fits on the 10x10 board. The data that must already be held will be the input row, the input column, the input orientation, and the size of the ship being placed.


Start
Has horizontal orientation been selected?
Yes
No
Take initial position (row number and column number).
Take initial position (row and column).
Does the position already have an object at this location?
Yes
No
Does the position already have an object at this location?
Yes
No
Is column number > 9?
Yes
No
Is row number > 9?
Yes
No
Increase column number by 1.
Increase row number by 1.
Is column number < ship size + initial column number?
Yes
No
Is column number < ship size + initial column number?
Yes
No
Return false (not a valid position).
Stop
Return true.

‘Is Sunk’ Algorithm

Using the previous flowchart, I can easily represent how the ‘IsSunk’ algorithm will function as they work in very similar ways. The aim of this function is to checks whether a ship has been sunk. The criteria for a sunken ship are that all its positions have a true value for ‘hit’. The data that must already be held will be the initial row, the initial column, the orientation, and the size of the ship being placed.


Start
Does the ship have horizontal orientation?
Yes
No
Take initial position (row number and column number).
Take initial position (row and column).
Has this position been hit?
No
Has this position been hit?
No
Increase column number by 1.
Return false (ship has not sunk).
Increase row number by 1.
Yes
Yes
Is column number < ship size + initial column number?
Yes
Is column number < ship size + initial column number?
Yes
Stop
No
No
Return true.

Board Layout

The two boards will be displayed adjacent to each other, and one will remain ‘invisible’ to the other. This means that only the current player will be able to see their own board, swapping view when it is a different players’ turn. The board will contain ‘A’ to indicate the aircraft carrier, ‘B’ to indicate the battleship, ‘C’ to indicate the gunboat, and ‘D’ to indicate the minesweeper. Furthermore, a ‘*’ will be allocated to positions that have been hit, and ‘o’ to any positions the player has missed.

An example of what the user will see:

	0	1	2	3	4	5	6	7	8	9
0	A	A	A	A	A					
1										
2										
3					B					
4					B					
5					B					
6					B					
7										
8										
9										

	0	1	2	3	4	5	6	7	8	9
0										
1										
2										
3										
4										
5										
6										
7										
8										
9										

Inputs

The game will require inputs to function with a single or two players, so It is necessary to determine what variable type they will be stored in. The potential inputs are listed:

1. Number of players (second player will be computer or human).
2. Name (can by any input).
3. Row (must be 0-9 on the board).
4. Column (must be 0-9 on the board).
5. Orientation (horizontal or vertical placement)
6. Does the player want to play again (yes or no response)?

I have chosen that inputs 1,3,4,5 will all be stored using integer defined variables, thus will require an integer to be input by the user. This would mean error checking that the input was between 0-9 and that they have not put a character.

Input 2 will not be error checked as it will only be displayed and not needed for any processes.

Input 6 will be a character input (‘y’ or ‘n’)

Enumerations

This table details the enumerations that will be used in the solution.

Enum Name	Contained Enumerators	Purpose
PlayerType	Human, Computer	To differentiate between human and computer players.
GuessType	None, Hit, Miss	To identify if the players have guessed this position, and whether it has hit or missed a ship.
ShipType	None, Aircraft Carrier, Minesweeper, Gunboat, Battleship	To identify which boats are placed in positions on the board.
ShipOrientation	Horizontal, Vertical	To identify the orientation of the ship.

Data Structures

This table details the structures that will be used in the solution.

Struct Name	Contained Variables	Purpose
Player	Player type, player name, ships, board	To differentiate between human and computer players.
Ship	Ship type, orientation, position	To identify if the players have guessed this position, and whether it has hit or missed a ship.
ShipPosition	Row, Column	Contains the row number and column number for the ship position.
ShipPart	Ship type, hit, miss.	Will be used for positions on the board to contain the identity of the ship (including null) and if it has been hit or missed.

General Functions

Function Name	Function Type	Parameters Required	Purpose
GetPlayer2Type	PlayerType		To ask the user if they want to play with a computer or not, returning the result.
PlayGame	void		To start the main part of the game, containing each steps of play.
SetupBoards	void	Player	Sets up the players boards, asking them to place their ships.
ClearBoards	void	Player	Clears all the boards, changing each of the positions to null values.
SetupComputerBoards	void	Player	Sets up the computers board, placing each ship randomly.
GetRandomPosition	ShipPosition		The function that generates the random values for the row and column.
IsValidPlacement	bool	ShipPosition, ShipOrientation	Checks that the position of the ship selected stays on the board and does not overlap with another ship. Returns the result.
PlaceShip	void	ShipPosition, ShipOrientation	Places a ship on the board with the position and orientation.
DrawBoards	void	Player	Draws the 10x10 boards. Will use the ship locations stored with each player.
GetShipRepresentation	char	row, col	Will obtain the ship type that is at a position and show the symbol for it accordingly.
GetGuessRepresentation	char	row, col	Will obtain the object that is at a guessed position, including a hit or a miss.
GetShipName	char	ShipType	Returns the ship name.
GetBoardPosition	ShipPosition		Asks the user to enter a position (row and column). Contains error check.
GetShipOrientation	ShipOrientation		Asks the user to enter an orientation. Contains error check.

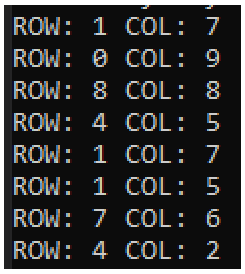
GetComputerGuess	ShipPosition		Acts as the computers turn, going to GetRandomPosition to guess.
SwitchPlayers	void		Represents a new turn by using a temporary variable that indicates whose turn it is.
IsGameOver	bool	Player	Performs a check to see if the game is over. Returns the result.
AreAllShipsSunk	bool	Player	Checks if every ship has been sunk by going through function 'IsSunk' multiple times. Returns the result.
IsSunk	bool	Player, Ship	Checks if a ship has been sunk by checking its starting row and column and seeing if it has been hit on every point adjacent. Returns the result.
DisplayWinner	void		Displays who won the game.
PlayAgain	bool		Asks the player if they want to play again, returning the result.

Solution

Documented Error

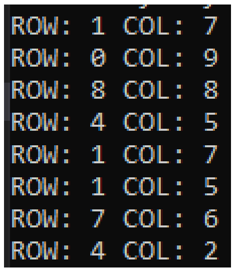
When coding the solution, I encountered an issue in which every time I would play against the computer, the ships would always be placed vertically and none appeared horizontally.
To test the potential error, I implemented a line of code that would output every random row and column number generated.

Test 1:



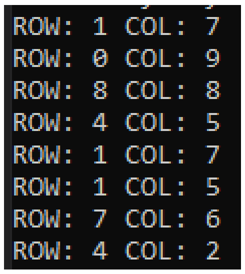
```
ROW: 1 COL: 7
ROW: 0 COL: 9
ROW: 8 COL: 8
ROW: 4 COL: 5
ROW: 1 COL: 7
ROW: 1 COL: 5
ROW: 7 COL: 6
ROW: 4 COL: 2
```

Test 2:



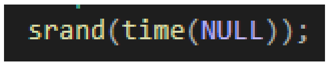
```
ROW: 1 COL: 7
ROW: 0 COL: 9
ROW: 8 COL: 8
ROW: 4 COL: 5
ROW: 1 COL: 7
ROW: 1 COL: 5
ROW: 7 COL: 6
ROW: 4 COL: 2
```

Test 3:



```
ROW: 1 COL: 7
ROW: 0 COL: 9
ROW: 8 COL: 8
ROW: 4 COL: 5
ROW: 1 COL: 7
ROW: 1 COL: 5
ROW: 7 COL: 6
ROW: 4 COL: 2
```

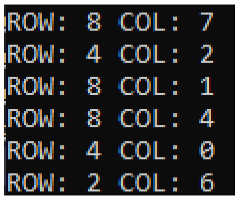
The images indicate that the output was not truly random. This meant that every time the computer board was generated, it would place the ships in identical positions and would make the user always have the same guess board. To solve this, I would require a new 'seed' that would guarantee new random values every game. I believed using time was a good solution because it is largely unlikely for the user to receive the same random values between games.



```
srand(time(NULL));
```

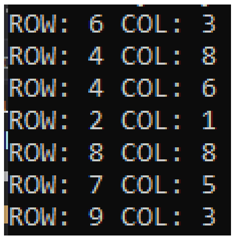
After implementing the change, I ran the tests again and the rows and columns generated where not identical. Therefore, I could assume that the ship placement was now random and could be fully tested in the final stages of the project

New test 1:



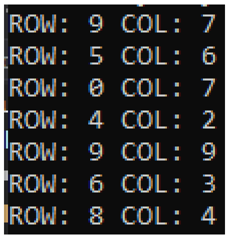
```
ROW: 8 COL: 7
ROW: 4 COL: 2
ROW: 8 COL: 1
ROW: 8 COL: 4
ROW: 4 COL: 0
ROW: 2 COL: 6
```

New test 2:



```
ROW: 6 COL: 3
ROW: 4 COL: 8
ROW: 4 COL: 6
ROW: 2 COL: 1
ROW: 8 COL: 8
ROW: 7 COL: 5
ROW: 9 COL: 3
```

New test 3:



```
ROW: 9 COL: 7
ROW: 5 COL: 6
ROW: 0 COL: 7
ROW: 4 COL: 2
ROW: 9 COL: 9
ROW: 6 COL: 3
ROW: 8 COL: 4
```

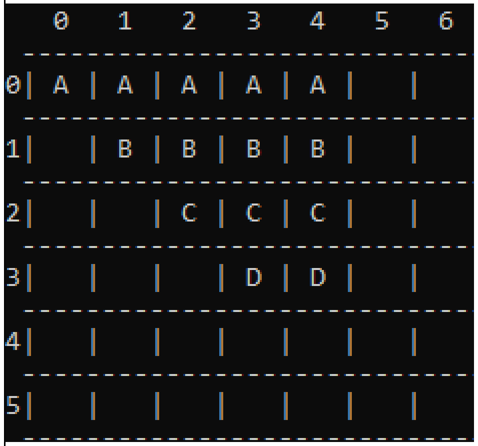
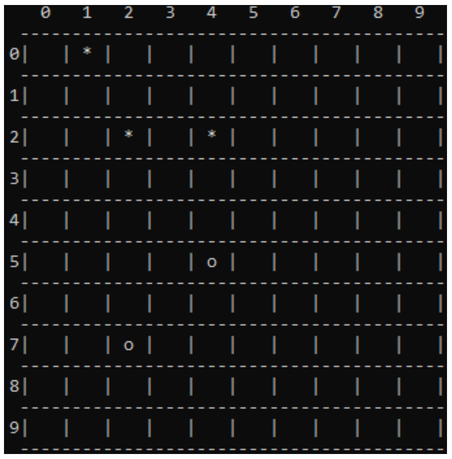
Final Functions

Function No.	Function Name	Function Type	Parameters
1	GetPlayer2Type	PlayerType	
2	GeneratePlayer	void	player, playerName
3	GenerateShip	void	ship, shipSize, shipType
4	PlayGame	void	player1, player2
5	SetupBoards	void	player
6	ClearBoards	void	player
7	SetupComputerBoards	void	player
8	GetRandomPosition	ShipPosition	
9	IsValidPlacement	bool	player, currentship, shipPosition, shipOrientation
10	PlaceShip	void	player, currentship, shipPosition, shipOrientation
11	DrawBoards	void	player
12	DrawColumnsRow	void	
13	SeperatorLine	void	
14	DrawShipBoardRow	void	player, row
15	GetShipRepresentation	char	player, row, col
16	DrawGuessBoardRow	void	player, row
17	GetGuessRepresentation	char	player, row, col
18	GetShipName	char	shipType
19	GetBoardPosition	ShipPosition	
20	MapBoardPosition	ShipPosition	rowInput, colInput
21	GetShipOrientation	ShipOrientation	
22	GetComputerGuess	ShipPosition	ComputerPlayer
23	UpdateBoards	ShipType	guess, currentplayer, otherplayer
24	SwitchPlayers	void	currentplayer, otherplayer
25	IsGameOver	bool	player1, player2
26	AreAllShipsSunk	bool	player
27	IsSunk	bool	player, ship
28	DisplayWinner	void	player1, player2
29	PlayAgain	bool	

Testing

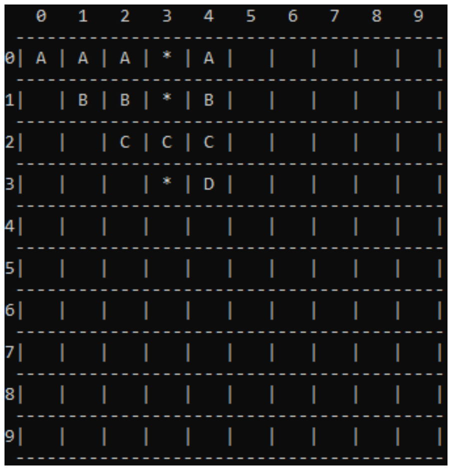
Test Plan

No.	Function(s)	Purpose	Expected Outcome	Result
1	GetPlayer2Type	To test that the user can select to play with a computer or another player.	Entering a ‘1’ enables play with player 2, leading to the program to ask both of their names. Entering a ‘2’ enables play with a computer.	Success
2	GetPlayer2Type	To test that the error check for GetPlayer2Type works correctly.	Entering an input that is not a ‘1’ or ‘2’ makes the program ask the user to input again.	Success
3	SetupBoards, GetBoardPosition	To test that the position input by the player is obtained successfully and continues the game.	Entering an input from 1 to 9 is followed by the program asking for the orientation. Must work in every case throughout the game.	Success
4	SetupBoards, GetBoardPosition	To test that the error check on GetBoardPosition works correctly.	Entering an input that is not 1 to 9 (such as a character or larger number) leads to	Success
5	SetupBoards, GetShipOrientation	To test that the orientation input by the player is obtained successfully and continues the game.	Entering a ‘1’ or ‘2’ makes the program check the validity of the position for vertical and horizontal placement, respectively. Must work in every case throughout the game.	Success
6	SetupBoards, GetShipOrientation	To test that the error check on GetShipOrientation works correctly.	Entering an input that is not a ‘1’ or ‘2’ makes the program ask the user to input again.	Success
7	SetupBoards, IsValidPositon	To test that the program tells the player when their input position/orientation is invalid when it does not fit on the board and prompts them to input the values again.	After inputting ‘9’ for column and ‘9’ for row, as well as horizontal orientation (‘1’) the program will ask the user to input again.	Success
8	SetupBoards, IsValidPositon, PlaceShip, DrawBoards	To test that when the player places ship onto the board that passes the validation check, they can see where it was placed immediately after.	Placing the aircraft carrier at row 0, column 0 in the horizontal position will immediately display the board with the character ‘A’ in coordinates (0,0), (0,1), (0,2), (0,3), (0,4). (Row, column)	Success

9	SetupBoards, IsValidPositon, PlaceShip, DrawBoards	To test that the user can place every ship when placed in valid positions. It also will show the final board.	Placing at the aircraft carrier at (0,0), battleship at (1,1) gunboat at (2,2), minesweeper at (3,3) all with horizontal orientation will place the ships as shown in the image: 	Success
10	UpdateBoards	To check that player 1 guesses work as intended, and the 'guess board' is presented correctly.	Hitting and missing ships results in the placement of a '*' and 'o' respectively (on the guess board). An example is shown: 	Success

```
   0   1   2   3   4   5   6
  -----------------------------
0| A | A | A | A | A |   |
  -----------------------------
1|   | B | B | B | B |   |
  -----------------------------
2|   |   | C | C | C |   |
  -----------------------------
3|   |   |   | D | D |   |
  -----------------------------
4|   |   |   |   |   |   |
  -----------------------------
5|   |   |   |   |   |   |
  -----------------------------
```

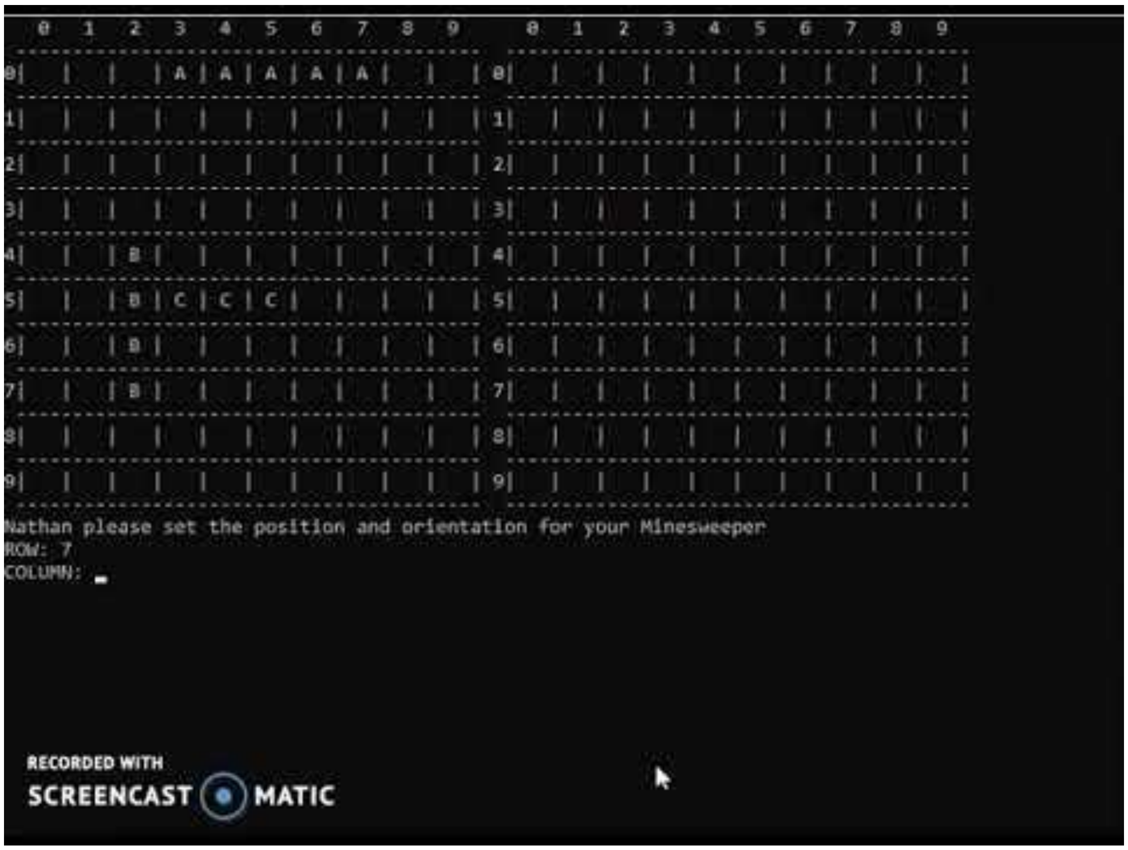
```
   0   1   2   3   4   5   6   7   8   9
  -----------------------------------------
0|   | * |   |   |   |   |   |   |   |   |
  -----------------------------------------
1|   |   |   |   |   |   |   |   |   |   |
  -----------------------------------------
2|   |   | * |   | * |   |   |   |   |   |
  -----------------------------------------
3|   |   |   |   |   |   |   |   |   |   |
  -----------------------------------------
4|   |   |   |   |   |   |   |   |   |   |
  -----------------------------------------
5|   |   |   |   | o |   |   |   |   |   |
  -----------------------------------------
6|   |   |   |   |   |   |   |   |   |   |
  -----------------------------------------
7|   |   | o |   |   |   |   |   |   |   |
  -----------------------------------------
8|   |   |   |   |   |   |   |   |   |   |
  -----------------------------------------
9|   |   |   |   |   |   |   |   |   |   |
  -----------------------------------------
```

11	UpdateBoards	To check that other player guesses work as intended.	Having your own ships hit will result in the placement of a '*' and there is no indication for missed ships. An example is shown: 	Success
12	IsSunk	To check that a ship being sunk makes the program inform the user.	Sinking a computers ship will output "ship has been sunk" with specification of which ship.	Success
13	IsGameOver, AreAllShipsSunk	To test that when all the computer's ships have been sunk that the game will end.	After sinking all the computer's ships. the game will announce the player as the winner and will then ask them if they want to play again. (Will use names)	Success
14	IsGameOver, AreAllShipsSunk	To test that when all the computer's ships have been sunk that the game will end.	After sinking all player 2's ships. the game will announce player 1 as the winner and will then ask them if they want to play again. (Will use names)	Success
15	IsGameOver, AreAllShipsSunk	To test that when all the computer's ships have been sunk that the game will end.	After all of player 1's ships have been sunk, the game will announce player 2 as the winner and will then ask them if they want to play again. (Will use names)	Success
16	PlayAgain	To test that the game successfully implements a play again function.	After the game is over, the players will be asked if they want to play again. Inputting a 'y' will start a new game with a cleared board.	Success
17	PlayAgain	To test that the game successfully implements a play again function.	After the game is over, the players will be asked if they want to play again. Inputting a 'n' will end the program.	Success
18	PlayAgain	To test that the game successfully implements a play again function.	After the game is over, the players will be asked if they want to play again. Inputting a value that is not a 'y' or 'n'	Success

```
   0   1   2   3   4   5   6   7   8   9
 ---------------------------------------
0| A | A | A | * | A |   |   |   |   |   |
 ---------------------------------------
1|   | B | B | * | B |   |   |   |   |   |
 ---------------------------------------
2|   |   | C | C | C |   |   |   |   |   |
 ---------------------------------------
3|   |   |   | * | D |   |   |   |   |   |
 ---------------------------------------
4|   |   |   |   |   |   |   |   |   |   |
 ---------------------------------------
5|   |   |   |   |   |   |   |   |   |   |
 ---------------------------------------
6|   |   |   |   |   |   |   |   |   |   |
 ---------------------------------------
7|   |   |   |   |   |   |   |   |   |   |
 ---------------------------------------
8|   |   |   |   |   |   |   |   |   |   |
 ---------------------------------------
9|   |   |   |   |   |   |   |   |   |   |
 ---------------------------------------
```

			will result in the game prompting the player to input again.	
19	main	Confidence trial of the main program to test that the game achieves its purpose.	Playing the whole game with ‘intent to win’ (i.e. beating the computer or second player) ends with the program displaying me as the winner and asking to play again.	Success
20	main	Confidence trial of the main program to test that the game achieves its purpose.	Playing the whole game with ‘intent to lose’ (i.e. losing to the computer or second player) ends with the program displaying the opposition as the winner and asking to play again.	Success

Running Example



```
   0   1   2   3   4   5   6   7   8   9       0   1   2   3   4   5   6   7   8   9
 -----------------------------------------    -----------------------------------------
0|   |   |   | A | A | A | A | A |   |   | 0|   |   |   |   |   |   |   |   |   |   |
 -----------------------------------------    -----------------------------------------
1|   |   |   |   |   |   |   |   |   |   | 1|   |   |   |   |   |   |   |   |   |   |
 -----------------------------------------    -----------------------------------------
2|   |   |   |   |   |   |   |   |   |   | 2|   |   |   |   |   |   |   |   |   |   |
 -----------------------------------------    -----------------------------------------
3|   |   |   |   |   |   |   |   |   |   | 3|   |   |   |   |   |   |   |   |   |   |
 -----------------------------------------    -----------------------------------------
4|   |   | B |   |   |   |   |   |   |   | 4|   |   |   |   |   |   |   |   |   |   |
 -----------------------------------------    -----------------------------------------
5|   |   | B | C | C | C |   |   |   |   | 5|   |   |   |   |   |   |   |   |   |   |
 -----------------------------------------    -----------------------------------------
6|   |   | B |   |   |   |   |   |   |   | 6|   |   |   |   |   |   |   |   |   |   |
 -----------------------------------------    -----------------------------------------
7|   |   | B |   |   |   |   |   |   |   | 7|   |   |   |   |   |   |   |   |   |   |
 -----------------------------------------    -----------------------------------------
8|   |   |   |   |   |   |   |   |   |   | 8|   |   |   |   |   |   |   |   |   |   |
 -----------------------------------------    -----------------------------------------
9|   |   |   |   |   |   |   |   |   |   | 9|   |   |   |   |   |   |   |   |   |   |
 -----------------------------------------    -----------------------------------------
Nathan please set the position and orientation for your Minesweeper
ROW: 7
COLUMN: _
```

Evaluation

Overview

When designing my system, I identified each step the game would require, and began assigning functions to each. After designing the more complex functions such as the Valid Placement Algorithm I was able to visualise the layout of the code and formulate a final solution.

I could have performed a more detailed analysis, especially when researching structures, enumerations, pointers, and references because during my solution phase there was an initial slow down as I was getting used to using them correctly. I believe my design phase was sufficient as the system was not overly complex and the algorithms use did not require more planning, therefore creating pseudo code or a prototype was not necessary. My use of flowcharts was very useful throughout this project as I was able to identify and group smaller components to achieve a larger goal. This was highly beneficial and proved to be useful when coding my final solution as I often was only required to translate what I had already created onto my project. Planning out the boards and how they would be positioned on the screen (next to each other) helped saved time when creating the code, particularly for the 'DrawBoards'', 'DrawColumnsRow' and SeperatorLine' functions. Overall, the steps taking to complete this project were sufficient, with an area of improvement being focused on the analysis stage.

Moreover, although the general flowchart created was useful in designing the main solution, the win condition checking did not function as planned. In the flowchart, a hit would create a sunk check, followed by a win check. However, when creating the solution, I considered it unnecessary as in order to efficiently loop the rounds of play in the 'PlayGame' function, the win condition would have to remain false. Therefore, the final code worked by having hits checked after each guess, then a sunk check. After the round has ended the will be a win check which contains a full sunk check of all the ships in the game. If I were to repeat this project, I would consider this type code layout (using the 'do' function). This is one area where use of pseudo code or prototyping may be beneficial.

Objective Assessment

1. Allow the user to play with the computer or with another player.
   1.1. They will be able to select their choice at the start of the game.
   1.2. If the user selected to play with a computer then the user will begin first, and they will not be able to see the computer's turns.
   1.3. If the user selected to play with another player then each player will take turns, and they will have to 'look away' when it is not their turn to avoid seeing the other board.

Objective 1 was fully implemented. Player selection contains error checking which improves the player experience and both the computer and player 2 roles work correctly. However, the implementation on having to 'look away' with both boards present is not technically an efficient solution to the problem and will be addressed the limitations section.

2. Use player names throughout the program.
   2.1. Players will be asked their name at the start of the game.
   2.2. A player name will appear to indicate when it is their turn to perform an action.

Objective 2 was fully implemented. Names were fully functional and were output across the game to address players.

3. Generate player and guess boards.
    3.1. A player board will show the player where their ships are with updated information on hit locations.
    3.2. A guess board will have no ship locations but will show any hits or misses.
    3.3. A board will be formed by a 10x10 grid.
    3.4. The rows and columns will go from 0 to 9.

Objective 3 was fully implemented. The player board not showing missed locations by the computer or second player was the correct approach because showing the ship type, hit and misses could potentially overwhelm the player with too much appearing on a single board. However, if further improvements were made to the computer AI, then it may be useful to show their misses. Currently, the computer can miss in the same location more than once, making it redundant to identify misses.

4. Allow the user to place their ships in selected location and in a desired orientation.

Objective 4 was fully implemented.

5. Allow the computer to place their ships in a random location and a random orientation.

Objective 5 was fully implemented. From final testing and trial runs there have been no issues found regarding random locations. It seems likely that the location and orientation of ships each game is random.

6. Have error checks on all inputs to prevent crashes.
    6.1. If the user is asked for a number
    6.2. If the user is asked for a letter
    6.3. When the user is asked for their name it will take any input they desire.

Objective 6 was fully implemented. All error checks work sufficiently and catch any undesired inputs. When entering the name of a player, the program saves their name regardless of what was input, allowing for number or letter names to be used and causes no issues to the game.

7. Have validity checks on number inputs.
    7.1. If a position is already taken (another ship is found here) then it will ask (or generate) a new position and orientation.
    7.2. If the position chosen does not fit on the board it will ask (or generate) a new position and orientation.

Objective 7 was fully implemented. As the tests indicate, all validity checks work correctly and enable the game to function without any issues like ships being placed off the board. This also works for when the computer randomly places a ship.

8. Allow the users or computer to guess where the other ships are on the other board.
    8.1. Computer guesses will always be different random locations on the board.

8.2. If the user or computer hits a ship, it will be indicated with a symbol and the user cannot select that position. It will also check if the ship has been sunk.
8.3. If the user misses a ship, it will be indicated with a symbol and the user cannot select that position.
8.4. If a ship has been sunk, it will inform the user, indicating which ship was sunk. It will also check if all the ships have been sunk.
8.5. If all the ships have been sunk, then the game will inform the user if they have won or lost.

Objective 8 was fully implemented. Having the computers guesses always be in different random places works as a standard game of Battleship, however there is room for more complexity that adds a layer of difficulty for the player. In the limitations sections there will be a solution to this potential improvement.

9. Allow the users to play again after the game has been won or lost.
   9.1. If the player selects yes, then the boards will be reset, and they will start a new game.
   9.2. If the player selects no, then the program will end.

Objective 9 was fully implemented.

Limitations and Extensions

1. Limited computer AI.

As indicated in the objective assessment, although the computer randomly guesses a different location each round, the overall AI is not as complex as it could be. In many cases, when a player initially hits a ship, they would then begin to guess locations around that area and begin creating a line of attacks. Implementing AI like this would improve the difficulty of the game, so it could be used as a 'hard mode'. The improvement would involve the player selecting between 'easy' or 'hard' at the start of the game after selecting to play with the computer. Choosing 'easy' would continue the original code but selecting 'hard' would use the more complex AI and have the computer target locations next to a 'hit' until that ship has been sunk. To do this I would require a counter once there are adjacent attacks, the computer will have to begin selecting locations the same row/column.

2. Changing player turns.

When there are 2 players, one must look away if it is not their turn to not see the others board. Since the game was designed for use on a single screen, there is no real solution to the 'look away' problem that occurs when playing with 2 people. However, a more complex design of this game could use the internet and function on multiple screens to allow for 2 players to play whilst not giving away their boards.

3. No surrender options.

The entire game could last a long time, so to improve user experience the program could benefit from having a surrender option. A simple implementation could be to ask the players each round if they wish to surrender, and when the input is yes then the game ends. However, this could negatively impact the user experience as having that option appear too often would interrupt the game. To prevent this, a counter could be used which counts the number of rounds up to 10, then resets. When the counter hits 10, the game will offer the surrender option to each user.

4. More testing.

When performing each test, I understood that there were many ways to play each game, and there could be hidden errors that have not been identified. Although the tests were overall sufficient in showing a working game, there could have been more to guarantee the success of all functions in the program.

5. No user feedback acquired.

As this was a smaller project and I knew the desired outcome, I did not seek feedback from other users. However, for a larger project it would be effective to get feedback from multiple different potential users of the system to collect experiences and identify weaknesses.

Conclusion

With every objective being implemented followed by a successful testing phase, the project was a success and the analysis and design steps have created an easy way to approach the solution. Although more could be done to completed and I have learnt more about documented design as well as structures, enumerations, pointers, and references.

Sources and Links

- Link used to learn about structures: http://www.cplusplus.com/doc/tutorial/structures/
- Link used to learn about Pointer and References: https://www.tutorialspoint.com/pointers-vs-references-in-cplusplus#:~:text=References%20are%20used%20to%20refer,referenced%20but%20pass%20by%20reference.
- Link used to learn about enumerations: https://docs.microsoft.com/en-us/cpp/cpp/enumerations-cpp?view=vs-2019
- Coded using C++ on Visual Studio
- Code uploaded to my GitHub account https://github.com/12nathanking