Contents

[1. Introduction to Checkers 2](#_Toc15912755)

[1.1. Explanation of the board 2](#_Toc15912756)

[1.2. How to play? 2](#_Toc15912757)

[1.3. Objective of the game 2](#_Toc15912758)

[2. Representing checkers in solution 3](#_Toc15912759)

[2.1. Initial Draft 3](#_Toc15912760)

[2.2. Finalize Draft 4](#_Toc15912761)

[2.2.1. Representing pieces in terms of bit 4](#_Toc15912762)

[2.2.2. Initializing pieces 4](#_Toc15912763)

[2.2.3. Final board output 5](#_Toc15912764)

[Advantage of using bitboard 5](#_Toc15912765)

[3. Movement in checkers: 6](#_Toc15912766)

[3.1. Using array approach 6](#_Toc15912767)

[3.2. Using bitboard approach 7](#_Toc15912768)

[3.2.1. For basic movement: 7](#_Toc15912769)

[3.2.1.1. Table for Basic movement using bit: 7](#_Toc15912770)

[3.2.2. For jumping movement: 8](#_Toc15912771)

[3.2.2.1. Table for jumping movement using bit: 8](#_Toc15912772)

[4. Generating of possible moves 9](#_Toc15912773)

[4.1. Creating the list of normal moves in CPU 9](#_Toc15912774)

[4.2. Creating the list of jump moves in CPU 9](#_Toc15912775)

[4.3. The mask used for verification 10](#_Toc15912776)

[4.4. Creating the list of moves in GPU 10](#_Toc15912777)

[5. Getting the best move set available 10](#_Toc15912778)

[5.1. Using CPU minimax 10](#_Toc15912779)

[5.2. Using GPU minimax 10](#_Toc15912780)

# Introduction to Checkers

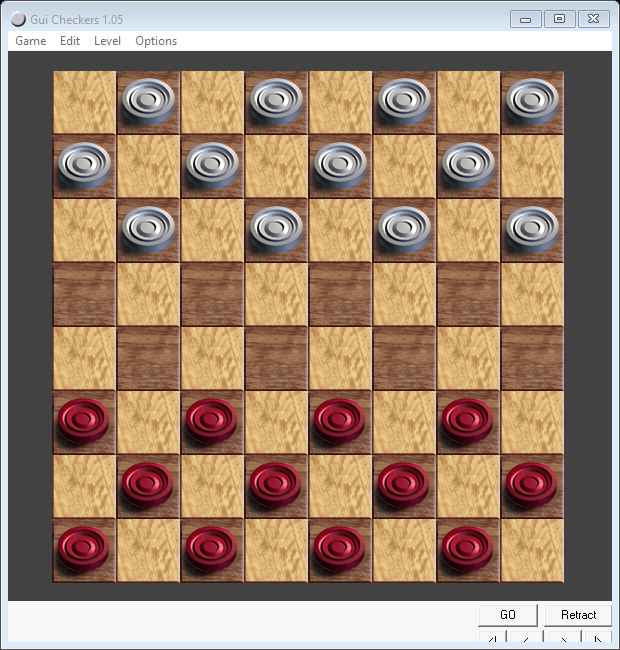


Figure 1. A normal 8 x 8 Checker board

## Explanation of the board

This is a standard 8x8 checker board with pieces on both side of the board, with black on the bottom and white on the top.

## How to play?

Each player is only allowed to move sideways on the color tiles and they must capture the opponent piece if they are close. Upon reaching the end of each of the opponent side, the piece will become a “King” in which that piece is allowed to move backwards.

## Objective of the game

The objective of this game is to capture all of your opponent pieces such that your opponent is unable to play the game. There can be a draw in the game if both players are unable to capture each other’s pieces within a certain amount of moves.

With the explanation of checkers, we will now proceed to represent the game into our project, starting with the data structure and the move generation.

# Representing checkers in solution

## Initial Draft

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
| 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 |
| 16 | 17 | 18 | 19 | 20 | 21 | 22 | 23 |
| 24 | 25 | 26 | 27 | 28 | 29 | 30 | 31 |
| 32 | 33 | 34 | 35 | 36 | 37 | 38 | 39 |
| 40 | 41 | 42 | 43 | 44 | 45 | 46 | 47 |
| 48 | 49 | 50 | 51 | 52 | 53 | 54 | 55 |
| 56 | 57 | 58 | 59 | 60 | 61 | 62 | 63 |

Figure 2a. Initial representation of the board

In this initial draft of the data structure, we planned to go with an array of 64 chars for both cuda and CPU but we soon realized that this might cause a lot of unnecessary memory allocation on the cuda side as there are some fields in the checker board that would not be accessed or use in the calculation of further moves, such that we decide that bit board representation for valid moves would be more preferred.

## Finalize Draft

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
|  | 28 |  | 29 |  | 30 |  | 31 |
| 24 |  | 25 |  | 26 |  | 27 |  |
|  | 20 |  | 21 |  | 22 |  | 23 |
| 16 |  | 17 |  | 18 |  | 19 |  |
|  | 12 |  | 13 |  | 14 |  | 15 |
| 8 |  | 9 |  | 10 |  | 11 |  |
|  | 4 |  | 5 |  | 6 |  | 7 |
| 0 |  | 1 |  | 2 |  | 3 |  |

Figure 2b. Bit representation of the board (where 0 to 11 = black pieces, 12 to 19 = free movement, 20 to 31 = white pieces)

As each piece in checkers will only touch the black titles, such that it would be better to store the relevant 32 moves as compared to using an array of 64 chars where most of the time only half of the array would be referenced to perform validations of move. Also, the data representation of each pieces would be different as the starting index would be beginning on the bottom right instead of the top right for array structure. Lastly as x64 and cuda both are little endian so our bit board looks like this

## Representing pieces in terms of bit

As mentioned previously, translating the board into a 32 bit representation is slightly different as compared to array representation, rather than starting from 0 to 63 in a standard array, in the bit board version, it begins from the MSB to the LSB of each rows eg the piece at 4 starts at 0x0000 0000 0000 0000 0000 0000 1000 0000 where each corresponding 0000 representing a row

## Initializing pieces

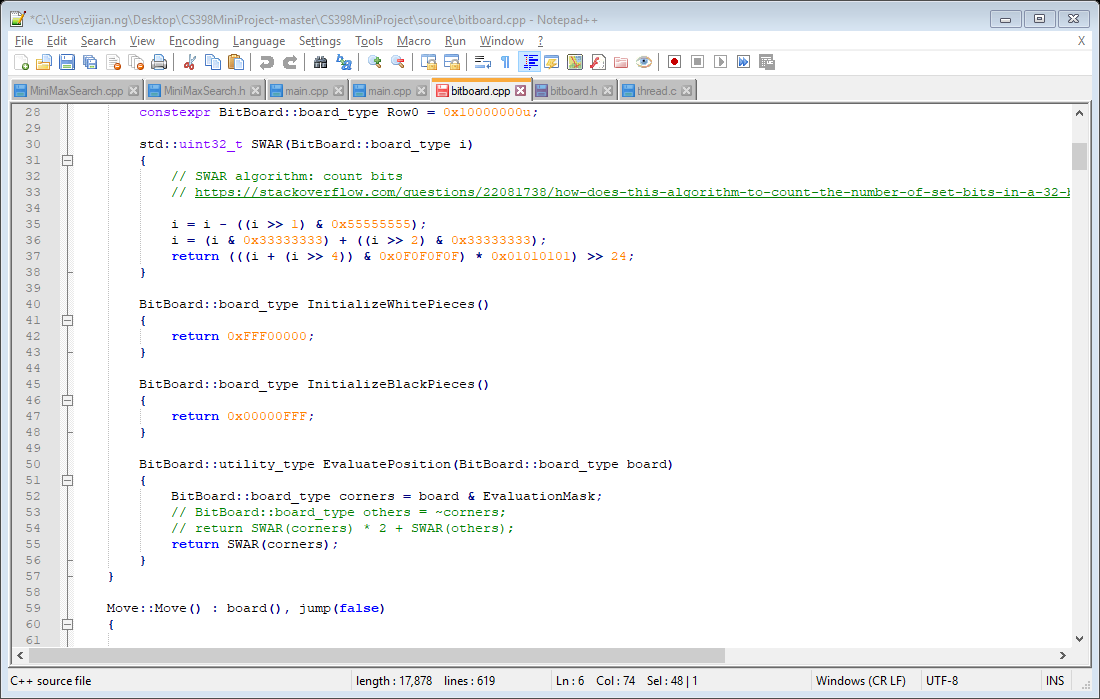


Figure 2c. Initialization of pieces.

In Figure 2c, we create two 32 bit binary to represent both black and white where white begins from the LSB to MSB while black begins from the MSB to LSB.

## Final board output

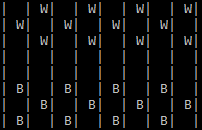


Figure 2d. A Checker board represented in console

This is the final representation of the checkers that will be seen when the program once starts up from the initialization of black and white pieces in figure 2c.

## Advantage of using bitboard

Performing bit wise operation would also be great in a sense that we do not need to perform countless of loops to get the list of possible moves. With the representation of the board done, let's move on to the processing of movement in checkers.

# Movement in checkers:

## Using array approach

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
| 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 |
| 16 | 17 | 18 | 19 | 20 | 21 | 22 | 23 |
| 24 | 25 | 26 | 27 | 28 | 29 | 30 | 31 |
| 32 | 33 | 34 | 35 | 36 | 37 | 38 | 39 |
| 40 | 41 | 42 | 43 | 44 | 45 | 46 | 47 |
| 48 | 49 | 50 | 51 | 52 | 53 | 54 | 55 |
| 56 | 57 | 58 | 59 | 60 | 61 | 62 | 63 |

Figure 3a. Array representation of the board

For movement in checkers, if we are using array of 64 chars then the movement will be much simpler as it will only be only dereferencing of the current pieces such as here for eg moving from a piece at 46 to 38 would be simply changing the piece in the array location of 46 to 38, this also applies for capturing of pieces.

## Using bitboard approach

### For basic movement:

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
|  | 28 |  | 29 |  | 30 |  | 31 |
| 24 |  | 25 |  | 26 |  | 27 |  |
|  | 20 |  | 21 |  | 22 |  | 23 |
| 16 |  | 17 |  | 18 |  | 19 |  |
|  | 12 |  | 13 |  | 14 |  | 15 |
| 8 |  | 9 |  | 10 |  | 11 |  |
|  | 4 |  | 5 |  | 6 |  | 7 |
| 0 |  | 1 |  | 2 |  | 3 |  |

Figure 3b. Bit representation of the board

In the case of using bit representation where each bit above represents a valid move, in order to move from 11 to 14, the program first has to move from 11 to 12 to 13 to 14 which is a total of 3 shift such that we can apply bit shift operations to move pieces to the designated direction of upper right, upper left, lower right and lower left. However, do take note that there are certain restrictions of movement on both black and white unless they have become king pieces, as well as there is a type of shift for even and odd rows which now comes into the picture.

### Table for Basic movement using bit:

|  |  |  |
| --- | --- | --- |
|  | Odd rows | Even rows |
| Moving Upper right | <<5 (left shift by 5) | <<4 (left shift by 4) |
| Moving Upper left | <<4 (left shift by 4) | <<3 (left shift by 3) |
| Moving Lower right | >>4 (right shift by 4) | >>5 (right shift by 5) |
| Moving Lower left | >>3 (right shift by 3) | >>4 (right shift by 4) |

Table 3a. Possible basic movement

The table above is the concluded possible normal move available.

### For jumping movement:

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
|  |  |  |  |  |  |  |  |
|  |  |  |  | W |  |  |  |
|  |  |  | W |  |  |  |  |
|  |  |  |  |  |  | W |  |
|  |  |  | B |  | B |  |  |
|  |  |  |  | B |  |  |  |
|  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |

Figure 3c. An example of a possible scenario

For capturing of pieces in bit representation, it follows the same implementation of the array approach of checking if there are opponent pieces in the possible direction, in terms of bit represent it will be checking with the existing white location, while at it, there is also a consideration to be added such as jumping to the next possible cell with even and odd rows as well.

### Table for jumping movement using bit:

|  |  |  |
| --- | --- | --- |
|  | Odd rows to even rows | Even to odd rows |
| Moving Upper right | <<5 <<4 (left shift by 5 then left shift by 4) | <<4<<5 (left shift by 4 then left shift by 5) |
| Moving Upper left | <<4 <<3 (left shift by 4 then left shift by 3) | <<3 <<4 (left shift by 3 then left shift by 4) |
| Moving Lower right | >>4 >>5 (right shift by 4 then left shift by 5) | >>5 >>4 (right shift by 5 then right shift by 4) |
| Moving Lower left | >>3 >>4 (right shift by 3 then left shift by 4) | >>4 >>3 (right shift by 4 then right shift by 3) |

Table 3b. Possible jump movement

The table above is the concluded possible jump move available.

# Generating of possible moves

With the list of possible moves derived in Section 3, it is time to start generating list of moves to be considered in Section 5.

## Creating the list of normal moves in CPU

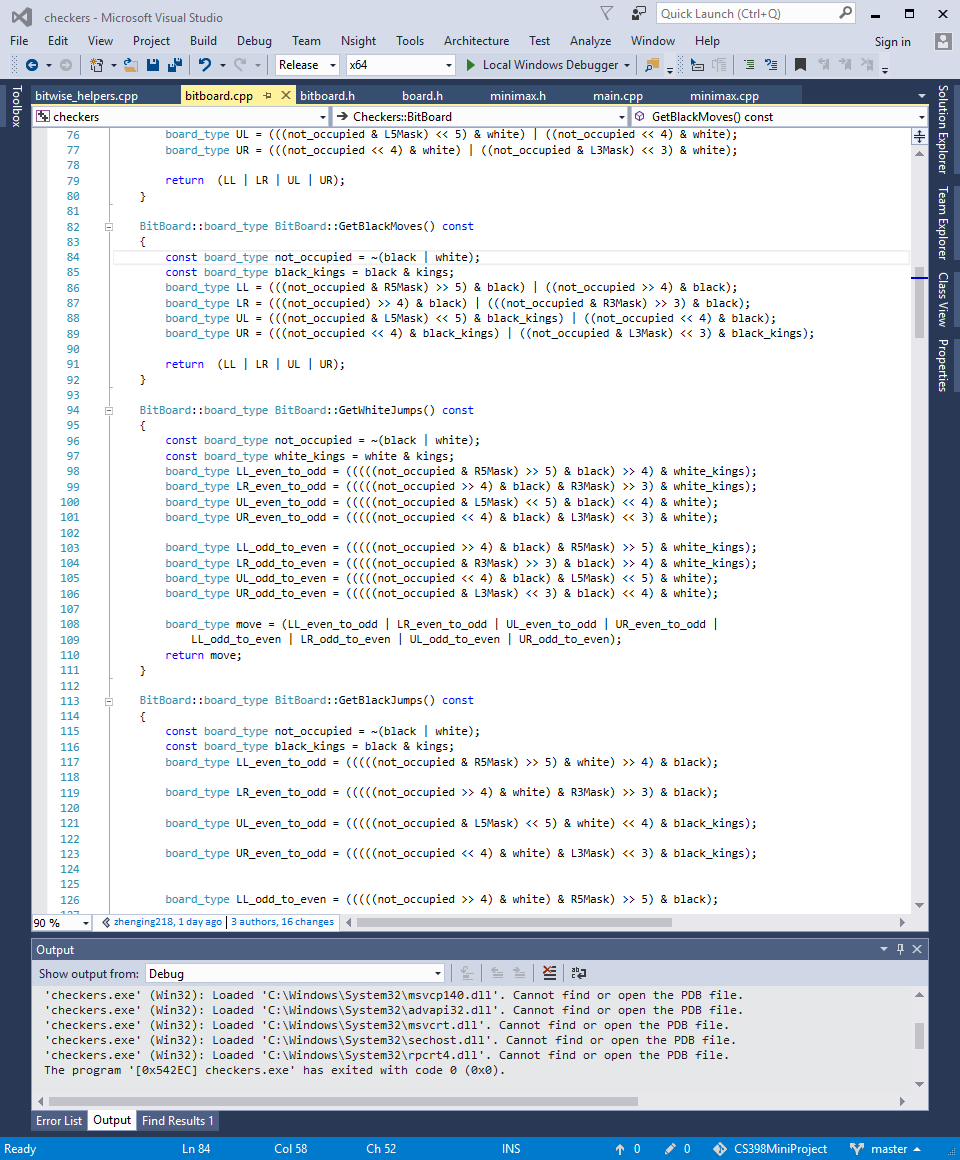


Figure 4a. An implementation of creating normal moves

As seen in Section 3, there is a possibility of a piece being able to move in 4 directions if there is nothing in the adjacent titles such that there will be lower left, lower right, upper left and upper right for both even and odd titles. Since we are going with bit implementation, we are constantly keep track of where the black and the white pieces are, to retrieve the empty cells for the moves considerations. While checking for the possible moves, it is also important to check if the moves are valid via the masks.

## Creating the list of jump moves in CPU

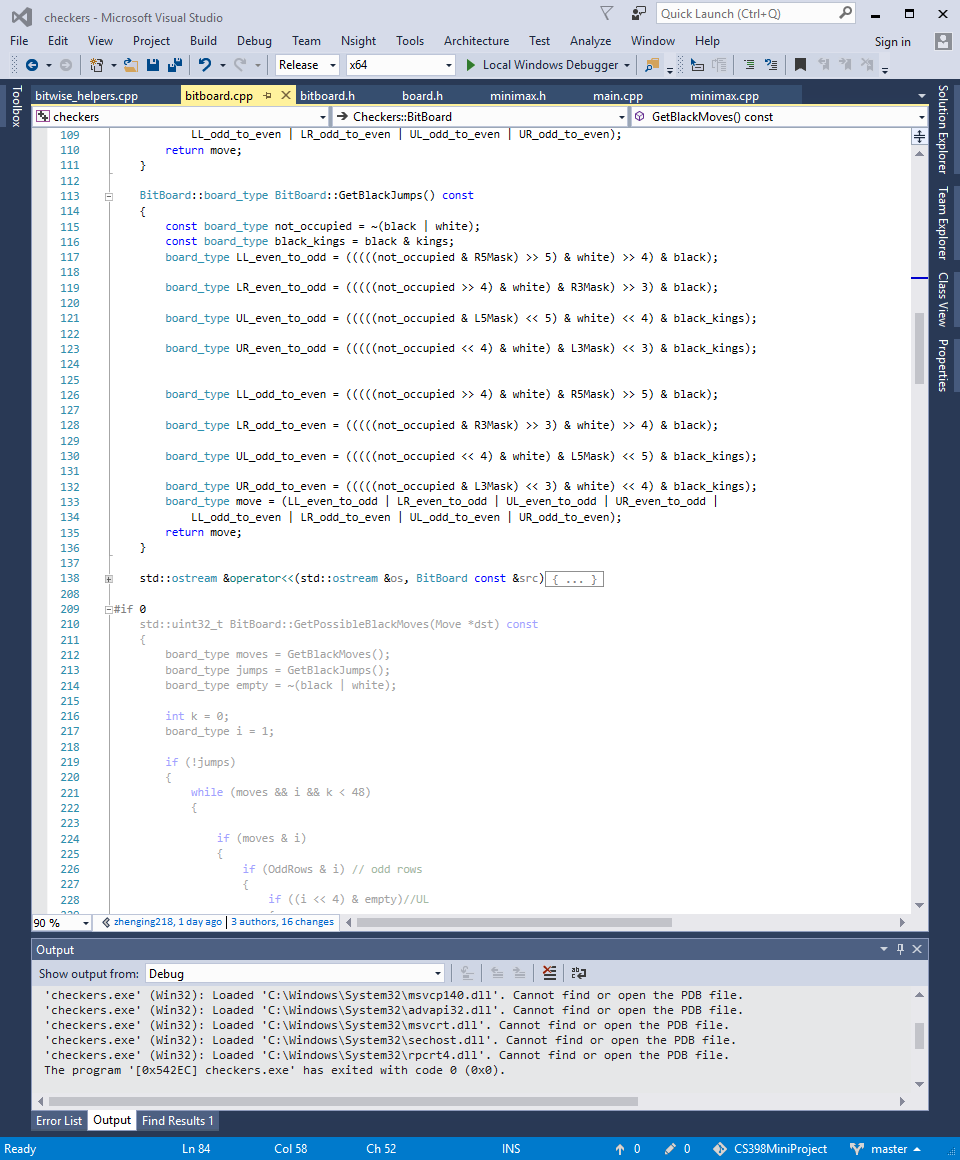


Figure 4b. An implementation of creating jump moves

With some implementation similar to creation of normal moves in Section 4.1, this time there is a need to check if the piece that is in the adjacent tiles are the opponent pieces. If the piece that is moving to any adjacent pieces has opponent tiles, the piece is then moved to the next tile after the opponent piece with their shifting based on the piece current position.

## The mask used for verification

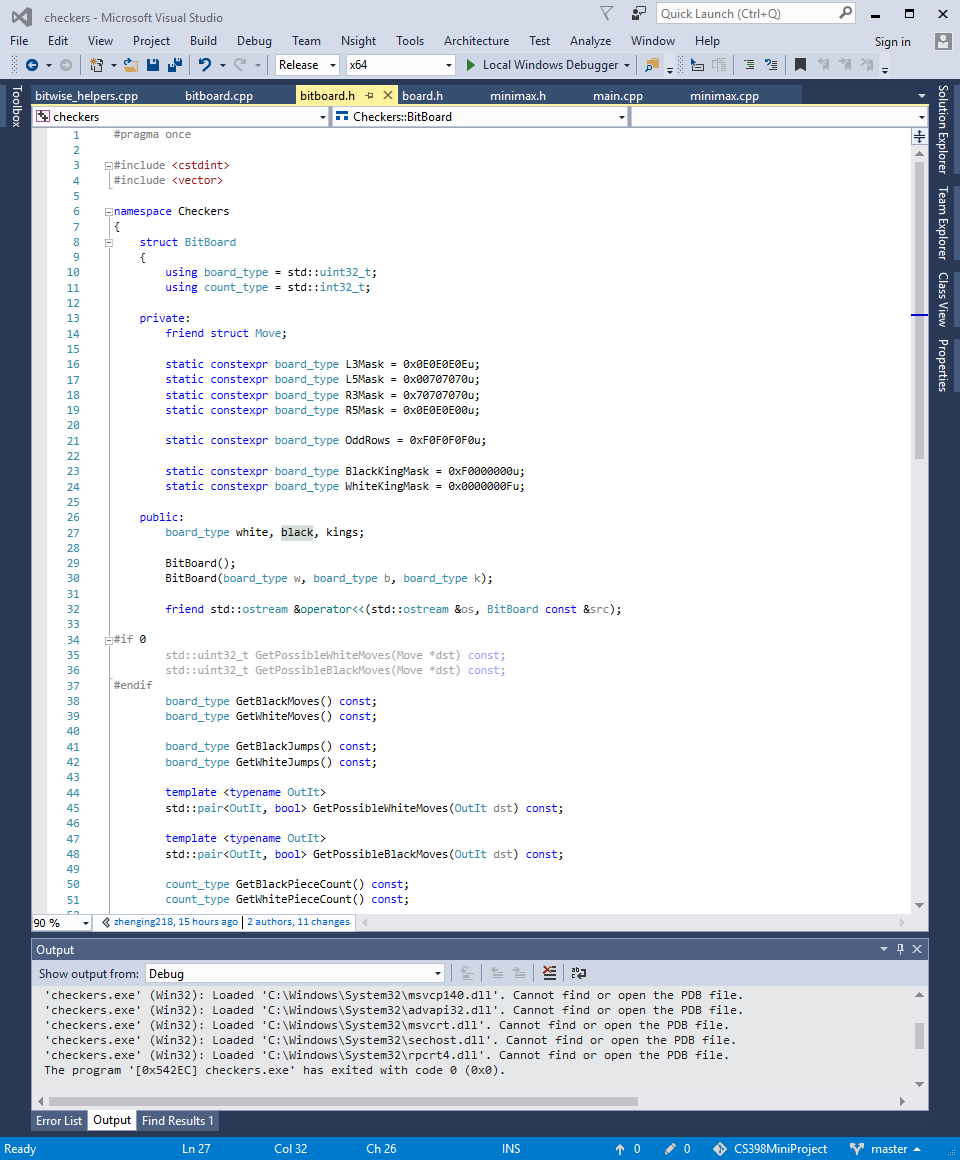


Figure 4c. The masks for checking valid moves

The L3,L5,R3,R5 masks are what is used to make a comparison with to see if the pieces can move to the corresponding even or odds tiles, as the king pieces in this case black could move additionally in the opposite direction thus the check of UL and UR is catered to the king pieces and for white pieces LL and LR. There is no need for if statements if we are using bit wise operations. Finally, after getting the possible moves in the 4 directions, all that is left is to return the bit results.

## Creating the list of moves in GPU

# Getting the best move set available



Figure 5. An illustration of minimax algorithm

## Using CPU minimax

## Using GPU minimax