Assignment 4: Gekitai

Due: 20:00, Sat 6 Nov 2021 File name: `gekitai.cpp` Full marks: 100

Introduction

The objective of this assignment is to let you practice (1) using 2D arrays, and (2) modularizing your program into appropriate functions.

You will implement a two-player board game called *Gekitai* (a Japanese word which means "repel" or "push away"). Here is the game description from the designer (we rephrased a little bit): Gekitai is a 3-in-a-row game played on a 6x6 grid. Each player has eight colored pieces and takes turns placing them anywhere on any open space on the board. When placed, a piece pushes all adjacent pieces outwards one space if there is an open space for it to move to (or off the board). Pieces shoved off the board are returned to the player. If there is not an open space on the opposite side of the pushed piece, it does not push (a newly played piece cannot push two or more other lined-up pieces). The first player who either (1) lines up three of their color in a row (horizontal or vertical or diagonal) at the end of their turn (after pushing), or (2) has all eight of their pieces on the board (also after pushing), is declared the winner. To quickly understand how to play this game, you may also watch this game review video or play this online game implementing Gekitai.

The key idea of this game is the "repel" effect when placing a piece onto the board. For example, refer to Figure 1 below. Suppose that the player (Red) of the current turn is to put a piece at location B2. Then all the three pieces (both the player's and opponent's) adjacent to B2 will be pushed away by one square outward. So, the black piece originally at A1 is shoved off the board and recycled to the player (Black) for making future turns whereas the pieces at C2 and B3 will be repelled to new positions D2 and B4 respectively.

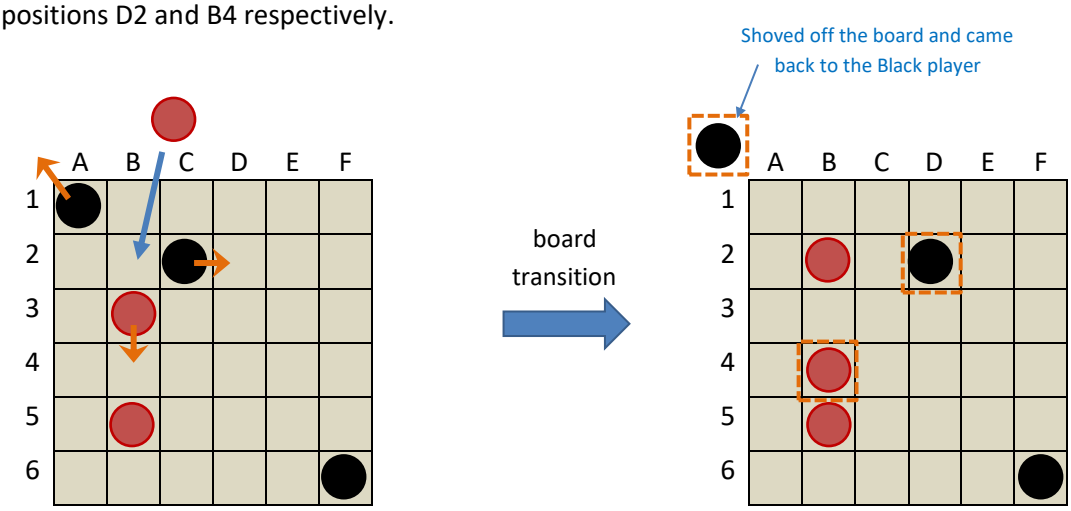

Figure 1: An example move on the game board and its effect on the adjacent pieces

However, remember (from the above description) that a move cannot push away a piece which has another adjacent piece (both the player's and opponent's) occupying the square it is being pushed onto. For example, look at Figure 2 below. If the player (Black) puts a piece at B3, it won't repel the

piece at B4 downward because there is no open space (B5 is already occupied), and only the piece at B2 will be pushed upward to position B1.

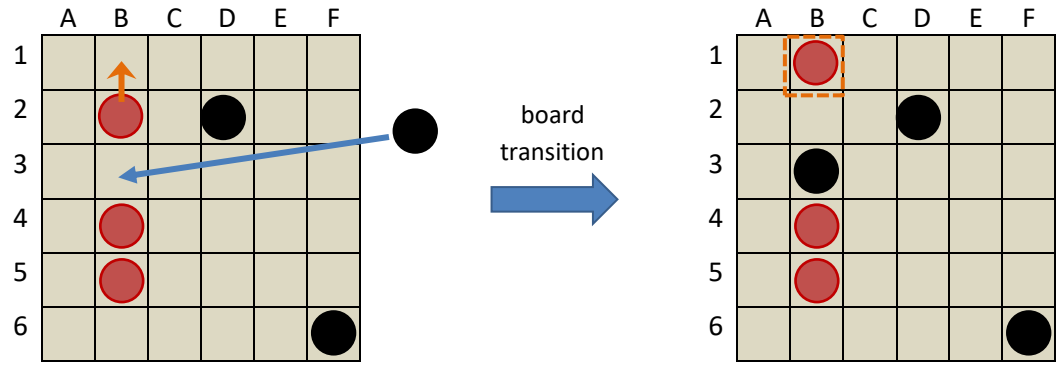

Figure 2: Another example move on the game board and its effect on the adjacent pieces

The last move made by Black was indeed a bad one – if Red puts a piece at B6 now, Red will win the game because three red pieces have formed a vertical line (see Figure 3).

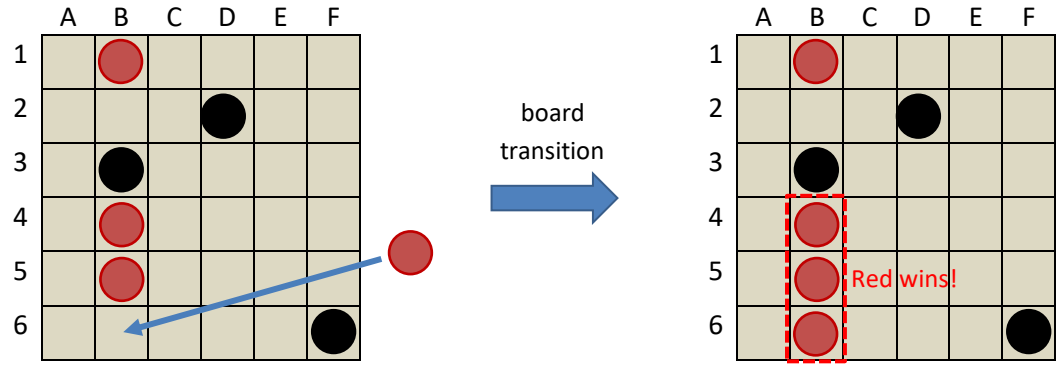

Figure 3: A move that lets Red win the game

To achieve the repelling effect, your program may need to scan all the eight directions (N, NE, E, SE, S, SW, W, NW) around the target position for a piece placement. For example, to put a piece at D3 in Figure 4, it should check if there exist any piece(s) at all the blue cells, and if any, it should check if the corresponding purple cell(s) are empty before moving the pieces outward into the purple cells.

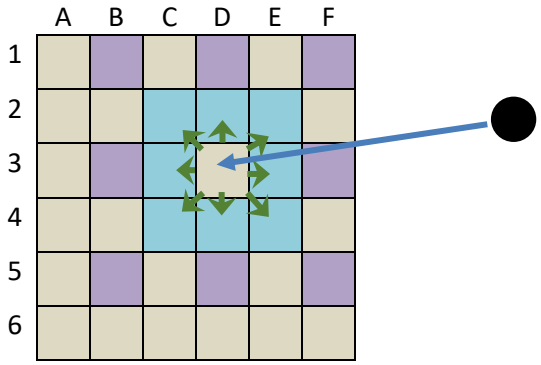

Figure 4: Positions to check to repel pieces adjacent to a target cell

Figure 5 below shows other possible winning conditions. Recall that a line can be formed horizontally, vertically, and diagonally, and that there is another winning condition: if all the eight pieces of the same color are placed on the board, the player of that color will win.

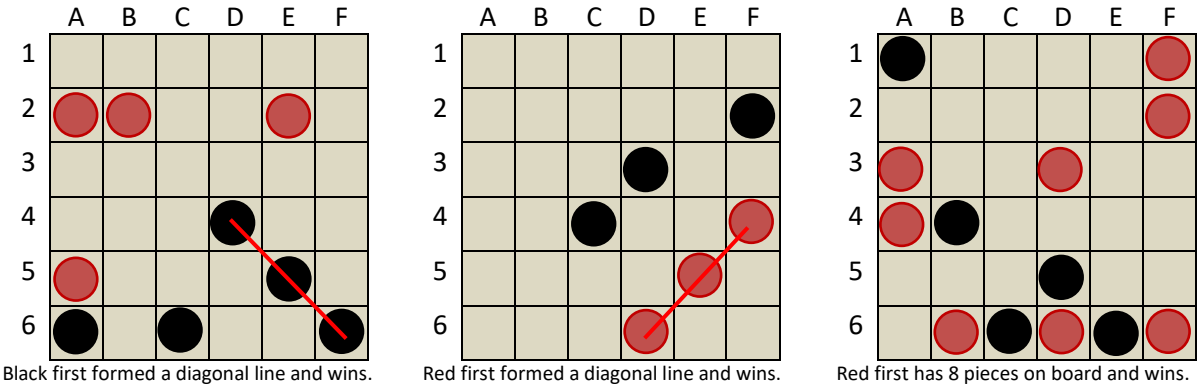

Figure 5: Other winning conditions

Program Specification

This section describes the representation of a board in a game, the necessary functions, and the flow of your program.

Basic Requirements

- You *cannot declare any global variables* (variables declared outside any functions), but global constants or global constant arrays are both allowed.
- The board size, number of pieces per player, and number of pieces required to form a row to win should be made easily scalable (define them as global constants near the top of your code), so our TAs can grade your program with different board settings by modifying them.

```
const int N = 6;    // board size
const int P = 8;    // number of pieces per player
const int L = 3;    // number of pieces required to form a line
```

Game Board Representation

We are going to implement this game as a C++ console-based program and the game board will be represented by characters arranged in Figure 6 below:

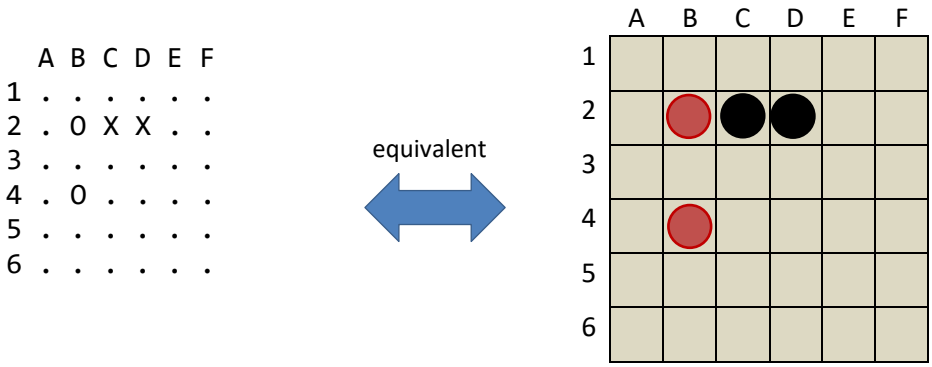

Figure 6: An example game board in the console.

In the beginning, all locations (or cells) on the board are empty (represented by a dot symbol). Pieces of player 1 and player 2 are denoted by 'X' and 'O' symbols, respectively.

Required Functions

Your program must contain the following functions, and they must be called *somewhere* in your program. You must ***not*** modify the prototypes of all these functions (if you like, you may change the parameter names but not their order and types. Please follow the specified function names exactly, including letter case and underscores). You can design extra functions if you find necessary.

Besides testing your program via your written main function, ***all the functions below will be graded individually.*** That is, we shall replace your `main()` with our testing code to call the functions one by one for grading. So, your code for each function shall implement the description of that function only. You shall ***not*** write any code in a function that is beyond that function's description. But you are always allowed to call the extra functions that you designed from these required functions to carry out the specified tasks.

`bool is_valid_move(char board[][N], char player, int y, int x)`

This function returns `true` if it is valid to move a piece of the specified `player` onto the board at row `y` and column `x` (both zero-based indexes), i.e., the array element `board[y][x]` is storing a dot symbol (representing an empty space, which is not yet occupied and not beyond the board's boundaries). Otherwise, it returns `false`.

`int pieces_on_board(char board[][N], char player)`

This function returns the number of pieces that are on the board and belong to `player`. Calling this function will be useful to assist determining whether the game is over. When it returns a value that is equal to the `P` constant, that means `player` has won.

`bool pieces_in_line(char board[][N], char player)`

This function returns `true` if it can find `L` consecutive pieces in a line on the board that belong to the specified `player`, and `false` otherwise. Calling this function will be useful to assist determining whether the game is over. When it returns `true`, that means `player` has won.

`void move(char board[][N], char player, int y, int x)`

This function carries out updates of the `board` array to actualize the effects of a move at row `y` and column `x` (both zero-based indexes) made by the specified `player`. This includes setting the array element `board[y][x]` to `player` and repelling its adjacent pieces (8 possible cells surrounding the cell `board[y][x]`) outward by one space if there is open space for them to move into.

`void print(char board[][N])`

Prints `board` to the screen using the format in Figure 6 (see more from our provided sample runs).

Program Flow

The program flow of the game is described as follows. You should call the functions above to aid your implementation.

1. The program starts the game with an empty board (filled with dots). Player 1 takes the first turn.
2. Then, prompt the current player to enter a position to put a piece in. Users will *always enter an alphabet (column) followed by an integer (row)* here (See more details in next section).
3. A piece cannot be placed ("Invalid move!") to the input position if the board array element at that position is not empty (a dot) or if the position is outside the board's boundaries. In case it is not placeable, display a warning message and go back to Step 2.
4. Update the board by putting the input piece to the corresponding position and repelling existing adjacent pieces away.
5. Swap player to take the next turn.
6. Repeat steps 2–5 until a player wins (forming a line of L pieces or having all P pieces put on the board) or the game draws (See more details in next section).
7. When a game finishes, display the message *"Player 1 wins!"* or *"Player 2 wins!"* or "*Draw game!*" accordingly.

Restrictions and Assumptions

- Please use a character array to represent the board, and pass it as arguments between functions. Avoid using other container classes like array, vector, map, etc. available in the standard library.
- The board is always a square. So only a single global constant N is needed to define its size.
- The normal board configuration is: N=6, P=8, L=3. To check if your program is scalable on board sizes, we may have test cases on N other than 6 although this may affect the original game rules. You may assume the range for N is between 5 and 8. For N = 5, P will be set to 6 only.
- The row index markers on the left of the printed board are always right adjusted at a fixed width of 2 characters. For single-digit row indexes, there is a single space ahead of them. (Although we assumed N is bounded by 8, this is still good for paving way for 2-digit row indexes in the future.)
- We assume that cell address inputs always follow the format of one letter (A-Z or a-z) plus one integer (1-26). Uppercase or lowercase inputs like "A1", "h7", or even having a space between them like "b 5" will be accepted as normal. But you need NOT handle weird inputs like "AA1", "A01", "abc", "A3.14", "#a2", … for cell addresses, which will cause `cin` fail. The behavior upon receiving these is unspecified, probably running into forever loop or program crash.
- Due to the repelling effect, it is possible that a move may cause both players to have L pieces forming a line at the same time. The official game rules may regard the moving player as the winner. But we will treat this as a draw game. So, your program should handle a draw game situation by checking if both players get a line of L pieces simultaneously, and print the draw game message. It is not possible for both players to put all their P pieces onto the board at the same time, so there is no need to check against this condition.

Sample Run

The following shows a sample run. The blue text is user input and the other text is the program printout. You can try the provided sample program for other input. *Your program output should be exactly the same as the sample program* (same text, symbols, letter case, spacings, etc.). Note that there is a space after '**:**' in the printout.

```
Round 1:
   A B C D E F
 1 . . . . . .
 2 . . . . . .
 3 . . . . . .
 4 . . . . . .
 5 . . . . . .
 6 . . . . . .
Player X's turn: H3↵
Invalid move!
Player X's turn: Z -1↵
Invalid move!
Player X's turn: A 1↵
Round 2:
   A B C D E F
 1 X . . . . .
 2 . . . . . .
 3 . . . . . .
 4 . . . . . .
 5 . . . . . .
 6 . . . . . .
Player O's turn: a6↵
Round 3:
   A B C D E F
 1 X . . . . .
 2 . . . . . .
 3 . . . . . .
 4 . . . . . .
 5 . . . . . .
 6 O . . . . .
Player X's turn: a6↵
Invalid move!
Player X's turn: c3↵
Round 4:
   A B C D E F
 1 X . . . . .
 2 . . . . . .
 3 . . X . . .
 4 . . . . . .
 5 . . . . . .
 6 O . . . . .
Player O's turn: d4↵
Round 5:
   A B C D E F
 1 X . . . . .
 2 . X . . . .
 3 . . . . . .
 4 . . . O . .
 5 . . . . . .
 6 O . . . . .
Player X's turn: c3↵
```

```
Game over:
   A B C D E F
 1 X . . . . .
 2 . X . . . .
 3 . . X . . .
 4 . . . . . .
 5 . . . . O .
 6 O . . . . .
Player X wins!
```

Another sample run:

```
Round 1:
   A B C D E F
 1 . . . . . .
 2 . . . . . .
 3 . . . . . .
 4 . . . . . .
 5 . . . . . .
 6 . . . . . .
Player X's turn: e6↵
Round 2:
   A B C D E F
 1 . . . . . .
 2 . . . . . .
 3 . . . . . .
 4 . . . . . .
 5 . . . . . .
 6 . . . . X .
Player O's turn: a2↵
Round 3:
   A B C D E F
 1 . . . . . .
 2 O . . . . .
 3 . . . . . .
 4 . . . . . .
 5 . . . . . .
 6 . . . . X .
Player X's turn: e4↵
Round 4:
   A B C D E F
 1 . . . . . .
 2 O . . . . .
 3 . . . . . .
 4 . . . . X .
 5 . . . . . .
 6 . . . . X .
Player O's turn: c2↵
Round 5:
   A B C D E F
 1 . . . . . .
 2 O . O . . .
 3 . . . . . .
 4 . . . . X .
 5 . . . . . .
 6 . . . . X .
Player X's turn: e3↵
```

```
Round 6:
   A B C D E F
 1 . . . . . .
 2 O . O . . .
 3 . . . . X .
 4 . . . . . .
 5 . . . . X .
 6 . . . . X .
Player O's turn: d2↵
Round 7:
   A B C D E F
 1 . . . . . .
 2 O O . O . .
 3 . . . . . .
 4 . . . . . X
 5 . . . . X .
 6 . . . . X .
Player X's turn: c5↵
Round 8:
   A B C D E F
 1 . . . . . .
 2 O O . O . .
 3 . . . . . .
 4 . . . . . X
 5 . . X . X .
 6 . . . . X .
Player O's turn: e1↵
Round 9:
   A B C D E F
 1 . . . . O .
 2 O O . . . .
 3 . . O . . .
 4 . . . . . X
 5 . . X . X .
 6 . . . . X .
Player X's turn: b6↵
Round 10:
   A B C D E F
 1 . . . . O .
 2 O O . . . .
 3 . . O . . .
 4 . . . X . X
 5 . . . . X .
 6 . X . . X .
Player O's turn: c4↵
Game over:
   A B C D E F
 1 . . . . O .
 2 O O O . . .
 3 . . . . . .
 4 . . O . X X
 5 . . . . X .
 6 . X . . X .
Draw game!
```

Submission and Marking

- Your program source file name should be `gekitai.cpp`. Submit the file in Blackboard (https://blackboard.cuhk.edu.hk/).
- Insert *your name*, *student ID*, and *e-mail* as comments at the beginning of your source file.
- You can submit your assignment multiple times. Only the latest submission counts.
- Your program should be *free of compilation errors and warnings*.
- Your program should *include suitable comments as documentation*.
- ***Do NOT plagiarize.*** Sending your work to others is subject to the same penalty for copying work.