Sabancı University

Faculty of Engineering and Natural Sciences
CS204 Advanced Programming
Spring 2020

Homework 6 – A two-party simple board game with object sharing

Due: 22/04/2020, Wednesday, 21:00

PLEASE NOTE:
Your program should be a robust one such that you have to consider all relevant programmer mistakes and extreme cases; you are expected to take actions accordingly!

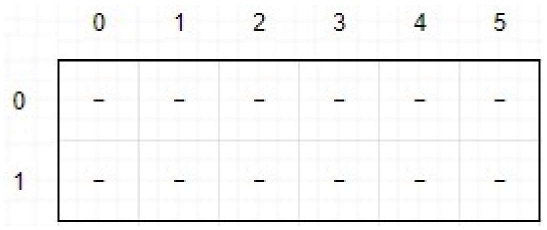
You can NOT collaborate with your friends and discuss solutions. You have to write down the code on your own. Plagiarism will not be tolerated!

Introduction

In this homework, you are asked to implement a simple board game using the *object sharing* concept of C++. The main program, which includes the game implementation using classes, is given to you as well as the .exe file to try out the game. You are expected to write two classes: *Board* class and *Player* class. We are going to explain these classes in more details in the following sections. Before that, we start with a general description of the game.

The Game

The game is two-player turn-based game, which is to be played on a 2×6 board. The board is simulated with character matrix of `char` type. Please see Figure 1 for an empty board. Each player has a pawn and moves it on the board randomly. Each player starts from 0 0 but one player moves clockwise, while other one moves counterclockwise on the board.



	0	1	2	3	4	5
0	-	-	-	-	-	-
1	-	-	-	-	-	-

Fig 1. Initial State of the Board

The players play in turns. At first, *player A* starts the game. Then *player B* plays. This continues in turns until the game finishes. Before playing in each turn, the player first rolls a dice[1] to determine the amount of cells to move; the program uses a random number generator that returns an integer between (and incl.) one and six to simulate a dice (given in the main program; you will not implement this). *Player A* moves its pawn on the board *n* cells in clockwise manner, where *n* is the current dice value. If the new cell of *player A* is vacant, then *player A* claims ownership of this cell. If that cell is not vacant, then it does not change

[1] The word *dice* is the plural form of *die* (*zar* in Turkish). Although a single *die* is used here, we will use the term *dice* in a grammatically wrong manner in order not to create confusion with the other meaning of the word *die*.

anything. After that, turn passes to *player B*, which plays similarly but counter-clockwise. The first player that owns seven cell wins the game. Of course, the game may end with a draw if both players have same amount of cells and there are no vacant cells left.

Here is a sample iteration of the game, which may help to illustrate the gameplay. The game begins with an empty board as shown in Figure 1. Initially all cells are vacant that are represented with '-' character.

Player A rolls the dice and assume it returns 5. Thus, *player A* moves 5 cells in clockwise manner. It ends up in (0,5) and gets ownership of this cell (by changing that cell's value to `'A'`) as shown in Figure 2.

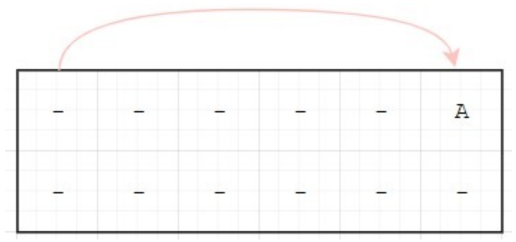


Fig 2. After *player A* went from 0 0 to 0 5, player A gets ownership of this cell.

Then, it is *Player B*'s turn. Assume dice returns 2. As mentioned before, *Player B* moves counter-clockwise. Thus, it ends up in cell (1,1) and gets ownership of that cell (by changing that cell's value to `'B'`) as shown in Figure 3.

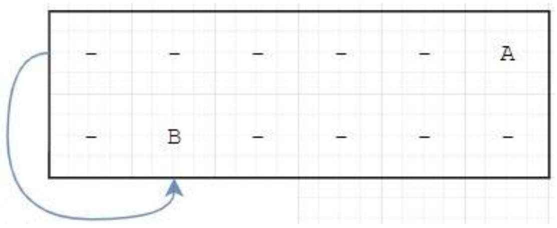


Fig 3. After *player B*'s turn, from 0 0 to 1 1 and gets that cell

In the next turn, suppose *player A* rolls 4, moves to (1, 2) and gets that cell as shown in Figure 4.

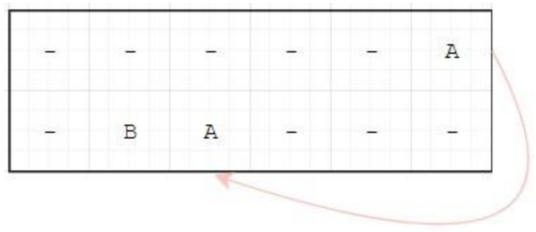


Fig 4. After *player A*'s turn, from 0 5 to 1 2 and gets that cell

In the next turn, suppose *player B* rolls 5 and moves to (0, 5). Since that cell has already been owned and not vacant, it will not get a new cell ownership in this turn. So the board layout will not change as shown in Figure 5.

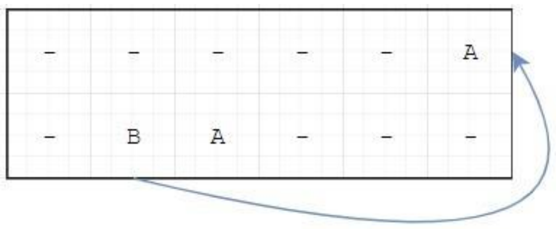


Fig 5. After *player B*'s turn, from 1 1 to 0 5

Then *player A* rolls 6 and moves to (0, 3). It gets that cell as shown in Figure 6.

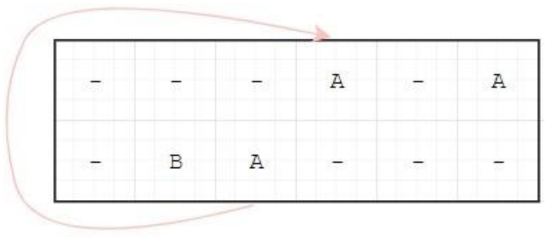


Fig 6. After *player A*'s turn, from 1 2 to 0 3 and gets that cell

Player B was at (0,5). In this turn, suppose it rolls 5, moves to (0,0) and gets this vacant cell as shown in Figure 7.

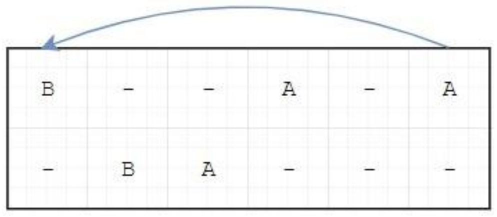


Fig 7. *Player B* moves from 0 5 to 0 0 and gets 0 0 cell.

Then, *player A* rolls 2 and moves to (0,5), which was already owned by itself. Thus, the board layout does not change after this move as shown in Figure 8.

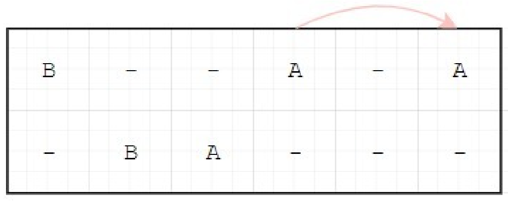


Fig 8. After *player A*'s turn from 0 3 to 0 5

After that, *player B* rolls 1 and moves to (1,0). It gets that cell as shown in Figure 9.

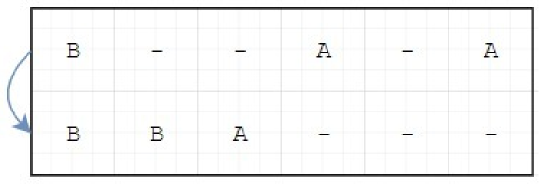
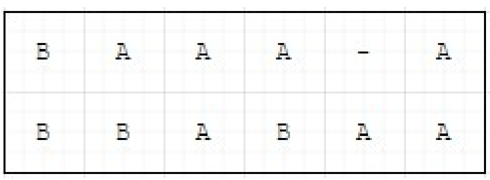


Fig 9. *Player B* went from 0 0 to 1 0 and gets it.

After some turns at which the dice values are 1, 3, 1, 5, 6, 4, 1, the final state of the board becomes as shown in Figure 10. Here, *Player A* wins since it has 7 cells owned.



```
B   A   A   A   -   A

B   B   A   B   A   A
```

Fig 10. *Player A* wins.

This is just an example and at the end *Player A* won. Of course, there could be other outcomes such a *Player B*'s victory or a draw. We provide you the .exe file of our implementation; you can try it to see other cases.

In the example above, only the ownerships are shown on the board; not the pawn positions. Actually, you will do the same in the program, but of course you have to keep track of the pawn positions to implement the movements.

Program Flow and Input

There is only one input which is the seed[2] of the random number generator that is used to simulate the dice concept. This input operation is already handled in the main given to you. Input for the seed must be integer value; if you enter a non-integer while trying the provided .exe file, the program may generate same random numbers for different runs. Thus, please do not enter non-integer inputs for seed value.

The game will be played in turns starting Player A and will end by either victory of one of the players (owning 7 cells) or by a draw (having no vacant cell and each player owns 6 cells). The game logic has been partially implemented in main and it is provided to you. You are not supposed to change it; what you will do is just the class design and implementation for the Board and Player classes.

The `Board` Class

The `Board` class will be used to create and manipulate a board on which the game will be played. A board is represented by a built-in matrix, which is a private data member of the class. The matrix will have fixed number of rows and columns. You can define this private data member as shown below.

```
char theBoard[2][6];
```

Now, we will give the constructor and some of the member function explanations of the `Board` class.

Default Constructor: You need to initialize all of the matrix elements to the dash '–' character, which will indicate that the cell is vacant.

displayBoard: The displayBoard function does not take any parameters. It only displays the current state of the board. You need to display the game board exactly as you will see when

[2] In computer systems, random number generation works as a recursive mathematical function. The initial value of this function is called the *seed*. The sequence of random numbers generated becomes the same for the same seed value. That is why we input the seed value to control the random number generation during grading process.

you run the executable file that comes with this homework pack. Please remark that there should be one blank between two horizontal cells for a tidy output.

getOwner: This function basically returns the owner of the cell at given position. It takes two `int` parameters and returns a `char` value. The integer parameters specify the row and column indices of the board, respectively. This function is supposed to return `'-'` (dash) character if cell is vacant at the specified row and column on the board; otherwise it returns the owner, i.e. either `'A'` or `'B'`.

setOwner: This function basically sets the ownership of the specified position on the board. It takes two `int` parameters and one `char` parameter. These parameters represent the location of the cell (row and column) and who will be the owner of this cell (`'A'` or `'B'`).

isFull: This function is used to check whether the board is full or not. As you can predict it should return Boolean value; *true* if the board is full, *false* if it is not. Here, full board means having no vacant cells.

countOwnedCells: This function takes one `char` parameter. It should return number of cells, that are owned by player specified by the parameter.

Two of the above functions (namely `displayBoard` and `isFull`) are explicitly called in the game implementation given to you in main.cpp. However, the others are not directly called in main.cpp; but these need to be used by the `Player` class member functions. Since the use of friend functions and friend classes are **not** allowed in this homework, in the implementation of the `Player` class, you will need to use them to manipulate the shared board object. You do **not** need other functions (especially other types of accessors/mutators) for the `Board` class; thus please do not add them.

The `Player` Class

The `Player` class will be used to manage the players of the game. There will be two player objects playing on the same board in a game. Thus, player objects must share a Board object using *object sharing* concept and principles of C++ as we have seen in the lectures. We have seen two different methods for object sharing in the course; due to our main function implementation, which is provided to you, you must use the reference variable method.

The `Player` class should keep its identity as a `char`, current coordinate on the board as two `int`s (one for row and one for column), and one more `int` for the direction of movement, as private data members. Now, we will give constructor and member function explanations of the `Player` class.

Constructor: The constructor of the `Player` class takes three parameters, which are the `Board` object that will be played on, the id character of the player, and an integer to determine direction (1 for clockwise; 0 for counter-clockwise). These parameters are used to initialize the corresponding private data members. Since both players start at (0,0), current row and column values should be initialized to 0.

move: It takes one `int` parameter representing the amount of cells to move in the direction of the player object on which it is called. Movement is continuous such that when the player reaches the end points, it turns right (for clockwise) or left (for counter-clockwise). This function updates the position data members of the player object accordingly.

claimOwnership: This function basically sets the ownership of a vacant cell. It does not take any parameter, since the private members of the player object are enough. In this function, you have to check if the current location of the player is vacant or not. If it is vacant, then you can set its ownership to the player object. If the current cell has already been owned by any of the players, you will not change anything. You should use the `Board` class' member functions, such as getOwner and setOwner, to manipulate the board.

wins: This function does not take any parameters. It will return *true* if the player has won the game; it returns *false* otherwise. You have to use countOwnedCells function of the `Board` class in the implementation.

getRow: This function does not take any parameters and returns the player's current row.

getCol: This function does not take any parameters and returns the player's current column.

This member functions are sufficient; you do not need any other member functions and please do not add more functions.

Using Object Sharing Principles and Object Oriented Design

In your program, the *Board* object must be shared by the *Player* objects. For this object sharing, you have to employ the method that uses reference variables.

It should be clear that you will write two classes for `Board` and `Player`. You need to analyze the requirements carefully and make a good object-oriented design for these classes. In this context, you have to determine the data members and design/implement member functions of each class correctly. We will evaluate your object oriented design as well. Moreover, you are not allowed to use friend class or friend functions in your design (i.e. you are not allowed to use the `friend` keyword anywhere in your code). Our aim by this restriction is not to make your life miserable, but to enforce you to proper object oriented design and implementation.

Provided files

game.exe: This is the executable file of our implementation for this board game. In this homework, we do not provide any sample runs due to the interactivity of the game. Instead we provide this executable so that you can try and understand the requirements.

main.cpp: This file contains the `main` function and the related code of the game implementation that uses class functions. In this homework, our aim is to reinforce object oriented design capabilities; thus, we did not want you to deal with the class usage, but focus on their design and implementation. Please examine this provided file in order to understand how the classes are used and how the game is played. We will test your codes with this main.cpp with different inputs. You are not allowed to make any modifications in this file (of course, you can edit the file to add `#include`s, etc. to the beginning of the file); you have to create and use other files for class definitions and implementations.

Do not forget to submit all .h and .cpp files (including the classes' .h and .cpp files, and the main .cpp file).

Please see the previous homework specifications for the other important rules and the submission guidelines

Good Luck!
Albert Levi, Vedat Peran