ENEE 150: Intermediate Programming Concepts for Engineers

Fall 2020 **Handout #7**

Project #1: Backgammon: Due Sept. 24 at 11:59pm

In this project, you will build a program that allows two human players to play backgammon. The program will keep track of the backgammon board and check that each player's moves are valid. It will support checking of several types of moves, including basic moves, entry moves from the bar, and bearing off moves. You will leverage the code for printing the board that you wrote in homework #1. In this project, you will provide the rest of the code that enforces the rules of backgammon.

1 The Board

In backgammon, each player starts with 15 *checkers*, colored either white or red. The checkers are placed on a board containing 24 spaces, called *points*. The points are grouped into 4 quadrants, each containing 6 points. Initially, each player's checkers are placed on 4 points–5 checkers on two different points, 3 checkers on one point, and 2 checkers on one point. Your program should display the points on the board using "|" characters, or the letters "W" or "R". Each "W" character denotes a white checker while each "R" character denotes a red checker. The bottom two quadrants of points are numbered 1–12 from right to left below the board, while the top two quadrants of points are numbered 13–24 from left to right above the board. Checkers on points in the bottom two quadrants are stacked from the bottom upwards, while checkers on points in the top two quadrants are stacked from the top downwards. While a point may contain any number of checkers, you should never print more than 5 checkers per point. There is also a "Bar" that appears below the board with counters for each player–this will be explained later. Here's how your program should display the board at the beginning of the game:

```
13 14 15 16 17 18    19 20 21 22 23 24
W  |  |  |  R  |     R  |  |  |  |  W
W  |  |  |  R  |     R  |  |  |  |  W
W  |  |  |  R  |     R  |  |  |  |  |
W  |  |  |  |  |     R  |  |  |  |  |
W  |  |  |  |  |     R  |  |  |  |  |

R  |  |  |  |  |     W  |  |  |  |  |
R  |  |  |  |  |     W  |  |  |  |  |
R  |  |  |  W  |     W  |  |  |  |  |
R  |  |  |  W  |     W  |  |  |  |  R
R  |  |  |  W  |     W  |  |  |  |  R
12 11 10 9  8  7     6  5  4  3  2  1
  Bar:  White-0  Red-0
```

You should use your print_board function from homework #1 to display the board. Just like in homework #1, you should use the same two arrays, "the_board_white" and "the_board_red" (each containing 25 elements), to track the position of checkers on the board. In addition, you should use two additional variables, "the_bar_white" and "the_bar_red," to keep track of the number of white and red checkers, respectively, on the

bar. Finally, you should also use two additional variables, "num_white" and "num_red," to keep track of the total number of white and red checkers on the board, respectively. You can download the "board.c" file from ELMs that declares all of these variables globally.

2 Object of the Game

Each player has a "home quadrant": the home for the player of the white checkers is the lower-right quadrant (points 1–6), and the home for the player of the red checkers is the upper-right quadrant (points 19–24). Each player takes turns moving their checkers (see Section 3). White checkers move from larger-numbered points towards smaller-numbered points, while red checkers move from smaller-numbered points towards larger-numbered points. In other words, players move checkers towards their home quadrant. While making moves, a player can cause one of his/her opponent's checkers to be placed on *the bar* (see Section 4). The "Bar" label in the above display indicates how many of each player's checkers are on the bar. Checkers on the bar must "enter" the board to continue movement towards the home quadrant. Eventually, a player will move all of his/her checkers into his/her home quadrant. At that point, the player is allowed to move his/her checkers off the board, a process called "bearing off" (see Section 5). The first player to move all of his/her checkers off the board wins the game.

3 Basic Moves

3.1 Dice

At the beginning of a turn, a player rolls two 6-sided die. The following function generates a pseudo-random number between 1–6 to "simulate" the rolling of one die:

```
int roll_die()
{
  return ((random() % 6) + 1);
}
```

You should copy this code into your program, and call it each time a player rolls a die–*i.e.*, you will call this function twice at the beginning of each player's turn. After rolling the two die, your program should print the dice values, and prompt the appropriate player for his/her moves. In your project, you should begin the game by allowing the player of the white checkers to move first.

IMPORTANT: do not use the pseudo-random number generator, "random," anywhere else in your code, and only call "roll_die" exactly twice at the beginning of each player's turn. (This will ensure that the sequence of dice values in your program will exactly match the sequence in our solution code, which will be important for testing and grading purposes).

3.2 Performing Moves

During a player's turn, the prompted player is allowed two moves, one for each die value. Each move requires the prompted player to specify two numbers that identify one of the player's checkers to move from an originating point (first number) to a destination point (second number). The conditions for a valid move are:

1. Each of the two numbers in the move must be between 1–24.
2. The difference between the first number and the second number in the move must be positive for white checker moves, and negative for red checker moves.
3. The absolute value of the difference between the two numbers in a move must match one of the rolled dice. In particular, the first move on a turn must match one of the die values while the second move must match the *other* die value.
4. The originating point in a move must contain one or more checkers belonging to the moving player.
5. The destination point in a move must not contain more than one checker belonging to the moving player's opponent. In other words, the destination point can be either empty, contain any number of checkers belonging to the moving player, or contain exactly one checker belonging to the moving player's opponent.

After a player enters a move, your program should check the validity of the move. If the move is valid, your program should update the board with the move, and display the new board. If the move is invalid, your program should print "Invalid move" and re-prompt the player for another move. After processing two valid moves, the current player's turn ends, and a new turn for the opposing player begins. Note, during a turn, a player can move a single checker twice, or two different checkers once each. Here is some output that illustrates basic moves assuming the moves start from the beginning of the game:

```
Rolling dice:  4 1
White's move:  24 22
Invalid move
White's move:  24 25
Invalid move
White's move:  24 23
13 14 15 16 17 18    19 20 21 22 23 24
W  |  |  |  R  |     R  |  |  |  W  W
W  |  |  |  R  |     R  |  |  |  |  |
W  |  |  |  R  |     R  |  |  |  |  |
W  |  |  |  |  |     R  |  |  |  |  |
W  |  |  |  |  |     R  |  |  |  |  |

R  |  |  |  |  |     W  |  |  |  |  |
R  |  |  |  |  |     W  |  |  |  |  |
R  |  |  |  W  |     W  |  |  |  |  |
R  |  |  |  W  |     W  |  |  |  |  R
R  |  |  |  W  |     W  |  |  |  |  R
12 11 10 9  8  7     6  5  4  3  2  1
  Bar:  White-0  Red-0
White's move:  13 9
13 14 15 16 17 18    19 20 21 22 23 24
W  |  |  |  R  |     R  |  |  |  W  W
W  |  |  |  R  |     R  |  |  |  |  |
W  |  |  |  R  |     R  |  |  |  |  |
W  |  |  |  |  |     R  |  |  |  |  |
|  |  |  |  |  |     R  |  |  |  |  |
```

```
R  |  |  |  |  |     W  |  |  |  |  |
R  |  |  |  |  |     W  |  |  |  |  |
R  |  |  |  W  |     W  |  |  |  |  |
R  |  |  |  W  |     W  |  |  |  |  R
R  |  |  W  W  |     W  |  |  |  |  R
12 11 10 9  8  7     6  5  4  3  2  1
  Bar:  White-0  Red-0
Rolling dice:  2 3
Red's move:  12 10
Invalid move
Red's move:  12 14
13 14 15 16 17 18    19 20 21 22 23 24
W  R  |  |  R  |     R  |  |  |  W  W
W  |  |  |  R  |     R  |  |  |  |  |
W  |  |  |  R  |     R  |  |  |  |  |
W  |  |  |  |  |     R  |  |  |  |  |
|  |  |  |  |  |     R  |  |  |  |  |

|  |  |  |  |  |     W  |  |  |  |  |
R  |  |  |  |  |     W  |  |  |  |  |
R  |  |  |  W  |     W  |  |  |  |  |
R  |  |  |  W  |     W  |  |  |  |  R
R  |  |  W  W  |     W  |  |  |  |  R
12 11 10 9  8  7     6  5  4  3  2  1
  Bar:  White-0  Red-0
Red's move:  17 20
13 14 15 16 17 18    19 20 21 22 23 24
W  R  |  |  R  |     R  R  |  |  W  W
W  |  |  |  R  |     R  |  |  |  |  |
W  |  |  |  |  |     R  |  |  |  |  |
W  |  |  |  |  |     R  |  |  |  |  |
|  |  |  |  |  |     R  |  |  |  |  |

|  |  |  |  |  |     W  |  |  |  |  |
R  |  |  |  |  |     W  |  |  |  |  |
R  |  |  |  W  |     W  |  |  |  |  |
R  |  |  |  W  |     W  |  |  |  |  R
R  |  |  W  W  |     W  |  |  |  |  R
12 11 10 9  8  7     6  5  4  3  2  1
  Bar:  White-0  Red-0
Rolling dice:  2 4
White's move:
```

4 Entry Moves

If a player's checker moves onto a destination point containing a single checker belonging to his/her opponent, the moving player's checker replaces his/her opponent's checker, and the replaced checker is moved off the board onto "the bar." In this case, in addition to updating the destination point on the board, your program should also increment the counter on the bar appearing below the board corresponding to the player who's checker was moved off the board.

Once on the bar, a checker must "enter" the board before making more basic moves. White checkers enter starting at point #24, whereas red checkers enter starting at point #1 (*i.e.*, a checker enters at the point furthest away from its home quadrant). A player can make such an entry move by specifying "0" as the first number in the move. (The second number should be the destination point on the board to move to, similar to basic moves). Moreover, the number of points from entry to the destination point must match one of the rolled dice. For example, if a white checker is on the bar and its player rolls a "4" and a "1", the checker can enter at either point 21 or point 24 by specifying "0 21" or "0 24", respectively, as the move. (Note, these moves would also be subject to validity condition #5 from Section 3.2). If a player tries to make an entry move but violates any of these conditions, your program should print "Invalid move" and re-prompt the player for another move.

Finally, during a player's turn if the player has any checkers on the bar, **he/she must perform an entry move**. Only after all of a player's checkers are off the bar is the player allowed to perform a basic move. If a player has a checker on the bar and tries to perform a basic move, your program should print "Invalid move" and re-prompt the player for another move. Here is some output that illustrates entry moves assuming the move sequence from Section 3.2 has already occurred:

```
Rolling dice:  2 4
White's move:  24 20
13 14 15 16 17 18    19 20 21 22 23 24
W  R  |  |  R  |     R  W  |  |  W  |
W  |  |  |  R  |     R  |  |  |  |  |
W  |  |  |  |  |     R  |  |  |  |  |
W  |  |  |  |  |     R  |  |  |  |  |
|  |  |  |  |  |     R  |  |  |  |  |

|  |  |  |  |  |     W  |  |  |  |  |
R  |  |  |  |  |     W  |  |  |  |  |
R  |  |  |  W  |     W  |  |  |  |  |
R  |  |  |  W  |     W  |  |  |  |  R
R  |  |  W  W  |     W  |  |  |  |  R
12 11 10 9  8  7     6  5  4  3  2  1
  Bar:  White-0  Red-1
White's move:  6 4
13 14 15 16 17 18    19 20 21 22 23 24
W  R  |  |  R  |     R  W  |  |  W  |
W  |  |  |  R  |     R  |  |  |  |  |
W  |  |  |  |  |     R  |  |  |  |  |
W  |  |  |  |  |     R  |  |  |  |  |
|  |  |  |  |  |     R  |  |  |  |  |

|  |  |  |  |  |     |  |  |  |  |  |
R  |  |  |  |  |     W  |  |  |  |  |
R  |  |  |  W  |     W  |  |  |  |  |
R  |  |  |  W  |     W  |  |  |  |  R
R  |  |  W  W  |     W  |  W  |  |  R
12 11 10 9  8  7     6  5  4  3  2  1
  Bar:  White-0  Red-1
Rolling dice:  1 2
Red's move:  14 16
Invalid move
Red's move:  0 2
13 14 15 16 17 18    19 20 21 22 23 24
W  R  |  |  R  |     R  W  |  |  W  |
W  |  |  |  R  |     R  |  |  |  |  |
W  |  |  |  |  |     R  |  |  |  |  |
W  |  |  |  |  |     R  |  |  |  |  |
|  |  |  |  |  |     R  |  |  |  |  |

|  |  |  |  |  |     |  |  |  |  |  |
R  |  |  |  |  |     W  |  |  |  |  |
R  |  |  |  W  |     W  |  |  |  |  |
R  |  |  |  W  |     W  |  |  |  |  R
R  |  |  W  W  |     W  |  W  |  R  R
12 11 10 9  8  7     6  5  4  3  2  1
  Bar:  White-0  Red-0
Red's move:  19 20
13 14 15 16 17 18    19 20 21 22 23 24
W  R  |  |  R  |     R  R  |  |  W  |
W  |  |  |  R  |     R  |  |  |  |  |
W  |  |  |  |  |     R  |  |  |  |  |
W  |  |  |  |  |     R  |  |  |  |  |
|  |  |  |  |  |     |  |  |  |  |  |

|  |  |  |  |  |     |  |  |  |  |  |
R  |  |  |  |  |     W  |  |  |  |  |
R  |  |  |  W  |     W  |  |  |  |  |
R  |  |  |  W  |     W  |  |  |  |  R
R  |  |  W  W  |     W  |  W  |  R  R
12 11 10 9  8  7     6  5  4  3  2  1
  Bar:  White-1  Red-0
Rolling dice:  2 5
White's move:  13 11
Invalid move
White's move:  0 20
13 14 15 16 17 18    19 20 21 22 23 24
W  R  |  |  R  |     R  W  |  |  W  |
W  |  |  |  R  |     R  |  |  |  |  |
W  |  |  |  |  |     R  |  |  |  |  |
W  |  |  |  |  |     R  |  |  |  |  |
|  |  |  |  |  |     |  |  |  |  |  |
```

```
|  |  |  |  |  |  |     |  |  |  |  |  |
R  |  |  |  |  |     W  |  |  |  |  |
R  |  |  |  W  |     W  |  |  |  |  |
R  |  |  |  W  |     W  |  |  |  |  R
R  |  |  W  W  |     W  |  W  |  R  R
12 11 10 9  8  7     6  5  4  3  2  1
  Bar:  White-0  Red-1
White's move:  13 11
13 14 15 16 17 18    19 20 21 22 23 24
W  R  |  |  R  |     R  W  |  |  W  |
W  |  |  |  R  |     R  |  |  |  |  |
W  |  |  |  |  |     R  |  |  |  |  |
|  |  |  |  |  |     R  |  |  |  |  |
|  |  |  |  |  |     |  |  |  |  |  |

|  |  |  |  |  |  |     |  |  |  |  |  |
R  |  |  |  |  |     W  |  |  |  |  |
R  |  |  |  W  |     W  |  |  |  |  |
R  |  |  |  W  |     W  |  |  |  |  R
R  W  |  W  W  |     W  |  W  |  R  R
12 11 10 9  8  7     6  5  4  3  2  1
  Bar:  White-0  Red-1
Rolling dice:  4 6
Red's move:
```

5 Bearing Off Moves

Eventually, a player will move all of his/her checkers into his/her home quadrant. Once all of a player's checkers are in the home quadrant, the player is allowed to move the checkers off the board one by one, a process known as "bearing off". White checkers bear off the board after moving past point #1, whereas red checkers bear off the board after moving past point #24. After such a bearing off move is made, the checker is removed from the board for the rest of the game.

A player can make a bearing off move by specifying "0" as the second number in the move. (The first number should be the originating point on the board to move from, similar to basic moves). Moreover, the number of points from the originating point until the checker bears off must be equal to or less than the value of one of the rolled dice. (In other words, rolled dice values do not have to exactly match the number of steps needed to bear off). For example, if all red checkers are in their home quadrant, bearing off a particular red checker at point #3 can occur by specifying the move "3 0" as long as one of the rolled dice takes on a value between 3–6. Here is some output that illustrates bearing off moves:

```
13 14 15 16 17 18    19 20 21 22 23 24
|  |  |  |  |  |     R  R  R  R  R  |
|  |  |  |  |  |     R  R  R  R  R  |
|  |  |  |  |  |     R  R  R  |  |  |
|  |  |  |  |  |     R  |  |  |  |  |
|  |  |  |  |  |     R  |  |  |  |  |

|  |  |  |  |  |  |     W  |  |  |  |  |
|  |  |  |  |  |     W  W  |  |  |  |
|  |  |  |  |  |     W  W  |  |  |  |
|  |  |  |  |  |     W  W  |  W  W  |
|  |  |  |  |  W     W  W  W  W  W  |
12 11 10 9  8  7     6  5  4  3  2  1
  Bar:  White-0  Red-0
Rolling dice:  4 1
White's move:  4 0
Invalid move
White's move:  7 3
13 14 15 16 17 18    19 20 21 22 23 24
|  |  |  |  |  |     R  R  R  R  R  |
|  |  |  |  |  |     R  R  R  R  R  |
|  |  |  |  |  |     R  R  R  |  |  |
|  |  |  |  |  |     R  |  |  |  |  |
|  |  |  |  |  |     R  |  |  |  |  |
```

```
|  |  |  |  |  |  |     W  |  |  |  |  |
|  |  |  |  |  |  |     W  W  |  |  |  |
|  |  |  |  |  |  |     W  W  |  W  |  |
|  |  |  |  |  |  |     W  W  |  W  W  |
|  |  |  |  |  |  |     W  W  W  W  W  |
12 11 10 9  8  7     6  5  4  3  2  1
  Bar:  White-0  Red-0
White's move:  3 2
13 14 15 16 17 18    19 20 21 22 23 24
|  |  |  |  |  |  |     R  R  R  R  R  |
|  |  |  |  |  |  |     R  R  R  R  R  |
|  |  |  |  |  |  |     R  R  R  |  |  |
|  |  |  |  |  |  |     R  |  |  |  |  |
|  |  |  |  |  |  |     R  |  |  |  |  |

|  |  |  |  |  |  |     W  |  |  |  |  |
|  |  |  |  |  |  |     W  W  |  |  |  |
|  |  |  |  |  |  |     W  W  |  |  W  |
|  |  |  |  |  |  |     W  W  |  W  W  |
|  |  |  |  |  |  |     W  W  W  W  W  |
12 11 10 9  8  7     6  5  4  3  2  1
  Bar:  White-0  Red-0
Rolling dice:  2 3
Red's move:  23 0
13 14 15 16 17 18    19 20 21 22 23 24
|  |  |  |  |  |  |     R  R  R  R  R  |
|  |  |  |  |  |  |     R  R  R  R  |  |
|  |  |  |  |  |  |     R  R  R  |  |  |
|  |  |  |  |  |  |     R  |  |  |  |  |
|  |  |  |  |  |  |     R  |  |  |  |  |

|  |  |  |  |  |  |     W  |  |  |  |  |
|  |  |  |  |  |  |     W  W  |  |  |  |
|  |  |  |  |  |  |     W  W  |  |  W  |
|  |  |  |  |  |  |     W  W  |  W  W  |
|  |  |  |  |  |  |     W  W  W  W  W  |
12 11 10 9  8  7     6  5  4  3  2  1
  Bar:  White-0  Red-0
Red's move:  20 0
Invalid move
Red's move:  21 0
Invalid move
Red's move:  22 0
13 14 15 16 17 18    19 20 21 22 23 24
|  |  |  |  |  |  |     R  R  R  R  R  |
|  |  |  |  |  |  |     R  R  R  |  |  |
|  |  |  |  |  |  |     R  R  R  |  |  |
|  |  |  |  |  |  |     R  |  |  |  |  |
|  |  |  |  |  |  |     R  |  |  |  |  |

|  |  |  |  |  |  |     W  |  |  |  |  |
|  |  |  |  |  |  |     W  W  |  |  |  |
|  |  |  |  |  |  |     W  W  |  |  W  |
|  |  |  |  |  |  |     W  W  |  W  W  |
|  |  |  |  |  |  |     W  W  W  W  W  |
12 11 10 9  8  7     6  5  4  3  2  1
  Bar:  White-0  Red-0
Rolling dice:  2 4
White's move:  2 0
13 14 15 16 17 18    19 20 21 22 23 24
|  |  |  |  |  |  |     R  R  R  R  R  |
|  |  |  |  |  |  |     R  R  R  |  |  |
|  |  |  |  |  |  |     R  R  R  |  |  |
|  |  |  |  |  |  |     R  |  |  |  |  |
|  |  |  |  |  |  |     R  |  |  |  |  |

|  |  |  |  |  |  |     W  |  |  |  |  |
|  |  |  |  |  |  |     W  W  |  |  |  |
|  |  |  |  |  |  |     W  W  |  |  |  |
|  |  |  |  |  |  |     W  W  |  W  W  |
|  |  |  |  |  |  |     W  W  W  W  W  |
12 11 10 9  8  7     6  5  4  3  2  1
  Bar:  White-0  Red-0
White's move:  2 0
13 14 15 16 17 18    19 20 21 22 23 24
|  |  |  |  |  |  |     R  R  R  R  R  |
|  |  |  |  |  |  |     R  R  R  |  |  |
|  |  |  |  |  |  |     R  R  R  |  |  |
|  |  |  |  |  |  |     R  |  |  |  |  |
|  |  |  |  |  |  |     R  |  |  |  |  |

|  |  |  |  |  |  |     W  |  |  |  |  |
|  |  |  |  |  |  |     W  W  |  |  |  |
|  |  |  |  |  |  |     W  W  |  |  |  |
|  |  |  |  |  |  |     W  W  |  W  |  |
|  |  |  |  |  |  |     W  W  W  W  W  |
12 11 10 9  8  7     6  5  4  3  2  1
```

```
  Bar:  White-0  Red-0
Rolling dice:  1 2
Red's move:
```

6 Termination

In this project, program termination occurs explicitly. Specifically, at any time if a player enters "0 0" as a move in response to a move prompt, your program should terminate. This is the only way your program should exit. In particular, your program does not need to detect the end-of-game condition when a player bears off all of his/her checkers.

7 Unsupported Moves and Rules

In this project, you only need to support the moves described in Sections 3–5. All other moves and rules in backgammon do not need to be supported. For example, you do not need to support the additional moves granted to a player when he/she rolls a double (*i.e.*, both dice take on the same value). Also, your program does not need to detect a player's inability to move because no checkers have a valid move. (In a normal backgammon game, the affected player would skip his/her turn). This means that it is possible for your program to become "stuck" and unable to make forward progress. (In this case, the only option is to issue the termination move, "0 0").

8 Functional Decomposition

As discussed in class, functional decomposition is crucial for building large programs. It maximizes code understanding, incremental testing and debugging, as well as reuse of your code. For this project, **your code is required to implement the functions in Section 8.1**. The discussion in Section 8.1 not only describes what each required function does, it also specifies the arguments and return values for each required function. **Your code must implement these interfaces exactly as described.**

8.1 Required Functions

There are 7 required functions. These functions are described below, and are listed in the "backgammon.h" header file provided on ELMs. In the description that follows, references are made to constants. These constants are in all uppercase letters, and are also defined in the backgammon.h file. While you must implement these required functions, you are encouraged to create more functions since some of the required functions can themselves be fairly complex, and can benefit from further decomposition.

1. `void print_board()`

This function prints the board along with the current position of all the checkers on the board. The output should match the format in the examples from this handout. You should have already implemented this function as part of homework 1, and can use your code for this function.

2. `int check_move(int mover, int from, int to, int die)`

This function takes 4 input arguments. The first argument, "mover," specifies the player (WHITE or RED) making the move. The next two arguments, "from" and "to", specify the two numbers in the move. And the last argument, "die" specifies one of the rolled die values for the player's turn. This function determines if the specified move, "from, to," is valid for the specified player, "mover," given the die value, "die." If the move is valid, the function returns VALID_MOVE. If the move is invalid, the function returns INVALID_MOVE. (The check_move function should call the next three functions to check the individual move types). Note, check_move only performs a check against one die value; you must call check_move multiple times to check against all desired die values.

3. `int check_basic_move(int mover, int from, int to, int die)`

This function takes the same 4 input arguments as check_move. It determines if the specified move, "from, to," is a valid basic move (as described in Section 3) for the specified player, "mover," given the die value, "die." If the move is a valid basic move, the function returns VALID_MOVE. If the move is not a valid basic move, the function returns INVALID_MOVE. (The check_basic_move function should call the check_die and check_board_bounds functions described below).

4. `int check_entry_move(int mover, int from, int to, int die)`

This function takes the same 4 input arguments as check_move. It determines if the specified move, "from, to," is a valid entry move (as described in Section 4) for the specified player, "mover," given the die value, "die." If the move is a valid entry move, the function returns VALID_MOVE. If the move is not a valid entry move, the function returns INVALID_MOVE. (The check_entry_move function should call the check_die and check_board_bounds functions described below).

5. `int check_bear_off_move(int mover, int from, int to, int die)`

This function takes the same 4 input arguments as check_move. It determines if the specified move, "from, to," is a valid bearing off move (as described in Section 5) for the specified player, "mover," given the die value, "die." If the move is a valid bearing off move, the function returns VALID_MOVE. If the move is not a valid bearing off move, the function returns INVALID_MOVE. (The check_bear_off_move function should call the check_board_bounds function described below).

6. `int check_die(int mover, int from, int to, int die)`

This function takes the same 4 input arguments as check_move. This function checks validity conditions #2 and #3 described in Section 3.2. If the two conditions are met, it returns VALID_MOVE. If the two conditions are not met, it returns INVALID_MOVE.

7. `int check_board_bounds(int index)`

This function takes 1 input argument, "index," which is a coordinate on the backgammon board. This function checks validity condition #1 described in Section 3.2. If the condition is met, it returns VALID_MOVE. If the condition is not met, it returns INVALID_MOVE.