ECE 209
Fall 2023

Program 1: `trouble`

Due Friday, Oct 27 @ 11:59pm

This programming assignment must be completed individually. Do not share your code with or receive code from any other student. Evidence of copying or other unauthorized collaboration will be investigated as a potential academic integrity violation. **The minimum penalty for cheating on a programming assignment is a grade of 0 on the assignment.** If you are tempted to copy because you're running late, or don't know what you're doing, you will be better off missing the assignment and taking a zero. Providing your code to someone is cheating, just as much as copying someone else's work.

DO NOT copy code from the Internet, or use programs found online or in textbooks as a "starting point" for your code. Your job is to write this program on your own. **You may use GitHub Copilot** to assist with code development, but you are responsible for the code that you submit. Evidence of copying code from any source will be investigated as a potential academic integrity violation.

This program implements a version of the board game ***Trouble***. Players advance their pieces around a track, with the potential of removing other players' pieces, and the first player to get all of their pieces to the finish line wins the game. Hopefully, you've played this before, or one of several similar games (Sorry!, Aggravation, Parcheesi), so that the rules will be familiar. The rules are described below, but a full set of rules can be found here: https://www.ultraboardgames.com/trouble/game-rules.php.

The learning objectives of the program are:

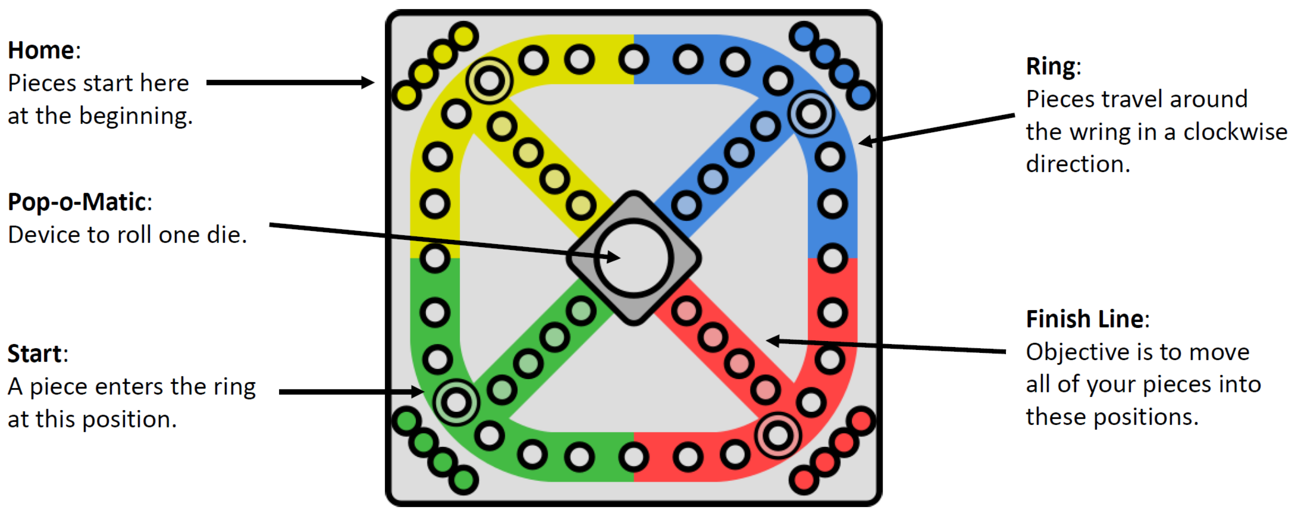
- Implement functions to complete a C program.
- Use global variables to communicate among functions.
- Use one-dimensional and two-dimensional arrays, and simple string operations.

Background

Trouble is played with 2-4 players using a board shown in the figure[1] on the next page. Each player is associated with a color. Each player's pieces begin in its Home area. When allowed, a piece enters the Ring (at the designated starting position) and travels around the ring in a clockwise direction. When it has completed the ring, it moves into its Finish Line. The first player to get all of its pieces into its Finish Line wins the game.

Pieces are advanced using a roll of a single six-sided die. The Trouble board uses a special device, called the Pop-o-Matic to roll the die. It is a clear plastic dome in the middle of the board, with the die inside. The player pushes down on the dome, which causes the die to "pop." This prevents the die from getting lost.

[1] Source: https://en.wikipedia.org/wiki/Trouble_(board_game).


Home:
Pieces start here at the beginning.
Pop-o-Matic:
Device to roll one die.
Start:
A piece enters the ring at this position.
Ring:
Pieces travel around the wring in a clockwise direction.
Finish Line:
Objective is to move all of your pieces into these positions.

Here are the rules for moving the pieces:

- Each position can only be held by a single piece. A piece can jump over occupied spaces, but it cannot land on a position that is occupied by a piece of the same color. If the piece lands on a position occupied by a different color, that other player's piece is removed from the Ring and placed back in its Home area.
- A roll of 6 is required to move a piece from Home to the Start position. If the Start position is occupied by a piece of the same color, the new piece cannot enter. If the Start position is occupied by a piece of a different color, that piece is removed and sent Home.
- If no moves are available for a given roll, the player's turn ends.
- Any roll of 6 allows the player to roll again.
- A move into the Finish Line requires an exact roll. In other words, the number of positions moved must exactly match the roll.
- A piece may only move around the Ring once. Once on the Ring, it cannot land on or pass its Start position again.

Data Representation

One of the most important aspects of designing software is deciding how information about the problem being solved will be represented by data that can be manipulated by the program. A good data representation will allow you to efficiently perform operations that match the behavior of the information.

In this case, the information is the state of the board during game play, namely the location of the pieces on the board. There is no score to track during the play of the game, but we do need to be able to detect when the game has been won. We also need be able to model the behavior of a random roll of a single die.

For this program, I am specifying the data representation, and you must follow it exactly. Having a common representation allows us to create code that will test the behavior of your program. You might have other ideas for data representations, and I encourage you to experiment with them on your own; your idea might be better than this one! But the code you submit for grading must conform to the representation described here.

Colors

There are four colors involved in the game. We will arbitrarily assign 0 to Red, and will then go clockwise around the board: Red = 0, Green = 1, Yellow = 2, Blue = 3.

Positions on the Board

First, we need a way to identify all of the locations on the board. Internally, we will use an integer for this, because it's easy to manipulate. When we communicate with users, however, we will use a string representation, because that is more suitable.

The different locations in the Home area are all the same and do not need to be distinguished. Also, a piece will only be in its own color's Home area, so there's no need to represent color information. We use -1 (negative one) to represent any location in the Home area. For the string representation, we use "H".

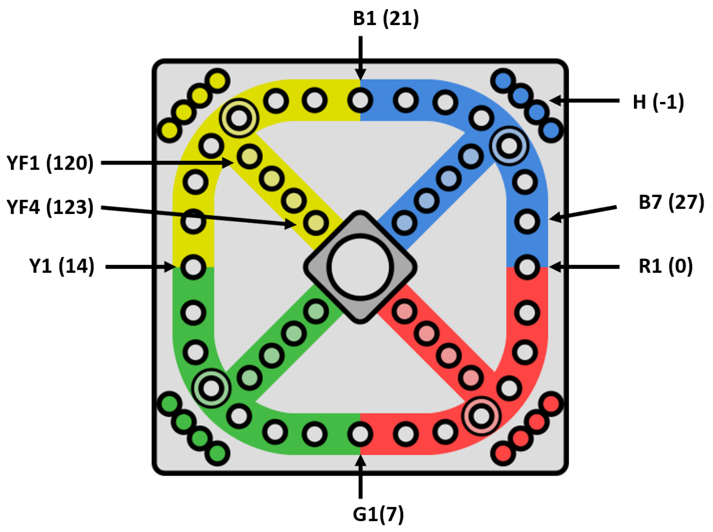
There are 28 locations on the Ring, divided into four color areas. The colors aren't very important for moving around the Ring, but it's helpful to the user. We will use integer 0 through 27 to represent the 28 locations. Location 0 is the beginning of the Red region and numbers increase in clockwise order. Location 27 is the last location in the Blue region.

For the string representation, for communicating with users, we use the color of the region and the location within that region. Because humans are not used to starting counting with zero, the positions within each region will be labeled as 1 through 7. The string uses the first character of the color and the number. For example, location 0 is "R1" and location 27 is "B7".

NOTE: Given a location integer `x`, its color is `x/7`, and its number is `x%7+1`.

The start position for a given color is position 5 in its region: R5 (4), G5 (11), B5 (18), Y5 (25).

Locations in a player's Finish Line area are different than those on the Ring, and they have a different numbering scheme: `100 + 10*color + location`. Therefore, Red's locations are 100, 101, 102, 103; Green's locations are 110, 111, 112, 113, and so on. As a string, they are represented by the color, followed by "F," followed by the position number (starting with 1), as in "RH1", "RH2", "RH3", and "RH4".



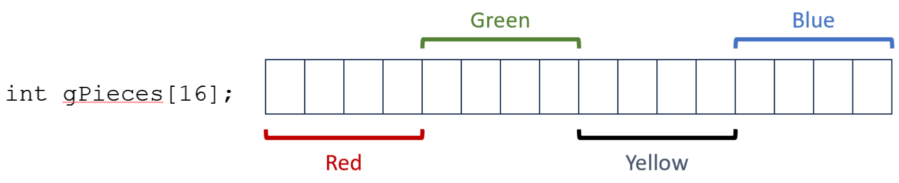
Location of Each Piece

Now that we know how to represent each position on the board, we need to track the position of each piece. Each player (color) has four pieces. We assign each piece a unique integer identifier: 0 through 15. We assign the numbers in the order of colors: Red = 0, 1, 2, 3; Green = 4, 5, 6, 7; Yellow = 8, 9, 10, 11; Blue = 12, 13, 14, 15.

NOTE: The color of piece `x` can be computed by `x/4`.

State of the Game

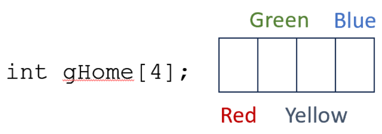
To represent the state of the game at any time, we just need to indicate the position of each piece. For this purpose, we use a *global array* of integers. The index to the array is the piece's identifier, and each element contains the position of that piece.



```
int gPieces[16];
```

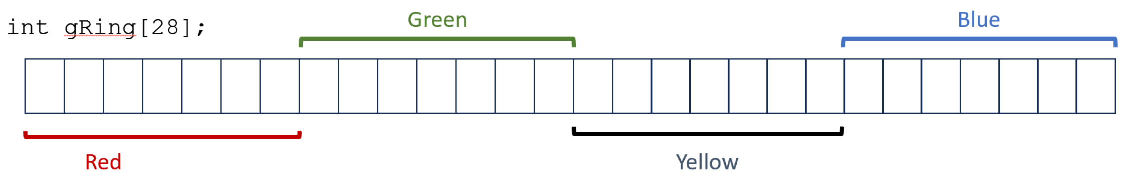
This would be sufficient, because it completely captures the state of the game. However, it is also convenient to represent the board itself, and the pieces that occupy the various spaces. This makes it easy to "look at" a position to see whether it contains a piece, the color of that piece, etc.

For the Home areas, we don't need to track individual pieces, because all Home positions for a given color are the same. Therefore, we just use a global array that contains the number of pieces in the home area for each color. The index is the color, and the element contains an integer from 0 to 4.



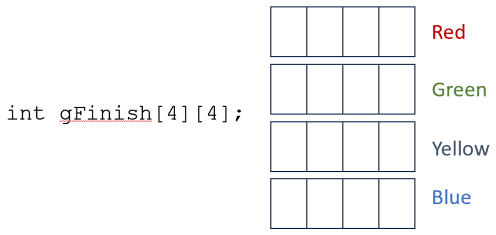
```
int gHome[4];
```

For Ring locations, we use a global array of integers. The index is the position number, and the element is the ID of the piece in that position, or -1 if the position is empty.



```
int gRing[28];
```

For Finish Line positions, we need an array of four locations per color. (Since the pieces can't cross from one color's Finish Line to another, there's no reason to make them a single array.) For convenience, we use a global two-dimensional array of integers. The first index is the color and the second is the position. As with the Ring, the data in the array specifies which piece occupies that location, or -1 if the location is empty.



```
int gFinish[4][4];
```

Why global variables? Didn't I say that global variables are evil, and you should never use them? No. I said you should avoid them, and you should have a very good reason for using them. It makes sense in this case: (1) There is only one board and only one game being played, so there would be no reason to have multiple versions of the arrays. (2) Almost all of the functions in the program will need to read and/or write this information, so it's simpler to make them global.

Program Specification

You are given a template file that already contains the main function and declarations of the global variables described in the previous section. There are also some functions provided for generating pseudorandom numbers for the die rolls.

The main function implements the user interface for the game.

1. The user enters an integer that will be used as the "seed" for the generation of pseudorandom numbers. This is to allow you to rerun the same game over and over, with the same die rolls, for testing purposes.
2. The user enters the number of players: 2, 3, or 4. When there are two players, it will be Red and Yellow (on opposite sides of the board). When three players, it will be Red, Green, and Yellow.
3. The **initializeBoard** function is called, which sets up the board for the beginning of play. (You are responsible for writing this function.) Assertions are provided to test the arrays for proper setup. This will hopefully make sure you do it correctly, and it also reminds you how to add assertions to test your code during development.
4. The program then enters a loop that allows each player to take their turn. For each loop iteration, the **playerTurn** function is called with the player number. (This is also a function that you must write.) The function returns 1 if the player wins the game on this turn, and 0 otherwise.
5. When the game has been won, the loop exits and the program ends.

You may not create any additional global variables, and you may not change the names or types of the global variables provided in the template file.

Your program must implement the following three functions. You may choose to implement other functions, but these three functions must be implemented exactly as specified. ***Please write the definitions of these functions (and the declarations/definitions of any other functions that you choose to implement) in the middle section of the program file, right after the main function.*** This will make it easier for the TAs to find your code, grade your code, and help you debug when you need it. This section of code is clearly marked by comments. FOLLOW INSTRUCTIONS!!!

Here are the declarations of the six functions that you must implement. (These declarations are already in the file. Don't write them again.) For any other functions you choose to write, add their declarations into the provided area, and write their definitions in the space below the main function.

```
void initializeBoard();
```

Initialize the board to begin play. Every piece is in the Home area. The Ring and the Finish Line locations are all empty.

```
void printStatus();
```

Print the location of each color's pieces, using the string representation described above. Print one color per line, followed by the position for the pieces associated with that color. When printing the color, use %6s to specify a fixed width for this string, so that everything gets lined up as in the examples below.

Here's what the status should look like at the beginning of the game, when no player's pieces have moved out of Home.

```
   Red: H,H,H,H
 Green: H,H,H,H
Yellow: H,H,H,H
  Blue: H,H,H,H
```

Here's an example where some pieces are still in Home and others are at various places along the Ring and in the Finish Line.

```
   Red: R1,H,RF3,Y3
 Green: H,G4,B6,H
Yellow: R2,G5,Y5,B3
  Blue: H,H,B5,R5
```

This function will be called at the beginning of a player's turn, so that they can decide what move to make based on their roll.

`int printMoves(int player, int roll);`

During a player's turn, they will roll a number from 1 to 6. This function prints all of the legal moves that can be made by the player, based on that roll. This depends on (a) the locations of each player's pieces, and (b) the location after moving the rolled number of spaces. Remember the following rules:

- Piece can only leave Home on a roll of 6. If the starting location is occupied by a piece of the same color, the piece cannot leave.
- Piece cannot land on a location that contains a piece of the same color.
- Moving into the Finish Line requires an exact roll.

All legal moves must be printed using the format "X to Y" with X being the starting location and Y being the ending location. If there are no legal moves, the function does not print anything.

The function returns the number of legal moves, an integer between 0 and 4.

`int movePiece(int player, int from, int roll);`

This function moves a piece on the board. The piece is specified by the player and the starting position. The number of spaces to move is specified by the roll. If the move is legal, the program data structures are updated and the function returns 1. If the move is illegal, the function returns 0.

NOTE: This function must not print anything!!!!

`int checkWin(int player);`

This function checks whether the specified player has won the game. The function returns 1if the player has won, and 0 otherwise.

NOTE: This function must not print anything!!!!

`int playerTurn(int player);`

This function does everything necessary to carry out a player's turn. There are a lot of steps, and it involves both printing and reading information from the user. Below are the steps required. See the appendix for examples of moves.

1. Print a message that announces the player's turn. This looks like the following. (Note that all examples shown here are indented for easy reading, but they must start at the left margin of the console output. Other than that, spacing should be exactly the same as in the examples. Also, unless otherwise specified, there must be a linefeed at the end of each line.)

   ```
   ---- Red's turn
   ```

   Replace "Red" with the appropriate color of the player. Hint: Use the `gPlayerNames` array to print the string that matches the player's color.
2. Print the board status, using the **printStatus** function.
3. Call the **pop_o_matic** function to roll the die. Print the following message, replacing X with the number rolled.

   ```
   You rolled X.
   ```
4. Based on the roll, call **printMoves** to print all legal moves available to the player.
5. If there is no legal move, print the following message and skip to step 9.

   ```
   You have no legal moves.
   ```
6. Ask the user to enter the current position of the piece they wish to move. Print the following prompt, and then read a string from the user. You may assume that the user will enter a valid location string (but it be could be any location, not just the ones printed in step 4.

   ```
   Enter position of piece to move:
   ```

   NOTE: There is no way for the player to decline to make a move. If there is at least one legal move, the player must choose one. This may be against the official rules of the game.

   NEW in V2: If the user enters "Q", return 1, which will end the game.
7. Given the position (step 6) and the roll (step 3), try to move the specified piece. If it's not a legal move, print the following message and go back to step 6.

   ```
   Illegal move, try again.
   ```
8. Check whether the player has won the game. If so, return 1. If not, continue to step 14.
9. If the roll was 6, print the following message and return to step 7.

   ```
   You rolled 6, so you get to roll again.
   ```
10. Return 0, which indicates that the turn is over and the player has not won the game.

Other Functions

As mentioned earlier, you are encouraged to create your own functions to modularize your code and help with development. For example, you may want functions to convert between the integer and string representations of a board location. Declare each function in the area provided above the main function, and then define each function in the designated area after the main function.

Developing and Submitting the Program

You will submit this code as a zyBook lab, but you will develop the program outside of the zyBook. It is expected that you will use CLion, but you are free to use whatever development environment that you want.

1. In CLion, create a new project. Specify a C Executable, and use the C17 standard. Name the project whatever you like, but I recommend something meaningful like "prog2" or "trouble". This will create a directory in the place where your CLion projects are kept.
2. Download the template file from zyBook 15.1 and *overwrite* the **main.c** file in your CLion project. (a) Download the ZIP file. (b) Extract main.c from the ZIP file. (c) Copy it to your project director and overwrite the main.c file that's there.
3. Now use CLion to complete the program, using the editor, compiler, and debugger to meet the program specifications. NOTE: You will need to provide definitions of the required functions before the code will compile. You can use "stub" versions of the functions that don't do anything except return a value of the expected type.
4. When you are ready, upload your **main.c** file to the zyBook assignment and submit. This will run the tests.

If some tests fail, go back to Step 2 and work on your program some more. The input for the failed tests will give you some idea of what's not correct, but use the debugger to figure out what's happening with your implementation.

Several iterations of Steps 2 and 3 might be necessary. In fact, it's a reasonable strategy to write a program that only passes the first test (or some tests), then improve it to pass the next test, etc. (This even has a fancy name: test-driven development.)

There is no limit to the number of times you can submit. Each submission will overwrite the earlier ones, so make sure that your new code does not break tests that passed earlier.

Hints and Suggestions

- Don't overcomplicate the program. Do not use anything that we have not covered in class. (In particular, do not use a struct.)
- Work incrementally. Get one part of the code working, and then move to the next. You can define **playerTurn** to always return 1, and keep adding code to call the other functions until you are ready to test the full program.
- I recommend implementing and testing the functions in the following order.
    - **initializeBoard**
    - **printStatus**
    - **printMoves**
    - **movePiece**
    - **checkWin**
    - **playerTurn**

You can alter the **main** function to test your functions. For example, after the board is initialized, call a test function (that you write) to set up the board in a certain way and call one or more of the other functions.

These changes must be removed in your final submission, because the **playerTurn** tests require the provided **main** function to be run and to execute according to this specification. (Don't leave debug print statements in your code!)

- When developing and debugging, use the same seed value each time, so that you get the same set of dice rolls. This allows you to run the same code repeatedly, to debug and figure out what's happening. When it appears to work correctly, change the seed to get a new set of rolls.
- For general questions or clarifications, use Piazza, so that other students can see your question and the answer. For code-specific questions, post a private message to Piazza and attach your code as a file. (Do not copy and paste!)

Administrative Info

Updates or clarifications on Piazza:

Any corrections or clarifications to this program spec will be posted on Piazza. It is important that you read these postings, so that your program will match the updated specification.

What to turn in:

- Submit your **main.c** file to the zyLab assignment in *15.1 Program 1: Trouble*.
- Suggestion: Until you are ready to test the **playerTurn** function, define the function to always return 1. You can add more and more code t

Grading criteria:

15 points: Program compiles and executes. The first test will use a seed of -1, which will exit the program. This is only to test whether your program will compile and execute correctly. This requires initializeBoard and playerTurn to be defined. (They don't have to work correctly, but they must have a definition with an appropriate return.)

10 points: Proper coding style, comments, and headers. **No additional global variables.** No goto. See the Programming Assignments section on Moodle for more style guidelines. (You will not get these points if you only submit trivial code.)

8 points: **initializeBoard** works correctly.

10 points: **printStatus** works correctly. (Must be graded manually. ZyBook will report that the test failed, because the grader must look at the output to check it.)

8 points: **printMoves** works correctly. (Must be graded manually.)

10 points: **checkWin** works correctly.

14 points: **movePiece** works correctly. (Must be graded manually.)

15 points: **playerTurn** works correctly.

10 points: Complete **GitHub Copilot** survey on Moodle page.

Do not make assumptions. Write your code to pass any test consistent with this specification. We reserve the right to run your code on tests that were not provided ahead of time.

APPENDIX: Sample Run

This is not a complete game, but it hopefully represents a lot of the rules being applied. Comments in the text boxes to the right are not part of the game input/output. User input is shown here in bold and highlighted in yellow; this is only to highlight what is entered by the user.

```
Enter a random seed: 1234
Enter the number of players (2, 3, 4): 2

---- Red's turn
   Red: H,H,H,H
 Green: H,H,H,H
Yellow: H,H,H,H
  Blue: H,H,H,H
You rolled 5.
You have no legal moves.
```

Nothing happens until someone rolls a 6. Since we only have two players, alternates between Red and Yellow.

```
---- Yellow's turn
   Red: H,H,H,H
 Green: H,H,H,H
Yellow: H,H,H,H
  Blue: H,H,H,H
You rolled 2.
You have no legal moves.

---- Red's turn
   Red: H,H,H,H
 Green: H,H,H,H
Yellow: H,H,H,H
  Blue: H,H,H,H
You rolled 6.
H to R5
Enter position of piece to move: H
You rolled 6, so you get to roll again.
   Red: R5,H,H,H
 Green: H,H,H,H
Yellow: H,H,H,H
  Blue: H,H,H,H
You rolled 5.
R5 to G3
Enter position of piece to move: R5
```

Red rolls a 6. Brings a piece to the ring. Notice the change in status is printed.

```
---- Yellow's turn
   Red: G3,H,H,H
 Green: H,H,H,H
Yellow: H,H,H,H
  Blue: H,H,H,H
You rolled 3.
You have no legal moves.
```

```
---- Red's turn
   Red: G3,H,H,H
 Green: H,H,H,H
Yellow: H,H,H,H
  Blue: H,H,H,H
You rolled 1.
G3 to G4
Enter position of piece to move: G3

---- Yellow's turn
   Red: G4,H,H,H
 Green: H,H,H,H
Yellow: H,H,H,H
  Blue: H,H,H,H
You rolled 4.
You have no legal moves.

---- Red's turn
   Red: G4,H,H,H
 Green: H,H,H,H
Yellow: H,H,H,H
  Blue: H,H,H,H
You rolled 4.
G4 to Y1
Enter position of piece to move: G4

---- Yellow's turn
   Red: Y1,H,H,H
 Green: H,H,H,H
Yellow: H,H,H,H
  Blue: H,H,H,H
You rolled 5.
You have no legal moves.

---- Red's turn
   Red: Y1,H,H,H
 Green: H,H,H,H
Yellow: H,H,H,H
  Blue: H,H,H,H
You rolled 6.
H to R5
Y1 to Y7
Enter position of piece to move: H
You rolled 6, so you get to roll again.
   Red: Y1,R5,H,H
 Green: H,H,H,H
Yellow: H,H,H,H
  Blue: H,H,H,H
You rolled 1.
Y1 to Y2
R5 to R6
Enter position of piece to move: R5
```

Now Red has two players on the ring, and Yellow still has none.

```
---- Yellow's turn
   Red: Y1,R6,H,H
 Green: H,H,H,H
Yellow: H,H,H,H
  Blue: H,H,H,H
You rolled 2.
You have no legal moves.

---- Red's turn
   Red: Y1,R6,H,H
 Green: H,H,H,H
Yellow: H,H,H,H
  Blue: H,H,H,H
You rolled 3.
Y1 to Y4
R6 to G2
Enter position of piece to move: R6

---- Yellow's turn
   Red: Y1,G2,H,H
 Green: H,H,H,H
Yellow: H,H,H,H
  Blue: H,H,H,H
You rolled 6.
H to Y5
Enter position of piece to move: H
You rolled 6, so you get to roll again.
   Red: Y1,G2,H,H
 Green: H,H,H,H
Yellow: Y5,H,H,H
  Blue: H,H,H,H
You rolled 1.
Y5 to Y6
Enter position of piece to move: Y5

---- Red's turn
   Red: Y1,G2,H,H
 Green: H,H,H,H
Yellow: Y6,H,H,H
  Blue: H,H,H,H
You rolled 5.
Y1 to Y6
G2 to G7
Enter position of piece to move: Y1

---- Yellow's turn
   Red: Y6,G2,H,H
 Green: H,H,H,H
Yellow: H,H,H,H
  Blue: H,H,H,H
You rolled 4.
You have no legal moves.

---- Red's turn
```

Finally, Yellow rolls 6.

Unfortunately, Red also lands on Y6, which sends Yellow's piece back Home.

```
   Red: Y6,G2,H,H
 Green: H,H,H,H
Yellow: H,H,H,H
  Blue: H,H,H,H
You rolled 2.
Y6 to B1
G2 to G4
Enter position of piece to move: Y6

---- Yellow's turn
   Red: B1,G2,H,H
 Green: H,H,H,H
Yellow: H,H,H,H
  Blue: H,H,H,H
You rolled 1.
You have no legal moves.

---- Red's turn
   Red: B1,G2,H,H
 Green: H,H,H,H
Yellow: H,H,H,H
  Blue: H,H,H,H
You rolled 3.
B1 to B4
G2 to G5
Enter position of piece to move: B1

---- Yellow's turn
   Red: B4,G2,H,H
 Green: H,H,H,H
Yellow: H,H,H,H
  Blue: H,H,H,H
You rolled 3.
You have no legal moves.

---- Red's turn
   Red: B4,G2,H,H
 Green: H,H,H,H
Yellow: H,H,H,H
  Blue: H,H,H,H
You rolled 3.
B4 to B7
G2 to G5
Enter position of piece to move: B4

---- Yellow's turn
   Red: B7,G2,H,H
 Green: H,H,H,H
Yellow: H,H,H,H
  Blue: H,H,H,H
You rolled 5.
You have no legal moves.

---- Red's turn
```

```
   Red: B7,G2,H,H
 Green: H,H,H,H
Yellow: H,H,H,H
  Blue: H,H,H,H
You rolled 1.
B7 to R1
G2 to G3
Enter position of piece to move: B7

---- Yellow's turn
   Red: R1,G2,H,H
 Green: H,H,H,H
Yellow: H,H,H,H
  Blue: H,H,H,H
You rolled 3.
You have no legal moves.

---- Red's turn
   Red: R1,G2,H,H
 Green: H,H,H,H
Yellow: H,H,H,H
  Blue: H,H,H,H
You rolled 4.
R1 to RF1
G2 to G6
Enter position of piece to move: R1
```

Here, Red moves a player into its Finish Line.

```
---- Yellow's turn
   Red: RF1,G2,H,H
 Green: H,H,H,H
Yellow: H,H,H,H
  Blue: H,H,H,H
You rolled 2.
You have no legal moves.

---- Red's turn
   Red: RF1,G2,H,H
 Green: H,H,H,H
Yellow: H,H,H,H
  Blue: H,H,H,H
You rolled 5.
G2 to G7
Enter position of piece to move: G2

---- Yellow's turn
   Red: RF1,G7,H,H
 Green: H,H,H,H
Yellow: H,H,H,H
  Blue: H,H,H,H
You rolled 6.
H to Y5
Enter position of piece to move: H
You rolled 6, so you get to roll again.
   Red: RF1,G7,H,H
 Green: H,H,H,H
Yellow: Y5,H,H,H
  Blue: H,H,H,H
You rolled 4.
Y5 to B2
Enter position of piece to move: Y5

---- Red's turn
   Red: RF1,G7,H,H
 Green: H,H,H,H
Yellow: B2,H,H,H
  Blue: H,H,H,H
You rolled 6.
H to R5
G7 to Y6
Enter position of piece to move: H
You rolled 6, so you get to roll again.
   Red: RF1,G7,R5,H
 Green: H,H,H,H
Yellow: B2,H,H,H
  Blue: H,H,H,H
You rolled 1.
RF1 to RF2
G7 to Y1
R5 to R6
Enter position of piece to move:
```

Note: It's legal to move a piece within the Finish Line (RF1 to RF2, in this case), if the position is open.

The game goes on and on, but hopefully you get the idea...