*Exercise 1: Penalty Logic (Game.cpp)*

***Explained***

***This exercise …?***

[GameRunner.cpp]

[Game.cpp]

[Game.h]

1. [GameRunner.cpp] Code begins by creating a random generator and seeding it w/ a command line arg.
   * If there is only one cmd line arg (the default way of running the code), use the time as the seed, otherwise if a specific number is passed then use this as the seed.
2. [GameRunner.cpp] Next, a default “BasicBoard” object is created.
   * [Game.h] Default basic board object is a child of “Board” and passes up the number 12 to initializes the num\_places() array.
   * [Game.cpp] In addition to setting the num of board places, a for loop of length 50 (exclusive)(0-49) populates four different vectors with 50 questions relevant to each of the 4 categories.
3. [GameRunner.cpp] Three “BotPlayer” objects are now instantiated passing both a str and a bool.
   * [Game.cpp] BotPlayer(str, bool) is derived from the “Player” object and is using a move constructor to steal the contents of “Player” and transfer them to “BotPlayer” in order to initialize the following values with the ones passed:
     + m\_name{str},
     + m\_purse{0},
     + m\_place\_idx{0},
     + m\_is\_in\_penalty\_box{bool}
4. [GameRunner.cpp] (vector<reference\_wrapper<Player>> players;) Next a vector called “players” is created to hold **REFERENCES** to “Player” objects as opposed to “Player” objects themselves.
   * All three bot players are placed into the “players” array using .emplace\_back() instead of push\_back()
     + Still not 100% clear on this. The only difference appears to be that emplace\_back() constructs the string literal directly inside the array as opposed to making a copy and then placing that at the end of the array.
5. [GameRunner.cpp] Now a “Game” object is created passing the previously created “board” object and our “players” vector that we just filled.
   * [Game.cpp] The game object instantiates:
     + m\_board(board)
     + m\_players(players)
     + m\_cur\_player\_idx(-1)
6. [GameRunner.cpp] Finally, we call the “.play()” function on our “game” object and this last line of code is apparently where all the magic happens.
   * [Game.cpp] The .play() function takes no parameters and starts by calling the \_add\_players() function.
     + This function runs a loop the length of m\_players.size() which is the vector instantiated with the number of players, exclusive.
     + Within this loop a “new\_player” variable is created (of the Player object type) and is instantiated with the same information from before (name and the bool about printing messages). This player’s place on the board is set to 0 and so is their purse. This is done for all players added to the game (hence the loop).
     + If this is the first player added to the game (ie: the players position in the array == 0) we start them off in the penalty box, otherwise “set\_is\_in\_penalty\_box()” gets set to false.
     + Finally, the add\_player() method ends by printing a confirmation message to the screen saying that the player was successful added and displaying what number they are +1(if player is original #0 in the array, that plus one will make it display as player #1 as opposed to player #0)
   * [Game.cpp] Now we jump back up to the “.play()” function. The next line is a while loop that essentially keeps the constantly checks to see if anyone has won the game yet. If not, the game keeps going so long as the Boolean function “!\_is\_winner()” has a return value equal to false. We must be careful to note that the “\_is\_winner()“ function in this while loop has a “!” or a “not” sign in front of it. This basically means the inverse of what it originally means. !\_is\_winner means that this while loop will run so long as the function returns false, if it returns true the loop will end.
     + [Game.cpp] the “\_is\_winner()” function starts by instantiating a bool variable called “winner” == false.
     + It then runs through the list of players using a range-based-for-loop and checks each one to see if anyone has a purse greater than or equal to 6. If they do, then the value of “winner” becomes true, if not it stays false.
     + Finally, this function returns the value of the “winner” variable.
   * [Game.cpp] Now we jump back up to the “.play()” function. Inside the body of the aforementioned while loop we create a variable called cur\_player and instantiate it with the return value of the function, “move\_to\_next\_player()”
     + [Game.cpp] inside of “move\_to\_next\_player()”, the private Game object member variable “m\_cur\_player\_idx” is set to **the remainder of** “m\_cur\_player\_idx” incremented by 1 and then divided by “m\_players.size()”
     + Basically, the pattern of player’s turns is determined by this calculation which is heavily dependent upon how many players are in the game.
     + Essentially this just returns an array index for which player should go next. That player is then returned as the “cur\_player()”. The pattern for a game w/ only 3 players is show below:

Graphical user interface, application, table, Word, Teams

Description automatically generated

* + [Game.cpp] Now we jump back up to the “.play()” function and next we determine a players roll by setting the value of the “roll” variable to the results of “cur\_player\_.roll()”.
    - [Game.cpp] every Player object has a pure virtual function called “.roll()” which must be defined by everyone who inherits from the Player class (including Bot players). From the looks of it, normal “Player” rolls and “Bot Player” rolls are probably defined differently, but we will know for sure soon as we proceed.
    - The “cur\_player” object on which this “.roll()” function is called is derived from the normal “Player” object which has the “.roll()” function defined as such:
      * Bot Player .roll(): { return rand() % 5 + 1; }
        + Returns a random value between 1-5 inclusive.
  + [Game.cpp] Again we jump back up to the “.play()” function. Now we know that the “Game” object’s private member variable “roll” (**not to be confused with the Player Object virtual function: .roll()**) will be set to a random int\_32 value between 1-5 inclusive.
  + We then print out the results of the two variables we just calculated to the user on screen displaying both the current player’s name and the number they rolled:
    - auto cur\_player = \_move\_to\_next\_player();
    - auto roll = cur\_player.roll()
  + Finally the “\_handle\_player\_turn(m\_board, cur\_player, roll)” function is called, passing it the type of board we have, the current player object, and the value of their current roll.
    - [Game.cpp] **Looks like we’ve finally reached the meat and potatoes of our Game code!** Inside the “\_handle\_player\_turn(m\_board, cur\_player, roll)” function we first create a constant bool titled “is\_getting\_out\_of\_penalty\_box” and set to true if the players roll was an odd number and false if it was even.
      * Let’s also recall early that every player has a separate bool value that determines if they are currently in the penalty box or not. In the beginning, only player one starts off in the penalty box. Our top “if statement” checks to see if the current player is in the penalty box or not**. If not we simply set the current players new place on the board using the result of “m\_board.get\_new\_place\_idx(currentPlayersPlace, roll)”**.
        + [Game.cpp]“m\_board.get\_new\_place\_idx(currentPlayersPlace, roll)” is calculated by taking the players current board index, adding it to their current roll, then dividing it by the total number of places on the board (12) and returning the remainder:

Graphical user interface, text, application, email

Description automatically generated

Background pattern

Description automatically generated

* + - * + Contrarily, if the current player **IS** in the penalty box, then we proceed to another if statement:
        + This next if statement checks: **if the player’s roll was an even number (meaning “is\_getting\_out\_of\_penalty\_box” is false aka “!is\_getting\_out\_of\_penalty\_box”)** If so, we print on screen that the user will not be getting out this turn.
        + Also, we check to see if the current players response to the question is correct. There is a bug here because the we are checking the ]“.response\_is\_correct()” which already is returning a true or false value of whether the question was answered correctly, but then we inverse it by using the not operator. Removing the not operator would fix this bug. If the player answered incorrectly we say that the player gets sent to the penalty box (but they never actually get sent. This is another bug!) :

[Game.cpp]“.response\_is\_correct()” method determines if a player answered the question right by choosing a random number between 0-9 (exclusive). If the resulting number is not equal to the number 7 then we return true (meaning the players response to the question was correct), otherwise we send back false { return rand() % 9 != 7; }.

* + - * + Also, if the player is currently in the penalty box, but then is not set to get out of the penalty box, then why are we sending them back to the penalty box if they get an answer wrong? This block needs to be re-evaluated too.
    - [Game.cpp] “\_handle\_player\_turn” - Now we’ve gotten through all the “if statements” and are on to the questions. A question object is created and initialized using the “m\_board.get\_question\_for\_place\_idx(cur\_player.get\_place\_idx());” function
      * [Game.cpp] “m\_board.get\_question\_for\_place\_idx()” creates a category object using by calling “get\_category\_for\_place\_idx()” which simply uses the players place on the board to choose a question category.
      * Now that we have a category, “m\_board.get\_question\_for\_place\_idx()” uses an if statement to determine which category was selected and then proceeds to grab a question from the front of the vector of that particular category to display to the user.
      * There is also an exception handler in here that will throw an error if the size of the questions array is 0, meaning that all questions in this category have already been asked (this shouldn’t happen with only 3 players).
    - [Game.cpp] “\_handle\_player\_turn” – Now we’ve got a question loaded up and we display this new information on screen. We tell the current players name, their new location, the category they landed on, and their new question.
    - Next we determine if their response to the question was correct using the same formula from “.response\_is\_correct()”
      * return rand() % 9 != 7;
    - Now, using an “if statement” if the response is correct we incremnent the player’s purse.
    - Next we have what initially seems like a bug. We are checking again to see if player is in the penalty box and displaying either “answer is correct or corrent” depending on if they are. We also display the number of coins the user now has.
    - Conversely, If the player gets the answer wrong we set their “set\_is\_in\_penalty\_box” value to true and then display on screen that they were wrong and are going to jail!