Project 1

SORRY!

Course: CIS 17-C

April 24, 2021

Ezequiel Hernandez Jr.

1. Introduction

Sorry is a four-player board game consisting of 12 pawns and 45 cards. The objective of the game is to move all your pawns around the board and into your home space. I chose to make this game because it was my favorite board game to play growing up

1. Approach to Development

* **Concepts:**

My first step to developing this project was to look over chapters 6, 9, and 11 of Nicolai M. Josuttis ’ book, “The C++ Standard Library: A Tutorial and Reference.” Reading the book gave me an idea of how I can use the different data structure in my program. I thought that the multimap and the linked list would be especially useful when making a board game due to their ability to make elements easily accessible.

* **Version Control :**

I have a total of 6 or 7 version of the game. In my first version I outlined the private data types and functions I would need. I have 4 class 3 of them are aggregations of my Game class. I first developed my Deck class when we were assigned to make a program using any container from the STL. When making my second version. When developing versions 3 and 4 I realized that I did not need a players class in my game, since the Players only have a color and a choice, both can be represented by two variables.

1. Game Rules

Every player starts off with three pawns each in their start area. There are four start areas for each player, each with a different color(red, green, blue, yellow). On their turn the player must pick a card from the deck and follow the instructions on the card. Any forward moving card can move a pawn out of its start area. Pawns move clockwise on the board.

* **Occupied Space:**

When moving pawns forward, players must jump any pawns they pass and count that jumped pawn as one space. If a player lands on an opponent’s pawn, the opponent’s pawn is sent back to its start area and the player takes the space. Only one of each players pawns may occupy a space, except for the start area and home. If a player’s only possible move makes them land on space occupied by one of their pawns they forfeit their turn

* **Board Rules:**

**Slide – Only slide when a player lands exactly on the slide triangle and if that triangle shows the players color. When player slide all pawns that are passed are sent back to the start area including their own. (Unfortunately I was unable to implement this feature, I had planned to use a find algorithm to find the spaces in the range of the slide, but I spent a lot more time debugging than I though I would have.)**

**Safety Zone – Players cannot enter other players safety zone. Pawns can move backwards out of their safety zone. Players cannot move backwards into their safety zone. Player cannot move their pawn past their safety zone entry, except if moving backwards out of their safety zone.**

**Home – Once home pawns are not to be moved. The game is won when a player brings home all of their pawns.**

* **Card Rules:**

**#1-#5, #8, #12 – Move your pawn forward the number of spaces shown on the card.**

**#7- Move one of your pawns forward 7 spaces or split the move between two of your pawns.**

**#10- Move one of your pawns forward ten spaces or move your pawn backwards one space**

**#11- Move your Pawn forward 11 spaces or switch one of your pawns with an opponent’s. Can only switch with pawns on the open track. If the switch lands you on a triangle and the color is valid then slide.**

**SORRY! – Move a pawn from your start to switch places with an opponents pawn, the opponents pawn is sent back to their start area. Or move one of your pawns forward four spaces. If there is no pawn on your start you forfeit your turn**

1. **Description of Code**

Pseudo code:

*Initialize game with number of players*

*GameBoard fills start areas depending # of players*

*LOOP START-loops until winning condition is met*

*refreshBoard*

*PrintBoard*

*LOOP TURNS -loops until queue is empty*

*Player draws card*

*If move card, player choose pawn to move*

*Else if effect card choose effect and target*

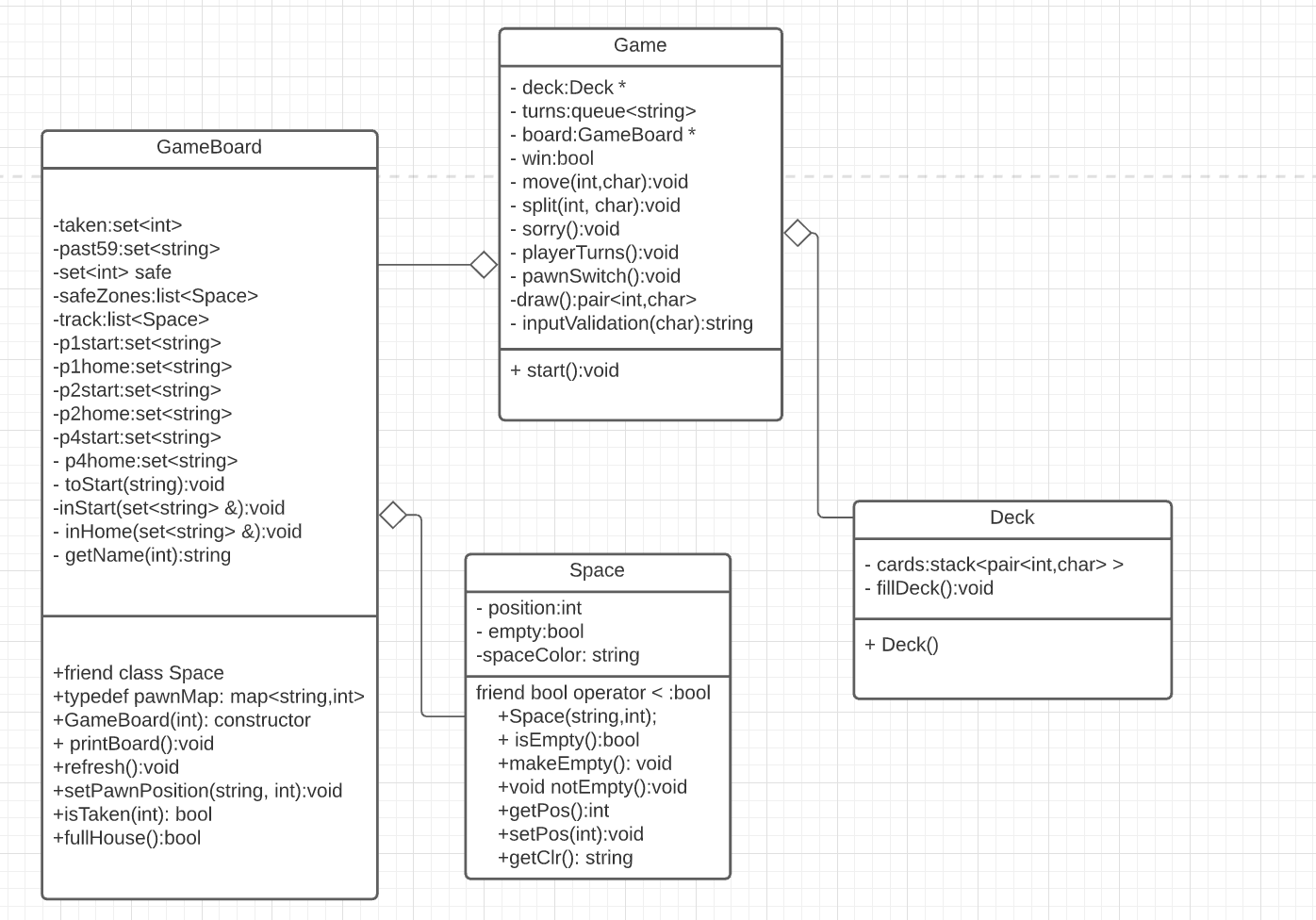
*Pop player from queue*

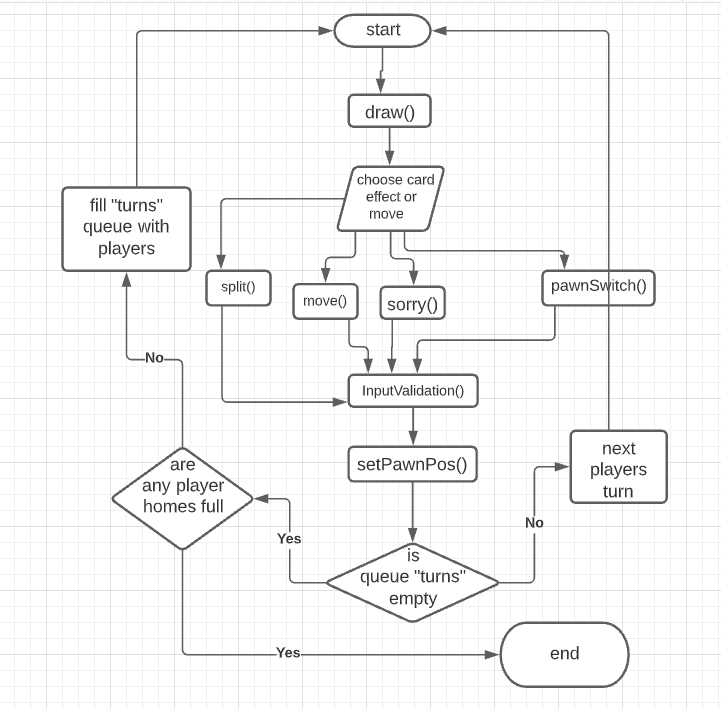
*If queue is empty*

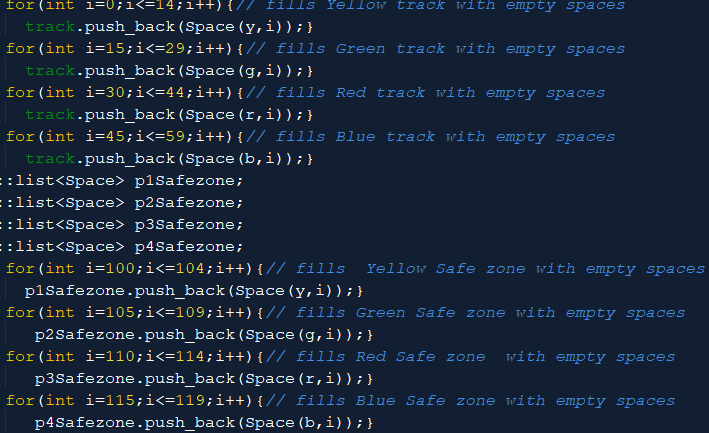
*END LOOP TURNS*

*Fill queue with players*

*END LOOP START*

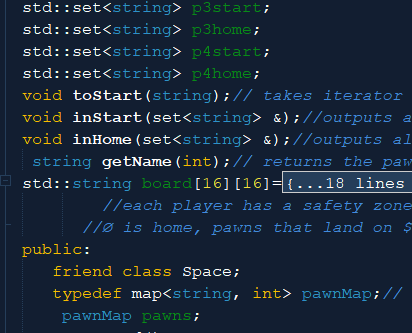
****

****

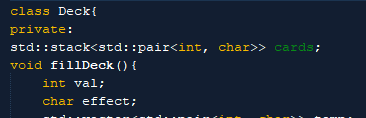
1. **Container Classes**
2. **Sequences**

**I used a list of Spaces to represent the gameboard. The main list “track” represents the outer track where all pawns must traverse. I also made a track for each player to represent the safety zone. After every action the lists are traversed with iterators to decide what information will be displayed for each space. The list “track” holds space objects, the space objects all have a unique position number 0-59. All spaces also have a bool “empty” that is use when determining what string should be displayed for that corresponding space on the board. All spaces also have a string color that was meant to be used for the slide functionality.**

1. **Associative Containers**

****

I used a map of string and ints to represent each pawn by name and position on the board. Using the map as an associative array was very useful for quick element access. The sets were used to represent home and star positions for each player.

1. Container Adaptors 

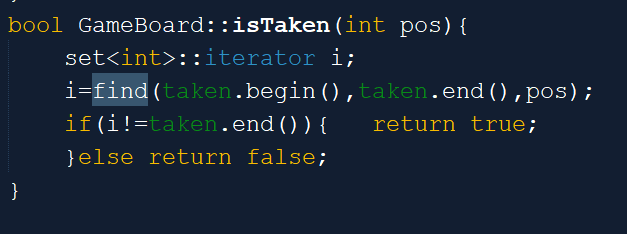
I used a stack and a queue for container adaptors. I used the stack to represent my deck because, a deck is a stack of cards. I then used a queue for the players turns because the players can be popped out and pushed back in the order of their turn.

6. Iterators

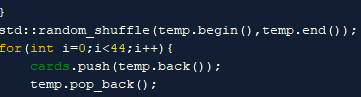
1. Input Iterator- I used input iterators to read the values of the sets of strings that were int the players start and home sets.
2. Forward Iterator- Can be used to read and write data. I used forward iterators when calling member functions of the elements in the list
3. Random-Access Iterator- I used random access iterators when using a map as an associative array. I also used them when using algorithms that require iterators to point to the beginning and the end of a list.

7.Algortihms

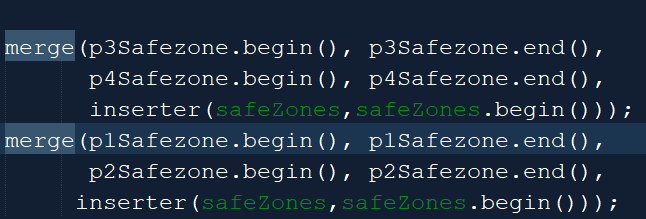
1. **Non-mutating algorithms:** I used the find algorithm in my GameBoard class to find if a certain position had been taken.



1. **Mutating algorithms:** I used random\_shuffle in my Deck class to shuffle the stack of cards. I first had to store the cards in a vector since the algorithm requires random access iterators.

****

1. **Organization:** I used the merge sort to in my GameBoard class to merge together all of the player’s safety zones. This made it easier for me to retrieve information about the spaces by only iterating through one list instead of four.

****