Project 2

<Sorry! Game>

CIS17A
Eduardo Brito
5/28/25

Introduction

This project is based on the board game “Sorry!” which is a 2+ player game where each player has pawns that they need to try to get to the home base. There are cards 1-12 that can be pulled and in order to leave the home base a 1 or 2 needs to be pulled, if not pulled within your turn, you are skipped. Then a “Sorry” card allows someone with their pawn still at home base to swap places with an opponent pawn on the board. Once you’re near the home base you must get the exact number to home in order to get your pawn at home base. So if you are three spaces away, and if you continuously get another number then your turn is skipped or must move another pawn if it’s on board. I want to recreate this board game as a project because this was one of the games me and my siblings used to play when we were kids. I tried to get it as close as the in person game as possible using the methods from the chapters we’ve covered. Plus adding a few more features to make it more user friendly.

Summary

The final version of this Sorry! board game program spans over 1101 total lines of code, broken down into:

- Main Program: 712 lines
- Game_Classes.hpp: 194 lines
- Counter.hpp: 30 lines
- Total Comments: 165 additional lines across all files
  PawnV4, PlayerV4, Bot, CardDeck
- 1 Template Class: Counter<T>
- 1 Enumerator: PosStat for pawn position states (START, HOME)
- OOP Concepts:
  Class inheritance (Bot inherits from PlayerV4)
  Polymorphism with virtual and overridden functions in Class Bot
  Operator overloading in PlayerV4 class
  Aggregation (players own pawns)
  Abstract classes (PlayerV4 has virtual methods like isBot)
- Advanced C++ Features:
  Templated Counter<T> class for tracking moves
  Exception handling for file operations in saveGame and loadGame
  Static members and UML-style layout in PlayerV4 class.
- Additional Features:
  Save/load functionality with file I/O
  Undo with backup() and restore() in PawnV4

Bot player modes with two difficulty choices (moveBot, moveBotHard)
Total game move counter tracked with the Counter template

Development

Version 1 - Struct-Based Foundation

- Goal: Convert the original raw logic into a structured program using C++ structs.
- Changes:
  Created PawnV1 and PlayerV1 structs to organize player and pawn data.
  Simplified the game logic using functions operating on these structs.
  The game was a little functional but lacked any object-oriented design.
  No bot logic, no undo, no classes, this was a the starting version.

Version 2 - Transition to Classes

- Goal: Start using classes within the main program and further update the classes
- Changes:
  Replaced PawnV1 and PlayerV1 structs with PawnV2 and PlayerV2 classes.
  Added private data members and public member functions.
  Used accessors (get/set) for behavior.
  Shrink the code using headers (Game_Classes_V2.h) better separation.
  Still did not use bot players, polymorphism, or advanced features.

Version 3 - Inheritance and Bot Integration

- Goal: Begin using class inheritance, polymorphism
- Changes:
  Introduced a new derived class Bot inheriting from PlayerV3.
  Implemented two bot difficulty modes:
    Regular Bot (moveBot): Makes the first legal move.
    Hard Bot (moveBotHard): Prioritizes pawns strategically, uses the SORRY! card to bump opponents farthest from home.
  Added virtual functions like isBot() and getName() to enable polymorphism.
  Converted the main game loop to treat human and bot players smoothly.
  Improved file handling with save/load features with try/catch.

Version 4 - Final version

- Goal: Finish the game with all remaining C++ advanced concepts from Chapters 13–16.
- Changes:
  Introduced the Counter<int> template class to track total moves.
  Used the STL map to maintain player stats and sort winners using the overloaded < operator.
  Added CardDeck class for shuffling and drawing from a full 45-card deck (with multiple 1–13 values).
  Implemented undo functionality using backup() and restore() methods in the PawnV4 class.
  Refined the game board display and end game stats summary.
  Applied:
  - Inheritance: Bot from PlayerV4.
  - Aggregation: PlayerV4 contains array of PawnV4 objects.
  - Static members: PlayerV4::plyrCnt tracks player count.
  - operator overloading: for comparing and printing player stats.
  - Polymorphism: Combined logic between bots and human players.
  - Documented the project using Doxygen like comments.

Description of Sample Input/Output

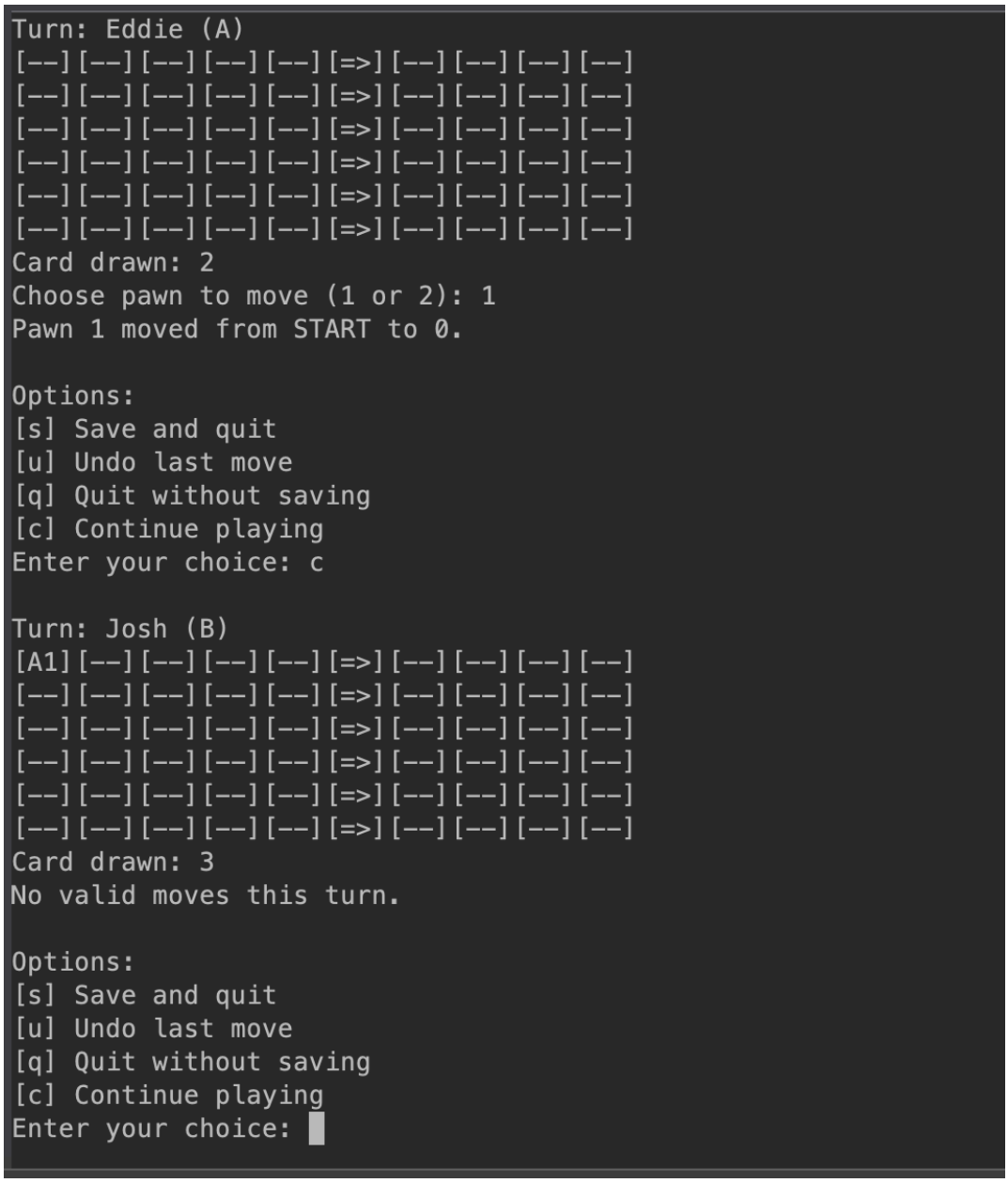
Input:

1

C

Output: Moves pawn from Home to spot #1

Then game continues



Psuedocode:

Constants:

B_SIZE = 60

P_CNT = 2

SLIDE_LEN = 3

SLIDE_STARTS = [5, 15, 25, 35, 45, 55]

Enum PosStat: START = -1, HOME = -2

MAX_PLAYERS = 4

Classes:

PawnV4:

```
private:
   int pos
   int prevPos
   bool home
   bool washome

public:
   Pawn()                -> sets pos and prevPos to START, home and washome to false
   int getPos() const          -> returns pos
   void setPos(int p)          -> sets pos = p
   void sendHome()               -> sets pos = HOME, home = true
   void sendStart()            -> sets pos = START, home = false
   void markHome()               -> sets home = true, pos = HOME
   bool isHome() const          -> returns home
   void backup()              -> sets prevPos = pos, washome = home
   void restore()            -> sets pos = prevPos, home = washome
```

PlayerV4:

```
private:
   string name
   char sym
   Pawn pawns[P_CNT]
   int moves
   static int plyrCnt

public:
   Player()                  -> sets name = "", sym = ' ', moves = 0, increments plyrCnt
   void setName(string n)       -> sets name = n
   string getName() const       -> returns name
   void setSym(char s)         -> sets sym = s
   char getSym() const          -> returns sym
   int getMoves() const        -> returns moves
   void incrMoves()           -> increments moves
   Pawn& getPawn(int i)          -> returns reference to pawns[i]
   bool allHome() const         -> returns true if all pawns are home, else false
   void reset()              -> sets moves = 0 and sends all pawns to START
   virtual bool isBot() const    -> returns false (default for Player)
   static int getPlyrCnt()      -> returns plyrCnt
   bool operator<(const Player&)  -> returns true if this.moves < other.moves
```

Bot:

private:
  bool isHard

public:
  Bot(bool hard = false) -> sets isHard = hard, name = "HardBot" or "Bot"
  bool isBot() const override -> always returns true
  bool hardMode() const -> returns isHard
  string getName() const override -> returns name
  void setName(string n) override -> sets name = n

Card Deck:

private:
  int cards[45]
  int idx

public:
  CardDeck() -> constructor calls shuffle()
  void shuffle() -> fills cards[] with 1–13 and shuffles them
  int draw() -> returns cards[idx++] or reshuffles if idx ≥ 45

Template Counter

private:
  T count

public:
  Counter() -> sets count = 0
  void add(T amt) -> adds amt to count
  T getCount() const -> returns count
  void reset() -> sets count = 0

Main:

Initialize random seed
Declare Player* array of MAX_PLAYERS
Declare CardDeck and Counter<int> for tracking moves

Show Main Menu:
  1 -> View rules
  2 -> Load game (if fail, fallback to new game)
  3 -> Start new game
  4 -> Exit

```
If New Game:
   Ask for player count
   If 1 -> Ask name, create 1 Player + 1 Bot
      Ask for difficulty: regular or hard
   Else -> Get names and create Player objects

Game Loop:
   while true:
      currentPlayer <- players[turn % playerCount]
      showBoard()
      card <- draw from deck
      if currentPlayer is Bot:
         moveBot() or moveBotHard()
      else:
         move() using player choice

      if all pawns HOME:
         announce winner
         showStats()
         showSortedStats()
         break

      Prompt: save & quit / undo / quit / continue
      if undo -> call restore(), retry move
      if save -> call saveGame() and exit
      if quit -> exit

     Post increment turn

   Delete players[]
```

Implementations:

```
draw(deck): return next card, reshuffle if needed

move(players, index, card, counter):
   If card == 13:
      search for opponent pawn to bump
      if none then skip turn
```

```
        else, ask player which pawn to bump
            bump + move own pawn to that position
    else:
        check all pawns for legal move
        if none then skip turn
        else -> ask for pawn index, move it
            bump -> slide -> bump again
            increment move counter

moveBot() / moveBotHard():
    Basic bot ->first legal move
    Hard bot -> prioritize furthest pawn, SORRY card logic

bump(players, index, pos):
    if another pawn is at pos -> send to START

slide(pawn):
    if pos in SLIDE_STARTS -> move pawn forward SLIDE_LEN

undoMove(players, index):
    restore previous position from pawn's backup()

saveGame(players, turn, file):
    write turn + all Player data to file

loadGame(players, turn, file):
    read data, return true if success

showStats(): display total moves, pawns HOME
showSortedStats(): display ranking using operator< overload
```

Flowchart:

Variables

#	Variable Name	Type	Description	Location

1	totalMoves	Counter<int>	Global counter object to track total moves made in game	Global constant (line 17)
2	plyrCnt	int	Number of players currently playing (1–4)	Line 19
3	turn	int	Current turn number	Line 43
4	choice	char	Stores menu option	Line 44
5	hardBot	bool	True if hard bot is selected; false otherwise	Line 45
6	players	PlayerV4* players[MAX_PLAYERS]	Array of pointers to PlayerV4 objects for all players	Line 152
7	deck	CardDeck	Deck object used to shuffle and draw cards	Line 149
8	currIdx	int	Index of the current player (calculated	Line 151

			as turn % plyrCnt)	
9	curr	PlayerV4*	Pointer to current player object during the game loop	Line 152
10	cont	char	User input option after a move (s = save, q = quit, u = undo, c = continue)	Line 173
11	moves	int	Number of moves by a player	Struct Plyr(line 34)
12	player	Plyr*	Dynamically allocated array for both players	Main function(line 66)
13	card	int	Function that draws random card	Line 155
14	moved	bool	Tracking for bool movement	move() line 366
15	leglExsts	bool	If any legal move exists for drawn card	move() line 367
16	atmpts	int	Counts attempts to select a valid pawn	move() line 439

17	ch	int	Player input for pawn selection for swapping	move() line 441
18	B_SIZE	const int	Board size, number of spaces on board	Game_Classes_h. Line 17
19	P_CNT	const int	Number of pawns per player	Game_Classes_h. Line 18
20	SLIDE_LEN	const int	Length of a slide	Game_Classes_h. Line 19
21	SLIDE_COUNT	const int	Number of slides on the board	Game_Classes_h. Line 20
22	SLIDE_STARTS	const int	Array of starting positions of each slide	Game_Classes_h. Line 21

Concepts Used

Chapter	Concepts	Location
13	Private Data Members	- PawnV4 has private: int pos, int prevPos, bool home, bool washome - PlayerV4 has private: string name, char sym, PawnV4 pawns[P_CNT], int

		moves Bot has private: bool isHard - CardDeck has private: int cards[45], int idx - Counter<T> has private: T count
13	Constructors	Each class has a default constructor that initializes values. Like PawnV4 sets position and flags, PlayerV4 sets moves to 0, and CardDeck shuffles the cards right away.(line 39)
13	Array of Objects	The PlayerV4 class contains an array of PawnV4 objects to manage each player's pawns. In main(), the players themselves are stored in an array of pointers.(line 86)
13	UML	PlayerV4 and Bot where Bot inherits PlayerV4.(line 146)
14	Static Members	- PlayerV4 has a static member plyrCnt that tracks how many players have been created. - In PlayerV4 class. Line (88), and

		initialized in main file before main.
14	Operator Overloading	bool operator<(const PlayerV4& other) is overloaded to compare players based on their moves count.(line 108)
14	Aggregation	PlayerV4 has an array of PawnV4 pawns[P_CNT]. So has an instance of another class. Line(86)
15	Protected	PlayerV4 has protected members. Line 83.
15	Base class to derived	Class bot is a derived class from PlayerV4. Line 146.
15	Polymorphism	PlayerV4* players[] array to store **both** PlayerV4 and Bot objects. This allows me to call other needed methods. (Line 41)
15	Abstract Classes	virtual bool isBot() const { return false; }. Line 134
16	Exceptions	In both void saveGame and bool loadGame. Using

		try/ catch exceptions. Line 300.
16	Templates	template class Counter<T> to track moves. Line 16.