Project 1

Orbito

Daniel Zarate

CIS-17C-48786

November 19, 2023

**Table of Contents**

**Section Title Page**

1. Introduction 3
2. Approach to Development 3
3. Game Rules 6
4. Description of Code 6
5. Sample Input/Output 7
6. Checkoff Sheet 7
7. Documentation of Code 8

**1 Introduction**

Everyone knows about the classic game Connect 4. I enjoy playing this game, so when I saw a game similar to Connect 4 with a twist, I knew this would be the game that I wanted to code for this project. I found Orbito on a Youtube video where a couple played the game, and I was fascinated by the uniqueness of this game.

I estimate that about 25 hours total were spent coding this game over the course of a little more than a week. It has 2 classes and a total of 574 lines of code. Link to GitHub repository: https://github.com/RealHumphry/CIS17-C

**2 Approach to Development**

Before beginning development, I wanted to make sure I had a good idea of how the game works. I do not own a copy of Orbito, so I had to search for the complete rules online. Eventually, I found a picture of the back of the box which explains the game’s basic rules (the picture can be found in the Game Rules section). The website also gives more explanation for what happens in more unusual cases, like a draw for example. With this in mind, I began development. Of course, I decided to begin with the base of the game: the board.

*Version 1*. To create a simple and visual way for the player to interact with the board, each position on the 4 by 4 game area would be represented by a letter followed by a number. The player simply types the position using this format to indicate what spot they want to interact with. I decided to use a map for the game board because I wanted to have each element to represent one spot on the board. The map’s key could be used to identify each spot on the board and a char variable could represent what was occupying that spot. In this version, the key was a char pointer.

*Version 2.* I created the board class. This contains the board map that was made in the first version. I found out that char pointers don’t work too well as keys, so I decided to make the key an int variable instead. I created functions in this class for printing and editing the board.

*Version 3*. Now that I can edit the board, I wanted to create a function for checking if someone had made 4 in a row. Since each spot on the board is numbered, I can jump to any other position from where I currently am and compare the value on those spots to check if there is a four-in-a-row match. The function returns a bool variable type which tells if there is a win or not.

*Version 4.* I used a queue to keep track of who’s turn it was. I wrote this in a way to prepare to move it to its own class.

*Version 5.* The “Player” class has been created. The main function loops a function in the Player class, which allows each player to edit the board and fill in spots in the board with their piece. Depending on which player’s turn it is, an ‘X’ or an ‘O’ fills in their selected spot. As mentioned before, a queue is used to keep track of who’s turn it is. It is filled with two elements, ‘1’ and ‘2’. When the player’s turn begins, the value at the front is retrieved and popped. When it is empty (which would occur on player two’s turn), the queue is refilled with the two int elements so that the value can again be retrieved for the next turn.

*Version 6.* Instead of using an int for the key of the board map, I decided to use a string. This would allow the player to enter the position and the container could directly accept this input as the key. I also made changes to the function that searches for any four-in-a-rows. To prevent repeated code, I made a function so that each row and column could be searched in a single loop. It searches by making complex manipulations to the key string. This would be later replaced with a simpler method. The game loop in the main now ends once a player has won (instead of looping for a certain number of times even if no one has won).

*Version 7.* A function for orbiting the board has been added to the Board class. I am using a stack and a list for this function. Two containers are used because the perimeter of the board orbits slower than the inside (just like how planets further from the sun take longer to complete a revolution). All the pieces on the perimeter (whether or not they are empty) are put into a stack, and then poured out back onto the board one position away from where they originally were. This allows the orbit to be made. An iterator for the map is used to traverse through the various positions on the board. The function is called after a player has chosen where to place their piece.

*Version 8*. The function that checks for any four-in-a-rows now uses iterators. Two iterators and nested loops are used. One iterator cycles through the first row. For each cycle, the second iterator moves down that column and their values are compared. If any difference is found, then there is no four-in-a-row. The same concept is used to check each row. Additionally, the two diagonals are checked. The function returns an int indicating whether player 1 won, player 2 won, there was a draw (both have at least 1 four-in-a-row), or neither has won yet.

*Version 9*. A function to the Board class has been added to check if a stalemate has occurred. A list is filled with all the (second) values in the board map and the count() function is used to count the number of O’s there are on the board. If half the board has player two’s pieces, then the other half must be full of player one’s pieces. Hence, the board is full and no more moves can be played. In the case that this event occurs and no player has completed a four-in-a-row, the board is orbited 5 times and once again we check if there is a winner.

*Version 10.* The orbit function now only uses a stack. Since there are only 4 inner pieces, I thought it would be simpler to just use the swap() function. Two iterators traverse through the board and only three swaps need to be made to orbit the inner pieces.

*Version 11.* Now another important part of each player’s turn is added: moving an opponent’s piece. This involves the user inputting a position which must be occupied by an opponent’s piece, and then asking them where they want to move it to. They may also choose not to move an opponent’s piece. The end position must be adjacent to its starting position. The user’s input must be checked to see if it passes all of these requirements. Various functions were made to do this. First, there is a function which takes and handles all of the user’s input. This calls another function, which fills a list with all the spots adjacent to the spot entered by the user. This was quite challenging because filling with all adjacent spots must take into account if the spot is along one of the sides or in a corner, where there are not as many adjacent spots as there would be if it was a spot in the middle. The merge() function is used to combine the list that was passed by reference as a parameter and the list that was filled within the function. Another function that takes the list of adjacent spots checks if there are any open spots in any of those adjacent spots. This is used to prevent the player from selecting an opponent’s piece that has no where to move to. The player function that is called on each turn gives the player the option to move an opponent’s piece except on the first turn when there is no opponent piece on the board yet.

*Version 12.* Continuing with checking the adjacent spots, a function confirms whether or not the adjacent spot that the player selected is in fact adjacent to the original spot. This uses the find() function to determine if the spot the player entered is found in the list of adjacent spots. The function that fills the list fills it with the board’s map keys (instead of the second value as it previously did). In the function that checks whether or not the adjacent square is occupied, a set is filled with the board’s map second value. The find() function is used to check its occupancy. Text is also added to appear at the beginning of the game. This includes the title, slogan, general rules, and a small note which explains how to input positions on the board. At this point, the game may be considered completed.

*Version 13.* The final version adds finishing touches. The text that appears at the beginning is put into its own function to keep the main function clean. Functions are added to the Player class. The functions in the Board method which previously asked for user input are now moved to the newly created Player functions for organization and simplicity reasons. Small changes throughout are made to accommodate these changes.

A box with instructions on it

Description automatically generated**3 Game Rules**

Orbito is easy to pick up, but difficult to master. The game is simple, but the strategy is intricate. It requires deep thinking and visualizing how things will change as the game progresses.

The game is played by two players on a four-by-four grid. They each take turns making moves. Each turn can be simplified into 3 steps. First, the player takes one of the opponent’s pieces and moves it to an adjacent square (diagonals included). This step is optional. Of course, this is not done on the first move since your opponent has not yet placed one of their pieces on the board. Second, you choose an open spot to place your piece. Finally, the board orbits. Because the board is constantly changing, this game requires thinking ahead and predicting how things will change. The first player to get four pieces in a row, either horizontally, vertically, or even diagonally, wins! A stalemate occurs when the board is

completely filled and there is still no winner. No moves can be played by either player, so the board is orbited 5 more times. If there is now a four-in-a-row revealed at the end of this orbit, then that player wins. A game that gets to this point will surely be a nail-biting close call.

**4 Description of Code**

The code has just 2 classes and one function inside the main.cpp file besides the main function. I wanted to keep the main function as simple and clean as possible. One basic loop in the main function runs the entire game and one int variable keeps track of how many turns have gone by.

The Player class keeps track of which player’s turn it is using a queue container from the STL. This class has 8 functions. They handle requesting and verifying the user’s input so it can be passed as parameters into the functions in the Board class. It also manages some data from the Board class and displays the results to the user, especially when it comes to declaring the winner and other similar details.

The Board class creates a game board and has many functions for manipulating this board. The board is a map container from the STL. Stacks, sets, and lists are also STL containers that are used for checking and moving data from the board. Several functions in this class utilize STL algorithm functions for manipulating data within these containers. This class handles everything behind the scenes so that the Player class can output everything in a way that matters to the user. Only one class displays output: the class for displaying the board.

**5 Sample Input/Out**

See folder “Screenshots of Output.”

**6 Checkoff Sheet**

*Container Classes*.

*Sequences*

*List.* Lists are the container used when keeping track of all the spots adjacent to another spot. They are also filled with all the elements in a map and a count of a certain element within that list reveals if there is a stalemate.

*Associative Container*

*Set.* A set is filled with all the elements in a map to search for a certain element. More specifically, the set searches for any unoccupied spots adjacent to another spot.

*Map.* A map is used as the gameboard. Each key marks a spot on the board, and the value in the key says what occupies that spot (‘X’, ‘O’, or nothing, which is ‘ ’). The key is a string and the second value is a char.

*Container Adaptors*

*Stack.* When orbiting the board, a stack is filled with all the outer pieces. The top values of the stack are replaced into the board but in a new position; they are placed where they would be after 1 step in the orbit.

*Queue.* A queue is used to keep track of who’s turn it is. It is filled with all the players by number. At the start of each turn, the number at the beginning of the queue is retrieved and popped. If the queue has reached the last player, it is refilled, putting player 1 at the beginning, so that play can resume.

*Iterators*

*Bidirectional Iterator*. Both lists and maps use this type of iterator.

*Algorithms*

*Non-mutating algorithms*

*Find.* This algorithm is used to look for a certain value in a set and list. It is also used to assign an iterator to wherever a certain key is found in a map.

*Count.* This algorithm is used to get the number of times a certain key appears in a map. This helps determine if the value entered by the user exists as a key in the map. If not, their input is considered invalid. It is also used to find out the number of times a certain value appears in a list. This helps determine if the gameboard is full, which would be a stalemate.

*Mutating algorithms*

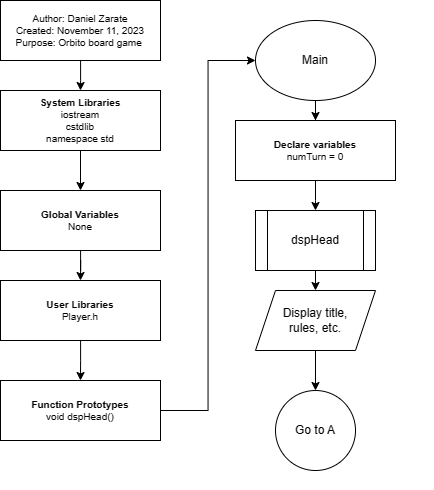
*Swap.* This algorithm is used to swap the inner values of the board when orbiting it.

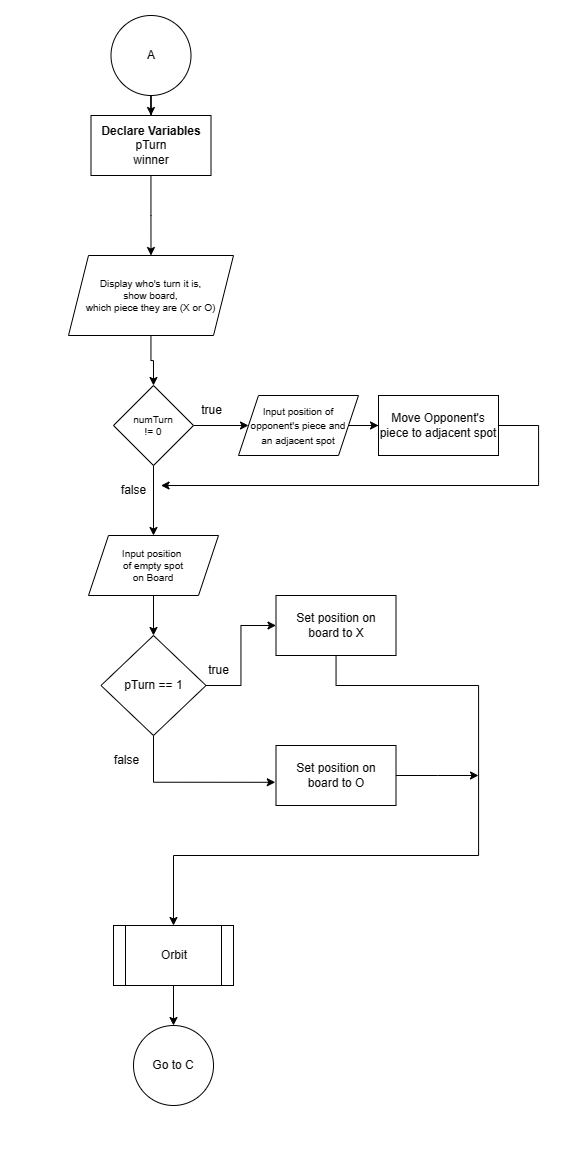
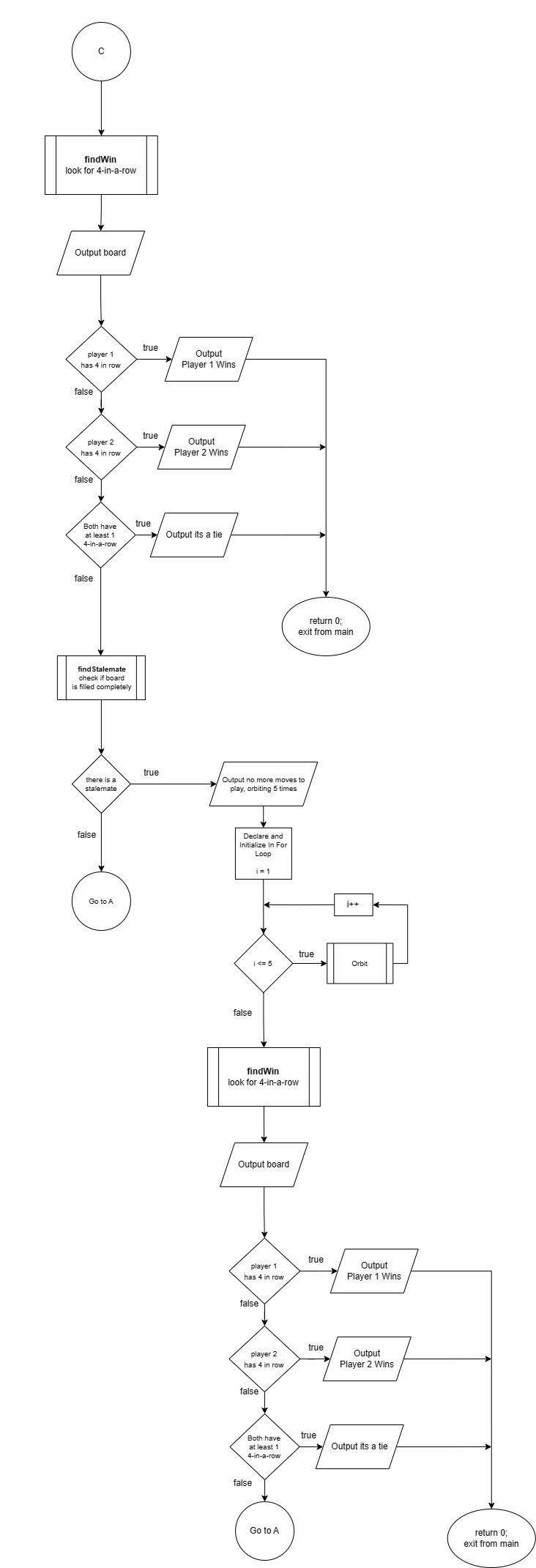
*Organization*

*Merge.* This algorithm is used to combine two lists. One list is passed as a parameter to a function by reference. Another list is created within that function and the two lists are combined using the merge algorithm.

**7 Documentation of Code**

*Flowchart*

**

*  
*