Project 2 - Connect 4

Michael Cooper

17C - 43484

Dr. Lehr

June 12, 2022

Introduction	2
Approach To Development	2
Game Rules	3
Description of Code	3
Sample Input/Output	5
Project 2 Additions	6

Introduction

I chose to code the game Connect 4 because I thought it would tie all the requirements together nicely and efficiently. I spent over 2 weeks before reaching the final build reaching **1264 lines** with 4 overarching classes for the board, game, game state, and players.

The repository can be found here: https://github.com/SouL909/Cooper_Michael_CSC17C_43484/tree/main/Projects/Project2

Approach To Development

Concepts

I used separate header files for each aspects of the game needed to make a complete build with all the necessary components. Each respective file manages a different part of the game including the visual representation of the board itself and each players markers as well as the state of the game and its players. The game state files tie in all the separate classes and organizes the phases of the game such as accessing the menu, reading the rules, creating a game, and keeping track of rounds/wins. In addition the wins and related data are saved via an output file and are able to be accessed when returning to the game after exiting and saving.

Version Control

The build took four versions to create with the first two taking a great deal of time to determine how to best separate each file and make relevant classes while encompassing all the needs of the project. I also ran into some difficulty getting the build to run with the restraints of my Mac and not having C++11 and had to do lots of troubleshooting to include all the requirements. Once finally getting the build to run and work correctly it was

just a matter of optimization and making the inputs and outputs user friendly.

Game Rules

Game Title: Connect Four

The classic table top game which involves trying to successfully complete a consecutive sequence of four markers, while preventing your opponent from doing the same. The “Board” consists of an 8 by 4 grid in which players can drop a marker, even if they must stack on top of another marker.

Objective:
To be the first player to connect 4 of the same markers in a row (either vertically, horizontally, or diagonally)

How To Play:

- First, decide who goes first and which marker they will use (X or O)
- Players must alternate turns, and only one disc can be dropped in each turn.
- On your turn, drop one of your colored discs from the top into any of the available slots.
- The game ends when there is a 4-in-a-row or a stalemate.

Description of Code

Line size: 1264
While initially focusing on making sure the prompts from the program were clear and easy to understand, I also tried to lay out the game just like an arcade version. Two separate players should be able to easily navigate through the board and log their progress throughout, while having the program automatically detect wins in

any direction. The program will be able to loop for another round or save players data for later use.

Layout:

This version includes the use of 9 header files and 4 source files, so that the main is extremely simplified and the data is passed through the various files. The player information (player.h/.cpp) has been stored in public and private classes and allows that information to be completely separate from the actual turn processes.

A Look at the Main:

```
int main(int argc, char** argv) {
 // Initializes and runs game
  Game game;
  game.start();
  game.run();
    return 0;
}
```

The use of header and source files resulted in the main becoming extremely brief. An instance of a class is created from the respective .h file and then passes that data into game.start() to start the actual game processes.

Header and Source Files Examples:

board.h/cpp:

The board files keep track of the board size and display, wins, and each player's markers on the board by creating respective classes. The Cpp file Uses a constructer to create the board and uses a bidirectional iterator to check each column and determine where a marker can be placed. The board is then printed and wins are checked for in each possible direction.

```
  const int BOARD_SIZE = 8*4;
  //Class declaration, keeps track of each individual tile for X's or O's
  //Also keeps track of which player is X or O
  class Board {
  public:
    std::list<char> board; // Sequences: list
    std::map<std::string, char> player_markers;
    Board(const std::list<Player>& players);
  public:
    bool mark(std::string name, int col);
    char vertWin();
    char hortWin();
    char diagWin();
    char win();
    void printBoard();
  };
```

```
//Constructor to create board
Board::Board(const std::list<Player>& players) {
  _player_markers = &player_markers;
  for_each(players.cbegin(), players.cend(), fe);
  _player_markers = nullptr;
  board.resize(BOARD_SIZE);
  fill(board.begin(), board.end(), *MARKERS.cbegin()); // Mutating Algorithm: f
}

//Checks each column of array and sees where it can place marker
bool Board::mark(string name, int col) {
  if(col < 0 && col >= 8) {
    return false;
  }
  for(std::map<string, char>::iterator itr = player_markers.begin(); itr != pla
    if(itr->first == name) {
      for(int i = 3; i >= 0; i--) {
        auto spot = next(board.begin(), (col + 8 * i));
        if(*spot != '#')
          continue;
        else {
          *spot = itr->second;
          break;
        }
      }
      return true;
    }
  }
  return false;
}
```

```
Leaderboards and selection
--------------------------------------------------
(1) Player: Michael Wins: 1
--------------------------------------------------
--------------------------------------------------
(2) Player: Michael Wins: 1
--------------------------------------------------
--------------------------------------------------
(3) Player: Jacob Wins: 0
--------------------------------------------------
--------------------------------------------------
(4) Player: Jacob Wins: 0
--------------------------------------------------
Would you like to create new players? (y/n)
```

```
Would you like to create new players? (y/n)
n
Player 1 select your profile: 1
Player 2 select your profile: 3
Playing...

# # # # # # # #
# # # # # # # #
# # # # # # # #
# # # # # # # # Michael's turn!
Enter a number from 0-7
```

GameState.h/cpp:

The game state files use a series of glasses for each respective part of the game including the menu, creating a game, tracking the progress of the games, and the rules themselves. The cpp file executes each part of the game relevant to the established classes and utilizes text output save data to check if there are already players saved.

Sample Input/Output

```
Welcome to Connect 4!
Enter the following options...
1. Rules
2. Play
3. Exit
```

Upon starting the game the player is prompted to read the rules, play, or exit. If the player decides to check the rules they are displayed replicating the “board” that will be displayed. The player will then have the option to return to the menu or quit.

If the program detects there is save data it will display a leader board where the players can select their names or create new players. The board is then displayed with the players name whose turn it is. By entering a number 0-7 your respective marker is dropped into the available slot on the board. After

```
x # # # # # # #
x # # # # # # o
x # # # # # # o
x # # # # # # o Michael WINS!
Play again? (y/n):
```

```
# # # x # # # #
# # x o x # # #
# x o o x # # #
x o o x o # # # Michael WINS!
Play again? (y/n):
```

```
# # # # # # # #
# # # # # # # #
o o o # # # # #
x x x x # # # # Michael WINS!
Play again? (y/n):
```

a win is detected the winning player will be displayed and a prompt will be displayed to start another game. The different win scenarios are displayed below.

Project 2 Additions

Recursion/Recursive Sorts:

In GameState.cpp a recursive bubble sort for player data is used until a base case is reached which ties into the win counts and leaderboard.

```
190     } else {
191       // Uses the comp function to sort the
192       // players by their win count
193       // sort(players.begin(), players.end(), comp);
194       rec_bubble(&players[0], players.size());
195       // game->players = list<Player>(players.begin(), players.end());
196       int i = 0;
```

Recursion is also used in Tree.h for the insert and print function as it sorts data through the tree as shown below:

```
53          void printPostorder() {
54                  if (stem == nullptr)
55                          return;
56                  // first recur on left subtree
57                  printPostorder(stem->left);
58
59                  // then recur on right subtree
60                  printPostorder(stem->right);
61
62                  // now deal with the node
63                  std::cout << stem->data << " ";
64          }
65
66          void printPostorder(Leaf<T>* node)
67          {
68                  if (node == nullptr)
69                          return;
70
71                  // first recur on left subtree
72                  printPostorder(node->left);
73
74                  // then recur on right subtree
75                  printPostorder(node->right);
76
77                  // now deal with the node
78                  std::cout << node->data << " ";
79          }
```

Hashing:

Hash.h is used to go through each character and apply a bitwise operation tied to LinkedList.h and a key.

```
  LinkedList<Node>* table;
  unsigned int ELFHash(const std::string& key) {
    unsigned int hash = 0;
    unsigned int x    = 0;

    for(std::size_t i = 0; i < key.length(); i++)
    {
      hash = (hash << 4) + key[i];
      if((x = hash & 0xF0000000L) != 0)
      {
         hash ^= (x >> 24);
      }
      hash &= ~x;
    }

    return hash;
  }
public:
  Hash() {
    table = new LinkedList<Node>[SIZE];
    for(int i = 0; i < SIZE; i++) {
      table[i] = LinkedList<Node>();
    }
  }
```

Trees and Graphs:

Tree.h and the use of a tree is primarily done to implement win counts, while Graph.h and the use of a graph is tied to general player data.

```
35  template<class T>
36  class Tree {
37          Leaf<T>* stem;
38  public:
39          Tree() {
40                  stem = nullptr;
41          }
42          void push(T data) {
43                  if (stem == nullptr) {
44                          stem = new Leaf<T>;
45                          stem->data = data;
46                          stem->left = nullptr;
47                          stem->right = nullptr;
48                  } else {
49                          insert(stem, data);
50                  }
51          }
```

```
46  class Graph
47  {
48  public:
49          Graph(int vertices, std::vector<Edge> list)
50          {
51                  edges = new std::vector<Edge>[vertices];
52                  this->vertices = vertices;
53                  for (std::vector<Edge>::iterator itr = list.begin(); itr != list.end(); itr++)
54                  {
55                          pushEdge(*itr);
56                  }
57          }
58          ~Graph()
59          {
60                  delete[] edges;
61          }
```