C++ Pente Manual

Bug Report:
To the best of my knowledge, there are no bugs within my implementation. I have extensively tested my program, but *unforeseen* bugs could exist - but again, to the best of my knowledge none exist.

Feature Report:

- Missing features: My computer does not delay winning for points; if sees that a move results in a win it will take it
- Extra features: none, the current implementation does not have any extra features

Description of Data Structures/Classes:
Classes: Board, BoardView, Codes, Player, Human, Computer, Round, Tournament and stdafx

Class Board:
Description:
Board handles all the logic of the game board. This includes placing a stone on an intersection, checking for 5 in a row stones, checking and updating if there is a capture pair. It is responsible for handling the state of the Pente game board.
Inheritance:
No inheritance used for this class.
Composition/Data Structures:

- Uses a 2D vector of characters to represent the board itself. Uses a vector instead of array to allow for multiple board sizes - not included in current implementation but much easier to modify at a later time
- Has a move struct to store all the information about a move. Includes information like the position, the intersections left, and the stones around position before the board was updated. Created to facilitate undoing moves
- A stack of move structs that represent the previous moves made. It is used to be able to undo moves (helpful for the computers strategy)

Class BoardView:
Description:
Is the view module for the board. Its sole purpose and responsibility is to print out the board object. Specifically, it prints out the 2D vector of characters.
Inheritance:
No inheritance used for this class.

Composition/Data Structures:

No data structures, only a static public member function that takes in a board object to print.

<u>Class Codes:</u>

Description:

Handles the return codes and error handling that may occur anywhere throughout the program. It stores the error as an enum and has a member that allows you to get the error message associated with it.

Inheritance

No inheritance used for this class.

Composition/Data Structures:

Uses an enum to identify the return code. Works well for smaller projects.

<u>Class Player:</u>

Description:

The Player class serves as the base class for Pente players. It provides common functionality and attributes that both Human and Computer players share. The class holds player-related data, including the player’s name, color, scores, and the best move to make in the case of a Computer player.

Inheritance:

The Player class doesn't inherit from any other class. It serves as the base class for the Human and Computer players. It defines a pure virtual function MakeMove, which must be implemented by derived classes to make a move on the game board. And provides two protected members BestMove() and GetReasonMessage() for Human and Computer to use to get the strategy. It also placed two functions EvaluateMove() and DetermineBest() in protected that can be overridden by a base class to change how the strategy evaluates the move or determines the best one - strictly for modifiability at a later date.

Composition/Data Structures:

- Member variables such as m_name, m_color, m_tournamentScore, m_capturedPairs, etc., are used to store player-related data
- Has a struct called ComputerMove, which is used for the computer strategy, is composed within the class to store information about the best move to make
- m_bestMove is a protected member that is a ComputerMove struct that the derived classes can access
- It also contains MoveReason enum that identities why the computer played where it did

<u>Class Human:</u>

Description:

The Human class represents a human player in the game of Pente. It is a subclass of the Player class, inheriting common player attributes and functionality. The Human class is responsible for making moves on the game board based on human input and interacting with the game.

Inheritance:

The Human class inherits from the Player class, which provides the basic player-related attributes and methods. This inheritance allows the Human class to access the computer's strategy for playing defined in its GetHelp function. It specifically overrides MakeMove to get player input.

Composition/Data Structures:

- It implements the MakeMove method, which is a required override from the base Player class, and is used to make a move on the game board
- The GetHelp method allows the human player to receive the best move
- The CallToss method is used to determine the starting player in the game.

<u>Class Computer:</u>

Description:

The Computer class represents a computer player in the game of Pente. It inherits from the Player class. The Computer class is responsible for playing the best possible move.

Inheritance:

Inherits from the Player class, which provides common player attributes and methods. By inheriting from the Player class, the Computer class can use the BestMove protected member to make its move on the board.

Composition/Data Structures:

- It implements the MakeMove method, which is an override from the base Player class. Basically it just places the stone based on the BestMove

<u>Class Round:</u>

Description:

The Round class represents a round of the game Pente. It manages the gameplay between two players, a human player and a computer player. The class keeps track of the game state, including the game board and player turns, checking for a round's end and tallying scores.

Inheritance:

The Round class doesn't inherit from any other class. It is a standalone class.

Composition/Data Structures:

- Has two player point objects, a Human and a Computer, as member variables. This is needed as we need to know the human to perform a coinflip. Stored as pointers as tournament is “in control” of them
- It uses a player pointer vector vector, m_players, to store the two players in the order of their turns
- The class includes member Board object which represents the game state
- Other member variables are used to manage the game, including the current player index, the winner of the game, etc.

Class Tournament:
Description:
The Tournament class manages a tournament of the game Pente, handling the flow of the game, including saving and loading games. Its main purse is providing functionality for playing, saving, and resuming games, as well as displaying end results
Inheritance:
Doesn't inherit from any other class - standalone class
Composition/Data Structures:
- The class contains player objects of type Human and Computer as member variables
- It uses a pointer player vector, m_players, to store player objects for when it resumes from a save
- Interacts with game rounds through the Round class, and it includes methods for managing game state serialization and deserialization
- Includes several member variables for handling file paths, string constants for parsing and formatting saved games

Class stdafx:
Description:
The precompiled header that stores all the include files. Used to not have to reinclude everything in other classes.
Inheritance
None.
Composition/Data Structures:
None.

Log:
Sept 12th 2023:
- Started rudimentary implementation of Pente, i.e. structure of classes including: Player, Board, Round and Client.cpp. Included in the classes are TODO

labels/comments to provide meaningful direction on what needs to be implemented (30 min).

- Board class and its member functions are nearly fully implemented. Board constructor initializes the board to null/empty characters 'o' (subject to change). Its current implementation includes 1 private member of the board itself, which is a 2D array of characters, acting as a game board (10 min).
- Member function Board.PrintBoard() successfully implemented; formats the columns to "A" - "S" which is variable on board size. Implementation of allowing variable board sizes is considered, but for current implementation, board size is a constant at 19 (30 min).
- Member function Board.MakeMove() successfully implemented, with necessary bounds checking. Place a player's color symbol on the 2D board array, if the selected location is valid. The function will return INVALID_MOVE (-1) if the player's selected location was not valid. Still needs to check if the move can be placed in terms of if there is another piece already there (20 min).

- Player class started. Mainly its members have been defined, but polymorphism and class inheritance still needs to be added. More specifically its members score and color, along with its accessors and mutators Player.SetColor() sets objects color (needs proper checks still) and Player.GetColor() returns in color (10 min).

- Round class started. Adds a board and a player array for the specific round. Added needed members such as Board object to start the game, and a rudimentary move system Round.Play(), which is very bare bones providing a structure to where to proceed next. Essentially Round is currently acting as a way to test Board and moving for now - but will become where the ply/round loop resides (20 min).

- Client.cpp is where main will be held, and the driver of the entire program. Only includes main and a Round object currently (5 min).

- Wrote down different strategies computers could take to have the best chance at winning. Most serious and more difficult computers likely require some probabilistic AI model, which will likely take too long to not only create, but train as well. My current computer will prioritize winning instead of point collection, as it could spiral into eventually losing - though future work is needed on a full fledged computer strategy (1 hour).

- Design and structure of Pente and the C++ program fully drawn out and detailed on paper. It includes where inheritance and polymorphism will occur, members each class will need, constant values, where to ask for input, how each class will interact with one another, etc. Mainly this is the guide on how the game will be written in code, describing what needs to be done and my current best way of implementation (1.5 hours).

Total: 4.5 hours

Sept 17th 2023:
- Reworked most of Board Class. Instead of a 2D array, I opted to use a 2D vector of characters to represent the board. This is done in case board size were to either change among round, or just selected to be a different size. Generally it makes working with the board much easier (10 min)
- Added a default constructor that sets the board to a symbolic constant if not selected (currently 19) and initializes the board. And a parameterized for the boardSize member (10 min).
- Defined Board.InitBoard() helper function to initialize the vector to null characters (which the symbolic constant has since now been changed to '-') and called it in the constructors, based on size on the board - uses vector.resize() (10 min).
- Defined Board.PositionToIndex() member which takes the position as a string as a parameter and converts it into a valid row column index for the 2D vector. Row and column passed by reference for easier manipulation. Returns true if it could convert the string, false if it could not (20 min).
- Defined Board.IsValidPosition() helper function which validates the position of a row or column. This only checks if the selected move was inside the board bounds - not if the stone was already there, as it will need to be used in capturing board pieces (20 min).

- Defined a BoardView class to facilitate output of the board (10 min).
- Moved Board.PrintBoard() to the BoardView (now BoardView.PrintBoard() class as the Board should not be handling output, only manipulation of the board. It now takes a Board object as a parameter - it copies it instead of passing by reference as stated in the style guidelines. Reworked it so it uses Board's assessors instead of previously using members (30 min).

- Altered design to allow for the client to completely be in control of input and output - still working on some details, but will no longer ask for output within any Round, Tournament, or Player classes - will only accept input (1 hour).

Total: 3 hours

Sept 18th 2023:
- Created an enum in Board to facilitate a way to return mutator errors. Currently includes SUCCESS, INVALID_MOVE, and INVALID_BOARD (10 min).
- Created mutator for setting the board: Board.SetBoard() - will be used for serialization. Sets the m_gameBoard member to the passed 2D vector. Returns invalid board - if more than the MAX_BOARD_SIZE (currently at 26) (20 min).

- Defined m_capturedPairs, and it's accessor Board.GetCapturedPairs() called in Board. as a way to see if the move placed resulted in the other opponents pieces being captured (10 min).
- Defined Board.CapturePairs() to capture pairs if possible, called in Board.MakeMove(). Checks every direction (currently with an array - but looking into using cos and sin for checking), and sees if a pair can be captured based on current move. It does also call Board.IsValidPosition() to ensure where it is looking is not out of bounds. Made public, as computers could potentially use this for its strategy (1.5 hours).

- Finished Board.MakeMove() mutator function. It takes the color and position, e.g. ('W', "J10"), as a string, calls Board.PositionToIndex() to convert into indices, then checks if these indices are valid with Board.IsValidPoisiton() and if the piece we are on is not a null piece. If these conditions are met, we change the position on the board to that color of stone, and call Board.CapturePairs() to see if the move would update the board in any way. Returns INVALID_MOVE or SUCCESS, depending on if the piece was placed (1.5 hour).

- Wrote down questions doc regarding specific questions I have about style guidelines and overall program design. To be asked in office hours to get a better understanding of what needs to be reworked/implemented (20 min).

Total: 4 hours

Sept 19th 2023:
- Renamed select Board function members to better describe their purpose: Board.MakeMove() -> Board.PlaceStone(), Board.IsValidPosition() -> Board.IsValidIndex() (5 min).
- Defined Board.IsBoardFull() and called it in Board.PlaceStone(). Checks if there are any null pieces left on the board to see if the round ends in draw (20 min).
- Defined Board.FiveInARow() and called it in Board.PlaceStone(), along with adding a private member m_fiveInARow. Checks if the current placed piece resulted in 5 of the same colored tiles, i.e. winning the round, returning true or false (30 min).

- Defined private members m_players, m_currPlayer, and m_gameBoard to Round class.
- Defined Round.StartRound() that will act as a game loop for the round. Checks win conditions and tie conditions, and facilitates the moves for each player.
- Defined Round.CheckWin() that returns true if a player won the game. Called in Round.StartRound(). WIll be used in the game loop condition.
- Defined Round.GetNextPlayer() to get the next player in the round, whose turn it is now.
(1 hour for the above 4 items)

- Added mutators: Player.SetRoundScore(), Player.IncRoundScore(), Player.SetTournamentScore(), Player.IncTournamentScore() to facilitate encapsulation for Player class.
- Defined Player.GetStoneLocation() which is a virtual member that gets input from the player. Currently not overloaded and implementation for computers and humans are still needed. Acts as a rough plan of what to do next for the player and its subclasses.
(5 min for the above 2 items)

- Added function headers and improved comments in methods to Board class (1 hour).

Total: 3 hours

Sept 20th 2023:
- Updated Board.PlaceStone() to check if proper conditions are met, i.e. guard clauses. Returns an enum to determine its success or specific error (10 min).
- Added Board.NInARow() to check if a stone's pieces were the same color for n pieces away. This was to reduce code duplication in Board.FiveInARow() and Board.CapturePairs. Returns a bool vector of 8 directions - if true it means yes that direction had N in a row, false otherwise (1 hour).
- Created algorithm for counting 4 uninterrupted stones in a row on paper - this was a complex algorithm that basically makes all 5 in a row stones a unique piece so that it doesn't double count these as 4 pieces. It goes from top to bottom counting the lines of 4, and "removing" the stones it previously counted (2 hours).
- Implemented Board.UninterruptedStones() to check at the end of the game for 4 uninterrupted stone pieces, which works based on the above algorithm (implementation of what I wrote). Return an int of how many 4 in a row if found - to be called at end of game (2 hours).

- Updated getters and setters based on encapsulation - minimal work but done for completeness of the class (30 min).
- Updated comments to read better and replaced literal constants with symbolic, wherever possible. Also included function headers (1 hour).
- Minimally tested board to see all proper functionality - basic cases confirmed. Board class is nearly completed(20 min).

Total: 7 hours

Sept 21st 2023:
- Renamed Round.StartRound() to Round.Play(), and updated it to check if round started on a serialized file to prevent unnecessary coin flips and board restriction. Add a ply count to add board restrictions of where a player could place a stone (1 hour).
- Added Board.SetBounds() method to facilitate board restrictions based on what the current ply is. Uses manhattan distance to see how far the piece was placed from center (30 min).
- Added Round.SetPlyOrder() to determine which player will go first if the round was not started from a serialized file. Performs a coin flip asking humans to call it, if guessed correctly they are set as the first player, color being white. Also outputs these results (30 min).
- Added Human.CallToss() to perform a coin toss with a random number generator - input and output for the specific call is here as well. Coin flip of random generation not performed in Round class to prevent a miscommunication of inputs. E.g. if more than 2 players - we would need to update this, so having it one place is ideal (20 min).

- Fixed game loop in Round.Play() to check if there is a winner. I.e. if the move resulted in round points higher than 5, or if we got a 5 in a row.
- Added Round.TallyScores() to tally scores at the end of the round, i.e. there was a winner or a tie. Adds up the total captured pairs, the 4 in a row with Board.UniterupttedStones() and the amount of points received for placing 5 in a row. Outputs the amount of points received (should be its own method).
- Added Round.OutputEndPly() to display round details after each ply. Includes all players and their colors, including captured pairs and tournament scores.
- Added Round.OutputEndRound() to display winner and details - uses Round.OutputEndPly() to display player details.
- Added enum return codes for Round.Play() to tell the caller how the round ended.
(2 hours for the above 5 items).

- Move the check win to its own separate member to reduce clutter and improve readability - named Round.CheckEndRound(). Returns true if the round is ended, false otherwise. Will output the winner or if it is a tie, outputs as such. Needs tweaking to make it more readable (1 hour).

- Added some accessors to help with serialization and overall a better user experience for a Round object. As well as some mutators (30 min).
- Did very basic testing to see if game loop works and players move accordingly, will need more work once serialization is implemented (20 min).

Total: 6 hours

Sept 22nd 2023:
- Added Tournament.Start() which handles the tournament loop, i.e. starting new rounds, asking if a player wants to resume and handling the end rounds. Basic layout formed - and items below in log were the works added to this “driver” function. Basically acts as the Pente game (1.5 hours).
- Added Round.AskToSerialize() to ask users if they would like to serialize after each round. Returns true if yes, false otherwise. If true Round.Play() returns SERIALIZE enum/returncode to let the caller know to save the tournament state (45 min).
- Added Tournament.SerializeGame() which handles serialization of an ongoing game. Creates a save folder if there is none, asks the user for the file name that should be saved, and saves it to that file (1.5 hours).
- Added Tournament.FormatSave() which takes each player, and the board from Round, and returns the string of the serialization file. This helps to improve Tournament.SerializeGame() readability (45 min).
- Added Tournament.AskFileName() to get the file name of the user to save the game to. Returns a filesystem::path object to the file for ease of use (45 min).
- Added Tournament.SaveGame() to write the output from Tournament.FormatSave() to the designated location by the user (30 min).
- Tested saving games based on files, fixing bugs where they came up (1 or 2 minor ones, nothing significant) (45 min).

- Created Tournament.LoadGame() - with basic functionality laying out where I needed to go to load a serialized game. Showing files of saved games, asking the user/controller to resume if they wanted to (1 hour).
- Added Tournament.PrintAvailableFiles() which prints the current saved games in the save folder. The save folder is a symbolic constant member to avoid trying to look for saves where they are not (30 min).

- Added Tournament.AskToResume() which asks the user/controller if they want to resume from a previous saved game, if any are located in the save folder. Returns true if the user does, false otherwise(30 min).

Total: 8.5 hours

Sept 23rd 2023:
- Added Tournament.GetFileAsPath() to reduce code duplication by making sure the file has the proper .txt extension. Returns file path with the proper extension (20 min).

- Added Tournament.ReadFile() which handles reading and making sure the serialized file is parsed. Basically gets all information from the saved file, and updates the objects such as players and rounds based on the save files. Returns true if successfully parsed the save file (1.5 hours).
- Added Tournament.ParseBoard() which handles the parsing of the board in the save. Returns a bool to indicate if parse was successful (2D vector passed by reference to act as board - may change to have better compatibility with Board Class) (45 min).
- Added Tournament.ParsePlayer() handles parsing of each player. A player object is passed by reference and updated accordingly. Needed some work with regex to ensure capturing right color and proper points (1.5 hours)
- Added Tournamnent.ParseNextPlayer(), handling parsing for the next player in the round (30 min).

- Added Tournament.LoadGame() to facilitate loading and starting of the game based on serial files. Calls Tournament.ReadFile() and asks the user for a save file and such - driver method to start from serialized game (1 hour).
- Created Round.SetGameState() to ensure encapsulation was not broken. Updates game board, and sets a flag for a serialized game to true, also updates the players. Returns an enum on its success/failure (45 min).

- Added Tournament.OutputEndResults(), for when the round ends and user/controller does not want to play another round (20 min).

- Updated Tournament.Start() to handle all of the above items. Essentially integrated all of the functions used to load from a save, and allows for saving a game - updated game loop. Used top down approach here - really resulted in derived functions (30 min).

- Started testing based on serialization files on a website, to ensure proper loading and realized a massive bug in code for checking five in a row. It only worked on end pieces as I failed to realize that a stone could be placed in the middle to make 5. Deleted half of my board class - as the underlying functionality of checking n in a row didn't take into account placing in the middle (30 min).
- Went for a walk (30 min).

Total: 8 hours

Oct 8th 2023:
- Rewrote Board.NInARow() to Board.CheckNInARow(). Instead of returning a boolean for each 8 directions, it returns the count of the color in each direction (30 min).

Total: 30 minutes

Oct 9th 2023:
- Added Board.ColorSeq() which gets the stone sequences for each direction in a vector of strings n places away. Includes the position stone is on as well, so the first value will be where the row/column index starts on (30 min).
- Added Board.AreSameStones() which returns true if the string of characters are the same. Meant to check if all the stones in the sequence are the same color (10 min).
- Added Board.UpdateSeqs() which updates the surrounding directions of stones based on the vector of string sequences. Updates the board to new sequences. (20 min).
- Fixed/Readded Board.CapturePairs() which checks all the sequences based on Board.ColorSeq() and see if the stone at the location "captures" a pair of stones (1 hour).

- Removed Board.CheckNInARow() logic in preparation for the new functions above
- Tested capturing pairs and all other related functions stated above on board (not with round) in multiple different scenarios - fixed bugs that presented themselves (30 min).

Total: 2.5 hours

Oct 10th 2023:
- Started work on Board.CheckNInARow(), set up basic layout and wrote down the rudimentary logic that needs to be implemented. This included testing regex matches

for specific patterns on how a N in a row can present itself - full implementation to come (1 hour).

- Created the design and layout of how the computer will work - wrote down in notes how to utilize functions from Board to essentially check a win condition, capture pair, blocking or creating a block. Implementation is needed, but design is worked out (1.5 hours).

Total 2.5 hours

Oct 12th 2023:
- Changed Board.AreSameStones to Board.CountSameStones which counts the number of stones in a row in a sequence of stones (1 hour).
- Changed Board.CheckNInARow() to Board.NumNInARow() adjusting some of the logic structure (10 min).
- Created Board.GetWinInARow() which returns the number of times a win sequence occurred in all 8 directions (1.5 hour).
- Created Board.CheckValidMove() as a guard function to prevent illegal moves. Separates the guard clauses from the logic in Board.PlaceStone() (20 min).

- Did significant testing on improvements made to the board to ensure code is working properly (30 min).

Total: 3.5 hours

Oct 13th 2023:
- Created Board.GetWinInARow() to access the count of wins the board made (5 min)
- Created Board.UnitterStones() which counts the total number of uninterrupted stones based on the input n. Ensured that it didn't double count in the same direction as the win (1.5 hours).
- Changes Board.NumNInARow() to get the right count for Board.UnitterStones() (30 min)
- Fixed bug in Board.CountSameStones that would improperly count stones based on the shareable middle stone from the sequence (30 min).
- Made a new member function Board.CardinalCount() which counts the number of stones in each cardinal plane, i.e. horizontal, vertical and the diagonals (1 hour).
- Remade Board.OffsetIndices() which includes a way to step on the row and column deltas (15 min).

- Created accessors Board.IsBoardFull() and IsWinner()
- Updated Board.PlaceStone() and fixed a bug where it would improperly count the win
- Updated Board.SetBoard() and ensured that the guard clauses worked
- Extensive testing base on these changes
- Created Board.InitGameBoard() for constructor
(2 hours for the above 5 items)

- Created Round.Reset() (30 min).
- Tournament.Start() allows for the game to be reset based on new Round.Reset() (10 min).

Total 7 hours

Oct 14th 2023:
- Fixed a bug in bounds for the Board. Created inner bounds and outer bounds to properly set a region where a player can place (1 hour).
- Created Board.IndicesToString() to convert row and column indices for the 2D into a string position based on their offsets (30 min).

- Created a Struct for moves in Board class
- Allows for storing previous moves to eventually undo them, stored in a stack
- Removal of member variables, stored in struct now as a m_currMove
- Created Board.UndoMove() which pops the previous move off the stack and sets it to the current move - updating whatever the previous move did.
(2 hours for the above 4 items)

- Tried implementing MiniMax in my player class to act as the strategy for my computer, but failed horribly (2 hours)

- Fixed bug in Board.UndoMoves() miscounting pieces left and win/captures (30 min)

- Created Player.BestMove() which plays as the current player and next player for every intersection on the board. Recording the best moves of the two (2 hours).
- Created Payer.EvulateMove() which evaluates the move made in Board.BestMove() (2 hours)
- Created Player.GetReason() to get the reason on why that was the best move (1 hour)

- Created Human.GetHelp() which gets the computers strategy for a move (30 min)

- Updated Human.MakeMove() to allow human to ask for help (30 min)

- Updated Computer.MakeMove() to allow the computer to place the stone base on the best move (15 min).

Total 13 hours

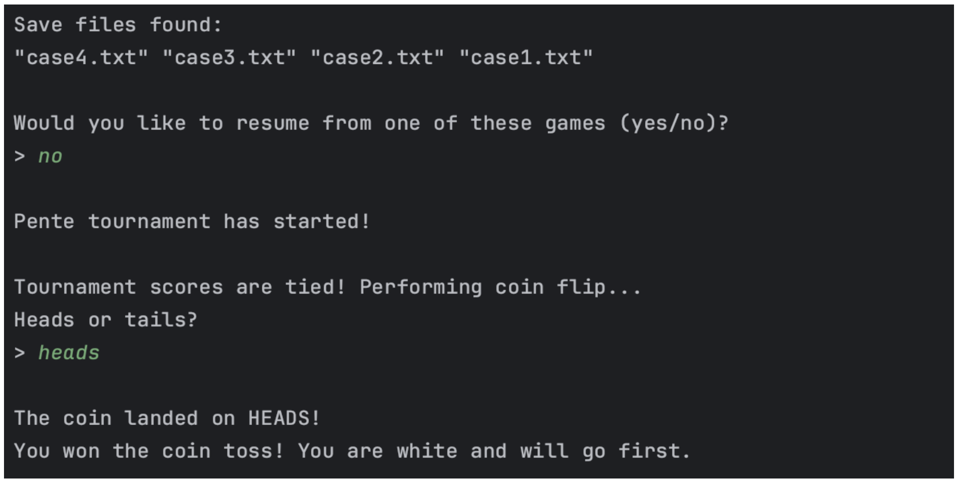
Oct 15th 2023:
- Added function headers to all functions in all the classes (2 hours).
- Improve comments to better explain semantics and design choices, removing all TODO: labels (2 hours).
- Refactored code in places where duplication existed. Including spots where it was clear that another function was needed as the function had more than 1 purpose (2 hours).

- Created C++ manual (2 hours)

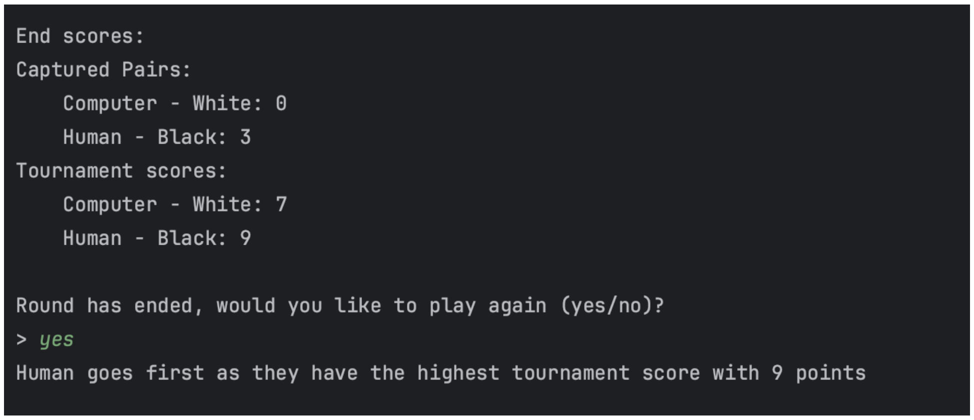
Total 8 hours

How to Run:
The entry point is in main.cpp. Compile for C++17, can do g++ main.cpp and the rest of the classes: `g++ -std=c++17 main.cpp Board.cpp BoardView.cpp Codes.cpp Human.cpp Computer.cpp Player.cpp Round.cpp Tournament.cpp stdafx.cpp`. There is a cmake-build-debug folder that can automatically generate the makefile for you and create an executable if you so wish.

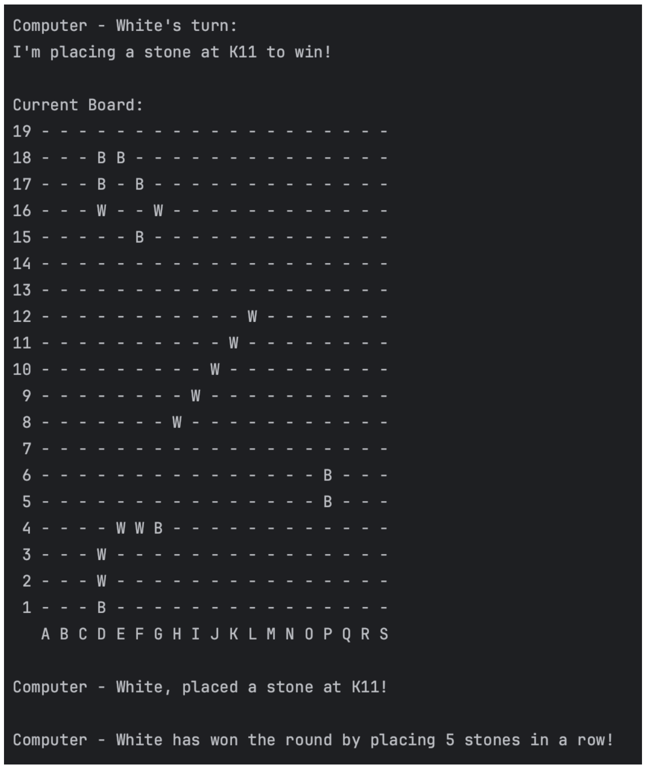
Screenshots:
<u>First player of round being determined</u>
Tied scores or first round:


```
Save files found:
"case4.txt" "case3.txt" "case2.txt" "case1.txt"

Would you like to resume from one of these games (yes/no)?
> no

Pente tournament has started!

Tournament scores are tied! Performing coin flip...
Heads or tails?
> heads

The coin landed on HEADS!
You won the coin toss! You are white and will go first.
```

One player has more points:

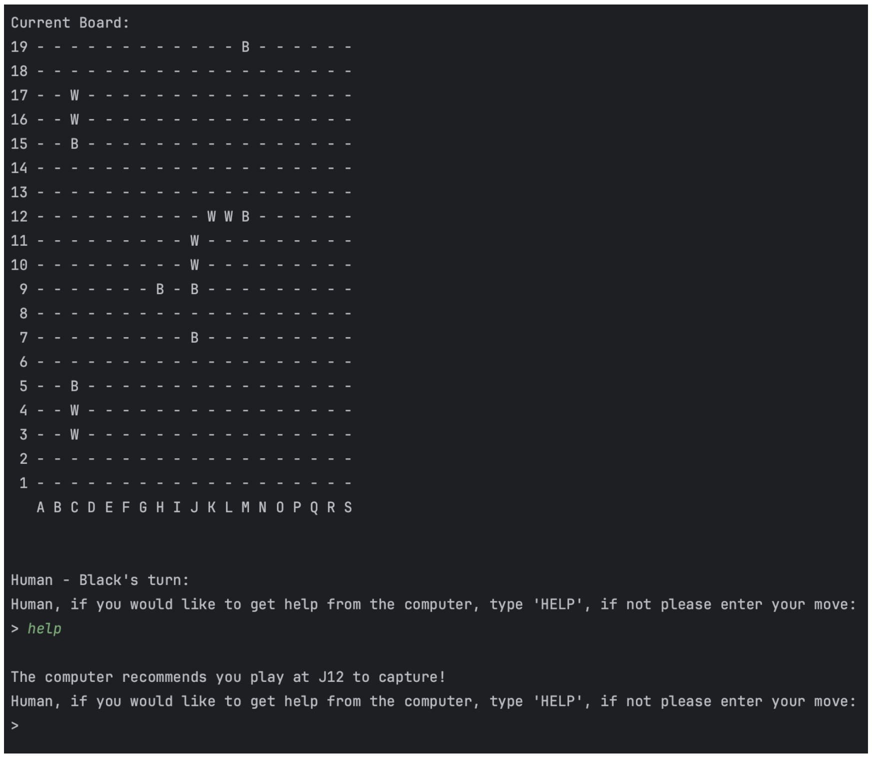

```
End scores:
Captured Pairs:
    Computer - White: 0
    Human - Black: 3
Tournament scores:
    Computer - White: 7
    Human - Black: 9

Round has ended, would you like to play again (yes/no)?
> yes
Human goes first as they have the highest tournament score with 9 points
```

<u>Computer's move being explained</u>



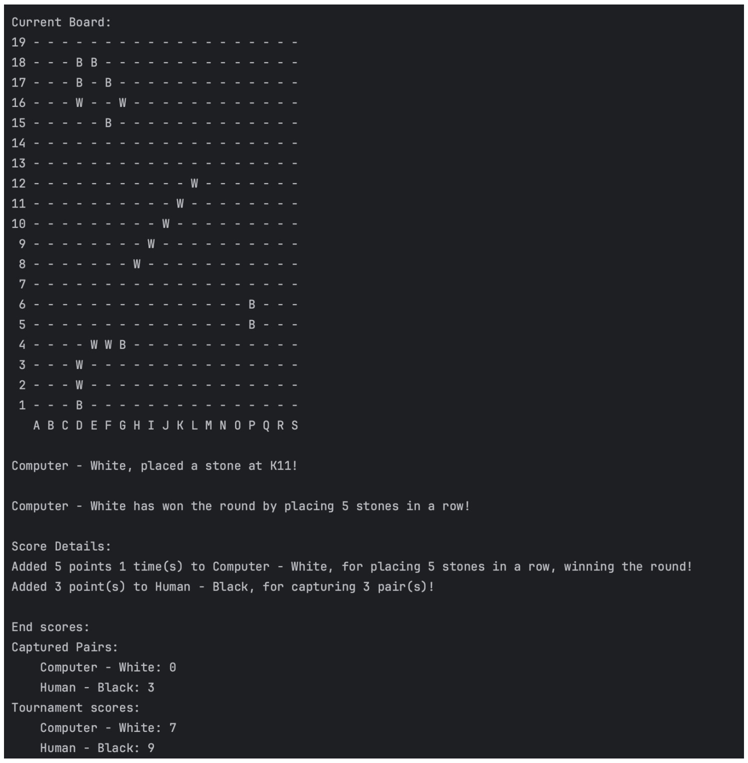
```
Computer - White's turn:
I'm placing a stone at K11 to win!

Current Board:
19 - - - - - - - - - - - - - - - - - - -
18 - - - B B - - - - - - - - - - - - - -
17 - - - B - B - - - - - - - - - - - - -
16 - - - W - - W - - - - - - - - - - - -
15 - - - - - B - - - - - - - - - - - - -
14 - - - - - - - - - - - - - - - - - - -
13 - - - - - - - - - - - - - - - - - - -
12 - - - - - - - - - - - W - - - - - - -
11 - - - - - - - - - - W - - - - - - - -
10 - - - - - - - - - W - - - - - - - - -
 9 - - - - - - - - W - - - - - - - - - -
 8 - - - - - - - W - - - - - - - - - - -
 7 - - - - - - - - - - - - - - - - - - -
 6 - - - - - - - - - - - - - - - B - - -
 5 - - - - - - - - - - - - - - - B - - -
 4 - - - - W W B - - - - - - - - - - - -
 3 - - - W - - - - - - - - - - - - - - -
 2 - - - W - - - - - - - - - - - - - - -
 1 - - - B - - - - - - - - - - - - - - -
   A B C D E F G H I J K L M N O P Q R S

Computer - White, placed a stone at K11!

Computer - White has won the round by placing 5 stones in a row!
```

Computer providing help



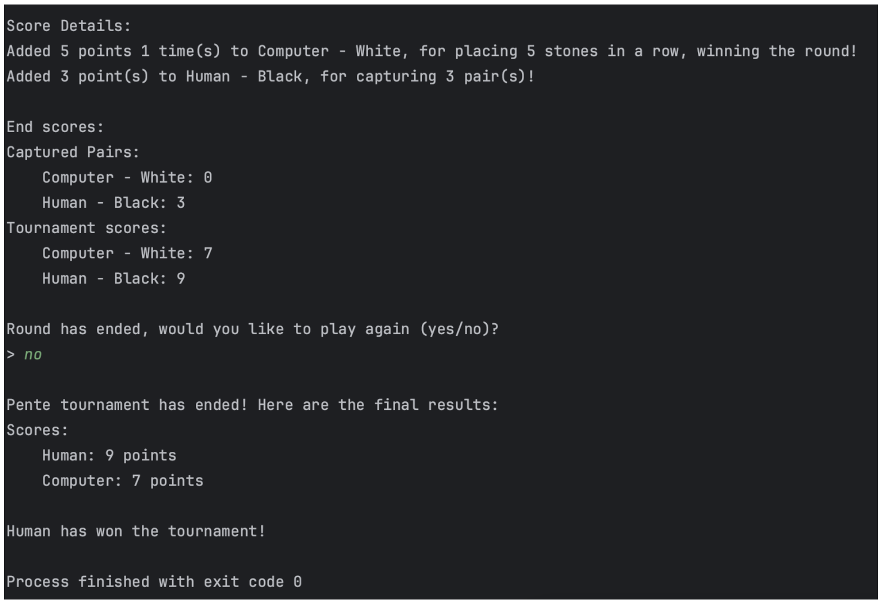
```
Current Board:
19 - - - - - - - - - - - - B - - - - - -
18 - - - - - - - - - - - - - - - - - - -
17 - - W - - - - - - - - - - - - - - - -
16 - - W - - - - - - - - - - - - - - - -
15 - - B - - - - - - - - - - - - - - - -
14 - - - - - - - - - - - - - - - - - - -
13 - - - - - - - - - - - - - - - - - - -
12 - - - - - - - - - - W W B - - - - - -
11 - - - - - - - - - W - - - - - - - - -
10 - - - - - - - - - W - - - - - - - - -
 9 - - - - - - - B - B - - - - - - - - -
 8 - - - - - - - - - - - - - - - - - - -
 7 - - - - - - - - - B - - - - - - - - -
 6 - - - - - - - - - - - - - - - - - - -
 5 - - B - - - - - - - - - - - - - - - -
 4 - - W - - - - - - - - - - - - - - - -
 3 - - W - - - - - - - - - - - - - - - -
 2 - - - - - - - - - - - - - - - - - - -
 1 - - - - - - - - - - - - - - - - - - -
   A B C D E F G H I J K L M N O P Q R S


Human - Black's turn:
Human, if you would like to get help from the computer, type 'HELP', if not please enter your move:
> help

The computer recommends you play at J12 to capture!
Human, if you would like to get help from the computer, type 'HELP', if not please enter your move:
>
```

Winner of the round being announced



```
Current Board:
19 - - - - - - - - - - - - - - - - - - -
18 - - - B B - - - - - - - - - - - - - -
17 - - - B - B - - - - - - - - - - - - -
16 - - - W - - W - - - - - - - - - - - -
15 - - - - - B - - - - - - - - - - - - -
14 - - - - - - - - - - - - - - - - - - -
13 - - - - - - - - - - - - - - - - - - -
12 - - - - - - - - - - - W - - - - - - -
11 - - - - - - - - - - W - - - - - - - -
10 - - - - - - - - - W - - - - - - - - -
 9 - - - - - - - - W - - - - - - - - - -
 8 - - - - - - - W - - - - - - - - - - -
 7 - - - - - - - - - - - - - - - - - - -
 6 - - - - - - - - - - - - - - - B - - -
 5 - - - - - - - - - - - - - - - B - - -
 4 - - - - W W B - - - - - - - - - - - -
 3 - - - W - - - - - - - - - - - - - - -
 2 - - - W - - - - - - - - - - - - - - -
 1 - - - B - - - - - - - - - - - - - - -
   A B C D E F G H I J K L M N O P Q R S

Computer - White, placed a stone at K11!

Computer - White has won the round by placing 5 stones in a row!

Score Details:
Added 5 points 1 time(s) to Computer - White, for placing 5 stones in a row, winning the round!
Added 3 point(s) to Human - Black, for capturing 3 pair(s)!

End scores:
Captured Pairs:
    Computer - White: 0
    Human - Black: 3
Tournament scores:
    Computer - White: 7
    Human - Black: 9
```

Winner of the tournament being announced



```
Score Details:
Added 5 points 1 time(s) to Computer - White, for placing 5 stones in a row, winning the round!
Added 3 point(s) to Human - Black, for capturing 3 pair(s)!

End scores:
Captured Pairs:
    Computer - White: 0
    Human - Black: 3
Tournament scores:
    Computer - White: 7
    Human - Black: 9

Round has ended, would you like to play again (yes/no)?
> no

Pente tournament has ended! Here are the final results:
Scores:
    Human: 9 points
    Computer: 7 points

Human has won the tournament!

Process finished with exit code 0
```