# CS32 Spring 2021 Project 3 Yash Shah

## Design of Classes

### Scaffold

**Private variables --**

m\_levels – stores integer of levels of the scaffold

m\_columns – stores integer of columns of the scaffold

m\_grid – stores object of type vector<vector<int>> to store a level x column grid

history\_level – stack that stores the history of a move’s level

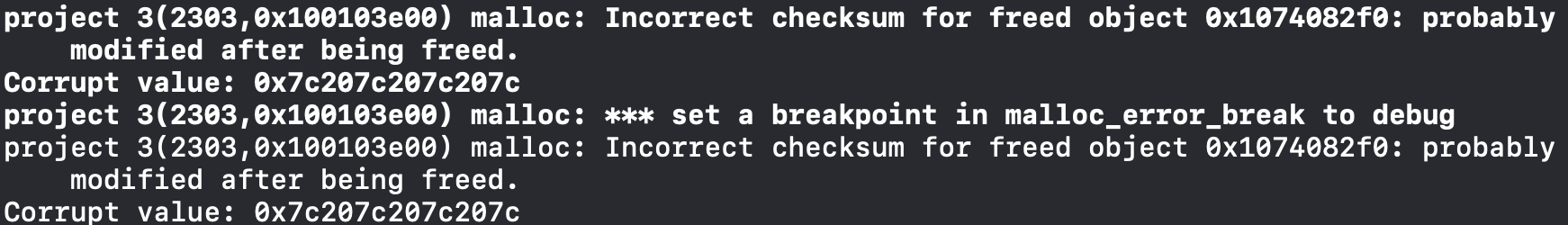
history\_level – stack that stores the history of a move’s column

\*a stack is used because the undo function requires the program to remove the checker in the scaffold that was most recently added, which implies a last-in first-out data structure, that is, a stack.

**ScaffoldImpl constructor –**

The constructor creates a user defined grid in the form of a vector of levels by vector columns where each entry is of type integer. The constructor defines its sizes and fills each spot with the constant VACANT.

Had a notable issue with my first implementation of the constructor where I tried to include the ‘+’,’-‘ and ‘|’ in the grid itself, but that implementation would cause malloc errors where memory already freed was trying to be accessed / modified due to some unknown error I couldn’t figure out. I then re-wrote the implementation to only include the game spots and add the borders in the display class.



**cols / levels getters –**

getters for m\_cols and m\_levels variables.

**numberEmpty --**

iterates through each spot in the grid and iterates a count for each VACANT spot.

**checkerAt –**

validates column and level inputs and then returns the spot at the respective level and column if valid.

**display –**

iterates through the grid and adds in the borders and grid values in form of the int to char conversion to represent the scaffold.

**makeMove –**

validates column and color. if valid, iterates through the column of the grid given to find the first vacant spot adds a token of color given at that spot in the grid. Then adds that move’s level and column in the stacks that store the respective moves.

**undoMove –**

validates that there are moves left to undo. If valid, gets the top values of the history level and column stacks and renders the spot at that grid position vacant, then deletes the move from the stack.

### Game

**Private variables --**

m\_gameboard – stores a pointer to a Scaffold object

m\_red – stores a pointer to a Player object that plays as the red checker

m\_black – stores a pointer to a Player object that plays as the black checker

m\_winner – stores an integer to the respective winner of a game

m\_turnCount – stores an integer that tracks the number of turns that have been played

m\_N – stores an integer that tracks the amount of numbers in a row (diagonally, vertical, or horizontal) it takes for a winner to be established

m\_lastCol – stores the last column that has been played on

**GameImpl constructor –**

Initializes each of the variables to their respective starting values, and creates a dynamically allocated scaffold for the gameboard defined by nColumns and nLevels.

**completed –**

tracks if the game has been completed by determining if m\_winner variables stores a value that would be interpreted as a win

**takeTurn –**

if the game isn’t already done, the function determines whose turn it is and chooses and makes a move for that player and tracks the last column that has been played on.

**play –**

while the game isn’t done, the gameboard is displayed and takeTurn is called for the respective player. When the game finishes, play outputs a message on who the winner is.

**checkerAt –**

calls the scaffold object’s checkerAt function.

### Player

#### HumanPlayer

**chooseMove –**

asks a user to select a column. If valid, returns that column. If not, the computer repeatedly asks for a column till given one.

#### BadPlayer

**chooseMove –**

selects the first column from the left where there is a vacant spot in the column.

#### SmartPlayer

**chooseMove –**

splits the columns to more efficiently parse through the columns and determine a valid move. Then calls the determineBestMove function to find the best column and returns that.

**rate –**

determines the rating to be given for a certain move on a scaffold.

**determineBestMove –**

determines the best move that a computer player can make using a minimax algorithm

### Support

#### Constant(s)

MAX\_DEPTH – holds an integer for the max depth of recursive calls (15 in my case)

#### Class Best Move

**Private variables –**

m\_bestCol – stores the column number of the object

m\_rating – stores the rating value of the object

**BestMove constructor –**

Initializes the m\_bestCol and m\_rating variables

#### Helper Functions

**isSequence –**

checks all possibilities (vertical, horizontal, and diagonal) whether there is a consequent sequence of checkers that is equal to the value of N passed into the function.

**determineWinner –**

uses the isSequence function to determine whether there is a winner.

## SmartPlayer::chooseMove

My smart player choose move function bases the decision of the column on a rating function and determineBestMove helper function that I defined in the SmartPlayer Class.

**The rate function works as follows –**

It takes in a paramaters of a Scaffold object (the gameboard), an integer N, an integer for the column that a move just has been acted on, and an integer depth,.

The function creates two winners, one for color 1 and the other for color 2, and determines if the move that just has been made (given by the integer of the column) results in a respective winner using a determineWinner function that I’ll describe below. If there is a winner, we return the result of the win (if winner 1 is the color that we passed in the function, we return a big number – depth; if winner 2 is the color that we passed in the function, we return a very small number – depth; 0 if a tie.) If no winner, then we set rating to -10000 which is an arbitrarily big number that isn’t in the range of -9999 and 9999 which would imply that the game isn’t finished.

This is used inside of the determineBestMove function that utilizes the rating to make the best move.

**determineBestMove ---**

the determineBestMove function takes in paramaters of an alarm clock, a gameboard of scaffold object, an integer N, an integer for a color to evaluate, an integer for the smart player’s color, and an integer depth.

The function defines a vector of BestMove objects, and uses that to store the results of all the moves that have been found.

The function then iterates through each column in the gameboard scaffold, checks if the alarm clock has timed out and then checks if the gameboard can support a valid move for the respective column. If so, we find the rating of the move, determine whether that rate function implies if the game is completed, and move on to a series of if and else statements to determine the recursive case. If the game is completed, then we push that value into the moveResults vector; if not and the depth is less than the max depth constant then the function recursively calls itself with the color variable set as the opposite checker checker until the game is completed and we can push that win/tie/loss value into the moveResults vector OR it hits the last else, in which case we force a tie and choose the column in the iteration.

After the moveResults object is filled with its respective bestMoves, we check if the color who we are determining is of the same color as the smart player—if so, we find the max value and submit that choice so that the smart player can establish winning moves. If the color is not the same color as the smart color, we find the minimum value so that the smart player can establish moves that don’t let the other player win.

Then we return that BestMove object which holds the best column and the rating for that column move.

## Pseudocode for Non-Trivial Algorithms

isSequence:

validate the input parameters

create temp variables for the count in a sequence, temp column and level variables, and a bool that tracks when to stop looping

check each spot horizontally, first to the left and then, in a similar process, check to the right and increment sequenceCount for each checker that is the same color. exit if the checker is a different color

check if the found sequenceCount is equal to the value of N passed—if so, return true

reinstantiate temp variables

check each spot vertically, from top to bottom, and increment sequenceCount for each checker that is the same color. exit if the checker is a different color.

check if the found sequenceCount is equal to the value of N passed—if so, return true

check each spot upward and to the right one column and level, and then in a similar process, check to the bottom and to the left one column and level. Increment sequence count for each checker that is the same color. exit if the checker is different color.

check if the found sequenceCount is equal to the value of N passed—if so, return true

check each spot upward and to the left one column and level, and then in a similar process, check to the bottom and to the right one column and level. Increment sequence count for each checker that is the same color. exit if the checker is different color.

check if the found sequenceCount is equal to the value of N passed—if so, return true

if nothing has already been returned, then there is no sequence so return false.

determineWinner:

validate the inputs to check if game is already completed based on N

find the level of the last move indicated

using the turnCount parameter, determine whose turn it is and store its color.

Using isSequence, check if there is any sequence with the color found

If there is, set the winner to that color. if not, set it to an arbitrary value

makeMove:

validate input parameters

check if the column that the parameter gives is full

if not full, iterate through the column to find the level that there exists a vacant spot

add the checker to that vacant spot and store the column and level of the move in the stacks of the respective variables

undoMove:

validate whether there is a move to undo or not (check if stacks of columns and levels is empty

find the top level and column of the stacks, and set the spot in the scaffold representing the level and column to VACANT

pop the values of the stacks

return the column of which the undo is being placed on