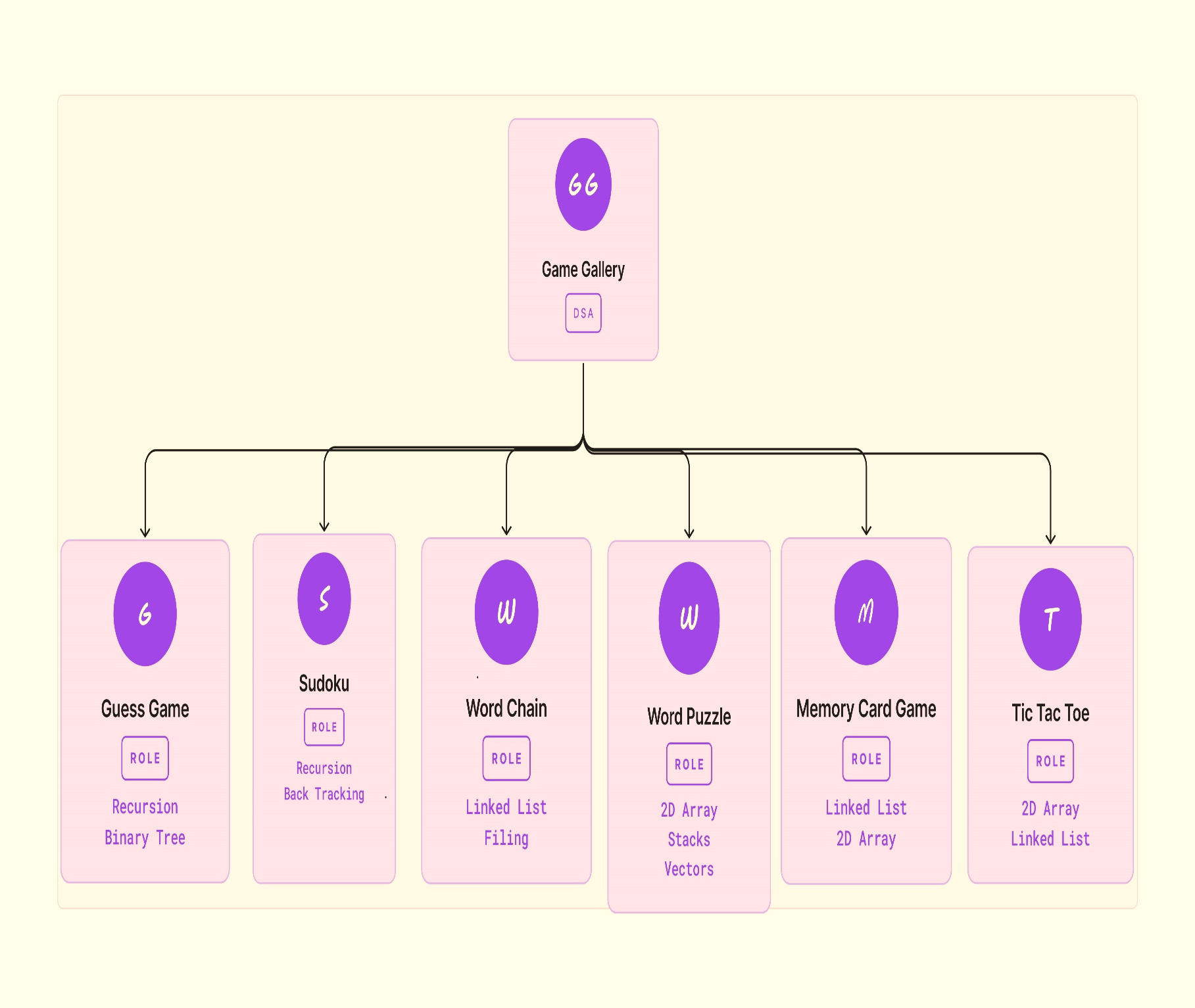
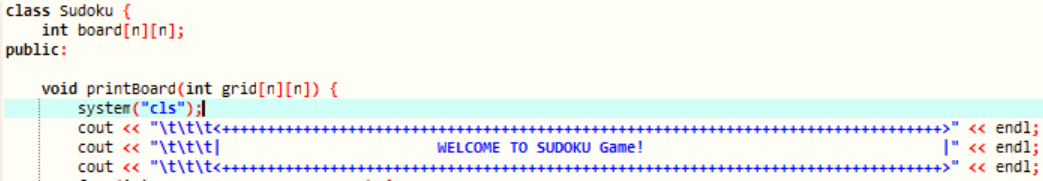
DSAA PROJECT FINAL REPORT

Manahil Abdul Kareem CT-22015

Aina Hyder CT-22017

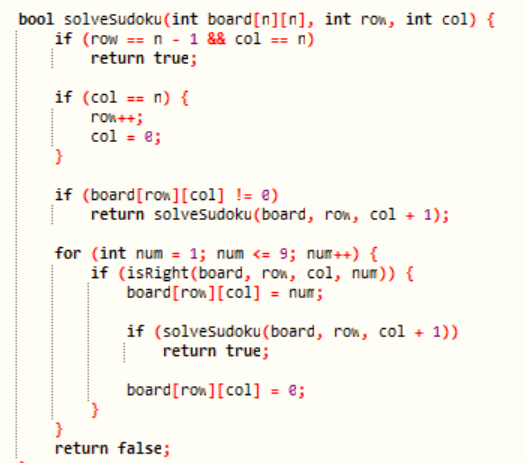
Iffat Iqbal CT-22013



SUDOKO

All functions included in Sudoku are:

**Class Definition:**

class Sudoku is defined to encapsulate the Sudoku-solving logic.

int board[n][n] represents the Sudoku grid.

**Printing the Sudoku Board:**

printBoard(int grid[n][n]) method prints the Sudoku board to the console.

It uses loops to iterate over rows and columns, separating blocks with "|" and "---" lines.

**Checking if a Number is Valid:**

isRight(int board[n][n], int row, int col, int num) checks if placing a number num at a specific position (row, col) is valid.

It checks the row, column, and the 3x3 box to ensure the number doesn't violate Sudoku rules.

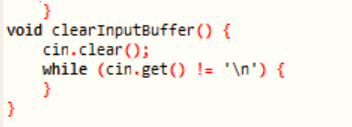
**Recursive Backtracking to Solve Sudoku:**

solveSudoku(int board[n][n], int row, int col) uses recursive backtracking to solve the Sudoku puzzle.

It starts from the top-left corner, tries placing numbers, and backtracks if a conflict is encountered.

**Checking if the Sudoku is Solved:**

isSolvedCompletely(int grid[n][n]) checks if the Sudoku grid is completely filled.



**Input Validation and Clearing Buffer:**

clearInputBuffer() clears the input buffer to handle invalid inputs.

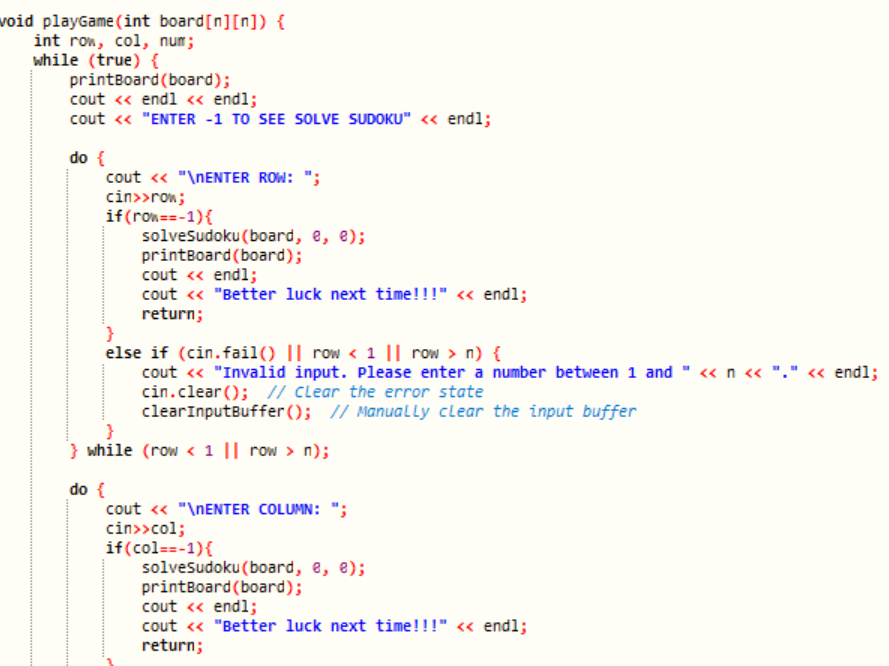
Input for row, column, and number is validated to ensure they are within the correct range.

**Playing the Sudoku Game:**

playGame(int board[n][n]) is the main function to play the Sudoku game.

It continuously takes user input for row, column, and number until the puzzle is solved or the user enters -1 to see the solution.

The game uses the previously explained functions for solving and validating the puzzle.



**Main Loop for User Input:**

The code uses a while (true) loop to keep taking user input until the puzzle is solved or the user chooses to see the solution.

**End Game Check:**

After the game loop, it checks whether the puzzle is solved completely or not.

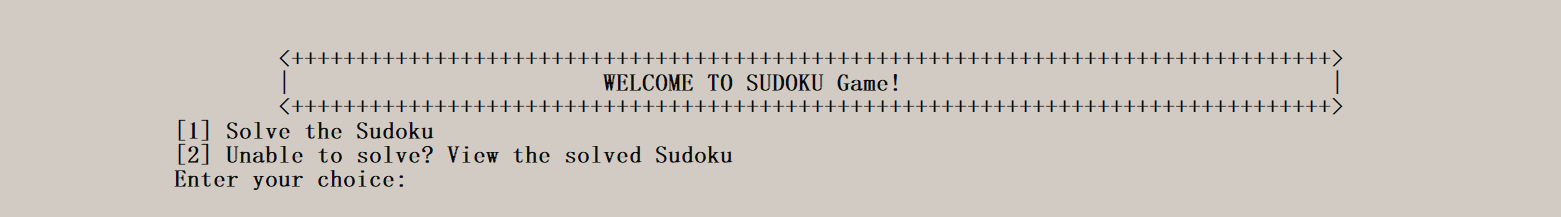
Prints a congratulatory message if solved, otherwise, suggests trying again.

**Usage of system("cls"):**

The code uses system("cls") to clear the console screen before printing the updated Sudoku board.

Conditional Printing of Solution:

If the user enters -1 for any input, the code solves the Sudoku puzzle and prints the solution, then terminates the game.

**Data Structures used in SUDOKO and why?**

**Recursion and backtracking** are utilized in Sudoku-solving algorithms due to the nature of the puzzle itself. Here’s why these techniques are favored over other data structures:

**Recursion:**

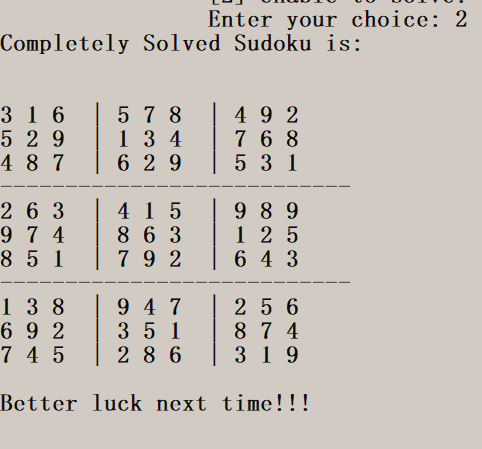
**Puzzle Structure:** Sudoku problems naturally lend themselves to recursive approaches due to their grid-based structure, which can be approached iteratively via recursive functions.

Solving Strategy: Recursive techniques provide an intuitive way to break down the problem into smaller, solvable subproblems. The Sudoku grid is split into cells, and the algorithm navigates these cells efficiently.

**Backtracking:**

**Exploring Solution Space:** Sudoku involves exploring possible solutions while ensuring they follow specific rules. Backtracking allows the algorithm to explore these solutions systematically and backtrack when encountering conflicts.

Efficiency: Backtracking helps in discarding incorrect paths efficiently, narrowing down the solution space.



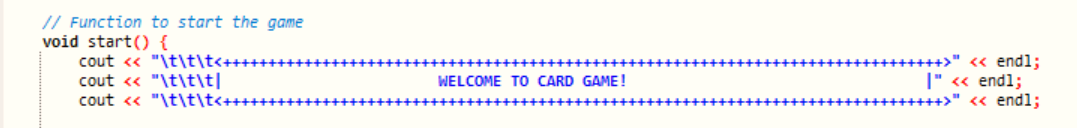
**2D Array Representation** (Used in the code as int board[n][n]):

**Direct Access:** Arrays offer direct access to individual cells in the Sudoku grid, essential for quick validation and manipulation.

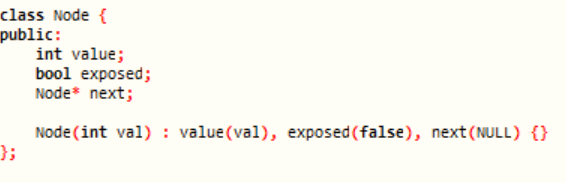
**Simplicity and Efficiency:** In Sudoku, where constant-time access to cells is crucial, arrays simplify operations like checking for valid moves and updating the board efficiently.

While other data structures (like linked lists, trees, or matrices) could represent the Sudoku grid, arrays are favored for their simplicity and efficiency in directly addressing and navigating the grid. For Sudoku, where the main challenge lies in efficiently validating and manipulating the grid cells, arrays are an optimal choice.

Recursion and backtracking align well with the problem-solving strategy required in Sudoku, providing a straightforward and efficient way to explore and solve the puzzle. Other data structures might introduce complexities in cell addressing or traversal, making them less suitable for this specific problem

Card game

. All functions included in Card Game are:



**Node Class for Cards:**

Node class represents a card with a value, exposure status, and a pointer to the next card.

**CardRow Class:**

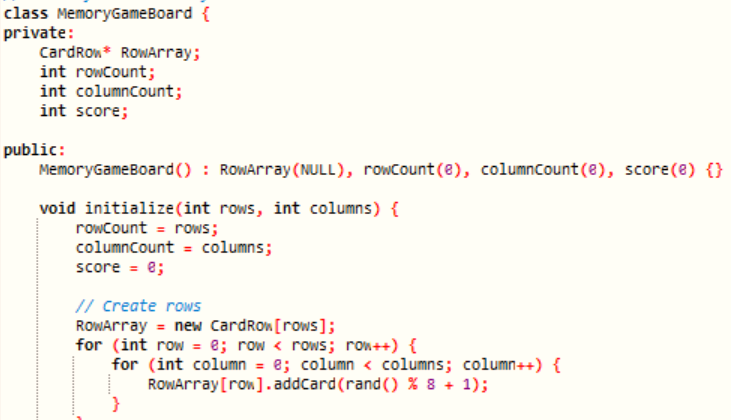
CardRow class manages a row of cards using a linked list of nodes.

Cards are added using addCard and the entire row is deleted with deleteRow.

**MemoryGameBoard Class:**

Manages the game board using an array of CardRow.

Initializes the board, displays it, checks for game completion, exposes, covers cards, and tracks points.



**MemoryGameFinal Class:**

Main class for the Memory Game.

Uses MemoryGameBoard to manage the game.

Handles game menu, user input, and the game loop.

**Dynamic Memory Allocation:**

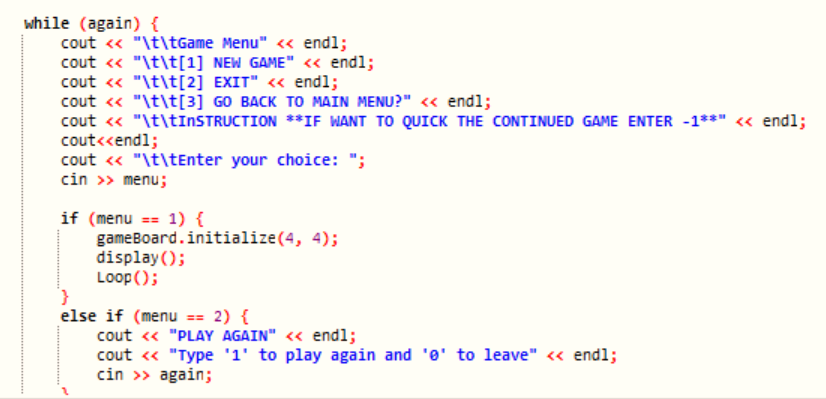
Nodes for cards are dynamically allocated using new.

**Destructor for Memory Cleanup:**

~MemoryGameBoard is a destructor to clean up dynamically allocated memory.

**Random Card Initialization:**

Cards are initialized with random values using rand().



**User Interface:**

Displays the game board, menu, and messages using cout.

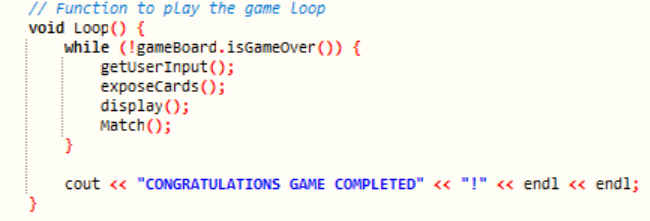
**Input Validation:**

Validates user input for row and column to ensure they are within the correct range.

**Game Logic:**

Game logic includes exposing cards, matching pairs, updating points, and checking for game completion.

**Conditional Game Restart:**

If the user enters a specific input (-1), the game restarts.

**Menu Handling:**

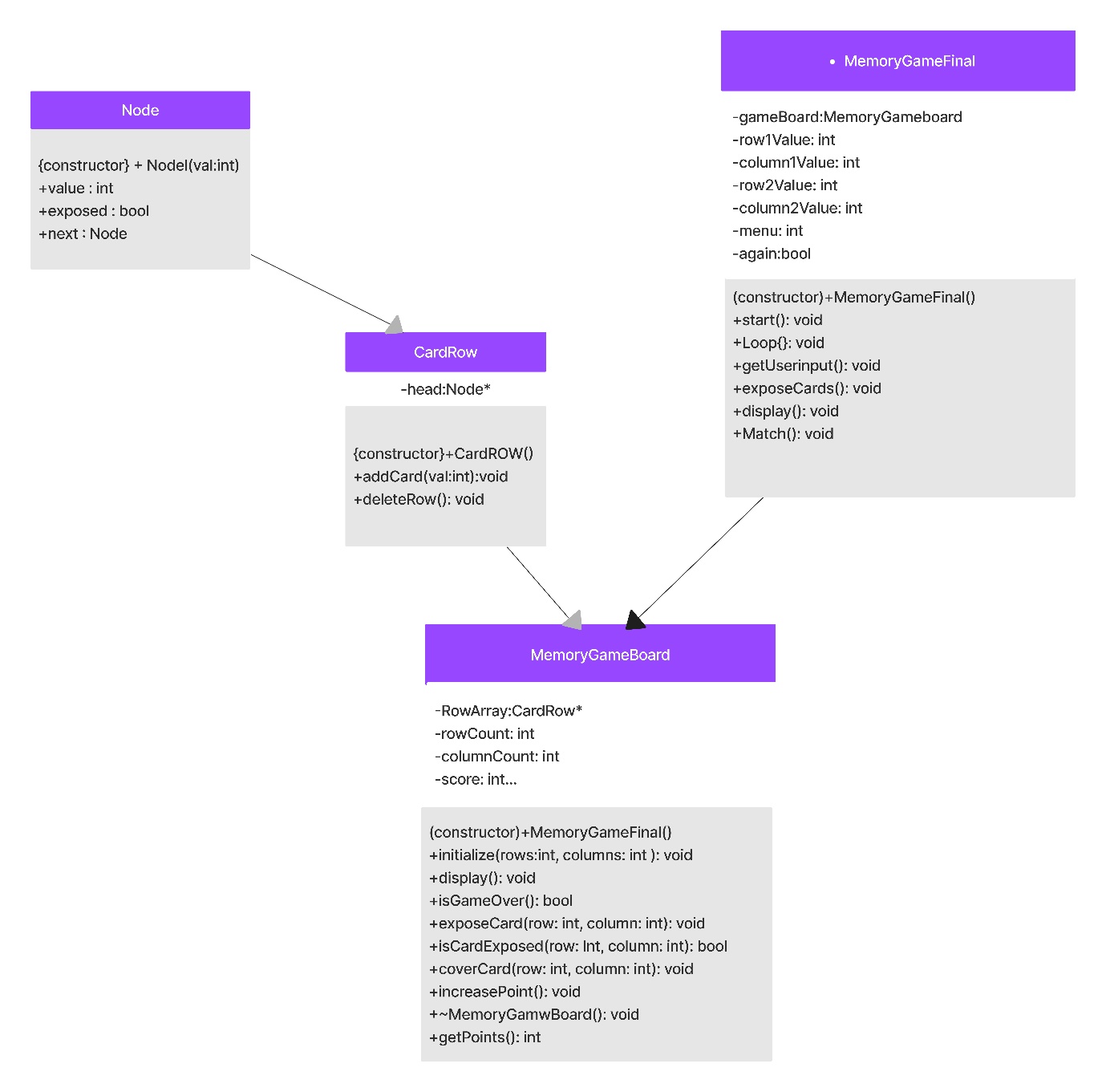
Provides a menu for starting a new game, exiting, or going back to the main menu.

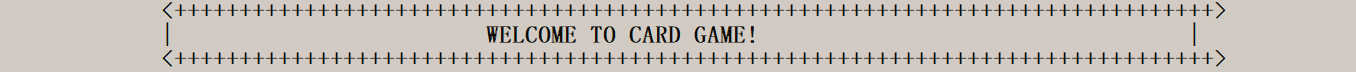
**Screen Clearing:**

Uses system("cls") to clear the console screen for a cleaner display.

**Reusability:**

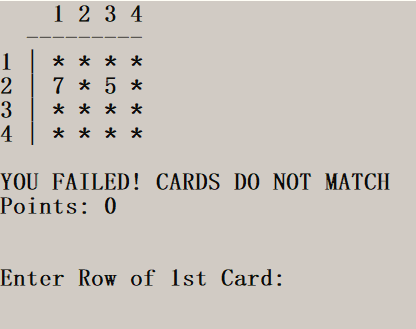
The code supports restarting the game or returning to the main menu for reusability.

Class Diagram of Card Game:

 **Data Structures used in Card Game and why?**

**Linked List** (for managing cards):

**Dynamic Structure:** Linked lists provide dynamic memory allocation, making it easy to add and remove cards during the game.

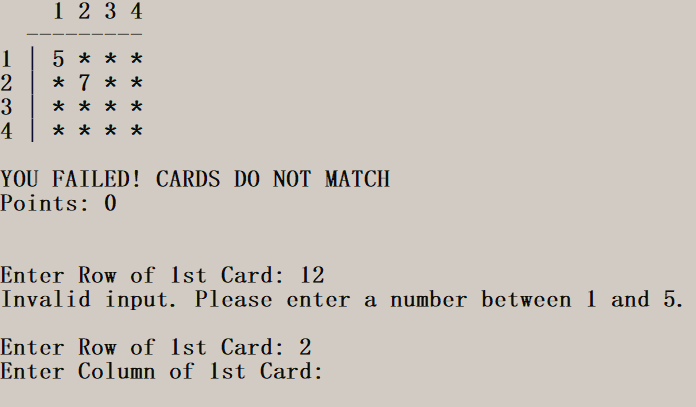
**Efficient Card Management:** The dynamic nature of linked lists allows efficient card management, such as adding new cards to the row and removing cards when needed.

Sequential Access: Cards are placed in a sequential manner, making linked lists a natural choice as they maintain the order of the cards in the row.

**2D Array** (for managing the game board):

**Grid Representation:** The game board is represented as a grid, making a 2D array an intuitive choice for organizing and accessing the cards in rows and columns.

**Constant-time Access:** Arrays allow constant-time access to specific positions in the board, which is crucial for quickly revealing, hiding, and checking cards' values.

**Efficient Memory Usage:** Arrays provide a contiguous block of memory, making them efficient for storing a fixed-size grid like a game board.

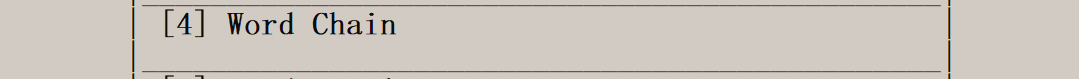
**Alternative Data Structure Considerations:**

**Arrays of Linked Lists:** While this could represent the game board, it might complicate memory management and traversal since the board has a grid-like structure.

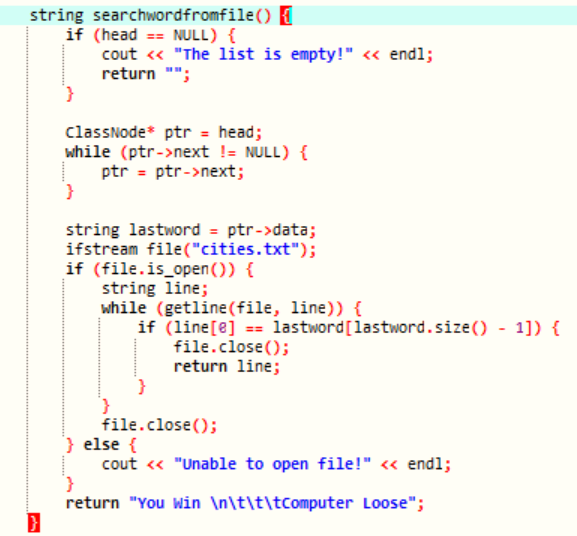
**Vectors:** Vectors could offer dynamic resizing similar to linked lists, but for the game board, where a fixed-size grid is used, arrays are more straightforward and efficient in terms of memory and access time.

In summary, linked lists are suitable for managing individual cards within a row due to their dynamic nature, while a 2D array efficiently represents the game board's grid-like structure, providing constant-time access and efficient memory usage. These structures align well with the requirements of card management and game board representation in the Memory Game implementation.ame or returning to the main menu for reusability.

Word chain



. All functions included in Word Chain and data structures used and why?

* **Linked List**
* Purpose: Maintains a chain of words entered during the game.
* Implementation: ClassNode is a node in the linked list with pointers to the next node (next) and additional pointers (left and right) for potential future expansion.
* Code Explanation:
* ClassNode Class:
* Purpose: Represents nodes in the linked list.
* Data: Contains word data, pointers to the next node, and additional pointers for future expansion.
* **Why Linked List?:**
* Enables dynamic chaining of words entered during the game.
* Game Class:
* Purpose: Manages the game logic and operations related to words and file handling.
* Data: Contains a head pointer for the linked list.
* insertNode() Method:
* Purpose: Inserts a new node (word) at the end of the linked list.
* Usage of Linked List: Adds new words to the existing word chain.

Word Validation Methods (checkword, checkRepitition, checkwordfromfile):

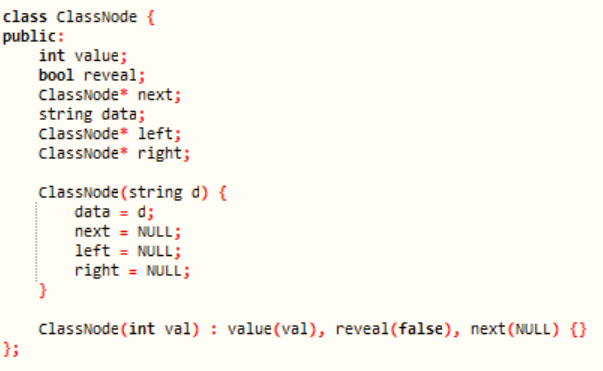
Purpose: Validate user-entered words against specific conditions.

Usage of Linked List: Verifies word chaining rules and checks if the word exists in a file (cities.txt).

* **File Handling Methods (deleteWordFromFile, searchwordfromfile):**
* Purpose: Read, modify, and delete words from a file.
* Usage of Linked List: Supports operations based on words stored in the linked list.

start() Method:

Purpose: Controls the flow of the game between the player and computer.

Usage of Linked List: Adds user-inputted words to the chain, checks validity, and allows the computer to play based on the chain's last word.

* printList() Method:

Purpose: Displays the current chain of words.

Usage of Linked List: Prints the sequence of words entered during the game.

* instructions() Method:

Purpose: Provides game instructions and starts the game.

Usage of Linked List: Initiates the game by calling start() method.

Why Linked List?

Advantages:

Allows dynamic addition and traversal of a sequence of words.

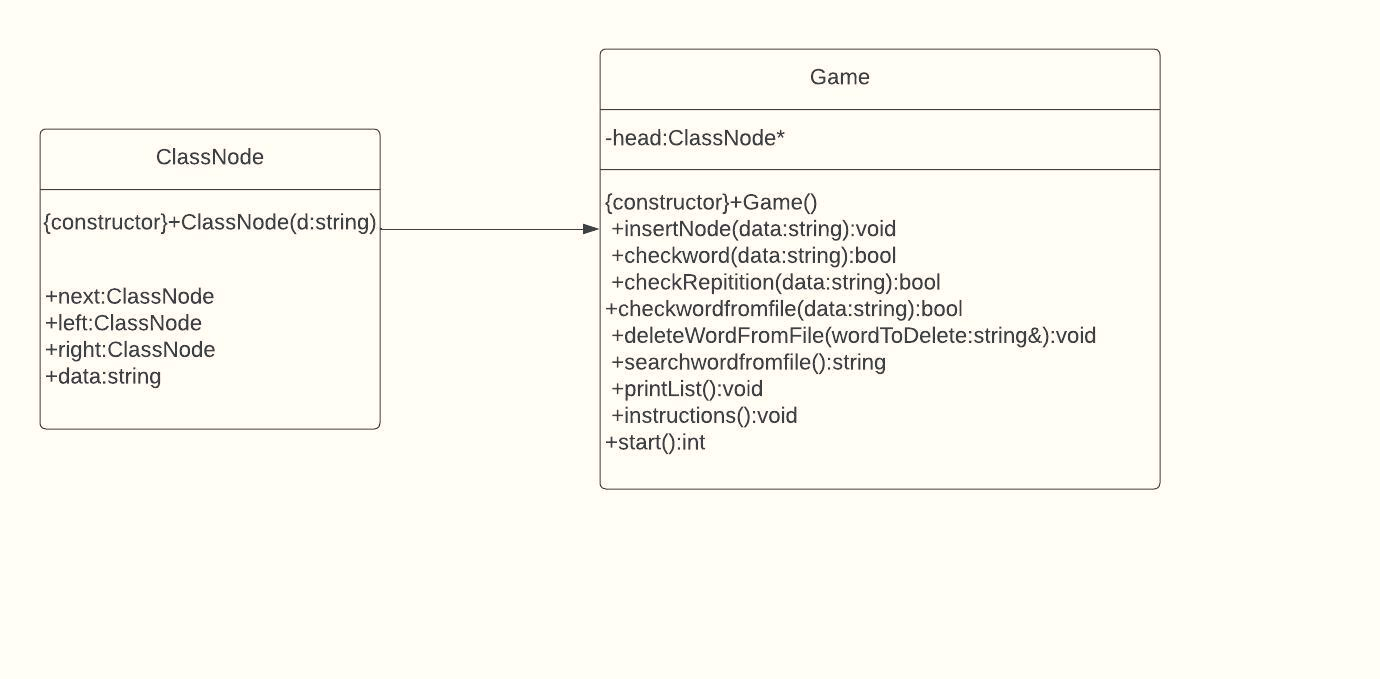
Supports sequential word chaining, crucial for this word-chain game.

**Other Data Structures Considered:**

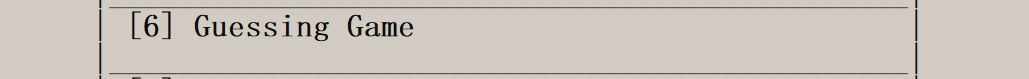
Arrays: Less efficient for dynamic word insertion and deletion.

Stacks/Queues: Not ideal for maintaining a sequential chain of words and validating word repetition.

The linked list's sequential nature and ability to dynamically add nodes align well with the game's requirement of chaining words based on specific conditions. Other data structures like arrays, stacks, or queues would not provide the necessary flexibility for maintaining and manipulating the word chain during the game.

Class Diagram for Word Chain:

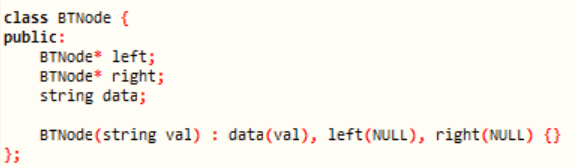
Guessing game



. All functions included in Guessing Game and data structures used and why?

* **Binary Tree Structure**

**Purpose:** Represents a hierarchical decision-making structure.

**Data Structure Used:** Binary tree (BTNode class with left and right pointers)

* Nodes: Each node represents a question or a potential guess.

**Left/Right Pointers:** Left child represents a "no" answer, and right child represents a "yes" answer.

**Explanation of Parts:**

**BTNode Class:**

Purpose: Represents nodes in the binary tree.

Data: Contains pointers to left/right children and a string for data.

**Why Binary Tree?:**

Allows for branching based on yes/no answers, efficiently organizing decision points.

**BinaryTree Class:**

**Purpose:** Manages game logic.

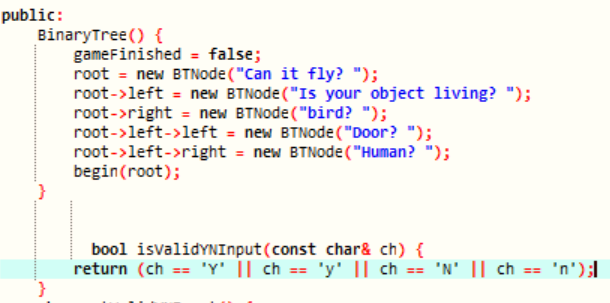
**Data:** Keeps track of game status and root node.

Why Binary Tree?:

Ideal for hierarchical decision-making where each node has at most two children.

Game Initialization (BinaryTree() Constructor):

**Purpose:** Sets up initial questions and potential guesses.

**Usage of Binary Tree:**

Constructing the initial decision tree structure to start the game.

**User Input Validation:**

isValidYNInput and getValidYNInput:

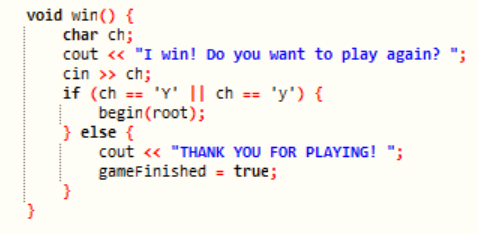
Ensure user inputs ('Y', 'y', 'N', or 'n') are valid.

**Why Binary Tree?:**

Validating simple yes/no user inputs for navigating through the tree's branches.

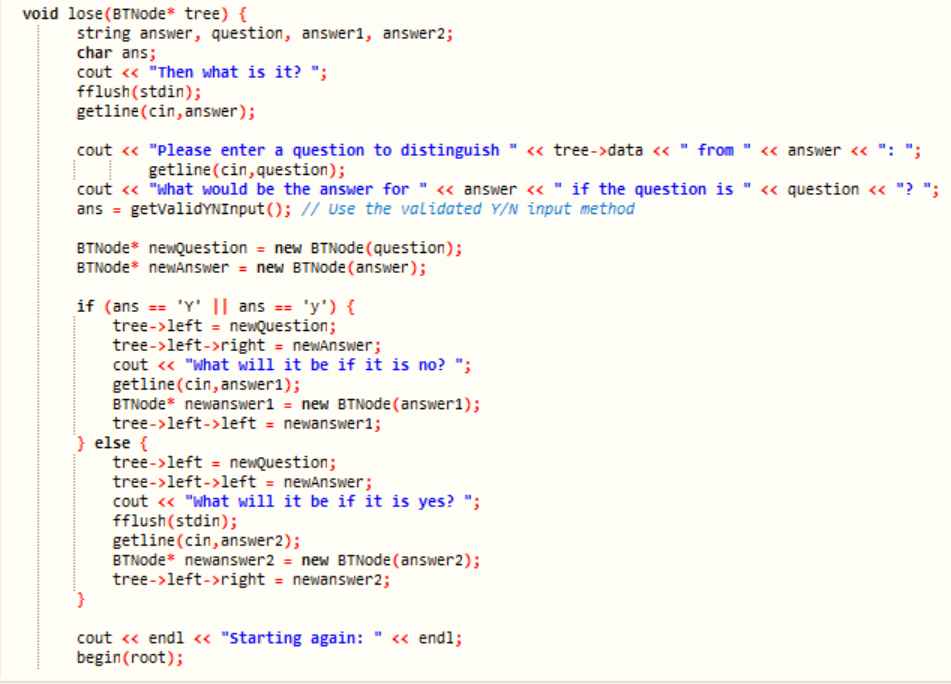
Game Flow (begin() Function):

**Purpose:** Guides the game based on user input.

**Recursion:** Traverses the tree based on user responses until a guess is reached.

**Why Binary Tree?:**

Recursively navigates the tree's structure to reach conclusions.

Win/Lose Handling:

**win() and lose() Functions:**

Manage game-ending scenarios upon successful guessing or failure.

Why Binary Tree?:

Handles game completion based on the tree's structure and user responses.

**Tree Manipulation:**

**Purpose:** Modifies the tree by adding nodes based on user input.

Why Binary Tree?:

Dynamically expands the tree structure based on user-provided information.

**Tree Destruction (~BinaryTree() Destructor):**

**Purpose:** Prevents memory leaks by deallocating memory.

Why Binary Tree?:

**Recursively** cleans up memory allocated for the tree's nodes.

Why a Binary Tree?

**Advantages:**

* Efficient decision-making structure for yes/no questions.
* Organizes information hierarchically.
* Other Data Structures Considered:
* Linked Lists: Not suitable for the branching nature of this decision-making game.
* Arrays: Less efficient for dynamic branching and node allocation.

Word puzzle game



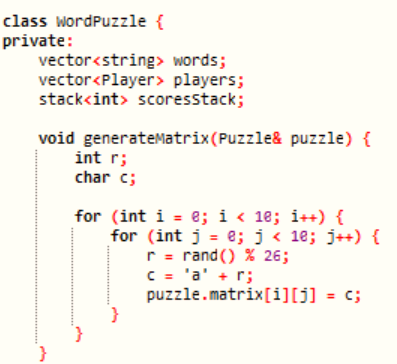
. All functions included in Word Puzzle Game and data structures used and why?

* **Class Definition:**

**WordPuzzle Class:** Manages the word puzzle game.

**Data Structures Used:**

**Vectors:** Used to store words and player information (vector<string> for words, vector<Player> for players).

**Stack:** Stores player scores (stack<int> scoresStack) during the game.

* **Methods:**

generateMatrix(Puzzle& puzzle):

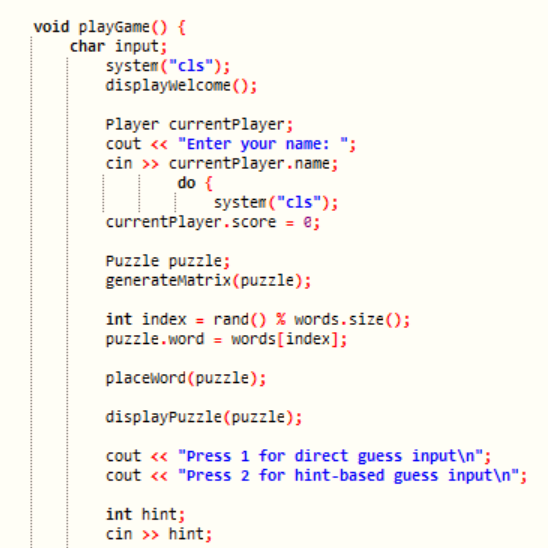
Generates a 10x10 matrix of random characters for the puzzle.

placeWord(Puzzle& puzzle):

Places a randomly selected word horizontally or vertically in the matrix.

WordPuzzle() (Constructor):

Initializes a vector of words for the game.

playGame():

Starts the game loop, allowing players to guess words.

Generates a puzzle, displays it, takes guesses, and updates player scores.

displayWelcome():

Displays the welcome message and instructions for the game.

displayPuzzle(const Puzzle& puzzle):

Displays the generated puzzle matrix.

displayScores():

Displays the scores stored in the stack.

* **Game Logic:**

Player Input: Allows users to enter their name and guess words.

Hint System: Provides hints for guessing words.

Scoring: Awards one point for each correct guess.

Replay Option: Asks users if they want to play again.

**Use of vector<string>:**

**Words Storage**: vector<string> words is used to store puzzle words.

* **Reasons for Choice:**

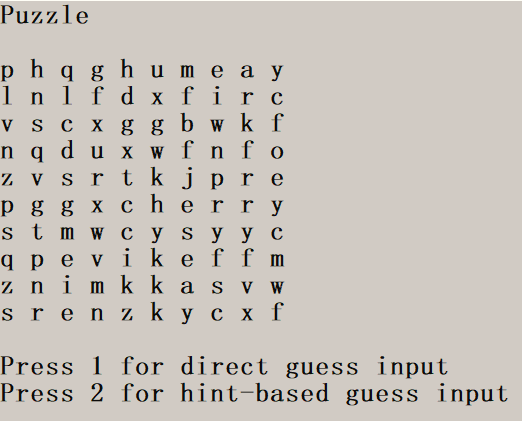
Dynamic Expansion: Vectors allow easy addition/removal of words.

String Storage: Suited for storing words as strings.

Standard Library: Offers convenient functions for dynamic arrays.

Use of vector<Player> and stack<int>:

Player Data: vector<Player> players stores player information, while stack<int> scoresStack tracks scores.

* **Reasons for Choice:**

Flexible Storage: Vectors can grow as new players join the game.

Sequential Scoring: Stack tracks scores in a LIFO (Last-In, First-Out) manner.

Rationale Behind 2D Array (Puzzle::matrix):

Storage of Puzzle Grid: The 2D array char matrix[10][10] represents the puzzle grid.

* **Reasons for Choice:**

Grid Representation: Efficiently stores the characters forming the puzzle grid.

Fixed Grid Size: For this game, a fixed-size grid is used (10x10).

* **Why These Data Structures?**

Vectors for Dynamic Storage: Ideal for handling variable-size collections (puzzle words, player data).

Stack for Sequential Scores: Efficient for tracking and managing player scores during gameplay.

2D Array for Grid Representation: Suited for the fixed-size puzzle grid.

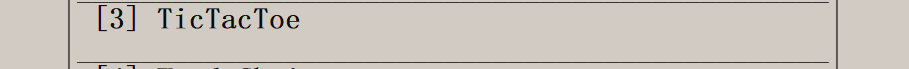
* **Why Not Other Data Structures?**

Arrays vs. Vectors: Arrays would require fixed sizes, limiting flexibility in handling dynamic elements.

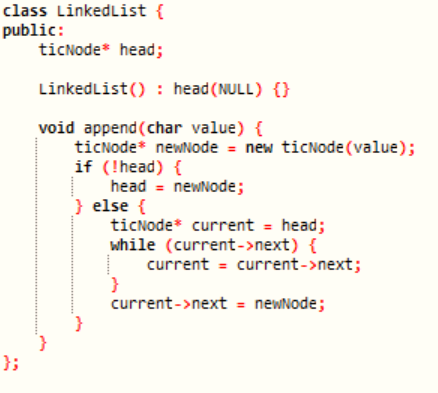
Linked Lists vs. Vectors: For the simplicity of linear access to player data and word storage, vectors suffice.

Queues vs. Stack: For scoring during gameplay, a stack provides a more straightforward solution.

Considering the nature of the game—dynamic word storage, sequential score tracking, and a fixed puzzle grid—these data structures offer a balance of efficiency, ease of use, and suitability for the specific requirements of the Word Puzzle game.

Tic TAC toe 

. All functions included in Tic Tac Toe and data structures used and why?

**LinkedList Class:**

LinkedList() Constructor:

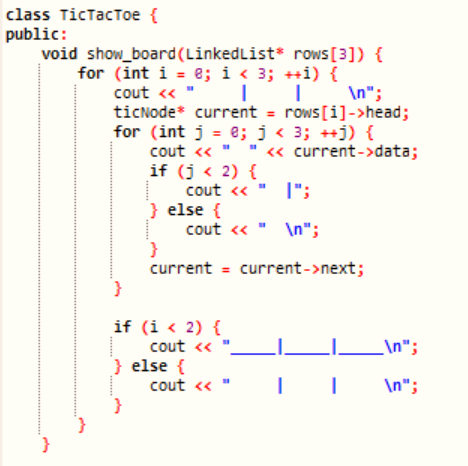
Initializes the linked list.

append(char value) Function:

Adds a new node at the end of the linked list.

**TicTacToe Class:**

**show\_board(LinkedList\* rows[3]) Function**:

Displays the Tic-Tac-Toe board using the linked list representation.

**get\_computer\_choice(LinkedList\* rows[3]) Function:**

Generates a random '0' move for the computer.

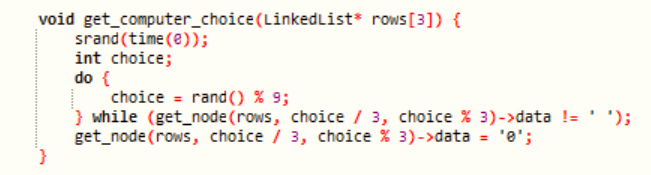
**get\_x\_player\_choice() & get\_0\_player\_choice() Functions:**

Accepts user input for 'X' or '0' moves on the board.

**count\_board(char symbol, LinkedList\* rows[3]) Function:**

Counts occurrences of a symbol ('X' or '0') on the board.

**check\_winner(LinkedList\* rows[3]) Function:**

Checks for a winner or a draw in the game.

**player\_vs\_player() & computer\_vs\_player() Functions:**

Manage different game modes: player vs. player or player vs. computer.

**get\_node(LinkedList\* rows[3], int row, int col) Function:**

Retrieves a specific node from the linked list grid.

**The 2D array (Puzzle::matrix)** represents the game grid, while the linked list (LinkedList) handles row-wise data management. These functions control user input, game logic, and game mode selection for Tic-Tac-Toe.

**Use of 2D Array (Puzzle::matrix):**

Storage of Game Grid: char matrix[10][10] represents the game grid for Tic-Tac-Toe.

**Reasons for Choice:**

Grid Representation: Efficient for maintaining the 3x3 game board structure.

Fixed Grid Size: Suitable for this game with a defined grid size.

**Use of Linked List (LinkedList and ticNode):**

Linked List for Rows: LinkedList manages rows of the Tic-Tac-Toe grid using a linked list of nodes (ticNode).

**Reasons for Choice:**

Dynamic Structure: Easily allows adding and deleting nodes in rows.

Sequential Access: Offers straightforward sequential access to elements within a row.

**Rationale Behind Using These Data Structures:**

2D Array for Fixed Grid: Ideal for maintaining a fixed-sized game grid for Tic-Tac-Toe.

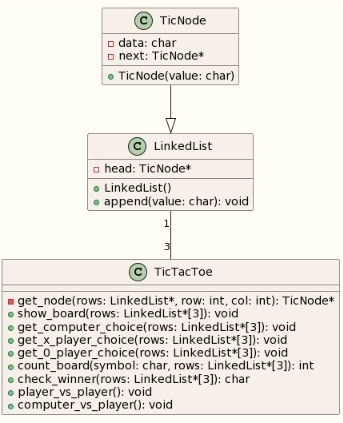
Linked List for Rows: Allows dynamic addition of elements for each row, offering flexibility in grid manipulation.

**Why Not Other Data Structures?**

Arrays vs. Linked Lists: Arrays could be used for the grid but might not be as flexible for dynamic changes as a linked list.

Linked Lists vs. Other Structures: Linked lists offer sequential access, aiding row-wise traversal, which fits well for grid structures.

Considering the nature of the Tic-Tac-Toe game—having a fixed grid but needing flexibility within rows for manipulation—the combination of a 2D array for the grid and linked lists for rows provides an efficient and balanced structure to manage the game state. Other structures might not offer the same level of flexibility or sequential access needed for this specific game's grid representation.

Class Diagram for TIC TAC TOE:

Thank

You

For

Your

Time