Mastermind STL
Project 1

Bryan Estrada

CSC-17C

Fall 2024

Project 1

11/10/2024

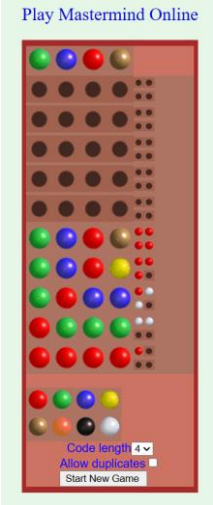
Table of Contents

Introduction

Mordecai Meirowitz, an Israeli postmaster, and telecommunications expert created the game Mastermind in 1970. Mastermind is a two-player code-breaking game. It bears similarities to a traditional pencil and paper game called Bulls and Cows, which might have existed for over a century. Mastermind is a game of deduction and logic, and it has been popular for decades due to its simple yet challenging gameplay.

How the Online Game Version Works

The game is played between two players: one player is an AI that creates a secret code, and the other player is a human user that tries to guess the code within a certain number of turns. The game consists of a decoding board and code pegs of different colors. The codemaker selects a secret code and places the pegs in a specific order on the board, hidden from the user. The user then makes a series of guesses, and after each guess, the AI provides feedback in the form of colored pegs. A red peg indicates a correct color in the correct position, and a white peg indicates a correct color in the wrong position. The game continues until the user guesses the correct code or until a predetermined number of turns have been exhausted.



My Approach to the Game

Similarities to the Board Game

My version of the game and the actual game are similar in a few ways. Both games are played by a user and an AI. The AI generates a random code according to user's settings (duplicates or not, code length). The user input their guess and the AI provides hints each turn. Both games consists of 10 turns. Lastly, both games offers an option for the user to read the rules of the games and how the game gives the user hints.

Differences from the Board Game

The games differ in the pegs. The online version consists of a code of color pegs, and the user has to guess the correct sequence of color pegs the AI generates. On the other hand, my program consists of numbers, the AI generates a sequence of digits from 1 – 8, and the user has to input their guess using numbers. Additionally, the game offers statistics of the game like the amount of victories and losses depending on the code length (4, 6, or 8) and if the user played with duplicates or not.

GitHub Repository Link

https://github.com/Bryan-EstradaC/CSC-17C-Project-1

Number of Lines

754 lines

Checkoff Sheet Contents

1. Container classes (Where in code did you put each of these Concepts and how were they used?
    1. Sequences (At least 1)
        1. ~~list~~
        2. slist
        3. bit_vector
    2. Associative Containers (At least 2)
        1. ~~set~~
        2. ~~map~~
        3. hash
    3. Container adaptors (At least 2)
        1. ~~stack~~
        2. ~~queue~~
        3. priority_queue
2. Iterators
    1. Concepts (Describe the iterators utilized for each Container)
        1. Trivial Iterator
        2. Input Iterator
        3. Output Iterator
        4. ~~Forward Iterator~~
        5. Bidirectional Iterator
        6. Random Access Iterator
3. Algorithms (Choose at least 1 from each category)
    1. Non-mutating algorithms
        1. for_each
        2. ~~find~~
        3. count

4. equal
5. search

2. Mutating algorithms
    1. copy
    2. Swap
    3. Transform
    4. Replace
    5. fill
    6. Remove
    7. ~~Random_Shuffle~~

3. Organization
    1. Sort
    2. Binary search
    3. merge
    4. inplace_merge
    5. ~~Minimum and maximum~~

The Logic of it All

Pseudocode

Start Game

Call setupGame()
Print welcome message

Repeat while playAgain is 'y' and quit is false:
Set endGame to false
Set skipTurn to false
Convert playAgain to lowercase

If playAgain is 'y':
Initialize turns with values from 1 to 10
Get valid code length and store in length
Get valid choice for duplicates and store in choiceDuplicate
Generate code and store in code
Print the generated code

Print instructions for the user

While the game is not ended, turns are available, and quit is false:

Reset skipTurn to false at the start of each turn
Print options to exit or show tutorial
Prompt the user to input their guess

If the input is "exit":
Call exitingGame to ask user for confirmation
Set skipTurn to true to skip the rest of the loop

If the input is "tutorial":
Show game instructions
Set skipTurn to true

If skipTurn is false:
Validate the guess input

If skipTurn is false:
Compare the guess with the code

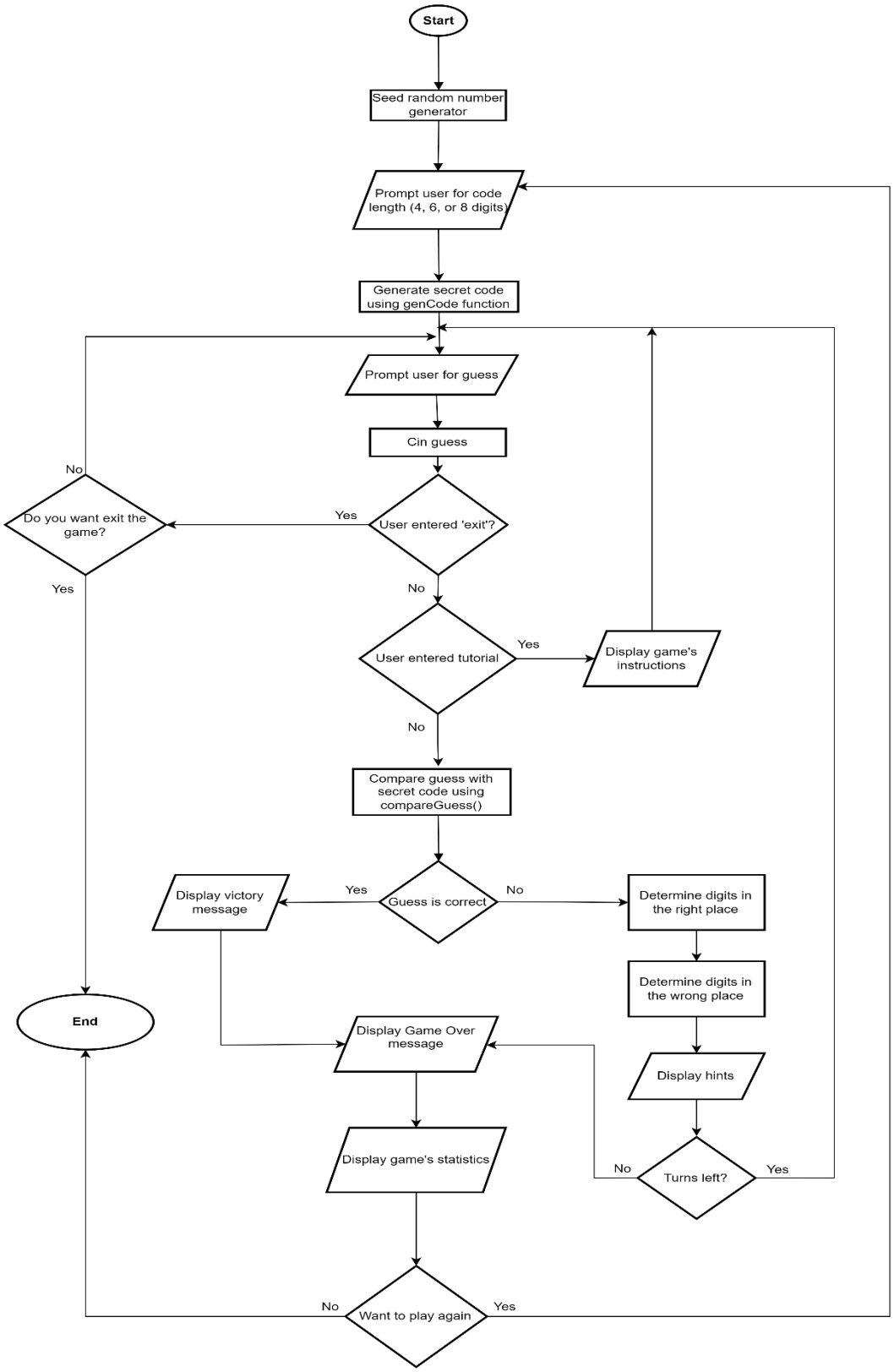
If the game is not quit:
Show game over message
Display statistics

Clear the code list

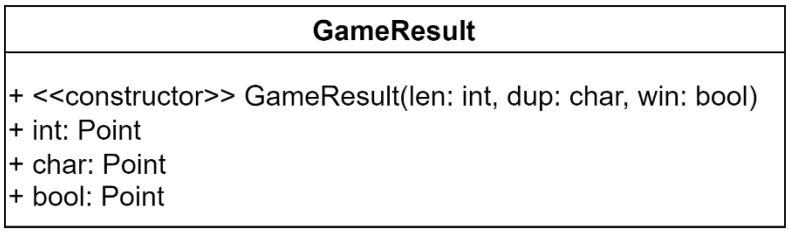
While playAgain is 'y' and quit is false

End Game

Flowchart


Start
Seed random number generator
Prompt user for code length (4, 6, or 8 digits)
Generate secret code using genCode function
Prompt user for guess
Cin guess
User entered 'exit'?
Yes
Do you want exit the game?
No
Yes
No
User entered tutorial
Yes
Display game's instructions
No
Compare guess with secret code using compareGuess()
Guess is correct
Yes
Display victory message
No
Determine digits in the right place
Determine digits in the wrong place
Display hints
Turns left?
No
Yes
End
Display Game Over message
Display game's statistics
Want to play again
No
Yes

UML Diagrams



List of Main Variables

STL:

Containers:

- **list<char> code:** A container that holds a string of chars that represents the secret code generated by the computer.
- **list<char> guess:** A container that holds a string of chars representing the guess entered by the user.
- **stack<int> turns:** Container that holds the number of turns that the user has left each time he or she enters their guess. Since it is a stack, it is initialized from 1 to 10 (turns). Then, 10 will be the element on the top of the stack, representing that the user will have 10 turns at the beginning of the game. When the user used one of their turns by entering a valid guessing, the program will pop the top turn from the stack until the user runs out of turns (turns = 0).
- **set<char> unique_number:** Container that holds unique numbers to help generate the code without duplicates. Since sets doesn't accept repeating elements, unique_number will hold numbers obtained from another STL container that is being shuffled until the code reaches the selected size by the user.
- **deque<char> numbers:** Container that holds the number from 1 to 8, and it is shuffled to generate a random code with duplicates or without duplicates.
- **map<char, int> code_count:** Container that holds elements from the user's guess that list<char> code also contains but user entered in the wrong position. code_count will contain the element as a key char, and the numbers of times the element appeared in the user's guess. This will help to count how many times an element was in the wrong from the user's guess even when the user's is playing with duplicates elements. Since maps only accepts unique elements, then code_count will hold the times a number was entered in the wrong position. code_count will help to give hints to the user on what possible numbers are entered in the correct or wrong position.
- **queue<GameResult> resultsQueue:** Container that holds a struct with the results of a game (length of the code, duplicates or not duplicates, and game win or lose). The program uses this container to store and print statistics of the user's games. For instance, it will help to store and print how many victories and losses the user has when he or she

has played with a code length of 4 with duplicates. Since queues operates in a FIFO type of arrangement, once the first element is stored and displayed, the element is popped from the back, leaving space for the next struct with the results of the next game.

Iterators:

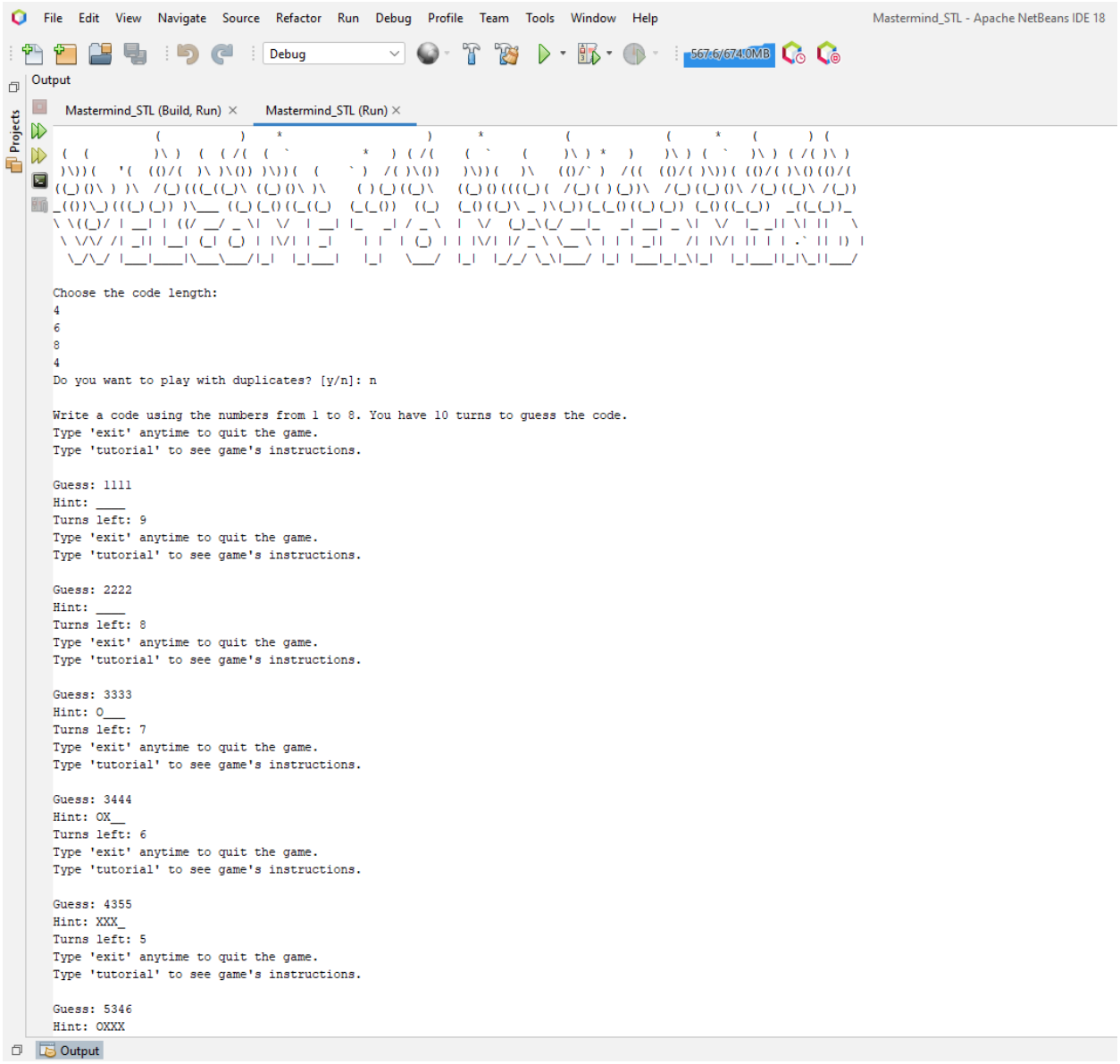
- **Forward iterator:** The program uses forward iterator when tries to access elements from the lists. In the function hint(), code_it and guess_it are initialized with the first element of the list containers code and guess, respectively. Since the function needs to compare each element of the containers, once the comparison is done, the iterator is incremented by one using the increment operator (++).

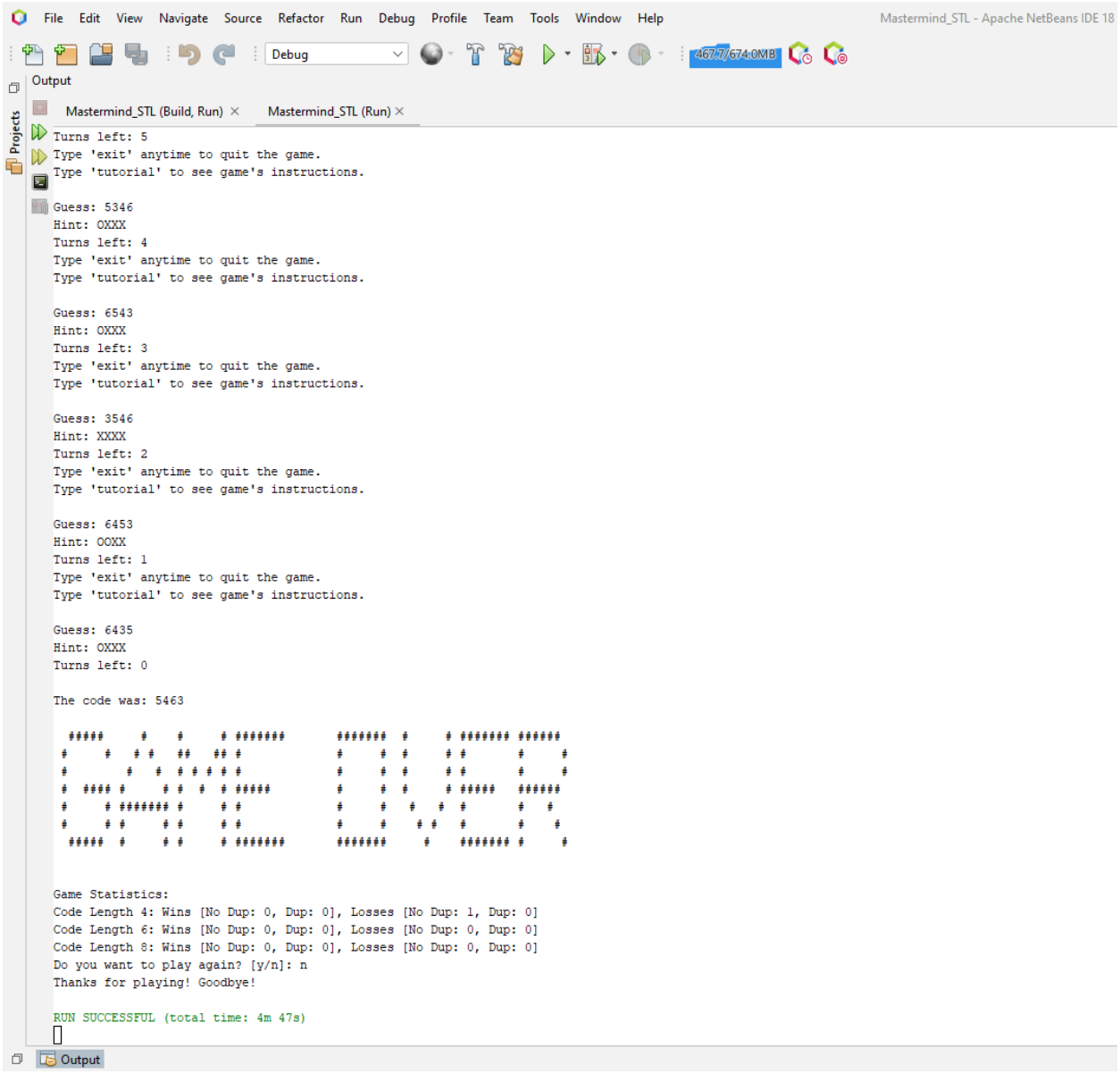
Algorithms:

- **find:** The program uses find() twice. find is used in the function genCode() to find an element in the set unique_numbers and determine if the element is not in the container, if it is not in the container, so the element can be stored in the list code when the user chose not duplicates. The second time find() is used, it is in the function hint() to find an element from the list guess, once the element is found, the function determines if the found element is in the list code.
- **Random_Shuffle:** It shuffles the elements of the container numbers in the function genCode() every time a while-loop runs, this will guarantee a better randomization of the elements stored in the list code, especially when the user chooses to play with duplicates.
- **max_element and min_element:** It determines what type of game the user has obtained more victories and fewer victories from (Duplicates or No-Duplicates) in the function displayStatistics().

Proof of a Working Product

This is an example of the program executing, the program gives accurate feedback.


File Edit View Navigate Source Refactor Run Debug Profile Team Tools Window Help
Mastermind_STL - Apache NetBeans IDE 18
Debug
567.6/674.0MB
Output
Mastermind_STL (Build, Run)
Mastermind_STL (Run)
Projects
Choose the code length:
4
6
8
4
Do you want to play with duplicates? [y/n]: n
Write a code using the numbers from 1 to 8. You have 10 turns to guess the code.
Type 'exit' anytime to quit the game.
Type 'tutorial' to see game's instructions.
Guess: 1111
Hint: ____
Turns left: 9
Type 'exit' anytime to quit the game.
Type 'tutorial' to see game's instructions.
Guess: 2222
Hint: ____
Turns left: 8
Type 'exit' anytime to quit the game.
Type 'tutorial' to see game's instructions.
Guess: 3333
Hint: O___
Turns left: 7
Type 'exit' anytime to quit the game.
Type 'tutorial' to see game's instructions.
Guess: 3444
Hint: OX__
Turns left: 6
Type 'exit' anytime to quit the game.
Type 'tutorial' to see game's instructions.
Guess: 4355
Hint: XXX_
Turns left: 5
Type 'exit' anytime to quit the game.
Type 'tutorial' to see game's instructions.
Guess: 5346
Hint: OXXX
Output



References

1. Dr. Lehr's github. https://github.com/ml1150258/CSC_7
2. Gaddis, Tony. *Starting Out With C++: From Control Structures through Objects*. 7th ed., Pearson, 2012.
3. Mastermind - Rules of the game. https://webgamesonline.com/mastermind/rules.php
4. Standard Template Library Programmer's Guide. http://209.129.8.7/~MarkLehrSyllabi/sgi-stl-docs/docs/
5. C++ STL Tutorial. https://www.geeksforgeeks.org/cpp-stl-tutorial/

Program

```
/***************************************
* Author    : Bryan Estrada            *
* Teacher   : Dr. Mark Lehr            *
* Class     : CSC-17C                  *
* Assignment: Project #1               *
* Title     : Mastermind with STL Library *
***************************************/

//Libraries
#include <iostream>
#include <cstdlib>   // Random Function Library
#include <ctime>     // Time Library
#include <string>
#include <set>
#include <list>
#include <stack>
#include <queue>
#include <map>
#include <utility>
#include <algorithm>
using namespace std;

//Structures
/*************************************************************
* STRUCT: GameResult
* ___________________________________________________________
* PURPOSE:
*    Holds the result of a single game of Mastermind,
*    including game settings and outcome.
*
* MEMBERS:
*    - int codeLength: Length of the code used in the game.
*    - char duplicateSetting: Character indicating whether
*                             duplicates are allowed ('Y' for
*                             yes, 'N' for no).
*    - isWin (bool): True if the player won the game, false
*                    if lost.
*
* CONSTRUCTOR:
*    GameResult(int len, char dup, bool win):
*       Initializes a GameResult with given code length,
*       duplicate setting, and win/loss outcome.
*************************************************************/
struct GameResult {
    int codeLength;
    char duplicateSetting;
    bool isWin; // true if user won, false if lost
    GameResult(int len, char dup, bool win)  {
        codeLength = len;
        duplicateSetting = dup;
        isWin = win;
    }
};
```

```
//Function prototypes
void setupGame();
char getDuplicateChoice();
int getCodeLength();
void genCode(int, list<char>&, char);
void printCode(const list<char>&);
void hint(const list<char>&, const list<char>&);
void showGameOverMessage(const list<char>&);
void showInstructions();
void validInput(const string&, bool&, const int&);
void compareGuess(list<char>&, const string&, const list<char>&, bool&,
                  stack<int>&, const int&, const char&, queue<GameResult>&);
void exitingGame(bool&);
void newGame(char&);
void recordResult(int, char, bool, queue<GameResult>&);
void displayStatistics(queue<GameResult>&);
void printWelcome();
void printGameOver();

/*************************************************************
* FUNCTION: main
*_____________________________________________________________
* PURPOSE:
*    The entry point for the Mastermind game. Initializes
*    game components, sets up the gameplay loop, processes
*    player input, and manages game flow, including player
*    statistics and exit conditions.
*
* LOCAL VARIABLES:
*    - queue<GameResult> resultsQueue: Holds results of each
*                                      completed game for
 *                                     later display.
*    - char playAgain: Indicates if the player wants to play
*                      another game ('y' or 'n').
*    - list<char> code: Stores the randomly generated game
*                       code.
*    - list<char> guess: Stores the player's current guess.
*    - char choiceDuplicate: Tracks whether duplicates are
*                            allowed in the code.
*    - int length: Represents the length of the code.
*    - stack<int> turns: Holds the number of turns (max 10)
*                        for each game.
*    - string guess_input: Stores player's input for each
*                          guess.
*    - bool quit: Flag to control game exit.
*    - bool endGame: Indicates if the current game is
 *                   complete.
*    - bool skipTurn: Skips the turn loop if necessary.
*************************************************************/
int main()
{
    queue<GameResult> resultsQueue;
    char playAgain = 'y';
    list<char> code;
```

```
list<char> guess;
char choiceDuplicate;
int length;
stack<int> turns;
string guess_input;
bool quit = false;  // Flag to control exit

setupGame();    //Setting up the random function
printWelcome();

do {
    bool endGame = false;
    bool skipTurn = false; // Flag to skip the turn without using `continue`
    playAgain = tolower(playAgain);

    if(playAgain == 'y') {
        for (int i = 1; i <= 10; i++) turns.push(i);

        // Get valid code length
        length = getCodeLength();

        // Get valid choice for duplicates
        choiceDuplicate = getDuplicateChoice();

        genCode(length, code, choiceDuplicate);
        cout << "\t\tCODE: ";
        printCode(code);
        cout << "\nWrite a code using the numbers from 1 to 8. You have 10 "
                "turns to guess the code.\n";

        while (!endGame && !turns.empty() && !quit) {
            skipTurn = false; // Reset skipTurn flag at the start of each turn
            cout << "Type 'exit' anytime to quit the game." << endl;
            cout << "Type 'tutorial' to see game's instructions." << endl;
            cout << "\nGuess: ";
            cin >> guess_input;

            // Check for exit command
            if (guess_input == "exit") {
                exitingGame(quit);
                skipTurn = true; // Skip rest of the loop for re-confirmation
            }

            if (guess_input == "tutorial") {
                showInstructions();
                skipTurn = true;
            }

            // First try-catch block: Check guess input
            if(!skipTurn){
                validInput(guess_input, skipTurn, length);
            }

            // Clear the previous guess and add the new one from input
            if(!skipTurn){
```

```
                    compareGuess(guess, guess_input, code, endGame, turns,
                                 length, choiceDuplicate, resultsQueue);
                }
            }

            if (!quit) {
                showGameOverMessage(code);
                displayStatistics(resultsQueue);  // Show statistics after each game
                newGame(playAgain);
            }
        }
        code.clear();
    } while (playAgain == 'y' && !quit);

    return 0;
}

/*********************************************************
* FUNCTION: genCode
*_______________________________________________________
* PURPOSE:
*    Generates a random code for the Mastermind game based on
*    the specified length and duplicate setting.
*
* PARAMETERS:
*    - int length: Desired length of the generated code.
*    - list<char>& code: Reference to a list where the
*      generated code will be stored.
*    - char choice: Character that indicates if duplicates
*      are allowed ('Y' for yes, 'N' for no).
*
* LOCAL VARIABLES:
*    - set<char> unique_numbers: Used to store unique
*      characters for codes with no duplicates.
*    - deque<char> numbers: Contains possible characters
*      ('1' to '8') for code generation.
*    - auto num_int: Iterator pointing to the start of
*      the numbers deque.
*_______________________________________________________
* RETURN:
*    Void: Modifies the 'code' list to hold the newly
*    generated random sequence of characters based on
*    the specified length and duplicate settings.
*********************************************************/
void genCode(int length, list<char>& code, char choice) {
    set<char> unique_numbers;
    deque<char> numbers = {'1', '2', '3', '4', '5', '6', '7', '8'};

    auto num_int = numbers.begin();

    while (code.size() < length) {
        random_shuffle(numbers.begin(), numbers.end());
        //char num = '1' + rand() % 8;
        if (toupper(choice) == 'Y' || unique_numbers.find(*num_int) ==
            unique_numbers.end()) {
```

```
            code.push_back(*num_int);
            unique_numbers.insert(*num_int);
            num_int++;
        }
    }
}

/*************************************************************
* FUNCTION: printCode
*  ___________________________________________________________
* PURPOSE:
*    Displays the generated code sequence by printing each
*    character in the code list.
*
* PARAMETERS:
*    - const list<char>& code: Reference to a list containing
*      the code characters to be printed.
*  ___________________________________________________________
* RETURNS:
*    Void: Outputs the code sequence to the console.
*************************************************************/
void printCode(const list<char>& code) {
    for (char num : code) {
        cout << num;
    }
    cout << endl;
}

/*************************************************************
* FUNCTION: hint
*  ___________________________________________________________
* PURPOSE:
*    Generates a hint to guide the player by indicating
*    the number of correct and misplaced digits in the guess.
*
* PARAMETERS:
*    - const list<char>& code: The actual code sequence the
*      player is attempting to guess.
*    - const list<char>& guess: The player's current guess of
*      the code.
*  ___________________________________________________________
* RETURNS:
*    Void: Outputs the hint string to the console, guiding
*          the player in future guesses.
*************************************************************/
void hint(const list<char>& code, const list<char>& guess) {
    map<char, int> code_count;
    int correct = 0;     // Counts correct positions (O's)
    int misplaced = 0;   // Counts misplaced digits (X's)

    // First pass: Check for correct positions and track remaining code digits
    auto code_it = code.begin();
    auto guess_it = guess.begin();
    while (code_it != code.end()) {
        if (*code_it == *guess_it) {
```

```
            correct++;  // Increment correct count for each matching position
        } else {
            code_count[*code_it]++;  // Track unmatched code digits for misplaced
checking
        }
        ++code_it;
        ++guess_it;
    }

    // Second pass: Check for misplaced digits (wrong position but correct digit)
    guess_it = guess.begin();
    code_it = code.begin();
    while (guess_it != guess.end()) {
        auto found = find(code.begin(), code.end(), *guess_it); // Search for the
guess in the rest of the code
        // Use find to check if *guess_it is in the remaining code characters
        if (*code_it != *guess_it && code_count[*found] > 0) {
                misplaced++;               // Increment misplaced count if digit is
in the wrong position
                code_count[*found]--;   // Decrement count to avoid double-counting
            }
        ++guess_it;
        ++code_it;
    }

    // Create the hint result string
    string hint_result(correct, 'O');   // Add all 'O's for correct positions
    hint_result += string(misplaced, 'X'); // Add all 'X's for misplaced digits
    hint_result += string(code.size() - correct - misplaced, '_'); // Add all '_'s
for incorrect digits

    // Print hint result
    cout << "Hint: " << hint_result << endl;
}

/*************************************************************
* FUNCTION: setupGame
*  ___________________________________________________________
* PURPOSE:
*    Initializes the random number generator with the current
*    time to ensure different random sequences in each game.
*  ___________________________________________________________
* RETURNS:
*    Void: Prepares randomization for generating elements.
*************************************************************/
void setupGame(){
    srand(static_cast<unsigned int>(time(0)));
}

/*************************************************************
* FUNCTION: getCodeLength
*  ___________________________________________________________
* PURPOSE:
*    Prompts the user to select a code length for the game
*    and validates the input, ensuring it is 4, 6, or 8.
```

```
*_____________________________________________________________
* RETURNS:
*    int: The validated code length chosen by the user.
*************************************************************/
int getCodeLength(){
    int length;

    do {
        try {
            cout << "Choose the code length: " << endl;
            cout << "4" << endl;
            cout << "6" << endl;
            cout << "8" << endl;
            cin >> length;
            if (cin.fail()){
                throw invalid_argument("Invalid input type. Please enter a number.");
            }
            if (length != 4 && length != 6 && length != 8){
                throw invalid_argument("Invalid code length. Please enter 4, 6, or
8.");
            }
        }
        catch (const invalid_argument& e) {
            cout << "Error: " << e.what() << endl;
            cin.clear();
            cin.ignore(100, '\n');
            length = 0;
        }
    } while (length != 4 && length != 6 && length != 8);

    return length;
}

/*************************************************************
* FUNCTION: getDuplicateChoice
*_____________________________________________________________
* PURPOSE:
*    Prompts the user to decide if duplicates are allowed in
*    the game code, validating the input as either 'y' or
*    'n'.
*_____________________________________________________________
* RETURNS:
*    char: The user's validated choice for duplicates ('y' or
*          'n').
*************************************************************/
char getDuplicateChoice(){
    char choiceDuplicate;

    do {
        try {
            cout << "Do you want to play with duplicates? [y/n]: ";
            cin >> choiceDuplicate;
            if (cin.fail()){
                throw invalid_argument("Invalid input type. Please enter 'y' or
'n'.");
```

```
            }
            choiceDuplicate = tolower(choiceDuplicate);
            if (choiceDuplicate != 'y' && choiceDuplicate != 'n'){
                throw invalid_argument("Invalid choice. Please enter 'y' or 'n'.");
            }
        }
        catch (const invalid_argument& e) {
            cout << "Error: " << e.what() << endl;
            cin.clear();
            cin.ignore(100, '\n');
            choiceDuplicate = '\0';
        }
    } while (choiceDuplicate != 'y' && choiceDuplicate != 'n');

    return choiceDuplicate;
}

/*************************************************************
* FUNCTION: showGameOverMessage
*  ___________________________________________________________
* PURPOSE:
*    Displays the end-of-game message, reveals the correct
*    code, and displays a game over message.
*
* PARAMETERS:
*    - const list<char> &code: The actual code used in the
*                              game, which is revealed to the
*                              player upon game over.
*  ___________________________________________________________
* RETURNS:
*    Void: Outputs the code and game-over message.
*************************************************************/
void showGameOverMessage(const list<char> &code){
    cout << "\nThe code was: ";
    printCode(code);
    printGameOver();
}

/*************************************************************
* FUNCTION: newGame
*  ___________________________________________________________
* PURPOSE:
*    Ask the user if he or she wants to play again. If the
*    the user responds 'y', the game will start over with a
*    new secret code. If 'n', the program ends.
*
* PARAMETERS:
*    - char &playAgain: User's choice ('y' or 'n')
*  ___________________________________________________________
* RETURNS:
*    Void: Outputs a message asking for a n0w game.
*************************************************************/
void newGame(char &playAgain){
    cout << "Do you want to play again? [y/n]: ";
    cin >> playAgain;
```

```
    playAgain = tolower(playAgain);
    if (playAgain == 'n') {
        cout << "Thanks for playing! Goodbye!" << endl;
    }
}

/*************************************************************
* FUNCTION: showInstructions
*  ___________________________________________________________
* PURPOSE:
*    Displays the instructions for the Mastermind game,
*    explaining the rules, the goal of the game, how guesses
*    and hints work, and how to enter valid inputs.
*  ___________________________________________________________
* RETURNS:
*    Void: Outputs the game's instructions to the console.
*************************************************************/
void showInstructions(){
    for(int i = 0 ; i < 80; i++){
        cout << "*";
    }

    cout << endl;
    cout << "*\t\t\tThis is Mastermind!" << endl << "*" << endl;
    cout << "*\tThe goal of the game is to guess the code the computer generated.";
    cout << endl << "*" << endl;
    cout << "*\tYou have 10 attempts to guess the code." << endl;
    cout << "*\tIn order to enter your guess, please type numbers from 1 to 8, "
            "\n*\taccording to the code size you selected (4, 6 or 8 digits).";
    cout << endl << "*" << endl;
    cout << "*\tFor every guess you entered, you will be given a hint in the form:";
    cout << endl << "*" << endl;
    cout << "*\tOOX_" << endl << "*" << endl;
    cout << "*\tThe symbols above represents the amount of digits in the right "
            "\n*\tposition, wrong position and incorrect digits from your guess: ";
    cout << endl;
    cout << "*\tO: One digit in the right position." << endl;
    cout << "*\tX: One digit in the wrong position. " << endl;
    cout << "*\t_: One incorrect digit." << endl << "*" << endl;
    cout << "*\tFor instance, if the secret code is '1234' and your guess was "
            "\n*\t'5247', the hint will be OX__, because '2' was in the right "
            "\n*\tposition, '4' was in the wrong position and '5' and '7' were "
            "\n*\tincorrect digits. As you can notice, the hint does not show you "
            "\n*\twhat digit's place was right, wrong or incorrect, it only shows "
            "\n*\tthe amount." << endl << "*" << endl;
    cout << "*\t\t\tHAPPY GUESSING! :D" << endl;

    for(int i = 0 ; i < 80; i++){
        cout << "*";
    }
    cout << endl;
}

/*************************************************************
* FUNCTION: validInput
```

```
*_____________________________________________________________
* PURPOSE:
*    Validates the player's guess input for correctness
*    in terms of format, length, and valid characters (1-8).
*
* PARAMETERS:
*    - const string& guess_input: The player's guess input to
*                                 validate.
*    - bool& skipTurn: A flag that, when set to true, skips
*                      the current turn if the input is
*                      invalid.
*    - const int& length: The expected length of the guess.
*_____________________________________________________________
* RETURNS:
*    Void: Outputs an error message and skips the turn if the
*          input is invalid.
************************************************************/
void validInput(const string &guess_input, bool &skipTurn, const int &length){
    try {
        if (guess_input.empty()){
            throw invalid_argument("Input cannot be empty. Please try again.");
        }
        if (!all_of(guess_input.begin(), guess_input.end(), ::isdigit)){
            throw invalid_argument("Guess contains invalid characters. Use only
numbers.");
        }
        if (guess_input.size() != length){
            throw invalid_argument("Guess length does not match the code length.");
        }
        for (char ch : guess_input) {
            if (ch < '1' || ch > '8')
                throw invalid_argument("Guess contains invalid numbers. Only use 1 to
8.");
        }
    } catch (const invalid_argument& e) {
        cout << "Error: " << e.what() << endl;
        skipTurn = true; // Skip turn if an invalid guess was made
    }
}

/************************************************************
* FUNCTION: compareGuess
*_____________________________________________________________
* PURPOSE:
*    Compares the player's guess to the generated code,
*    provides feedback through hints, and determines if the
*    game is won or lost.
*
* PARAMETERS:
*    - list<char>& guess: The list storing the player's
*                         current guess.
*    - const string& guess_input: The string containing the
*                                 player's guess.
*    - const list<char>& code: The list storing the generated
*                              code.
```

```
*    - bool& endGame: A flag that indicates if the game has
*                     ended.
*    - stack<int>& turns: A stack holding the remaining
*                         turns.
*    - const int& length: The length of the code/guess.
*    - const char& choiceDuplicate: A character indicating
*                                   whether duplicates are
*                                   allowed.
*    - queue<GameResult>& resultsQueue: A queue to store game
*                                       results.
*_____________________________________________________________
* RETURNS:
*    Void: Outputs the result of the guess, updates the game
*          status, and records the game result.
*************************************************************/
void compareGuess(list<char>& guess, const string& guess_input,
                  const list<char>& code, bool& endGame, stack<int>& turns,
                  const int &length, const char &choiceDuplicate,
                  queue<GameResult>& resultsQueue){
    guess.clear();
    for (char ch : guess_input){
        guess.push_back(ch);
    }

    if (code == guess) {
        endGame = true;
        recordResult(length, choiceDuplicate, true, resultsQueue); // Record win
        cout << "Congratulations!! You win !!" << endl;
        while(!turns.empty()){
            turns.pop();
        }
    } else {
        hint(code, guess);
        if (!turns.empty()) {
            turns.pop();
            cout << "Turns left: " << (turns.empty() ? 0 : turns.top()) << endl;
            if(turns.empty()){
                //cout << "\t\tTURNS ARE EMPTY" << endl;
                recordResult(length, choiceDuplicate, false, resultsQueue); // Record
loss
            }
        }

    }
}

/*************************************************************
* FUNCTION: exitingGame
*_____________________________________________________________
* PURPOSE:
*    Asks the player for confirmation to exit the game and
*    sets the quit flag if the player confirms.
*
* PARAMETERS:
*    - bool& quit: A reference to a flag that controls if the
```

```
*                  game loop continues or exits.
*_____________________________________________________________
* RETURNS:
*    Void: Exits the game if the player confirms, otherwise
 *         continues.
************************************************************/
void  exitingGame(bool &quit){
    char confirm;
    cout << "Are you sure you want to quit? [y/n]: ";
    cin >> confirm;
    if (tolower(confirm) == 'y') {
        cout << "Exiting game. Thanks for playing!" << endl;
        quit = true; // Set quit flag to exit loop
    }
}

/************************************************************
* FUNCTION: recordResult
*_____________________________________________________________
* PURPOSE:
*    Records the outcome of a single game (win/loss) and its
*    settings into a queue for tracking game results.
*
* PARAMETERS:
*    - int codeLength: The code's length used in the game.
*    - char duplicateSetting: The setting for duplicates ('y'
*                             or 'n').
*    - bool isWin: True if user won the game, false if lost.
*    - queue<GameResult>& resultsQueue: Reference to queue
*                                       where game results
*                                       are stored.
*_____________________________________________________________
* RETURNS:
*    Void: Adds the game result to the resultsQueue.
************************************************************/
void recordResult(int codeLength, char duplicateSetting, bool isWin,
                  queue<GameResult> &resultsQueue) {
    resultsQueue.push(GameResult(codeLength, duplicateSetting, isWin));
}

/************************************************************
* FUNCTION: displayStatistics
*_____________________________________________________________
* PURPOSE:
*    Displays the statistics of the game results, including
*    wins and losses for different code lengths (4, 6, 8) and
*    settings for duplicates, and compares the number of wins
*    with and without duplicates.
*
* PARAMETERS:
*    - queue<GameResult>& resultsQueue: A reference to the
*                                       queue containing the
*                                       results of previous
*                                       games to be analyzed.
*_____________________________________________________________
```

```
* RETURNS:
*    Void: Outputs game statistics to the console.
*************************************************************/
void displayStatistics(queue<GameResult> &resultsQueue) {
    static int wins_4[2] = {0, 0}, losses_4[2] = {0, 0};
    static int wins_6[2] = {0, 0}, losses_6[2] = {0, 0};
    static int wins_8[2] = {0, 0}, losses_8[2] = {0, 0};

    // Process results in the queue
    while (!resultsQueue.empty()) {
        GameResult result = resultsQueue.front();
        resultsQueue.pop();

        int dupIndex = (result.duplicateSetting == 'y') ? 1 : 0;
        if (result.isWin) {
            if (result.codeLength == 4) wins_4[dupIndex]++;
            else if (result.codeLength == 6) wins_6[dupIndex]++;
            else if (result.codeLength == 8) wins_8[dupIndex]++;
        } else {
            if (result.codeLength == 4) losses_4[dupIndex]++;
            else if (result.codeLength == 6) losses_6[dupIndex]++;
            else if (result.codeLength == 8) losses_8[dupIndex]++;
        }
    }

    // Output results
    cout << "\nGame Statistics:\n";
    cout << "Code Length 4: Wins [No Dup: " << wins_4[0] << ", Dup: " << wins_4[1] <<
"], "
         << "Losses [No Dup: " << losses_4[0] << ", Dup: " << losses_4[1] << "]\n";
    cout << "Code Length 6: Wins [No Dup: " << wins_6[0] << ", Dup: " << wins_6[1] <<
"], "
         << "Losses [No Dup: " << losses_6[0] << ", Dup: " << losses_6[1] << "]\n";
    cout << "Code Length 8: Wins [No Dup: " << wins_8[0] << ", Dup: " << wins_8[1] <<
"], "
         << "Losses [No Dup: " << losses_8[0] << ", Dup: " << losses_8[1] << "]\n";

    // Use max_element and min_element to find the most and least wins for duplicates
vs no-duplicates
    int wins[] = {wins_4[0] + wins_6[0] + wins_8[0], wins_4[1] + wins_6[1] +
wins_8[1]};
    int* maxWins = max_element(wins, wins + 2);
    int* minWins = min_element(wins, wins + 2);

    if(wins[0] != wins[1]){
        cout << "\nMore victories: ";
        if (*maxWins == wins[0]) {
            cout << "No Duplicates (" << *maxWins << " wins)" << endl;
        } else {
            cout << "With Duplicates (" << *maxWins << " wins)" << endl;
        }

        cout << "Fewer victories: ";
        if (*minWins == wins[0]) {
            cout << "No Duplicates (" << *minWins << " wins)" << endl;
```

```
        } else {
            cout << "With Duplicates (" << *minWins << " wins)" << endl;
        }
    }
}

/************************************************************
* FUNCTION: printWelcome
*  ____________________________________________________________
* PURPOSE:
*    Create a visually striking title screen for the game.
*  ____________________________________________________________
* RETURNS:
*    Void: Outputs the game's title to the console.
************************************************************/
void printWelcome(){
    cout << endl;
    cout << "              (           )    *                    )      *      "
            "     (            (      *    (       ) (      " << endl;
    cout << " (  (         )\\ )  (  ( /(  (  `          *   ) ( /(    (  `    ("
            "     )\\ ) *   )    )\\ ) (  `   )\\ ) ( /( )\\ )   " << endl;
    cout << " )\\))(   '(  (()/(  )\\ )\\()) )\\))(  (    ` )  /( )\\())   )\\))(
)"
            "\\   (()/` )  /((  (()/( )\\))( (()/( )\\()(()/(   " << endl;
    cout << "((_)()\\ ) )\\  /(_)(((_((_)\\ ((_)()\\ )\\    ( )(_)((_)\\
((_)(((((("
            "_)(  /(_)( )(_))\\  /(_)((_)()\\ /(_)((_)\\ /(_))  " << endl;
    cout << "_(())\\_)(((_)(_)) )\\___ ((_)(_)((_((_)  (_(_())  ((_)  (_)((_)\\ "
            "_ )\\\(_))(_(_)((_)(_)) (_)((_(_))  _((_(_))_   " << endl;
    cout << "\\ \\((_)/ | __| | ((/ __/ _ \\|  \\/  | __| |_   _| / _ \\  |  \\/
(_)"
            "_\\(_/ __|_   _| __| _ \\|  \\/  |_ _|| \\| ||   \\  " << endl;
    cout << " \\ \\/\\/ /| _|| |__| (_| (_) | |\\/| | _|    | |  | (_) | | |\\/| |/ "
            "_ \\ \\__ \\ | | | _||   /| |\\/| || | | .` || |) | " << endl;
    cout << "  \\_/\\_/ |___|____|\\___\\___/|_|  |_|___|   |_|   \\___/  |_|  |_/_/"
            " \\_\\|___/ |_| |___|_|_\\|_|  |_|___||_|\\_||___/  " << endl << endl;
}

/************************************************************
* FUNCTION: printGameOver
*  ____________________________________________________________
* PURPOSE:
*    Create a ASCII art-style game over message.
*  ____________________________________________________________
* RETURNS:
*    Void: Outputs the game's game over message.
************************************************************/
void printGameOver(){
    cout << endl;
    cout << "  #####     #    #     # #######       #######  #     # ####### ######
" << endl;
    cout << " #     #   # #   ##   ## #             #     #  #     # #       #     #
" << endl;
    cout << " #        #   #  # # # # #             #     #  #     # #       #     #
" << endl;
```

```
    cout << " #  #### #     # #  #  # #####         #     #  #     # #####   ######
" << endl;
    cout << " #     # ####### #     # #             #     #   #   #  #       #   #
" << endl;
    cout << " #     # #     # #     # #             #     #    # #   #       #    #
" << endl;
    cout << "  #####  #     # #     # #######       #######     #    ####### #     #
" << endl;
    cout << endl;
}
```