Assignment 2 :: Mastermind (Part A: Project Plan)

Fundamentals of C++ (FIT1048)

Author: James Jefferson Luu (30632749)
Tutor: Zac Lucarelli

Bachelor of Computer Science (C2001)
Monash University - Faculty of Information Technology

Due Date: 2019-09-07
Submission Date: 2019-09-07

Abstract

A white-faced figure appears on screen. As it speaks, the jaw moves as if a child was controlling it, or rather a psychotic engineer.

"Hello there, I want to play a game.

Fixed onto your skull is a death-trap. To disarm it, you will have to figure out a combination of colour letters on the wired tablet in front of you.

The death-trap can be as painful as a:
[0] *Stinger (Standard, 10 attempts, 4 letters, 6 selections, 3 clues)*
[1] *Amputator (Hard, 13 attempts, 5 letters, 8 selections, 2 clues)*
[2] *Decapitator (Very Hard, 15 attempts, 6 letters, 9 selections, 1 clue)*

Every time you get the combination wrong, a set of spikes will move closer towards your head. If you run out of attempts, the spikes will then dig into your brain (and you LOSE the game).

The only mercy you will get is markings on the tablet.
'!' means you have the right letter and position.
'?' means you have the right letter, but the wrong position.
Missing markings means that the letter isn't there at all.

You may also beg for a [*c*]*lue (depends on level), accept your* [*f*]*ate, play t*[*h*]*is recording again, or* [*q*]*uit the game.*

Live or die. Make your choice."

The above in italics is the supplementary story and game outline for my themed version of the Master Mind board game.

It will be stored in the text file `MasterMindIntro.txt` and loaded upon starting the program for the player to read before setting up the game.

Note: Throughout the following outline, "combination" and "code" are used interchangeably.

1 Development Outline for Master Mind

1.1 Game Classes and Functionalities

As per the brief, I have decided to use 3 classes in my game to represent the **Player**, the **Board**, and the **Application File** itself.

The **Player class** stores information relevant to the user player (the Code Breaker). This includes their name, score, the number of games played, and how many of those games they have won.
The use of this class simplifies my game code, as it separates those values from the application file.

The **Board class** contains details on the current game's board. This would be the number of rows (attempts that the player has), columns (length of the combination), the hidden combination, a set of letters the game uses to generate the combination, and the previous attempts of cracking the combination.

The **Application file (MasterMind)** contains the set difficulty level (extra), and all of the functions required for the game.
Through good programming practices, I will ensure that the game code is well-designed, and easy to test, debug, read and understand. How my code and game will be structured is covered throughout the rest of the outline.

1.2 Game Setup

- Load and display `MasterMindIntro.txt` and pause the game until the player presses "any key to continue".
- Initialise the game variables:
  - Ask for the player's name. The player's details will then be stored using the Player class to simplify the Application code.
  - Ask for the desired difficulty, which is assigned to the `skillLevel` variable (extra).
    This variable sets the `cluesLeft` variable, and the rows, columns and hidden code when sent to the Board class (see UML Diagrams).
  - Reset `previousMoves` in the Board class and generate a new string code for `hiddenCode` using `codeLetters` (see UML Diagrams).
  - Clear the screen in preparation for the main game loop.

1.3 Turns and Player Feedback Examples

After setting the game variables, the main game loop is loaded:

- A user interface resembling the following is generated by `displayBoard()` and displayed to the player.

```
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
                    Master Mind : Saw
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~


 /~~~~~~~~~~~~~~~~~\
 | Combo | _ _ _ _ |
 `-----------------'
 |     Guesses     |
 `-----------------'
 |  ____ | _ _ _ _ |
 \~~~~~~~~~~~~~~~~~/
```

- The player will then be asked to input their attempt to solve the combination (see 1.4).
- From here, the endgame conditions (see 1.5) are processed, if the player gets the combination wrong, the board updates to indicate which letters are correct or exist in the combination and unique feedback is given.

```
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
                    Master Mind : Saw
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~


 /~~~~~~~~~~~~~~~~~\
 | Combo | _ _ _ _ |
 `-----------------'
 |     Guesses     |
 `-----------------'
 |  ____ | _ _ _ _ |
 `-----------------'
 |  !___ | R R R R |
 \~~~~~~~~~~~~~~~~~/

 *Buzz* ... *Creak* (9 attempts left)

 Enter the combination:
```

- If there are still rows/attempts left on the board, the main game loop re-iterates for the next attempt.

1.4 Processing Player Input

I will include the three types of input methods that were covered throughout the unit's labs.

- Asking for a string input – used when asking the player for their name, their attempt to crack the code, or a game command (extra).
- Asking for a number input – used when setting the difficulty of the game (extra).
- Asking for a single letter input – used when asking if they want to play again, or to increase the difficulty level after winning five times in a row (extra).

All three methods are separate functions that use the same structure:

- `string askForString(string question);`
- `int askForNumber(string question);`
- `char askForLetter(string question);`

Based on the player's response, the game will perform the required action:

- The player's name will be sent to the Player class.
- The crack attempt will be sent to the Board class.
- The game commands will trigger a switch statement in the Application file (extra; see 1.6).
- The number will be used to set the combination, rows and columns of the board (sent to the Board class) and the amount of clues in a game (extra).
- The letter will determine whether the program exits or the game resets and starts again, as well as whether a higher difficulty level is set (extra).

1.5 Endgame Conditions

The game has two endgame conditions - when the player has guessed the combination correctly (and won), and if they have run out of attempts (and lost).

- After every guess the `checkGameOver()` function is called to determine if the endgame condition has been met.
- If NOT, then the main game loop continues (see 1.3).
- If it has, then the `gameOver()` function displays feedback depending on which condition is met and the player's stats. The player is then asked if they want to play again.

```
"Congraulations, you have beaten the game."

~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
 Subject:        James
 Pain Level:     Amputator
 Games Played:   1
 Times Survived: 1
 Game Score:     180
 Total Score:    180
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

"Do you want to play another? (y/n)" n

"Farewell, James."
```

- If the player has won five times in a row, then after they agree to play again, the game will prompt them if they want to proceed to the next difficulty level (extra).
- If the player has lost five times in a row, then before they are prompted to play again, the stats display will indicate a decrease of the difficulty level (extra).

1.6 Additional Features

Following the brief, I have decided to implement at least 6 of the 10 extra functionalities.

- **Incorporate a theme into the story and player feedback**
  This is already implied throughout the **Abstract** section.
  I plan for my version of Master Mind to be themed around the Saw film franchise, which involves a puppet and puzzles connected to death-traps. This would differentiate my Master Mind from the other versions online.

  I don't think this will be too difficult to implement using text files, strings and vectors.

  Examples of Feedback given can be seen in the **Turns and Player Feedback Examples** sub-section.

- **Implement selectable difficulty levels**
  This was also implied in the **Abstract** section.
  Information about the attributes the difficulty changes can be seen in the **Abstract** section and the **Game Setup** sub-section.

- **Implement an appropriate scoring system**
  This feature will be implemented differently from a traditional game of Master Mind due to the lack of a Code Maker player. The scores are determined at the end of the game from:

  - If the player has solved the code ($\times 1$, else $\times 0$).
  - The player's difficulty level
    ($\times 1$ for "Stinger", $\times 2$ for "Amputator", $\times 3$ for "Decapitator").
  - The amount of attempts left after solving the code
    (+20 per attempt).
  - Code Complexity (+10 per unique letter).
  - The amount of unused Clues (+10 per unused clue).

- **Player has additional command choices**
  This was also implied in the **Abstract** section.
  The main game loop will also accept single letter characters when asking for the player's code guess.
  From here, the player can use their clue/s to reveal a letter of the combination, give up on the current game, display `MasterMindIntro.txt` again, or quit the game entirely.

- **Display the board using ASCII art**
  The design of the board would resemble the following:

```
/~~~~~~~~~~~~~~~~~\
| Combo | _ Y _ _ |
`_________________'
|     Guesses     |
`_________________'
|  ____ | _ _ _ _ |
`_________________'
|  !!?? | R Y Y Y |
`_________________'
|  !!!_ | Y Y Y Y |
`_________________'
|  ____ | B B B B |
`_________________'
|  ____ | G G G G |
`_________________'
|  !___ | R R R R |
\~~~~~~~~~~~~~~~~~/
```

  The combination (shown when clues are used or the player loses) are displayed on the top row.
  The current turn is displayed on the row after the "Guesses" header, followed by the previous turns and how close they were to the combination.

- **Player promotion/demotion every 5 games won/lost**
  In addition to the difficulty levels, I also want to implement a promotion system that sets a higher/lower level every five sequential wins/losses. This will be tracked through the Application variable `gameStreak`, which increments upon solving a code, decrements (into negatives) upon running out of attempts, and resets when a streak is broken.

If I have enough time, I will also implement the following:

- **Allow the game to be saved and restored at the player's request**
  Although I might be able to implement this feature using by writing and reading to a text file called `savedMasterMind.txt`, the functions related to this feature (`saveGame()` and `loadGame()`) may probably take a lengthy amount of time to develop and test.

2 UML Diagrams

Board
- codeLetters : const static char*
- previousMoves : vector⟨string⟩
- hiddenCode : string
- boardRows : int
- boardColums : int
+ Board(skillLevel:int)
+ ~Board()
+ getMove(moveNum:int) : string
+ getCode() : string
+ getRows() : int
+ getColumns() : int
+ updateMoves(moveGuess:string) : void
+ resetBoard(skillLevel:int) : void

Player
- playerName : string
- playerScore: int
- playerWins : int
- playerGames : int
+ Player(name:string)
+ ~Player()
+ getName() : string
+ getScore() : int
+ getWins() : int
+ getGames() : int
+ updateScore(value:int) : void
+ updateGames(gameWon:bool) : void

MasterMind
skillLevel : int
cluesLeft : int
gameStreak : int
main()
displayBoard() : void
displayRules(fileName:string) : void
gameOver() : void
playGame() : void
resetGame() : void
askForString(question:string) : string
askForNumber(question:string) : int
askForLetter(question:string) : char
waitForPlayer() : void
checkGameOver() : bool
saveGame() : void (optional, see 1.6)
loadGame() : void (optional, see 1.6)