University of Nottingham
UK | CHINA | MALAYSIA

SCHOOL OF COMPUTER SCIENCE | YEAR 2018/2019 SPRING SEMESTER | 11.05.2019

C++ PROGRAMMING – COMP2045

BY JIAWEI LI

COURSEWORK 3 – REPORT

TIMUR MICHAEL CARSTENSEN | 20178811 | BIYTC4

TABLE OF CONTENTS

DESCRIPTION & PURPOSE

The game compiles with the following command on macOS 10.14.5:

```
g++ -std=c++17 main.cpp Game.cpp InputCompare.cpp
QuestionAlgorithm.cpp Timer.cpp Vocabulary.cpp -o game
```

The game also compiles on Microsoft Visual Studio 2019.

The game needs to be started with the following command line arguments:

```
./game PlayerName RoundNumber RoundDuration
```

RoundNumber and RoundDuration must both be positive integers greater than zero.

The game that I have designed is intended to be used as a revision exercise for the upcoming COMP2045 exam on the 21st of May. The user can add questions to the *vocabulary.csv* file in the following format:

- Question1,Answer1
- Question2,Answer2
- Etc.

The player needs to specify the player-name, the number of rounds to be played and the duration of each round as a command line argument when starting the game. When a player starts the game for the first time, he/she will be asked questions randomly and has to answer these in the set round duration. When the game finishes, the player's performance will be saved to the *player_config.csv* file. The performance is rated as a percentage. The lower the recorded percentage, the less the answer the user has given matches the correct answer.

If the user has played the game before, his previous performance will be loaded from the *player_config.csv* file. His previous performance will be grouped in the following brackets:

- 0 %– 25% → Bracket 1: Represented by attribute *QuestionAlgorithm.b1*
- 26% – 50% → Bracket 2: Represented by attribute *QuestionAlgorithm.b2*
- 51% – 75% → Bracket 3: Represented by attribute *QuestionAlgorithm.b3*
- 76% – 100% → Bracket 4: Represented by attribute *QuestionAlgorithm.b4*

The player will be asked the questions with the lowest performance rating first. Every time a question is asked which originates from the *player_config.csv file* the question will be removed from its corresponding *vector* in runtime. This ensures that the user will not encounter the same question in the next round.

Once all the *vectors* have been emptied, the game will use the **Vocabulary** classes' *returnRandomQuestionFromVocabulary()* and *returnQuestionIndex()* methods to randomly present the user with questions from the *vocabulary.csv* file.

When the game presents the player a question, a timer is started. Right after that, the *getline(std::cin, somestring)* function is called to record the players' answer.

Initially, I wanted to implement a timeout for *getline()* (E.g. once the timer runs out, the *getline()* would end). However, I found out that there is no portable solution for a *getline()*-timeout. As I developed this game on macOS, I was not able to implement a solution that would then compile on Linux. Thus, I devised the following workaround:

The user can take as long as he/she wants to input the answer. However, if he takes longer than the previously set round duration (in the command line argument when starting the game), instead of accepting the users' input, the game will enter an empty *std::string* as the input.

In the following step, the correct answer will be compared to the users' input (either the input from the user if he did not overstep the time limit or the empty s*td::string*) using the InputCompare classes' *fuzzyComparison()* method. As specified above, the method will return a percentage value as a double (between 0 and 1). At the end of the round, the performance as well as the index of the current question will be added to the *cfgElement* of the **QuestionAlgorithm** class (only index, performance pairs where the performance is less than 100% will be saved as there is no need to repeatedly ask the user questions that he/she is perfectly capable of answering already).

Once the game has finished, the *cfgElement* will be saved back to the *player_config.csv*.

RULES

There are no rules in particular that need to be followed. The user only has to use the format given above to start the game with the command line arguments. However, the user is highly encouraged to set the RoundDuration to a very low number (somewhere between 2 – 4 seconds) as this increases the difficulty of the game and should also increase the ability to learn the terms better.

EVALUATION

I believe that my game fulfills all of the "hard" constraints except for the displaying of results (in the form of a win, lose or tie). I do not think that it is necessary for the user to know how well they did in any given round or in one game as a whole. This is because the "AI" that determines which questions will be presented to the user is guiding the learning process. So, if the user answers a question poorly, the game will confront him/her with that question again, the next time he starts the game. This will continue happening until he/she improves his/her answer. Thus, over time, the user will be able to observe his own performance and learning progress based on which questions are asked repeatedly and which not.

Regarding the soft constraints:

- I have used inheritance and encapsulation.
- I added comments to all the source-code files.
- I cannot evaluate the complexity as I have nothing to compare with. However, I believe the game is quite complex.

- I have written a simple “AI” which determines which questions the user should answer. It mainly resides in the **QuestionAlgorithm** class.
- The game is very easy to start and has no rules to be learned. The players’ success (read: learning success/progress) is only dependent on their knowledge of the subject.
- An UML expression is included as *uml_expression.pdf*