CS 335 Exercise 5: Card Match

1 Introduction

The goal of this assignment is to build a stand-alone C++ program that will allow a user to play a card matching game using images of the standard 52–card deck (like the last assignment). The graphical user interface should support the basic elements of the game: play of the game itself, reset the game, quit the game, recording the number of guesses and matches so far, and detecting the winning condition.

2 Preparing the Class Repository

2.1 Update the Master Branch

Before you begin this exercise, make sure you return to the `master` branch by checking it out via SourceTree or the command line. To ensure you have all demo code available to you, after checking out, pull any changes your instructor might have posted since the last time you were on the master branch.

2.2 Create a New Branch and Project and Push to Personal

Create a new branch named `card-match`. Either use CLion to create a new project in a directory named `card-match` in the top-level of the class repository (`cs335`), or copy and rename the `qt-demos/griddemo` directory, and adjust the project and executable name in the `CMakeLists.txt` file accordingly. You can also work off your previous assignment, but do keep in mind that you will have to change your code considerably. Commit your new files and push the changes to your `personal` remote.

3 The Game

This version of *card matching* is played on a 4×13 board (4 rows and 13 columns) to accommodate all 52 cards. The elements of the 4××13 grid represent positions of cards on a playing board. The game starts with all positions of cards on the board turned face down (i.e. the card positions are defined, but turned face down, each showing a common image of the card back), There is a *guesses made* counter at 0 and a *matches made* counter at 0 in the application's status bar. The goal is for the player to discover matching cards, where a match is defined as two cards of the same rank. Since there are always 4 cards at the same rank across the four suits (i.e. there are four Queens: Queen of Diamonds, Queen of Hearts, Queen of Spades, and the Queen of Clubs. The player must find two pairs for every rank, pairing up 26 pairs to end play and win the game. Suits don't matter in the match, only rank. An image of the game is shown in Figure 1.

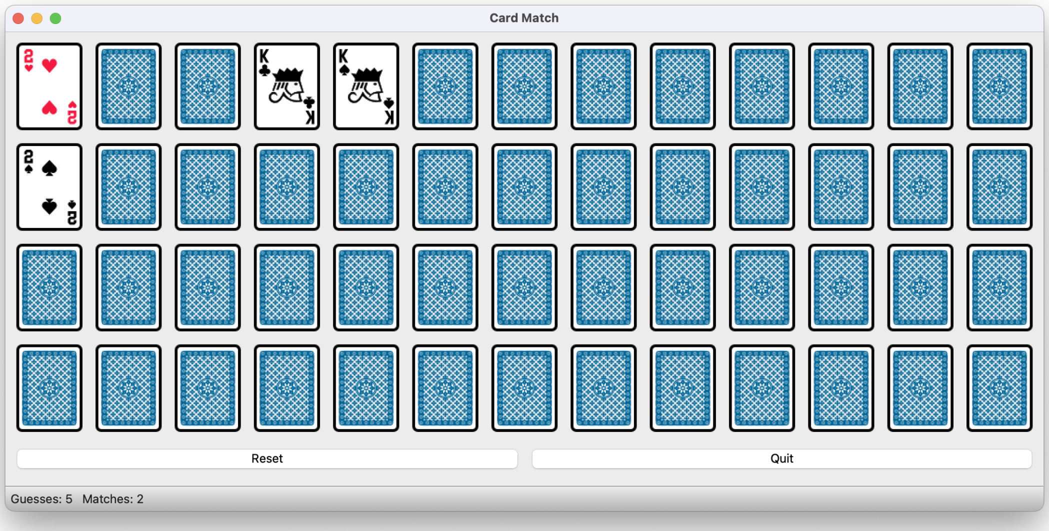

Figure 1: Card Matching Game

Play is as follows:
The GUI actions should function as follows:

- The player makes a guess by clicking with the mouse on one card. The game should toggle that card's state to reveal the face of the unique card in that position.
- The player makes a second guess by clicking on another face–down card with the mouse, trying to match the rank of the first card.
  - If the second card reveals a match with the first, the cards are turned face–up and permanently displayed: the cards remain unhidden and are no longer responsive to clicks. This causes the *matches made* counter to be incremented.
  - If the two selected cards do not match — *after three seconds* — both cards are toggled back to show *face–down* (i.e. the card back) *automatically.*
- The *guesses made* counter is incremented by 1 (in either case —- a match or a miss).
- The game ends when one of these things occurs:
  - All cards are revealed and therefore matched. A winning message should be displayed before the game is then reset.
  - The player quits.
  - The player resets the game.

4 The Approach and Implementation

Signals and slots will be crucial for this game. When a card's button is clicked, the button's `clicked()` signal can be connected to a card's `flip()` slot. The card can, in turn, emit a `flipped()` signal, which can be connected back to its button. Since the button needs to update its icon, the card can send a pointer of itself (i.e. `this`) along with the signal. Note that `QPushButton::setIcon()` is not a slot, so you may consider creating a new class that inherits from `QPushButton` and wraps the member function in a slot.

Note that the reset option will need to shuffle the cards and reestablish the connections with the buttons. In this case you must use `QObject::disconnect()` before the shuffling, and reconnect buttons and cards afterwards.

Also consider using a `QTimer` object to control the automatic hiding of non–matching card guesses (after 3 seconds). If there is no match, disable signals for all the card buttons, set a timer to go off in three seconds, and when it does, thank the timer for its service, hide the two cards that were not a match, and then re–enable all user events. Without disabling events while waiting for the timer to fire, the game flow can become weird if the user clicks again to start a match before the non–matching cards are hidden. You can use the `QObject::blockSignals()` member function to achieve this.

You will have to keep track of the selected cards, as well as which cards have been matched and which cards are still unmatched. For the matched and unmatched cards, I recommend a `std::set`. Your task will be considerably easier if you also keep track of which cards refer to which button. A `std::map` with cards as keys and buttons as values will allow you to do this.

The `griddemo` program introduced you to smart pointers in general, and the `std::unique_ptr` in particular. Since your maps and sets to keep track of card status require sharing of pointers, you can use `std::shared_ptr` in this case. Feel free to simply use raw pointers if you are uncomfortable with C++'s smart pointer concept.

You will have to define and connect quite a bit of signals and slots for this game. Remember that a circular connection of signals and slots will not necessarily lead to a stack overflow due to infinite recursion, but will still manifest itself in an infinite loop, which will render the program unusable. Breaking cycles like this temporarily can also be achieved using either `QObject::blockSignals()` or by manually disconnecting and reconnecting signals and slots as needed.

You will find that debugging event–based programs is fundamentally different from and more tedious than debugging sequential programs. I encourage you to liberally use assertions throughout your code to express your belief of what should and should not be true at a particular point of your program. The controlled crash caused by a failed assertion will definitely help you track down simple oversights (like a forgotten state update) as well as serious design issues (such as dereferencing a null pointer).

5 Submit Your Work

Commit your changes locally. Again, make sure to use an appropriate commit message. With your working directory being the top–level of the repo, create the archive with:

```
$ git archive -o card-match.zip HEAD card-match
```

Then upload the `card-match.zip` file to canvas. Submit the pull request (repo URL and commit ID) as the upload comment.

6 Work Summary

- Push the class repo to a personal remote
- Create the `card-match` branch
- Add a new project in the directory `card-match`
- Write the card matching game code
- Commit and push your work to the personal remote
- Submit the pull request and archive to Canvas