CSI2372

Project

Fall 2018

A Memory Game

In this year's project, you are asked to implement a memory card game called Memoarrr! as a console game. Various reviews and discussion of the games can be found on sites such as "The Opinionated Gamer" or "Board Game Geek".

The following specification is for the base game but you will also have to program two advanced variants of the game. In general, there are 25 memory cards that will show a combination of an animal and a background colour. There are five different animals: crab, penguin, octopus, turtle and walrus and five different background colours: red, green, purple, blue and yellow for a total combination of 25 cards. The game can be played with 2-4 players.

In the physical game, the cards are placed face down in a 5 times 5 square but the center position remains free for the volcano and treasure cards. (Note that means one of the regular animal card is not in the game). In our adaption, we don't use volcano or treasure cards but simply leave the center position empty. The players take turns to uncover a card where the uncovered card has to match the previously uncovered card's animal or background colour. The uncovered cards remain on the table face up. If the player uncovers a card that doesn't match the previously uncovered card, the player is out of the current round. A round of play ends with only one player remaining who then wins the current round and receives 1-4 rubies randomly. If there are no more cards to turn over then the players still in the game continue to take their turns but lose until only one player is left who wins the round. After the current round the cards remain in place but are turned face down again. After seven rounds, the game ends and the player with the most rubies wins. In the physical game there are 3 cards with one ruby, 2 cards with 2 rubies, and 1 card with 3 rubies as well as 1 with 4 rubies.

Base Game

In the base game, the rules as described above are used and we will display the card with a 3 x 3 array of characters and with a space of one character and row between cards hence the overall board is 19 rows by 19 characters. The animals and the background colour will be identified by their first letter in capital and small caps receptively. For example, the following symbolizes a **W**alrus card with **y**ellow background:

```
yyy
yWy
yyy
```

Cards that are not yet uncovered will be marked by all small caps z. An example of the game board with four cards uncovered is shown below. The position of the cards is marked with a letter for the row and a number for the column. The current round may have progressed A1 D1 B4 D3.

```
  yyy zzz zzz zzz zzz
A yWy zzz zzz zzz zzz
  yyy zzz zzz zzz zzz

  zzz zzz zzz bbb zzz
B zzz zzz zzz bPb zzz
  zzz zzz zzz bbb zzz

  zzz zzz     zzz zzz
C zzz zzz     zzz zzz
  zzz zzz     zzz zzz

  yyy zzz bbb zzz zzz
D yPy zzz bTb zzz zzz
  yyy zzz bbb zzz zzz

  zzz zzz zzz zzz zzz
E zzz zzz zzz zzz zzz
  zzz zzz zzz zzz zzz

   1   2   3   4   5
```

There can be 2-4 players. The game starts by each player looking secretly at the three cards directly in front of them. The ruby score of each player will only be revealed at the end of the game.

Expert Display Mode

In this version, the rules are the same as in the base mode but the board with the array of cards is removed. Instead only the face up cards are shown in a row along with the position they came from. Example:

```
yyy yyy bbb bbb
yWy yPy bPb bTb
yyy yyy bbb bbb
```

A1 D1 B4 D3

Expert Rules Mode

In this version, the cards or rather have some added meaning. When an octopus card is turned over, the card is exchanging position with an adjacent card in the same row or the same column (4-neighbourhood). The adjacent card may be face up or down and will remain unchanged. If a player turns over a penguin, then this player is allowed to turn a face-up card face-down. If the penguin is the first card turned up, no special action will take place. The walrus enables a player to block a face down card for the next player and hence the next player must choose another location. The player who turns over a crab card must immediately turn over another card. If that card does not fit, the player loses the current round. Finally, with the turtle the next player in the current round is skipped, i.e., the next player will not turn over a card and will not lose in the next step.

Your game implementation must allow for the combination of the expert modes.

Implementation

The implementation for the base game is specified below at the level of the public interface of the classes. You may add any private or protected method that you wish. Each class needs to provide a test driver that will allow to test the class without using the main loop. One must be able to turn the test driver on and off with a compile switch, e.g., testing the class Board must have a test driver in board.cpp that can be turned on and off with #DEFINE TEST_BOARD_. For testing, you need to create a version of your project that does not include main but all other files with the corresponding preprocessor switch enabled.

You will need to design the expert modes. Your implementation will be marked on the maintainability and extensibility of your code. As such, you will have to avoid code duplication, case statements and branching as much as possible and use instead generic (e.g., templates and automatic type derivation), object-oriented design patterns (e.g., class hierarchies) and the standard template library. The marks associated with each part of your implementation are noted in [] below.

Main Loop Pseudo Code [3]

```
Ask player to choose game version, number of players and names of
players.

Create the corresponding players, rules, cards and board for the game.

Display game (will show board and all players)

while Rules.gameOver is false

    update status of cards in board as face down

    update status of all players in game as active

    for each player

        Temporarily reveal 3 cards directly in front of the player

    while Rules.roundOver is false

        # next active player takes a turn

        get selection of card to turn face up from active player
```

```
                update board in game
                if Rules.isValid(card) is false
                    # player is no longer part of the current round
                    current player becomes inactive
                display game
        Remaining active player receives reward (rubies)
print players with their number of rubies sorted from least to most
rubies
print overall winner
```

Player [2]

Design a class Player which combines all information for a player including name, side of the board (top, bottom, left or right) and has the current count of rubies. The class should have the following public methods:

- string getName() const return the name of the player.
- void setActive(bool) set the status of the player as active or inactive.
- bool isActive() returns true if the player is active.
- int getNRubies() const return the number of rubies won by this player.
- void addReward(const Reward&) add a reward with a given number of rubies.
- void setDisplayMode(bool endOfGame)
- Side Player::getSide() and void Player::setSide(Side) where Side is a class enumeration of { top, bottom, left, right}

A player must be printable with the insertion operator cout << player. An example print out with endOfGame false could look as follows:
Joe Remember Doe: left (active)

Once endOfGame is true:

Joe Remember Doe: 3 rubies

Card [1.5]

Design a class Card which can take a face of one of the five possible animals and one of the five background colours. A card must also be "printable" as one string per row with the method:

```
Card c(Penguin,Red); // This constructor will be private
for (int row = 0; row <c.getNRows(); ++row ) {
     std::string rowString = c(row);
     std::cout << rowString << std::endl;
}
```

Note that `Penguin` and `Red` are enumeration values of type `FaceAnimal` and `FaceBackground.`

An object of type Card can not be copied or assigned and needs a private constructor but will give CardDeck (see below) friend access. The public interface of Card is to include conversion operators to type FaceAnimal and FaceBackground.

Reward [1]

Design a class Reward which can take one of four possible values from 1 to 4 rubies. A reward must also be printable with the insertion operator cout << reward.

An object of type Reward can not be copied or assigned and needs a private constructor but will give RewardDeck (see below) friend access. The public interface of Reward is to include conversion operators to type int returning the number of rubies.

Deck<C> [2]

Design a class Deck<C> as an abstract factory class that will be used to create a set of cards or a set of rewards. The type parameter <C> is intended to be one of {Card|Reward}. The class will need the following methods:

- void shuffle() shuffles the cards in the deck. You must use the function std::random_shuffle from the standard template library.
- C* getNext() returns the next card or reward by pointer. Will return nullptr if no more cards or rewards are available.
- bool isEmpty() const returns true if the deck is empty.

CardDeck [2]

Design a class CardDeck derived from Deck<Card>.

- static CardDeck& make_CardDeck() is the only public method for CardDeck. The method has to always return the same CardDeck during the execution of the program.

An object of type CardDeck can not be copied or assigned, and it has no public constructor.

RewardDeck [2]

Design a class RewardDeck derived from Deck<Reward> with the corresponding properties to CardDeck.

Board [2]

Design a class Board which holds an array of strings corresponding to the screen display of the game.

- bool isFaceUp(const Letter&, const Number&) const returns true if the card at a given position is face up. Letter and Number are enumerations. Throws an exception of type OutOfRange if an invalid Letter and Number combination was given.

- bool turnFaceUp(const Letter&, const Number&) changes the state of the specified card return false if card was up already. Throws an exception of type OutOfRange if an invalid Letter and Number combination was given.
- bool turnFaceDown(const Letter&, const Number&) changes the state of the specified card return false if card was down already. Throws an exception of type OutOfRange if an invalid Letter and Number combination was given.
- Card* getCard(const Letter&, const Number&) returns a pointer to the card at a given location. Throws an exception of type OutOfRange if an invalid Letter and Number combination was given.
- void setCard(const Letter&, const Number&, Card*) updates the pointer to card at a given location. Throws an exception of type OutOfRange if an invalid Letter and Number combination was given.
- void reset() changes the state to all cards to be face down.

A board must be printable with the insertion operator cout << board. The constructor for board should throw and Exception of type NoMoreCards if there are no more cards available to construct a board.

Game [2.5]

Design a class Game that encapsulates the current state of the game and it will have a class variable of type Board. It is responsible to print the current state of the game.

- int getRound() returns a number between 0 and 6 corresponding to the current round of the game
- void nextRound() increments the number of rounds and resets the Game state for the next round.
- void addPlayer(const Player&) adds or updates a player. If Player is already in Game, the player is updated, and if not it is added.
- Player& getPlayer(Side)
- const Card* getPreviousCard()
- const Card* getCurrentCard()
- void setCurrentCard(const Card*)
- Card* getCard(const Letter&, const Number&) which calls the corresponding method in Board.
- void setCard(const Letter&, const Number&, Card*) which calls the corresponding method in Board.

A game must be printable with the insertion operator cout << game. It should display the board and all players.

Rules [2]

Design a class Rules which has the main purpose to check if a selection of a player is valid.

- bool isValid(const Game&) returns true is previous and current card match; false otherwise.
- bool gameOver(const Game&) returns true if the number of rounds has reached 7.
- bool roundOver(const Game&) returns true if there is only one active Player left.
- const Player& Rules::getNextPlayer(const Game&)

The remaining marks are for the expert modes. Expert Display [2] and Expert Rules [4].