Systems Programming – Assignment #1

Reviiyot

2016/Fall

1 Before You Start

- It is mandatory to submit all of the assignments in pairs. It is recommended to find a partner as soon as possible and create a submission group in the submission system. Once the submission deadline has passed, it will not be possible to create submission groups even if you have an approved extension.
- Read the assignment together with your partner and make sure you understand the tasks. Please do not ask questions before you have read the whole assignment.
- Skeleton classes will be provided on the assignment page, you must use these classes as a basis for your project and implement (at least) all the function that are declared in them.

KEEP IN MIND

While you are free to develop your project on whatever environment you want, your project will be tested and graded ONLY on a CS LAB UNIX machine. It is your own responsibility to deliver a code that compiles, links and runs on it. Failure to do so will result in a grade 0 to your assignment.

Therefore, it is mandatory that you compile, link and run your assignment on a lab unix machine before submitting it.

We will reject, upfront, any appeal regarding this matter!!

We do not care if it runs on any other Unix/Windows machine.

Please remember, it is unpleasant for us, at least as it is for you, to fail your assignments, just do it the right way.

2 Assignment Goals

The objective of this assignment is to design an object-oriented system and gain implementation experience in C++ while using classes, standard data structures and unique C++ properties such as the “Rule of 5”. You will learn how to handle memory in C++ and avoid memory leaks. The resulting program must be as efficient as possible.

3 General Assignment Definition

In this assignment you will write a C++ program that plays "Reviiyot"(רביעיות).

- 2 or more players, each plays according to a given strategy. (See details below)
- (N+3)*4 cards (N>=1). An individual card consists of a VALUE and a SHAPE. The values are: 2,3,4,…,N, Jack, Queen, King and Ace. The shapes are Spade(S), Diamond(D), Club(C) and Heart(H). Each value has all 4 shapes. (It is as a standard card deck but with unlimited rank/value. For example a deck of N=12 consists of 2s,3s,...,12s, Jacks, Queens, Kings and Aces. A deck of N=1 has Jacks, Queens, Kings and Aces only. In both cases each value has all 4 shapes → Total for N=12: (12+3) * 4=60 cards, for N=1: (1+3)*4=16 cards.

- Rules:
  1. Each player has a unique name and a position around the table (Ex: Sarah at position 1, Yossi at position 2, Ruth at position 3, defined by the order at the input configuration file, see below). In the rest of this document we will use the pseudo names: Player1, Player2, etc. according to their allocation order.
  2. Each player gets 7 cards from the shuffled deck of cards, starting with Player1. All 7 cards are distributed at once before moving to the next player.
  3. A player that received 4 cards of a single value discards the set from his hand before the first play of the first player.
  4. Player1 plays first followed by Player2, Player3, …
  5. At his/her turn, the player, according to his/her strategy, asks another player for a card of a specific value (2-N, Jack, Queen, King and Ace). The asking player can

ask only for values that he/she holds at least one card of that same value. If the asked player holds K(>0) cards of that value he/she has to give all of his/her cards of that value to the asking player and draw K (see 6) new cards from the top of the deck. If the asked player doesn't have any card of that value the asking player picks up one card from the top of the deck.

6. A player draws card(s) from the deck ONLY if his/her hand is not empty, regardless if he/she is the asking or the asked player.
7. Any player that has all 4 cards of a single value discards the set from his hand. Be sure to handle all possible ways to complete a set.
8. The turn moves on to the next player after each turn regardless of the outcome of the current player.
9. If the deck of cards is empty the game continues without drawing new cards.
10. Each turn is played to its completion, i.e. asking player and asked player get to fulfill their actions (steps 5-7) – Hand over cards, draw cards, discard sets.
11. The first player(s) that gets rid of his/her cards wins and the game ends. Be aware that two players can win the game simultaneously.

4 Implementation requirements

Your implementation must include at least the following classes:

Card:** a single playing card. This class is abstract and holds the shape of the card only.

FigureCard: Inherits from **Card**. Used for cards with a figure on. That is: Jack, Queen, King and Ace.

NumericCard: Inherits from **Card.** Used for cards with numeric values. (2, 3, 4, 5, etc…)

** A card value is the combination of its value and its shape, when “value” is the most significant value while “shape” is the least significant value. Ex: 23C and not C23

Hand: one player's Cards. This is a collection (List, Hash Map, Vector - your choice) of **Card** objects representing the cards held by a single player. It has to include appropriate methods to add and remove cards, search for cards in the hand, getNumberOfCards and a toString function that returns a sorted hand where

2 < 3 < … < N < J < Q < K < A and C < D < H < S and
2C < 2D < 2H < 2S < 3C < 3D < 3H < 3S < ... < AC < AD < AH < AS

Player: This class is abstract, derives from **Hand,** and holds one player's info. A player has a name. It should include appropriate methods to add and remove cards, search for cards in his/her hand, getNumberOfCards and a toString function that returns the name, and the sorted hand (as describes in Hand).

PlayerType1, PlayerType2, PlayerType3, PlayerType4: Derives from **Player**: This class represents a player with a specific playing strategy (1, 2, 3, or 4).

Deck: This class represents the deck of cards. This is also a collection of **Card** objects with methods to deal cards, toString function that returns the number of cards in the deck and the cards from top to bottom.

Game: Holds the players and the deck. Handles the flow of the game, user input, output, error handling etc… **In order to take a snapshot of the game at any given time, you must provide a copy constructor of this class**. **Game** will also implement the **printState()** function. This function should be implemented with extra care, since it produces the expected output by which your work will be evaluated**. This function prints the cards remaining in the deck from top to bottom in a single line, followed by the name and sorted hand of all players (a single line per player).

Example:

```
Deck: JH 3C KS JC JD 3S JS 3D 3H
Ami: 2H KC KD AS
Tami: 2C 2D 2S KH AC AD AH
```

** 1. For your convenience, extra spaces and empty lines will be ignored.
2. The output is case sensitive. Ex: “3D” and not “3d”.

It is highly recommended to start by implementing the **Card** class and testing it, then the **Deck** class followed by **Hand** and finally the **Player**(s) and the **Game**. It is recommended to have a test driver (testing code) for each class (don’t submit them).

* * *

Main: Your main function is in **reviiyot.cpp** that is provided on the assignment homepage. You must use it and DO NOT change it.

makefile: You will need to write your own makefile.

5 Class Skeletons

The following files will be provided for you on the assignment homepage:

Card.h

Hand.h

Player.h

Deck.h

Game.h

You are required to implement the supplied functions. You are also free to add your own functions and variables to these headers as needed. We might use these functions to test your code, therefore any attempt to change their decorations might result in a compiling error and a 0 grade.

6. Player Strategies

PlayerType1: Requests the value he has the **most** from the player with most cards at hand. In case he has two or more values with the same highest amount, choose the **higher value**. In case two players have the most amount of cards, choose the one with the **higher** position.

Ex: The player's hand is 2, 2, 3, 3, 3, 6, 6, 6, K, K. Choose: 6.

PlayerType2: Requests the value he has the **least** from the player with most cards at hand. In case he has two or more values with the lowest amount, choose the **lower value**. In case two players have the most amount of cards, choose the one with **higher** position.

Ex: The player's hand is 2, 2, 3, 3, 3, 6, 6, 6, K, K. Choose: 2.

PlayerType3: Start from the first player (except for oneself), move to the next one at each turn in a cyclic order. The player requests for the **highest** value he/she has at hand.

Ex: The player's hand is 2, 2, 3, 3, 3, 6, 6, 6, K, K. Choose: K.

PlayerType4: Start from the first player (except for oneself), move to the next one at each turn in a cyclic order. The player requests for the **lowest** value he/she has at hand.

Ex: The player's hand is 2, 2, 3, 3, 3, 6, 6, 6, K, K. Choose: 2.

7. Configuration File:

The configuration file contains the arguments to your program, each in a single line, by the following order:

Parameter 1: Verbal: 0 or 1. See section 8 for details.

Parameter 2: N – Highest numeric value.

N=1 indicates only figures no numeric values, Ace is not numeric in this game.

Parameter 3: Initial deck ordered from top to bottom.

Parameter 4 to the end: List of players and strategies.

The order of the players in the game is determined by the order they are listed in the file.

In addition, a line that starts with '#' is a comment line. Empty lines and comment lines should be ignored when parsing the file.

Example:

```
# Verbal and highest numeric value
1
3

# Deck and players with their strategies
2H KD QS QC AS KC QH 2S 2D AH AD QD AC KH 2C KS JC JS JD JH 3C 3S 3D 3H
Ami 1
Tami 3
```

For your convenience you can assume that:

1. The given configuration file is valid. That is, there are enough cards in the deck to deal to all players, no invalid characters etc.
2. All games have a happy ending☺, i.e., we will NOT test for a game that runs infinitely.

8 Output

8.1 The trace of the game is defined as follows:

Before each turn print out:

1. A turn counter.
2. The state of the game.
3. A line that describes who is playing and what is his/her move (See example below).

Trace example:

```
Turn 1
Deck: 2C KS JC JS JD JH 3C 3S 3D 3H
Ami: 2H QC QH QS KC KD AS
Tami: 2D 2S QD KH AC AD AH
Ami asked Tami for the value Q

Turn 2
Deck: KS JC JS JD JH 3C 3S 3D 3H
Ami: 2H KC KD AS
Tami: 2C 2D 2S KH AC AD AH
Tami asked Ami for the value A

Turn 3
Deck: JC JS JD JH 3C 3S 3D 3H
Ami: 2H KC KD KS
Tami: 2C 2D 2S KH
Ami asked Tami for the value K
```

8.2 The printed summary of the game is defined as follows:

1. Winner(s) name.
   - In case there is one winner (ex. Ami):
     ***** The Winner is: Ami *****
   - In case there are two winners (ex. Ami and Tami, where Ami is ordered before Tami in the game):
     ***** The winners are: Ami and Tami *****
2. A line that states the number of turns: "Number of turns: 7"
3. A line “Initial State:”

4. A line of 10 dashes (----------)
5. Initial state of the game (use the printState(), see section 4)
6. A line of 10 dashes (----------)
7. A line: “Final State:”
8. Final state of the game (use the printState(), see section 4).

Summary example:

```
***** The Winner is: Ami *****
Number of turns: 7
----------
Initial State:
Deck: 2C KS JC JS JD JH 3C 3S 3D 3H
Ami: 2H QC QH QS KC KD AS
Tami: 2D 2S QD KH AC AD AH
----------
Final State:
Deck: 3S 3D 3H
Ami:
```

8.3 Requested Output:

For **verbal=0**: Print out a summary as described in 8.1

For **verbal=1**: Print out:

1. The trace as describe in 8.2
2. A line of 10 dashes (----------)
3. The summary as describe in 8.1.

See further examples in the assignment page.

9 Submission

- Your submission should be in a single zip file called ”student1ID-student2ID.zip”. The files in the zip should be set in the following structure:
  - src/
  - include/
  - bin/
  - makefile

- **src/** directory includes all .cpp files that are used in the assignment.
  Include/ directory includes the header (.h or *.hpp) files that are used in the assignment.
  bin/ directory should be empty, no need to submit binary files. It will be used to place the compiled file when checking your work.
- The makefile should compile the cpp files into the bin/ folder and create an executable named ”reviiot” and place it also in the bin/ folder.
- Your submission will be build (compile+link) by running the following commands: "make".
- Your submission will be tested by running your program with different configuration files, for example: "bin/reviiot test1.txt"
- You submission must compile without warnings or errors on the department computers.
- Compiler commands must include the following flags: `-g -Wall -Weffc++ -std=c++11`.

10 Recommendations

1. Take care not to duplicate objects unnecessarily. We will test your program for amount of used memory with heavy load, i.e. hundreds of players with thousands of cards.
2. Be sure to run in a reasonable amount of time. We will stop any program that runs more than 5 minutes on any configuration file and deduct points from your grade.
3. Be sure to implement destructors for each class in order to free all memory allocated to the object. We will test your program using VALGRIND in order to ensure no memory leaks have occurred
4. Read with extra caution the rules of the game and be sure that you understand the flow of the game and that you are covering all possibilities.
5. After you submit your file to the submission system, re-download the file that you have just submitted, extract the files and check that it compiles on the university labs. Failure to properly compile or run on the departments’ computers will result in a zero grade for the assignment
6. It is highly recommended that you use try and catch statements for error handling. This is the standard C++ exception handling and in many cases can prevent the program from crashing. For more details google "try catch exception handling C++"

בהצלחה