**Data Structures (EECS 2080C) -- Fall 2018 – Lab 5**

***Topics covered: Classes***

*Lab due:* ***Monday, Oct 8 at 6:0 PM***

**Objective:**

The objective of this Lab is to explore classes.

You will need to create a solution, with three projects. One will contain classes you implement, one will contain an executable using your classes you implemented, and finally one will contain the unit tests to test your classes.

**Task 1:** Create classes to implement a card game called 13.

It is expected that you defined and implement the at least the following classes.

Game (holds the players and the cards)

Card class to represent a card

Deck to represent deck of cards – implement using a queue

Player object – to represent a player

Player Stack which is owned by the player object (implement using a stack).

Player Hand of Cards (suggested using a vector, or simple array) –

HINT: you are going to need to several ToString methods on things to classes to test them

You must use pointers and not objects (sorry but I want you to practice with pointers). And you will need to write routines to free up the memory, as stack doesn’t have a clear method, which means you will need to iterate over the queue.

You will also want to consider which class is going to own the memory and will be responsible for calling delete on it. For example, you could have the memory ownership of the cards be maintained by the Players (and sub-objects) and the Deck. Or you could introduce another class to represent the World that owns all the cards, and hands out references to the pointers.

You also might want to consider using const pointers (cuz - with const you say: Don't change the data this pointer points to (quote I saw on StackOverflow). This way you don’t delete classes you don’t mean to until the end.

The rules are:

1. Each player must end their turn with 5 or fewer cards in their hand.
2. At the start of their turn, they must draw a card from the top of the community pile of cards (AKA as the Deck class - \*implement using a queue)
3. The player then tries to make a stack from their hand in order from 1 to 13 during their turn. \*implement using a stack
4. During their turn, the player may draw additional cards from the top of the pile so long as the total cards in their hand don’t exceed 6.
5. When the player cannot place any more cards in their stack and they have 6 cards in their hand, they must discard 1 card to the bottom of the community pile.
6. A player may choose to discard all of their cards to the bottom of the community pile and start with 5 new cards at the beginning of their turn. This action ends their turn without them getting a chance to draw a 6th card or discard any cards to their stack.
7. The game is played with a standard playing card deck (Ace = 1, Jack = 11, Queen =12, King = 13). The suits are not important for the purposes of determining rule 3 (i.e. a 4 of any suit can go on top of a 3 of any suit).
   1. Do we need to keep track of suits at all then? We should have a method for it, but you don’t have to.
   2. Can you put any number of cards on your stack per turn? yes

**Task 2:** Create a Unit Test library to test the classes you implemented. Write and have pass at least 3 tests per class.

Complete this before moving on to task 3.

**Task 3: Using the classes created in task 1 implement the** card game called 13 in a exe.

This should have 2 players with both players being controlled by the user. Include a screen shot of 1 turn of this game as well as a screen shot of 1 of the players winning the game in the lab report.

**Lab Submission:**

1. Write a word document or PDF that has the screenshots asked for in the Tasks.
2. Include all source code from all tasks, input and output files (if any), and any special instructions to compile and run those programs.
3. Package all files in a single zip folder and submit the file to blackboard.

**Lab Grading:**

1. 20% - Lab attendance
2. 20% - Task 1 has been correctly implemented and meets all requirements.
3. 10% - Task 2 has been correctly implemented and meets all requirements.
4. 30% - Task 3 has been correctly implemented and meets all requirements.
5. 20% - Lab report contains all required information and is well written.

If program fails to compile, 0% will be given for that Task.