CS 241, Spring 2018
Programming Project 1

The Game of War

In this project, you will implement the card game War. War is a game played between two people, with a regular deck of 52 playing cards. Each card in the deck has a suit (hearts, spades, clubs, diamonds), and a rank (2, 3, 4, 5, 6, 7, 8, 9, 10, jack, queen, king, and ace). Suits do not matter for playing War in any way, though we will still store them in the program.

The game begins with each player being given half of the playing cards as their own personal deck, face down. In each round of the game (known as a battle), each player turns over their top card. The player with the higher ranked card wins that round. The ordering of the ranks is as listed above, with 2 being the lowest rank and ace being the highest. After the battle, the winning player takes *both* of the cards and adds them to the bottom of his or her deck, and then the next battle begins.

If during a battle, both players turn over cards with the same rank, then a war begins. Each player takes the next card off the top of their deck and places it face down, and turns over the next card in their respective decks. These cards are now compared for the winning rank, and the winning player takes all six cards and adds them to his or her deck on the bottom. If the ranks are again tied, play continues in this manner, with each player turning over a card face down, then one face up, until the ranks differ on the face-up cards and someone wins the war.

The game continues until one player has all the cards; this player wins the game.

There are two situations left unspecified at this point:

- Most descriptions of this game are unclear as to what happens if one or both players run out of cards during a war. For our purposes, we will assume if one player runs out of cards during a war, the other one wins (and takes all the cards).

  Example: Suppose Alice and Bob are playing. Alice has only one card left, the ace of spades, while Bob has many cards left, with the ace of clubs the next one he will play. Alice plays the ace of spades; Bob plays the ace of clubs, so a war begins. Alice now has no cards left to use in the war, so she loses the battle (and is now out of cards, so she loses the game).

  Example: Suppose Alice and Bob are playing. Alice has only two cards left, the ace of spades and something else, while Bob has many cards left, with the ace of clubs the next one he will play. Alice plays the ace of spades; Bob plays the ace of clubs, so a war begins. Alice now plays her next card face down, as does Bob. However, Alice now needs to play a card face up, but has no cards left to use in the war, so she loses the battle (and is now out of cards, so she loses the game).

Though it is possible, we will assume that both players will never simultaneously run out of cards during a war.

- When playing in real life, the order the cards go back into a player's deck after each battle are not specified. Here, we will specify that cards always go back into the winning player's deck in the following order: first, all the cards played by the winning player during the battle, in the order they were played, followed by all the opponent's cards, in the order they were played. (See sample output at the end for examples).

Your program must use the following classes:

A card class (named Card, specified in the files card.h and card.cpp).

- This class must have two data members: a char called rank and a char called suit.
  - The ranks will be specified by the chars '2', '3', ..., '9', 'T', 'J', 'Q', 'K', 'A' (the last five for ten, jack, queen, king, and ace, respectively).
  - The suits will be specified by the letters 'H', 'C', 'D', 'S' (for hearts, clubs, diamonds, spades).

- This class must have the following methods (functions):
  - A constructor which takes two character arguments corresponding to the rank and the suit with which to initialize the card.
  - A function called toString() which converts and returns the card as a two-character string consisting of the rank followed by the suit. For instance, this function would return "7C" for the seven of clubs, and "KH" for the king of hearts.
  - A function called getValue() that returns the "value" of a card as an int. The value of a card is an integer between 2 and 14, based solely on the card's rank. Ranks 2-10 return those integers, while jack, queen, king, and ace return 11, 12, 13, and 14, respectively.
  - A function called beats(const Card & othercard) that takes a second card object as an argument and returns a Boolean. The return value should be true if and only if the card beats the argument card (in terms of rank, following the game).
  - A function called ties(const Card & othercard) that takes a second card object as an argument and returns a Boolean. The return value should be true if and only if the card ties the argument card (in terms of rank, following the game).

A deck class (named Deck, specified in the files deck.h and deck.cpp)

- This class must have one data member: a vector<Card> called cards that represents a player's personal deck during the game. The top of the deck is the beginning of the vector (low indices), and the bottom of the deck is the end of the vector (high indices). So cards are always added to the end of the vector and removed from the beginning.

- This class must have the following methods:

- A default constructor (does nothing).
- A function dealFromTop() that returns a card. This function should remove the top card from the deck and return it. All other cards shift one spot to the left (towards the top of the deck). The size of the deck decreases by one.
- A function addToBottom(const Card & card) that takes a card argument and adds this card to the bottom of the deck. The size of the deck increases by one.
- A function size() which returns the number of cards in the deck as an int.
- A function isEmpty() which returns a Boolean corresponding to whether or not the deck is empty.
- A function toString() which returns a string representation of the contents of the deck. The string that is returned should list out the cards from top to bottom, and should be enclosed in square brackets. For instance, if a deck consists of the cards king of hearts, 6 of diamonds, and 4 of spades, this string would look like "[KH 6D 4S]". If a deck is empty, "[]" should be returned.

A game class (named Game, specified in the files game.h and game.cpp)

- This class must have five data members:
  - Two decks of cards, one for each player.
  - Two strings, storing the names of each player.
  - An integer specifying what round (battle) of the game it is right now.

- This class must have the following methods:
  - A constructor that takes two strings, representing the names of the two players.
  - A function called setDeck(int player, const Deck & deck) that initializes a certain player's deck to whatever the argument passed in is.
  - A function called isOver() that returns a Boolean telling whether or not the game is over (the game is over when one player has no cards).
  - A function called getWinnerName() that returns a string specifying who the winner of the game is. You can return whatever you want if the game isn't over yet.
  - A function called playRound() that plays one round of the game. **This function is the only function, aside from code in main.cpp, that is allowed to print using cout**. This function should begin by printing the round number and the contents of both player's decks (using their names). Then it should simulate a round of the game, printing messages along the way indicating what is happening (see the sample output). It should print a message as to who wins the round. At the end of the round, print the contents of each player's deck again (this is how we will check your output).

You should have a seventh file, main.cpp, where the program begins running. This file should have a main() function (and any other functions you deem appropriate). The main program should prompt the user for the name of a text file, which will contain the names of the two players and a listing of the cards in their decks at the start of the game. Your program should read in this text file, divvying up the cards between two Deck objects, then initializing a Game

object with the players' names and their respective decks. Then your program should simulate the game of war (by calling playRound() repeatedly until the game is over). Lastly, print a message indicating who has won the game.

Note that you will never need to "shuffle" the deck in this game. The order of the cards is deterministic, and nothing is left to chance.

Text file format

Each text file will be organized as follows. The first two lines of the file will be the names of the two players (here just called P1 and P2). The remaining lines will all be the cards in the "starting deck" before they are dealt to each player. The first card listed goes to P1, the second card goes to P2, the third card goes to P1, the fourth to P2, and so on, alternating between players until the file ends. It is not guaranteed that all 52 cards will appear in the file. (Don't worry about that, I've purposely kept the decks small to make the games shorter and debugging easier.)

Submission

Turn in the files card.h, card.cpp, deck.h, deck.cpp, game.h, game.cpp, and main.cpp by the project deadline. Please upload them directly to Moodle as individual files; do not zip them first. Do not upload anything except source code.

Coding standards

C++ Code Style Guidelines:

- Use good **modular design**. Think carefully about the functions and data structures that you are creating before you start writing code.
    - The **main function should not contain low-level details**. It should be a high-level overview of your solution (remember **top-down design**).
    - As a general guide, **no function should be longer than a page long**. Of course there are exceptions, but these should truly be the exception.
- **Pick a capitalization style** for function names, local variable names, global variable names, and stick with it. For example, for function name style you could do something like "square_the_biggest" or "squareTheBiggest" or "SquareTheBiggest". By convention, variable names start with a lower case character.
- **Use descriptive names** for variables, functions, classes. You don't want to make function and variable names too long, but they should be descriptive (e.g. use "getRadius" or "get_radius" rather than "foo" for a function that returns the value of the radius of a circle). Also, stick with C++-style naming conventions (e.g. i and j for loop counter variables).
- Use good indentation. Bodies of functions, loops, if-else stmts, etc. should be indented, and statements within the same body-level should be indented the same amount.
- **Comment your code!**

  File Comments: Every .h and .cpp file should have a high-level comment at the top describing the file's contents, and should include your name(s) and the date.

Function Comments: Every function (in both the .h and the .cpp files) should have a comment describing:

- what function does;
- what its parameter values are
- what values it returns (if a function returns one type of value usually, and another value to indicate an error, your comment should describe both of these types of return values).

In header files, function comments are for the user of the interface. In a source file, function comments are for readers of the implementation of that function. Because of this, function comments in C source files often additionally include a description of how the function is implemented. In particular, if a function implements a complicated algorithm, its comment may describe the main steps of the algorithm.

My advice on writing function comments: write the function's comment first, then write the function code. For complicated functions, having a comment that lists the steps of the algorithm, will help you.

When commenting stick to a particular style. For example:

```
/*
 * Function:  approx_pi
 * --------------------
 * computes an approximation of pi using:
 *    pi/6 = 1/2 + (1/2 x 3/4) 1/5 (1/2)^3  + (1/2 x 3/4 x 5/6) 1/7 (1/2)^5 +
 *
 *  n: number of terms in the series to sum
 *
 *  returns: the approximate value of pi obtained by suming the first n terms
 *           in the above series
 *           returns zero on error (if n is non-positive)
 */

double approx_pi(int n) {
  ...

      (note: for this function, I'd likely have in-line comments describing
             how I'm computing each part of the next term in the series)

/*
 * Function: square_the_biggest
 * ----------------------------
 *   Returns the square of the largest of its two input values
 *
 *   n1: one real value
 *   n2: the other real value
 *
 *   returns: the square of the larger of n1 and n2
 */

double square_the_biggest(float n1, float n2) {
 ...
```

- **In-line Comments:** Any complicated, tricky, or ugly code sequences in the function body should contain in-line comments describing what it does (here is where using good function and variable names can save you from having to add comments). Inline comments are important around complicated parts of your code, but it is important to not go nuts here; over-commenting your code can be as bad as under-commenting it. Avoid commenting the obvious. Your choice of good function and variable names should make much of your code readable. For example, a comment like the following is unnecessary as it adds no information that is not already obvious from the C code itself, and it can obscure the truly important comments in your code:

```
  x = x + 1;  /* increment the value of x
```