Technical Elective: Programming Language (C++) Midterm Projects: Card, Decks and Hands

Objective:

This midterm is intended to familiarize you with basic program structure, data movement, execution control concepts, and more advanced techniques like the use of member operators and algorithms from the C++ STL, including:

- C++ header files and C++ source files,
- C++ STL input and output streams,
- C++ functions,
- C++ precompiler directives,
- basic C++ data types (including enumerations and strings) and basic uses of structs,
- adding a less-than comparison operator (operator<) to a struct,
- using the STL sort algorithm to order objects.

To do this, you will implement a C++ program that can (1) read sets of 5 cards from a file, sort the cards in each set into ascending order by rank and then suit, and then assign a score to the set based on its rank in poker (assuming a single standard 52 card deck with no wild cards), (2) encapsulate groups of cards within formal hand objects, (3) encapsulate groups of cards within formal deck objects, (4) "shuffle" the cards in a deck object, (5) "deal" cards from decks into hands, and (6) rank hands according to scoring functions that are defined outside the hand class.

Assignment:

Note: the details of this assignment are again intentionally somewhat under-specified, leaving you some room to choose what you think is the best way to implement them, **as long as what you do is reasonable, and you explain your design decisions in comments in the code and in your Readme.txt file.**

Project 1:

1. Open Visual Studio and create a new Visual C++ Empty Project for this project and add header and source files you need.

2. You will declare and define (respectively) the functions, structs, and other features of your program as you implement this project. **As you develop your code, please use good modularity:** for example, if a function's logic has several separable parts (or is long enough to fill the editor's window) you *may* want to break it up into other smaller functions.

3. In the header and source files, declare and define (respectively) a struct to represent a "playing card" (of type `Card`) that declares an enumeration for the different suits a card can have (clubs, diamonds, hearts, spades), another enumeration for the different ranks a card can have (two, three, four, five, six, seven, eight, nine, ten, jack, queen, king, ace), a member variable of the suit enumeration type, and a member variable of the rank enumeration type. Don't forget to include the header file for this struct in any source file that uses the struct type or the enumeration types it contains.

4. Declare and define a function for parsing a file (for example card_file.txt) containing text (character) strings that define different playing cards (we'll call these "card definition strings").

The *card definition strings* in such a file are separated by spaces, tabs, line breaks, or other forms of whitespace. *Valid* card definition strings in a file must be either two or three characters long. The last character of a valid card definition string encodes a card's suit (and must be C or c for clubs, D or d for diamonds, H or h for hearts, or S or s for spades). The first character encodes (or for a rank of 10, the first two characters encode) the rank of the card (and are 2, 3, 4, 5, 6, 7, 8, 9, 10, J or j for jack, Q or q for queen, K or k for king, and A or a for ace).

The file parsing function should take two parameters: a reference to a Standard Template library (STL) vector of card structs (of type `vector<Card> &`), and a C-style string (of type `char *`). This function should use the C-style string given in the second function parameter as the name of a file, open the file, and then read one card definition string at a time from that file until there are no more strings to read.

Each time a valid card definition string is read (invalid strings simply should be ignored), the function should push back a card struct instance with the rank and suit encoded in the string, into the vector that was passed as the first parameter. If the file cannot be opened or the function encounters any other problems (other than badly formed card strings, which it should skip over and keep processing good ones) during its execution it should print out a helpful error message indicating the problem and return a nonzero integer value; otherwise it should return 0 to indicate success.

5. Declare and define a function for printing out a vector of card structs to the standard output stream. The function should take a reference to a *const* vector of card structs (of type `const vector<Card> &`) as its only parameter and should print out valid card definition strings (with each string on its own line) for each of the card struct instances contained in the passed vector (hint: first construct each string and then use **cout** to print it). If the function encounters any problems during its execution, it should print out a helpful error message indicating the problem and return a non-zero integer value; otherwise it should return 0 to indicate success.

6. Declare and define a helpful "usage message" function that (1) prints out (to the program's standard output stream) the program's name followed by a helpful usage message telling the user how to run the program correctly (with the name of the program and the name of a file to read, e.g., **`card card_file.txt`**), and (2) returns a non-zero value.

7. For your main function:

    - Modify the signature of your main function so that it matches the one shown in class (the one that uses the variable names **argc** and **argv**).
    - Check that exactly one argument has been passed to the program (in addition to the program's name). If not, the program should call the usage message function and return the value returned from that call.
    - Declare a vector of card structs (of type `vector<Card>`) and pass that vector and the first program argument (after the program name) to the function that parses a file of card definition strings.
        - If that function returns a non-zero value the program should return that value, or otherwise the program should pass the vector to the function that will print it out, and then check the integer value returned from that function and if it's non-zero return it (note that the next step can be combined with this one by simply returning the result returned by the function).
    - Upon completion, if no errors have occurred, the program should return 0 to indicate success.

Project 2:

1. Modify your previous project, add a **`const`** less than operator (`operator<`) to your card struct, and define it so that it orders the cards by rank (in **ascending** order) and then by suit (also in **ascending** order).

2. Modify the function from the previous project that reads in cards from a file with a slightly richer structure (such as hands.txt) so that it:

    - reads a line at a time from the file,
    - reads card definition strings from that line until it either reaches the end of the line or it sees a `//` (comment symbol) in the line,
    - prints a warning message for any invalid card definition strings it sees but still continues processing the line if so,
    - checks whether or not it read exactly 5 valid card definition strings from the line (not fewer and not more) and if so pushes back 5 cards corresponding to those strings, into the vector that was passed to it (if it reads more or fewer than 5 it should generate a warning message but continue to process other lines from the file).

3. Declare and define a function that takes a reference to a const vector of cards as a parameter (for example of type `const vector<Card> &`) and taking five cards at a time in the vector (i.e., cards in positions 0-4, and then 5-9, etc. in the vector) prints out the poker hand rank of that group of cards:

    - straight flush: the 5 cards are consecutive in rank and have the same suit.
    - four of a kind: 4 of the 5 cards have the same rank.
    - full house: 3 of the 5 cards have the same rank and the other 2 have the same rank.
    - flush: the 5 cards are not all consecutive in rank but have the same suit.
    - straight: the 5 cards are all consecutive in rank but do not all have the same suit.
    - three of a kind: 3 of the 5 cards have the same rank and the other 2 cards have different ranks.
    - two pairs: 2 of the 5 cards have the same rank, another of the 2 cards are of a different rank, and the other card is of a different rank.
    - one pair: 2 of the 5 cards have the same rank and the other 3 cards have different ranks.
    - no rank: none of the other ranks apply.

    Hint: these ranks are much easier to determine if the cards in each set of 5 are sorted according to your card struct's less than operator, but you cannot just sort the entire vector since that may (likely) lose the groupings of the cards into hands. One straightforward way to get around this is to use an additional vector variable to hold 5 cards at a time, sort the cards in that additional vector, and then figure out the ranking. A more advanced and more efficient approach (though more susceptible to arithmetic mistakes) is to exploit two random-access properties of the vector's iterators: you can add a number to them to get another iterator, and you can also compare them for ordering (e.g., you can write expressions like `v.begin() + 5 < v.end()`).

4. In your main function, after the cards have been read into the vector from the file, pass the vector into the function that prints out a poker hand rank for each group of five cards. Your main function should then call the STL sort algorithm to sort the entire vector according to the less than operator you defined in your card struct and should then print out the cards in the order they appear in the sorted array.

Part III:

1. Modify your previous project, declare, and define a class (e.g., of type `Deck`) to represent a deck of cards, with a private member variable that is an STL-supplied **random access** sequence container containing the card type you created in your previous projects (e.g., of type `vector<Card>` but not `list<Card>`). The public interface for this deck class should be:

- A constructor that takes a file name and passes it into a call to the load method (see next). Make sure that if the load method fails the object remains in a valid (though perhaps uninitialized) state, especially if an exception is thrown (see load method failure handling discussion below).
- A destructor: if the compiler-supplied destructor is sufficient then you may use it rather than declaring/defining your own, but if you do please put a comment in the class declaration explaining why that is ok to do.
- A "load" method that takes a file name, opens the file, reads in valid card definition strings from the file as you did in previous projects (ignoring any invalid cards it may encounter and continuing to parse valid ones that follow) and inserts card objects (e.g., of type`Card`) corresponding to them into the container member variable. Feel free to reuse or modify code from your previous projects in your implementation of this method.
- As in the previous projects, if the input file cannot be opened an appropriate error message should be printed and the entire program should terminate and return a non-zero value to indicate failure. Since this method is called from the deck class constructor, and since constructors don't have return values, if this method fails it should either throw an exception which the constructor allows to propagate upward and the main function catches and handles, or this method could return an appropriate non-zero value, which the constructor then detects and throws, etc.
- A "shuffle" method that takes no parameters and randomizes the order of all the cards in the container member variable by calling the STL's `shuffle` algorithm (with begin and end iterators for the entire range of elements in the container and a callable object that returns random numbers, per the example in the C++ reference page for the shuffle algorithm noted above).
- A const "size" method that returns the number of elements in the container member variable.
- A non-member insertion operator (`operator<<`) that takes a reference to an **ostream** and a reference to a const deck object and uses the passed **ostream** to print out valid card definition strings on separate lines, for each card in the deck object's container member variable. Note that your implementation may use a friend declaration to grant this operator access to the deck object's private container member variable.

2. Declare and define a class (e.g., of type `Hand`) to represent a card player's hand of cards. This class should have a private member variable that is an STL-supplied container of cards (e.g., of type `vector<Card>` or `list<Card>`). The public interface for this class should be:

- A default constructor in which the container member variable is initialized to be empty.
- A copy constructor that takes a reference to another const hand object and initializes itself to contain the same sequence of cards as in the passed object.
- A destructor: if the compiler-supplied destructor is sufficient then you may use it rather than declaring/defining your own, but if you do please put a comment in the class declaration explaining why that is ok to do.
- An assignment operator that takes a reference to another const hand object, checks for self assignment (in which case it does nothing), and otherwise clears out any existing cards and updates itself to contain the same sequence of cards as in the passed object.
- A const "size" method that returns the number of elements in the container member variable.
- A const equivalence operator that takes a reference to a const object of the same type and returns true if and only if exactly the same sequence of cards appears in both objects.
- A const less than operator that takes a reference to a const object of the same type and returns true if and only if the sequence of cards in the object on which the operator was called should appear before the sequence of cards in the passed object according to a lexical (phone book style) ordering, based on the cards' less-than operator. Hint: Since cards are stored in sorted order (see the insertion operator below which moves a card from a deck to a hand), implementing this is straight forward compare the first card in each hand and if they are the same compare the second card, etc. The ordering of the first cards that differ determines the ordering of the hands with the

caveat that a hand whose cards are a prefix of the sequence of another hand's cards should be placed earlier, e.g., "9H" before "9H 10H".

- A const "as string" method that returns (by value) a C++ style string containing a space-separated sequence of valid card definition strings representing the sequence of cards in the container member variable.
- A non-member insertion operator (`operator<<`) that takes a reference to an **ostream** and a reference to a const hand object and uses the passed **ostream** to print out space-separated valid card definition strings on the same line, for each card in the hand object's container member variable. Note that your implementation may use a friend declaration to grant this operator access to the hand object's private container member variable.
- A non-member insertion operator (`operator<<`) that takes a reference to a hand object and a reference to a deck object, removes the card from the back of the deck object's container member variable, and adds it to the hand object's container member variable in such a manner that the cards in the hand object are kept in sorted order (according to the card's less than operator). Note that depending on which STL container you used, either pushing back and then sorting (e.g., for `vector` or `list`) or iterating until the right spot is reached and then inserting (e.g., for `list` but not vector since that would result in an inefficient re-copying of stored values) are reasonable approaches. Note that your implementation may use friend declarations to grant this operator access to the hand object's and deck object's private member variables.

3. Declare and define a "poker_rank" function that takes two references to const hand objects and returns a bool value that is true if (and only if) the first hand object ranks higher than the second hand object in the poker hand ranking definition.

4. In your main function:

   - Modify the signature of your main function so that it returns an int and takes an int (**argc**) and an array of pointers to char (**argv**) as its parameters.
   - Your main function should accept either 1 or 2 command line arguments (in addition to the program name): the name of an input file and an optional "-shuffle" argument **in either order**. Your program should print out an appropriate usage message and terminate with a different unique non-zero return value for each of the following cases:
     - only one command line argument is given but it contains "-shuffle".
     - two command line arguments are given but neither one contains "-shuffle".
     - no command line arguments are given.
     - more than 2 command line arguments are given.
   - Otherwise, your main function should construct a deck object using the input file name that was given.
   - If (and only if) the "-shuffle" command line argument was given to the program, it should then also call the deck object's shuffle method.
   - The program should then push back nine empty hand objects into an STL container (e.g., of type `vector<Hand>` or `list<Hand>`).
   - The program should then use the insertion operator you defined above to "deal" one card at a time from the deck to each of the nine hands in the container in turn, repeating the rotation until each of the hands has five cards.
   - The program should then print out the contents of the deck object and the contents of each of the hand objects to which the cards were dealt.
   - The program should use the STL `sort` algorithm to order the hands in the container (according to the hand class less than operator which is what it will do if you just call it with the begin and end iterator positions of the container), and then again print out the hand objects in the order they now appear in the container.

- The program should again use the STL `sort` algorithm to order the hands in the container, but this time it should do so according to the poker hand ranking function you defined (see above) by passing the name of that function as a third parameter to the sort algorithm; the program should then again print out the hand objects in the order they now appear in the container.
- Your main function should use try/catch logic to catch any exceptions that can be thrown by the functions, methods, or operators it uses, and upon catching an exception should print out an appropriate descriptive error message and return a unique non-zero value for each kind of problem. Otherwise, if no problem has occurred, your program should return 0 to indicate success.

Part IV:

1. Build each of your projects and fix any errors or warnings that occur. Please be sure to note all of the different kinds of errors or warnings you ran into (though you don't have to list every instance of each kind) in your ReadMe.txt file. **If you were fortunate enough not to run into any errors or warnings, please note that instead, where you would have listed errors or warnings in your ReadMe.txt file.**

2. Open a Windows console (terminal) window and change to the directory where your program's executable file was created by Visual Studio.

3. Run the executable program through a series of trials that test it with good coverage of cases involving both well-formed and badly formed inputs. For each project, an especially useful test is to check whether the output printed by your program consistently represents the contents of valid card strings from the file whose name it was given. Another one is to check the return code produced by the program, which you can get by running the program in the terminal window and then just after it finishes running issuing the command **`echo %errorlevel%`** which prints out the value of the **`%errorlevel%`** environment variable where the result returned by the most recently run program is always kept. In your ReadMe.txt file please document which cases you ran (i.e., what the command lines were) and summarize what your program did and whether that is correct behavior, in each case.

4. In your ReadMe.txt file, make sure that your name and student ID are at the top of the file, and that you've documented whether you ran into any errors or warnings while developing your solution (and if so what they were) and what the executable program did for each of the trials you ran. Be sure to save your changes to your ReadMe.txt file, as well as those to your source (and header) file(s) before preparing your solution zip file.

5. Prepare a zip file that contains each project's source, header, and ReadMe.txt files (except for `stdafx.h`, `stdafx.cpp`, or other Windows-generated files you did not modify), by:
   - Creating a new MidtermProjs folder with 3 subfolders (e.g., Proj1, Proj2 and Proj3) in it.
   - Copying the appropriate files (here, ReadMe.txt, and the source and header files where you wrote your code, but not the files that were created by Visual Studio such as `stdafx.h`, `targetver.h`, or `stdafx.cpp`) from Visual Studio into corresponding folders.
   - Right-clicking on the MidtermProjs folder and selecting the Send To->Compressed (zipped) Folder item in the menu that appears.