CS2311 Computer Programming (Semester A 2018/19)

Assignment 3

Due: Dec 1, 2018, 23:59 p.m.
NO late submission will be accepted. Plagiarism check will be performed.

Problem 1: Playing the High Card Game

Your task is to write a program that can fill a deck of cards, randomly shuffle them, and then play the High Card game with you as the only player. The program takes an integer from the user as the seed for random number generation and prints the process of playing the game. A skeleton of code is provided in *asg3_Q1_2018_skeleton.cpp*, and you need to complete the program.

Now to begin, a card has three attributes: *face*, *suit*, and *value*. There are 13 faces: "Ace", "Deuce", "Three", "Four", . . ., "Jack", "Queen", "King". There are four suits: "Hearts", "Diamonds", "Clubs", and "Spades". The value of a card simply depends on its face: an "Ace" has a value of 1, "Deuce" 2, "Three" 3, and so on, no matter the suit. A "Jack" has a value of 11, "Queen" 12, and "King" 13. A complete deck of cards thus has 52 cards in total. All faces and suits are already defined in the `main` function as two arrays of cstrings, i.e. `char face[][10]` and `char suit[][10].`

A class called `Card` is provided in the skeleton code, with the above attributes defined as *private* members. `face` and `suit` are both pointers to `char`, i.e. `char *`, or equivalently cstrings. Some access function prototypes are also defined. There is also a prototype of the default constructor. You need to implement these access functions and the default constructor in order to complete the class definition of `Card`.

To fill the deck and shuffle the cards, you need to implement 2 functions. A `deck` is simply an array of `Card` objects as you can see in `main` function. `filldeck(Card *, char[][10], char[][10])` is the function to fill the deck with 52 cards sequentially with the original order in Face and Suit defined in `main` function. That is, the first 13 cards in `wDeck` should be Ace of Hearts, Deuce of Hearts, ..., King of Hearts. The next 13 cards are Ace of Diamonds, ..., King of Diamonds, and so on. You also need to set the value for each card according to its face as mentioned before. Note that the deck is passed to the function as a pointer `(Card *),` and you should not change this. The next function you need to write is `shuffle(Card *wDeck)`. For each card in `wDeck`, it will be swapped with another random card.

Then the game starts. Initially, you have $10 dollars. In each round, the program deals one hand to you by drawing a card from the tip of the deck and shows it to you. You will place a bet on whether your next hand will be bigger or smaller than this hand. If your bet is correct, you win $10; otherwise, you lose $10. To place a bet, enter "b" to indicate that you think the next hand will be bigger, and enter "s" to mean otherwise. The game continues until one of these conditions are met: (1) Your balance becomes 0. Then the game ends automatically since you are broke. (2) You choose to leave the game with a positive balance. You can do this by entering "n" as your bet, the game

will end, and your balance will be shown. Note you have to play *at least* three hands before you can leave the game. (3) The entire deck of 52 cards are drawn. The game automatically ends, and your balance is shown as well.

The rules of comparison between cards are simple. If a card's face, or equivalently value, is bigger than that of the other card, then this card is bigger. Ace is the smallest, and King is the biggest face. If the two cards have the same face, then we compare their suits. Suits are arranged in an alphabetically ascending order. This implies that Clubs < Diamonds < Hearts < Spades.

What you need to do:

1. Based on, complete the class definition of `Card`. Do not add/remove attributes or member functions.
2. Finish the implementation for the default constructor and access functions.
3. Finish the implementation for `filldeck` and `shuffle` functions.
4. Finish the implementation for `compare` function.
5. Complete the High Card game as specified above in `main` function.

Note: The place required to fill the code is commented as "// your code here." Your output must be **EXACTLY** the same as the test cases on **PASS**. Note that your output may be different from the sample runs here if you are not using Mac OS X. Because different compilers implement `rand()` differently. You are encouraged to try different seeds and play different bets to test your own solution (it's a game anyway :-)). Please submit the completed source file to PASS system.

Running Examples: Inputs are indicated in red.

Example 1:

```
Enter the seed for random number generation: 2
### CS2311 High Card Game Begins! ###
Hand 1: Five of Hearts.
Your balance is 10 dollars. Your bet? b
Hand 2: Six of Spades. BIGGER than the previous hand. WIN! Your
balance is 20 dollars. Your bet? b
Hand 3: King of Diamonds. BIGGER than the previous hand. WIN!
Your balance is 30 dollars. Your bet? s
Hand 4: Queen of Hearts. SMALLER than the previous hand. WIN!
Your balance is 40 dollars. Your bet? n
Game ends, you have 40 dollars. Congratulations!
```

Example 2:

```
Enter the seed for random number generation: 3
### CS2311 High Card Game Begins! ###
Hand 1: Five of Hearts.
Your balance is 10 dollars. Your bet? b
Hand 2: Jack of Hearts. BIGGER than the previous hand. WIN!
Your balance is 20 dollars. Your bet? s
Hand 3: King of Clubs. BIGGER than the previous hand. LOSE!
Your balance is 10 dollars. Your bet? n
You can't quit now! Your bet? s
Hand 4: Five of Spades. SMALLER than the previous hand. WIN!
Your balance is 20 dollars. Your bet? n
Game ends, you have 20 dollars. Congratulations!
```

Problem 2: Contact person search

Interpersonal relationship has become very complex. This means we need to remember more and more friends' contact information. Though we may use a notebook to record their names and numbers, it is still troublesome.

Your task is to write a program to implement these functions:

1. Read the contact person list. Each record is made up with a name (a string that only includes English characters, no space or other char), and a telephone number (a string that only includes numbers).
2. Read a search string (maybe a part of name or telephone number). Search the contact list and find possible people.
3. Output the result according to the alphabetical order of the names. About alphabetical order, you can refer this link:
   https://en.wikipedia.org/wiki/Alphabetical_order
4. If no record in contact list satisfy the search request, just output "No result."

Input limitations:

1. The number of contact people is no more than 100.
2. The length of name and telephone number is no more than 100.
3. There will not be any invalid input, such as negative number, empty name or space in the name. You do not need to deal with these problems.

Challenging practice about pointer and dynamic memory allocation:

1. Try to write your own function to implement the string compare, not use the library function. This practice can help you to learn the basic operation of pointer.
2. Try to store the contact list not just use a big object array. You can create an object pointer array, and when you want to create a new contact person, find an empty pointer and use "new" to create a new object for it. This practice can help you learn the dynamic memory allocation.

Here is an example of using pointer of object:

```
Object Contact{
    char name[MAX_SIZE];
    void output(){
        cout << name << endl;
    }
}
int main(){
  contact foo;
  contact *ptr;
  ptr = &foo;
  ptr->output(); // using '->' and pointer to call member function
}
```

Here is an example of using pointer to iterate a char array:

```
…
char name[MAX_SIZE];
char *name_ptr;
cin >> name;
name_ptr = &name[0]; // Let the pointer points to the first element.
while(*name_ptr != '\0'){
    cout << *name_ptr;
    name_ptr++; // Let the pointer points to the next element.
}
cout << endl;
…
```

When we are handling with an array, it is usually too trouble to write tons of indexes. Using the pointer makes it easier to get access elements in an array. And as you see, we can do add and minus operation to the pointer to let it point to different elements.

Here is an example of dynamic memory allocation:

```
/* define of object pointer array*/
Contact *list[MAX_SIZE];
int list_num = 0;
…
/* create a new contact object*/
list[index] = new Contact(name, phone);//Construction method
…
```

Using this method, your program does not need to create a large array of useless objects and can save lots of memory space.

Running Examples: Inputs are indicated in red.

Example 1:

```
Input the number of contact people:
5
Input the contact list:
HenryXu 60000000
JinghuanYu 62601111
HongmingHuang 60009527
QianXu 61601234
HuWan 60002222
Input the search keyword:
an
Search result:
HongmingHuang 60009527
HuWan 60002222
JinghuanYu 62601111
QianXu 61601234
```

Example 2:

```
Input the number of contact people:
5
Input the contact list:
HenryXu 60000000
JinghuanYu 62601111
HongmingHuang 60009527
QianXu 62601234
HuWan 62002222
Input the search keyword:
62
Search result:
HuWan 62002222
JinghuanYu 62601111
QianXu 62601234
```

Example 3:

```
Input the number of contact people:
1
Input the contact list:
Jerry 10000000
Input the search keyword:
Tom
No result.
```