CS 112 - Homework 4

Deadline

11:59 pm on **Friday, September 30**

Purpose

To answer questions related to C++ classes, to practice creating and tastefully outputting an array of objects, and to create and test a class.

Note: you will be using and extending the class that you create in Problems 6-7 in future homeworks.

How to submit

You will complete **Problems 1, 2, and 3** on the course Canvas site (short-answer questions on C++ classes).

For **Problems 4 onward**, you will create the specified `.cpp`, `.h` , and (possibly) `.txt` files on the CS50 IDE, and then submit those to the course Canvas site.

NOTE: While I list the separate files you need to submit for each problem below, I am going to set up Canvas to *also* accept `.zip` files.

That is,

- you can submit each `.cpp`, `.h`, and `.txt` file to Canvas
- OR, if you prefer, you may compress your files to be submitted into a single `.zip` file and submit that `.zip` file to Canvas.

Problem 1 - 5 points

Problem 1 is correctly answering the "HW 4 - Problem 1 - Short-answer questions on array parameters. and one more on file i/o" on the course Canvas site.

Problem 2 - 11 points

Problem 2 is correctly answering the "HW 4 - Problem 2 - Short-answer questions on more C++ class basics" on the course Canvas site.

Problem 3 - 9 points

Problem 3 is correctly answering the "HW 4 - Problem 3 - Short-answer questions with a focus on constructors" on the course Canvas site.

Problem 4 - function read_words

This problem's purpose is to practice writing a function that includes amongst its arguments an array and its size.

First: consider the `guess_word_from_file` program from Homework 3.

Recall that this program assumes the existence of a file of words structured as follows:

- Its first line should contain an integer, assumed to be the number of words in that file.
- Its remaining lines should contain that many words, assumed to have just one word per line.

For example, such a file might contain:

```
4
eagle
bagels
glaring
regally
```

Wouldn't it have been convenient to have a function able to read the words from such a file into an array?

(This will still be awkward because of an argument array needing a size before it can be declared and used as an argument! But Homework 3's `get_size` function can still be called to conveniently get that in advance, to create an appropriate argument array that can then be used in this function.)

Using the design recipe, write a function `read_words` that expects a desired file name, assumed to have the structure described above, an array able to hold words, and its size, that has the side-effects of:

- trying to open that file for reading (and reading *past* the size on this file's first line, in this function's case)
  - (nice **OPTIONAL** touch: compare the size read to the array size provided, and complain and exit if they do not match! But I will let you decide if you wish to check that or not.)
- reading in the words in the file and storing them into the passed array (thus actually **changing** the argument array)

...and that returns nothing.

Submit your files `read_words.cpp`, `read_words.h`, `read_words_test.cpp`, and the `.txt` files you used in testing `read_words` in `read_words_test.cpp`.

Problem 5 - choice of another function with an array argument

As some additional array argument practice, along with more practice writing functions that use other functions, **choose ONE of the suggestions below** for this problem. (You are encouraged to try both if you like, but I will only grade one of them, in the interests of time.)

Problem 5 - choice 1: function `all_5_letters`

Using the design recipe, write a function `all_5_letters` that expects an array of words and its size, and returns `true` is **every** element in that array is a string containing exactly 5 letters (where a letter is defined as one of:

{`'a'`, `'b'`, ..., `'z'`, `'A'`, `'B'`, ... `'Z'`}

) and returns `false` otherwise.

For full credit, your function must:

- declare its array parameter such that it cannot be changed by `all_5_letters`
- appropriately used the Week 2 Lab Exercise function `five_letter_word` (which itself uses the function `is_letter`).

Problem 5 - choice 2: function `how-many-5-letters`

Using the design recipe, write a function `how_many_5_letters` that expects an array of words and its size, and returns how many of its elements is a string containing exactly 5 letters (where a letter is defined as one of:

{`'a'`, `'b'`, ..., `'z'`, `'A'`, `'B'`, ... `'Z'`}

).

For full credit, your function must:

- declare its array parameter such that it cannot be changed by `how_many_5_letters`
- needs to appropriately used the Week 2 Lab Exercise function `five_letter_word` (which itself uses the function `is_letter`).

Problem 6 - create a class definition

First: decide on a type of "playing card" you are interested in modeling with a C++ class **`PlayingCard`** or **`GameCard`** (your choice).

- It could be a "traditional" playing card, as is used in games such as Solitaire, Bridge, and Poker, for example.
  - Such a playing card has at least a suit (heart, club, spade, diamond) and a value (A, 2, 3, 4, 5, 6, 7, 8, 9, J, Q, K, although if you prefer this value could be expressed as 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12).
- It could also be a card as is used in a game such as Magic: The Gathering.
  - Such a card has quite a few features, including name, mana cost, power, toughness, and more.
  - (For our purposes here, you would NOT have to model ALL of the possible features!!, but a *subset* you find of interest, as long as it meets at least the minimum requirements given below.)
- It could also be a card for another game, a simplified version of a card for another game, or even a card you design for a fictitious game, as long as it meets at least the minimum requirements given below.

THEN: create **`PlayingCard.h`** or **`GameCard.h`** giving a **class definition** for your class, following the style of the posted `PlayerChar.h`, including at least the following:

- a **no-argument constructor** method
- a **multi-argument constructor** method allowing the user to specify initial values (as arguments) for all of your class' data fields it is reasonable for the user to be able to specify the value for
- a public **accessor method** for **each** data field it is reasonable for a user to be able to see
  - You probably need this for most if not all of your private data fields; BUT include a comment explaining why not for any data field you do **not** provide one for.
- a public **mutator method** for each data field you want the user **to be able to change directly**
  - This one will vary more, depending on the nature of your playing card. You might not have as many of these.
  - BUT do consider: if someone creates a card object using the no-argument constructor, will such a card object be usable with the mutator methods you *do* provide?
- a public "other" method **`display`** that expects nothing, has the side-effect of printing to the screen the data fields of the calling card object, and returns nothing
- a public "other" method **`to_string`** that expects nothing, and returns a `string` version of the calling card object (a `string` that includes the data fields for the calling card object)
- any other public or private "other" methods you would like to add
- declarations for at least two **`private`** data fields (but can have more if you wish)
  - for example, card suit and card value, or card name and card power

- any other private methods you would like to add

Submit your resulting `.h` file.

Problem 7 - implement your class' methods

Now create **`PlayingCard.cpp`** or **`GameCard.cpp`**, following the style of the posted `PlayerChar.cpp`, implementing each of your class' methods.

In your constructor methods' implementations, be sure to specify initial values for **each** of the new object's data fields.

(It is *not* required, but is a nice addition, to include error-checking of user-provided values in your multi-argument constructor(s) and mutator methods.)

NOTE: When you are implementing your card class' `to_string` method:

- Reminder: the `string` library function `to_string`, mentioned briefly in class on Thursday, September 22 (Week 5 - Lecture 2) and demo'd a bit in `PlayerChar-test.cpp`, is useful for including numeric data fields' values in this method's result.
- BUT: since we want to call our card class' method `to_string` as well, use `std` and the scope resolution operator `::` when you **call** the `string` library's `to_string` function in your class' method `to_string`, so it won't be ambiguous:

```
... std::to_string(13) ...
```

Submit your resulting `.cpp` file.

Problem 8 - test your class

Now create `PlayingCard-test.cpp` or `GameCard-test.cpp`, following the style of the posted `PlayerChar-test.cpp` and `Point-test.cpp`, that contains a `main` method that attempts to test your new class in the same way, being sure that this `main` method **at least**:

- uses your no-argument constructor to declare an object of your class
- uses (each of) your multi-argument constructor(s) to declare an object of your class
- uses your accessor methods to test if your objects were created with the expected data field values (and this can also be considered as testing your accessor methods, also, hopefully!)
  - This would be a good place to compile and run your test-so-far, to see if your constructors and accessors seem to be working!
- calls each of your mutator methods for an object of your class, then uses the corresponding accessor method to test if that object's data field was correctly changed
  - This would be a good place to compile and run your test-so-far, to see if your mutators seem to be working!
- tests your `display` method by printing a description of what it should print to the screen, followed by a call of your `display` method on at least one of the objects of your class
  - This would be a good place to compile and run your test-so-far, to see if your `display` seems to be working!
- tests your `to_string` method
- attempts to test any other methods you chose to add

- and you can add any additional statements/actions/playing around with your class that you'd like!

Submit your resulting `.cpp` file.