CS 202 - Computer Science II

Project 4

Due date (FIXED): Wednesday, 2/22/2017, 11:59 pm

Objectives: The main objectives of this project is to test your ability to create and use C++ classes, with multiple constructors, static members/functions, and expand to operator overloading. A review of pointers, structs, arrays, iostream, file I/O and C-style strings is also included.

Description:
This project will expand Project 3 by adding additional functionality, and implementing more abstract data types (ADTs) and their operations through classes. **Pointers must be used for array manipulation**, including arrays with ADTs (structs, classes) e.g, rental cars, rental agencies. **Pointers must be used in function prototypes and function parameter lists** - not square brackets. **Const** should be used in **parameter lists**, **functions**, and **function signatures** as appropriate. Any of your work (functions/ADTs/etc) produced up to Project_3 (such as your changed pointer-based string functions, e.g. string copy) have to be **modified** to use **const** (e.g. for string copy const-pointers) where appropriate. Square brackets should be used only when declaring an array, or if otherwise specified in the context of an overloaded ([]) operator. **Pointers can only be moved by incrementing or decrementing** (i.e., ++ or - -), or by **setting the pointer back to the base address** using the array name. You should now use the arrow operator (->) with any pointers where appropriate.

The additional functionality is as follows: You are given an updated data file where there is 1 rental car agency location (**RentalAgency**) which has **5** cars (**RentalCar**). Each car can also incorporate **up to 3** (0-3) special driving sensors (**CarSensor**). You will have the **same menu options**, but the **functionality has been updated** below. Note: using multiple helper functions to do smaller tasks will make this project significantly easier.

The CarSensor Class will contain the following private data members:

- **m_type**, a C-style string (name of sensor type), valid strings for CarSensor m_type are "gps", "camera", "lidar", "radar", "none".
- **m_extracost**, a float (additional rent cost per day for the car that carries the sensor, for "gps":=$5.0/day, for "camera":=$10.0/day, for "lidar":=$15.0/day, for "radar":=$20.0/day, for "none":=$0.0/day)
- **gps_cnt**, a static int member (keeps track of existing gps-type sensors)
- **camera_cnt**, a static int member (keeps track of existing camera-type sensors)
- **lidar_cnt**, a static int member (keeps track of existing lidar-type sensors)
- **radar_cnt**, a static int member (keeps track of existing radar-type sensors)

and will have the following methods:

- **Default Constructor** – will set the aforementioned data members to default initial values.
- **Parameterized Constructor** – will create a new object based on the values passed into it.
- **Copy Constructor** – will create a new object which duplicates an input CarSensor object.
- **Get/Set methods** for appropriate data member(s).
- **A Get and a Reset** member function to return and to reset each of the static member variables.

- **A Method to check if 2 CarSensor objects are the same.** You should try to make this an operator overload of (operator==). Try to make it a non-Class Member function which will have to access member data over Class methods.

The RentalCar Class will contain the following private data members:

- **m_make**, a C-style string (car make)
- **m_model**, a C-style string (car model)
- **m_year**, a (preferably const) int (year of production)
- **m_tank**, a (preferably const) float (max tank capacity in gallons)
- **m_sensors**, a CarSensor class type array of size 3 (max allowable number). (You should also use an auxiliary member variable to keep track of how many actual sensors exist onboard, also in case adding a new sensor is required).
- **m_baseprice**, a float (price per day for the sensorless vehicle)
- **m_finalprice**, a float (price per day with the increased cost of the car sensors)
- **m_available**, a bool (1 = true; 0 = false; try to display true/false using the "boolalpha" flag)
- **m_owner**, a C-style string (the current lessee; if no lessee, i.e. the RentalCar object is available), set to a '\0'-starting (0-length) C-string).

and will have the following methods:

- **Default Constructor** – will set the aforementioned data members to default initial values.
- **Parameterized Constructor** – will create a new object based on the values passed into it.
- **Copy Constructor** – will create a new object which duplicates an input RentalCar object.
- **Get methods** for data members.
- **Set methods** for data members except the **m_sensors**, and **m_finalprice**.
- **UpdatePrice** – a method to update the **m_finalprice** after any potential changes (to the **m_baseprice** or the **m_sensors**)
- **Print** – will print out all the car's data.
- **EstimateCost** – will estimate the car's cost *given* (a parameter passed to it) a number of days to rent it for.
- **A Method to Add a CarSensor** to the RentalCar object. You should try to make this an operator overload of (operator+). Hint: a RentalCar Class member method will be the better choice, since it will have access to private members.
- **A Method to Add a lessee (the name of a lessee)** to the RentalCar object. You should try to make this an operator overload of (operator+). Hint: bear in mind what "adding a renter" to a RentalCar object might imply about other data members.

The RentalAgency Class will contain the following private data members:

- **m_name**, a C-style string
- **m_zipcode**, an int array of size 5
- **m_inventory**, an array of RentalCar objects with a size of 5

and will have the following methods:

- **Default Constructor** – will set the aforementioned data members to default initial values.
- **Get/Set methods** for **m_name** and **m_zipcode** data members – **by-Value**.
- **A GetInventoryItem method** to index an object of the **m_inventory** data member – **by-Reference**. (Hint: This will allow you to access (read and write) to the agency's inventory like in Project_3.) You should try to make this an operator overload of (operator[]).
- **ReadAllData** – read all of the data for the agency from a user-provided file.
- **PrintAllData** - prints out all of the data for an agency (including car info).

- **PrintAvailableCars** – prints out all of the data (including car info) only for the available cars of the agency.

The menu must have the following updated functionality:

- **Read ALL** data from **file**. The file has been **structured** :
  The first line is the car agency info, followed by 5 cars.
  For each car the order is: year make model tanksize baseprice {sensors} available [lessee].
  The sensors are enclosed in {braces} and can be 0 up to 3 ws-separated names.
  The lesee name is [optional], it will only be there if the car is available.
- **Print out ALL** data for **the agency** and **all its corresponding cars** in a way that demonstrates this relationship (similarly to Project_3).
- **Print out the TOTAL number of sensors** built to equip the agency's car fleet (total number by sensor type).
- **Find** the **most expensive available** car – **ask the user** if they want to rent it – update that car's **lessee and availability status** if the user says yes.
- **Exit** program.

The following minimum functionality and structure is required:

- Ask the **user** for the **input file** name.
- The list of sensors must be stored in an **array of objects**.
- The list of cars must be stored in an **array of objects**.
- Use **character arrays** (i.e., C-style) to hold your strings. No string data type!
- Write **multiple functions** (Hint: each menu option should be a function).
- At least one function must use **pass by reference**.
- At least one function must use **return by reference**.
- Variables, data members, functions, function signatures, should all be **const** in a perfect program, unless there is no other way for the program to work. (This might seem as an overstatement. However, try to remember the **const** keyword and design around it as much as you can).
- **Pointers** must be used for **all array manipulation**.
- **Pointers** must be used in **function prototypes** and **function parameter lists** (not brackets).
- **Pointers** can only be **moved by incrementing or decrementing**:
  ```
  double d[3] = {1,2,3};
  double* d_Pt = d;
  for (int i=0; i<3; ++i,++d_Pt){ cout<<*d_Ptd; }
  ```
- Or by **setting** the pointer **back to the base address** using the array name.
  ```
  d_Pt = d;   cout<<endl<<*d_Pt<<endl;
  ```
- Write your **own string copy**, **string compare** (or other) functions as needed, modify any existing versions to account for the updated **specification** about using **pointers**.
- The other functionality and structure of the program should remain the **same as Project #3**, including **writing to screen** and **file** and **restrictions on string** libraries, **global variables** and **constants**, etc.
- For this, and hence for any other Project of similar requirements, **"Print out"** implies **Console** and **File Output** (the same information, but to a different file (with a characteristic name, e.g. "AllAgenciesAllCars.txt") for each different implemented functionality.

Implement the concepts of encapsulation and data hiding (necessary)!
Use the const keyword where appropriate (almost everywhere you can make it)!
Implement Class List-based Initialization as much as possible (advised)!

Implement operator overloads as much as you can (advised)!
This is a chance to experiment as much as possible with more advanced concepts and the intricacies of classes. It is not a strict requirement that **const** data members which can only be instantiated with a list initialization are there in your code right now, neither is it necessary to make your member functions in the operator overload approach. Try your best in order to acquaint yourself with these new concepts however, and figure out what might be giving you a hard time understanding and/or implementing at this early point.

The completed project should have the following properties:

- Written, compiled and tested using Linux.
- It must compile successfully using the g++ compiler on department machines. Instructions how to remotely connect to department machines are included in the Projects folder in WebCampus.
- The code must be commented and indented properly. Header comments are required on all files and recommended for the rest of the program. Descriptions of functions commented properly.
- A one page (minimum) typed sheet documenting your code. This should include the overall purpose of the program, your design, problems (if any), and any changes you would make given more time.

Turn in: Compressed .cpp file and project documentation.

Submission Instructions:

- You will submit your work via WebCampus
- Name your code file proj4.cpp
- If you have header file, name it proj4.h
- If you have class header and source files, name them as the respective class (CarSensor.h CarSensor.cpp RentalCar.h RentalCar.cpp RentalAgency.h RentalAgency.cpp) This source code structure is not mandatory, but advised.
- Compress your:
  1. Source code
  2. Documentation

  Do not include executable
- Name the compressed folder:
  PA#_Lastname_Firstname.zip
  Ex: PA4_Smith_John.zip

Late Submission:
A project submission is "late" if any of the submitted files are time-stamped after the due date and time. Projects will be accepted up to 24 hours late, with 20% penalty.