

CS 202 - Computer Science II

Project 4

Due date (FIXED): Wednesday, 10/4/2017, 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 significantly expand upon 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. Make sure all your C-string functions (e.g. string copy, string compare, etc.) work with pointers (parameters list and function implementation). **Const** should be used in **parameter lists, functions, and function signatures** as appropriate. Square brackets should be used only when declaring an array, or if otherwise you specify your own 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 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 Agency location (**Agency**) which has **5** (potentially) high-tech cars (**Car**). Each car can also incorporate **up to 3** (0-3) special driving sensors (**Sensor**). You will have **similar 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 Sensor Class will contain the following private data members:

- **m_type**, a C-string char array of 255 max characters (name of sensor type), valid strings for Sensor 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 Sensor Object.
- **Get/Set methods** for appropriate data member(s).
- **A Get and a Reset** static member function to return and to reset each of the static member variables.
- **A Method to check if 2 Sensor Objects are the same.** You should make this an operator overload of (operator==). You may want to try for practice to make it a non-Class Member function which will have to access member data over Class methods.

The Car Class will contain the following **private** data members:

- **m_make**, a C-string char array of 255 max characters (car make)
- **m_model**, a C-string char array of 255 max characters (car model)
- **m_year**, an int (year of production)
- **m_sensors**, a Sensor class type array of size 3 (max allowable number of sensors per car).
Hint: You are allowed to use an auxiliary member variable of your choice to keep track of how many actual sensors exist onboard, this will also help for instance 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 "std::boolalpha" manipulator like: cout << boolalpha << boolVariable;)
- **m_owner**, a C- string char array of 255 max characters (the current lessee; if no lessee, i.e. the Car 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 Car 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 Sensor** to the Car object. You should make this an operator overload of (operator+). *Hint:* a Car Class member method will be the better choice, since it will have access to private members. You may want to try that for practice.
- **A Method to Add a lessee (the name of a lessee)** to the Car object. You should make this an operator overload of (operator+). *Hint:* bear in mind what “adding a renter” to a Car object might imply about other data members.

The Agency Class will contain the following **private** data members:

- **m_name**, a C-string char array of 255 max characters (
- **m_zipcode**, a preferably const int number of size 5
- **m_inventory**, an array of Car 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 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 make this an operator overload of (operator[]). *Reminder:* Any calls to this operator are excluded from the Project's restrictions about using brackets.
- **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 Car Objects of the agency.

The menu must have the following updated functionality:

- 1) Read ALL data from file. The provided sample file HighTechAgency.txt is structured :
The first line is the car agency info, followed by 5 cars.
For each car the order is: year make model baseprice {sensors} available [lessee].
The sensors are enclosed in {braces} and can be 0 up to 3 ws-separated names.
The lessee name is [optional], it will only be there if the car is available.
- 2) Print to terminal ALL data for the Agency and all its corresponding Cars in a way that demonstrates this relationship (similarly to Project_3).
- 3) Print to terminal the TOTAL number of sensors built to equip the agency's car fleet (total number by sensor type).
- 4) 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.
- 5) 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 to hold your strings (i.e., C-style) exclusively (using the string data type is still not allowed).
- Write multiple functions (*Hint*: You could have each menu option be a function).
- At least one function must use Pass-by-Reference.
- At least one function must Return-a-Reference.
- Otherwise, as before, you are free to use pass by-Value, pass by-Reference, pass by-Address for your function parameters.
- 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). You are required to at least try.
- Pointers must be used for all array manipulation (iterating over elements to read/modify cannot be performed with bracket operator accessing). The only exception on [] is if you are using your own overload of operator [] .
- Pointers must be used in function prototypes and function parameter lists (the bracket notation is not allowed in parameters lists). 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 << *d_Pt << endl;
```
- Write your own C-string copy, C-string compare functions. Their prototypes will have the form (you must use the prototypes exactly as provided, with char * parameters):

```
// copies characters from source to destination until a NULL-  
character '\0' is found in source, then it NULL-terminates  
destination too, and returns  
void myStringCopy(char * destination, const char * source);
```

```
// returns 0 when the strings match, i.e. their characters are
// equal one-by-one until a NULL-character '\0' is found in both
// strings and at the same position as well
// returns a value <1 if the first character that does not match
// has a lower value in str1 than in str2
// returns a value >1 if the first character that does not match
// has a higher value in str1 than in str2
int myStringCompare(const char * str1, const char * str2);
```

- The other functionality and structure of the program should remain the **same as Project #3**, including **writing to screen** and **file**, as well as **restrictions on string** libraries, **global variables** and **constants**, etc.

Implement the concepts of encapsulation and data hiding (necessary)!

Use the const keyword where appropriate (almost everywhere you can make it)!

Implement Class Constructors with Initializer-Lists as much as possible (advised)!

Implement operator overloads as much as you can !

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 (Sensor.h Sensor.cpp Car.h Car.cpp Agency.h Agency.cpp) This source code structure is not mandatory at this point, but advised.
- Compress your:
 1. Source code
 2. DocumentationDo not include executable
- Name the compressed folder:
PA#_Lastname_Firstname.zip
([PA] stands for [ProjectAssignment], [#] is the Project number)
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.