CS 202 - Computer Science II
Project 3

Due date (FIXED): Wednesday, 2/14/2018, 11:59 pm

Objectives: The two main objectives of this project is to test your ability to (1) create and use pointers, and (2) create and use C++ classes. A review of your knowledge of structs, arrays, iostream, file I/O and C-style strings is also included.

Description:
This project will expand Project 2 by adding additional functionality, using pointers, and implementing abstract data types (ADTs) through classes. **Pointers must be used for all 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). Square brackets are to be used only when declaring an array. **For this project, pointers can only be moved by conducting post/pre-increment/decrement operations on them** (i.e., ++ or - -), or by **setting the pointer back to the base address** using the array name. All pointers must be passed by value. (Note: Try to use the arrow operator (->) with Class Object pointers for member access, if you use such in your code.)

The new functionality is as follows: You are given an updated data file (e.g. Agencies.txt) where there are 3 rental Car Agency locations, where **each of the 3** locations (**RentalAgency**) has **5** cars (**RentalCar**). 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. You may want to create a function that will get a car from a location based on the location and car indices.

The RentalCar Class will contain the following data members:
  ➢ **m_year**, an `int` (year of production)
  ➢ **m_make**, a C-string (`char` array of 255 maximum size)
  ➢ **m_model**, a C-string (`char` array of 255 maximum size)
  ➢ **m_price**, a `float` (price per day)
  ➢ **m_available**, a `bool` (1 = true; 0 = false; try to display true/false using the "std::boolalpha" manipulator like: `cout << boolalpha << boolVariable;`)
and will have the following methods:
  ➢ **Default Constructor** – will set the aforementioned data members to default initial values (*Hint*: Remember to use properly named constants where appropriate).
  ➢ **Parameterized Constructor** – will create a new object based on the values passed into it (All the above values are expected to be present in the function parameters list, and all should be used to initialize the instantiated Object member variables).
  ➢ **Separate Get and Set methods** for all data members.
  ➢ **Print** – will print out all the car's data.
  ➢ **EstimateCost** – will estimate the car's cost *given* (via a parameter passed to it) a number of days to rent it for.

The RentalAgency ADT will be a struct and will contain the following data members:
- ➤ **name**, a C-string (`char` array of 255 maximum size)
- ➤ **zipcode**, an `int` array of size 5 (*Hint*: You will *NOT* be able to use `cin` and `cout` –or any `fstream` objects– directly with this `int` array as you were doing so far with C-strings. The reason is that reading/writing is specially handled by C++ for `char` array types. You will need to manage reading/writing to an `int` array on your own.)
- ➤ **inventory**, an array of RentalCar objects with a size of 5

The menu must have the following entries, each implementing a functionality:
> **1) Ask** the user for the **input file name**. Then, **read ALL** data from that **file** (the file has been **structured** where the <u>first line is the Car Agency</u> info, <u>followed by 5 cars</u>). The data have to be stored into **arrays of Class Objects**.
- ➤ **2) Print out to terminal ALL** data for **all Agencies** and **all their corresponding Cars** in a way that demonstrates this relationship (see Sample Output section).
- ➤ **3) Estimate** car **rental cost** - **prompt** for: a) an **Agency** (e.g., Hertz – <u>you can do so with a 1-3 int number per-Agency</u>), b) a **Car number** (<u>where 1-5 are the cars at each agency</u>), c) the number of **days to rent** it for (`rental_cost=days_to_rent*price_per_day`).
- ➤ **4) Find** the **most expensive** Car – **Print to terminal** the single most expensive Car out of all 3 Agencies.
- ➤ **5) Print out** only the **available Cars** - from **all Agencies**, to a **separate output file** (when the user chooses menu entry 5, they should also get **asked** for an **output file name**).
- ➤ **6) Exit** program.

The following minimum functionality and structure is required:
- ➤ Ask the **user** for the **input file** name.
- ➤ The list(s) of cars must be stored into **array(s) of Class Objects**.
- ➤ The list of agencies must be stored into an **array of Struct 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 on function must use **pass by-Reference**. Otherwise, as before, you are free to use **pass by-Value, pass by-Reference**, **pass by-Address** for your function parameters. (*Note*: Remember that using pass by-Value will make the function work on a local internal copy of whatever variable you pass as an argument, therefore the change will not be made on the actual argument itself, and it will be left unaffected after the function call is complete).
- ➤ **Pointers** must be used for **all array manipulation** (iterating over elements to read/modify <u>cannot</u> be performed with bracket operator accessing).
- ➤ **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 length**, **compare**, **copy**, **concatenate** functions. Their prototypes will have the form (use the prototypes exactly as provided, with `char *` parameters):

```
// counts characters in str array until a NULL-character '\0' is
found, then it returns that number excluding the '\0' one
// the return type size_t represents an unsigned integral number
large enough to contain the maximum possible number of a storage
size that can appear on a target architecture
size_t myStringLength(const char * str);

// returns 0 when the C-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);

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

// appends the content of source to the destination array
// this means that the NULL-terminator of destination is
overwritten by the first character of source and a NULL-character
'\0' is appended at the end of the concatenated Cstring in
destination
// returns a pointer to the destination array
char * myStringCat(char * destination, const char * source);
```

➢ The other functionality and structure of the program should remain the **same as Project #2**, 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 as much as possible!

This is a chance to experiment as much as possible with classes, and their concepts as taught in class. It is not a strict requirement that all RentalCar data members are private at this point.

Try to structure your code as much as possible into class/struct header files (e.g. RentalCar.h, RentalAgency.h), class/struct source files (e.g. RentalCar.cpp, RentalAgency.cpp), and other source files (proj3.cpp).

Do your best in order to acquaint yourself with these new concepts at this early point, so that it pays off in future project which will impose such hard requirements.

Sample Output for menu option 2:

Hertz 93619

2014 Toyota Tacoma , $115.12 per day , Available: true

2012 Honda CRV , $85.1 per day , Available: false

2015 Ford Fusion , $90.89 per day , Available: false

2013 GMC Yukon , $110.43 per day , Available: false

2009 Dodge Neon , $45.25 per day , Available: true

Alamo 89502

2011 Toyota Rav4 , $65.02 per day , Available: true

2012 Mazda CX5 , $86.75 per day , Available: true

2016 Subaru Outback , $71.27 per day , Available: false

2015 Ford F150 , $112.83 per day , Available: true

2010 Toyota Corolla , $50.36 per day , Available: true

Budget 93035

2008 Ford Fiesta , $42.48 per day , Available: false

2009 Dodge Charger , $55.36 per day , Available: true

2012 Chevy Volt , $89.03 per day , Available: false

2007 Subaru Legacy , $59.19 per day , Available: false

2010 Nissan Maxima , $51.68 per day , Available: true

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 proj3.cpp
- ➤ If you have header file, name it proj3.h
- ➤ Compress your:
  1. Source code
  2. Documentation
  Do not include executable
- ➤ Name the compressed folder:
  PA#_Lastname_Firstname.zip
  ([PA] stands for [ProjectAssignment], [#] is the Project number)
  Ex: PA3_Smith_John.zip

Verify: After you upload your .zip file, re-download it from WebCampus. Extract it, compile it and verify that it compiles and runs on the NoMachine virtual machines or directly on the ECC systems.
- ➤ Code that does not compile will be heavily penalized –may even cost you your *entire* grade–. Executables that do not work 100% will receive partial grade points.
- ➤ It is better to hand in code that compiles and performs partial functionality, rather than broken code. You may use your Documentation file to mention what you could not get to work exactly as you wanted in the given timeframe of the Project.

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.