CS 202 - Computer Science II
Project 5

Due date (FIXED): Wednesday, 3/4/2020, 11:59 pm

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

Description:
For this project you may use **square bracket**-indexing, **pointers**, **references**, all **operators**, as well as the **`<string.h>`** or **`<cstring>`** library headers which offer their own **`strcpy`**, **`strcmp`**, **`strlen`**, (however the `std::string` type is still not allowed).

The required functionality is as follows: You are given the specifications for a 2 Classes that have an Inheritance relationship – one being the Base class (**Vehicle**) and one the Derived (**Car**). You have to translate these specifications into class implementations (header and source files) and test them against a small test driver program (**proj5.cpp**), which is provided.

You are also required to explain in your documentation the observed output from running the test driver, line-by-line.

The Vehicle (Base) **Class will contain the following protected data members:**
- **m_lla**, a `float` array of size 3 which represents the location of the vehicle on the earth (LLA stands for Latitude-Longitude-Altitude, which are the 3 values stored within the array).
- **m_vin**, a `const int` which represent a unique VIN – Vehicle Identification Number (no two vehicles can ever exist with the same m_vin).

and the following private data members:
- **s_idgen**, a `static int` which is used by the class to generate a unique identifier to initialize m_vin whenever a new Vehicle object is instantiated – how can you achieve this behavior? (*Hint*: Remember how you have been using static variables so far to keep track of the count of a class' active objects).

and will have the following public methods:
- **Default Constructor** – will leave everything uninitialized (except m_vin which has to follow the above described specifications). When it gets called, it should produce a debug output: "Vehicle #vin: Default-ctor" (where vin the actual member value).
- **Parameterized Constructor** – will create a new object based on a desired value for the VIN passed by-Value (it is however able to assign a different value if it runs into any danger of assigning conflicting values), and a desired set of values for LLA passed by-Address (a pointer to `float` data). When it gets called, it should produce a debug output: "Vehicle #vin: Parametrized-ctor" (where vin the actual member value).
- **Copy Constructor** – will create a new object based on the values of another Vehicle object (except m_vin which has to follow the above described specifications). When it gets called, it should produce a debug output: "Vehicle #vin: Copy-ctor" (where vin the actual member value).

- **Destructor** – called whenever an object gets destroyed. When it gets called, it should produce a debug output:
  "Vehicle #vin: Dtor" (where vin the actual member value).
- **Assignment operator=** – will assign member values to the calling object based on the values of another Vehicle object. When it gets called, it should produce a debug output:
  "Vehicle #vin: Assignment" (where vin the actual member value).
- **get/set methods** as appropriate for data members m_vin and m_lla.
- A **move** method which takes in a new LLA location by-Address (a pointer to `float` data) in order for the Vehicle object to move there. When it gets called, it should:
  - produce a debug output:
    "Vehicle #vin: CAN'T MOVE - I DON'T KNOW HOW" (where vin the actual value).

  Note: The Vehicle is a more general parent-Class: It does have its own data and a number of behaviors, and it does have a function to move (the function describes a moving behavior as it takes in a new LLA location), it just implements the behavior in this limited way.

- A **getIdgen** `static` member function to return the value of the `static` member variable s_idgen.

You should also provide an overload for this class for the:
- **Insertion operator<<.** When it gets called, it should produce an output:
  "Vehicle #vin @ [lat, lon, alt]" (where vin the actual member value and lat, lon, alt the LLA[0-2] values).

The Car (Derived) **Class will inherit from Vehicle and will contain the following private data members:**
- **m_plates**, a C-string `char` array of 256 max characters (car license plates)
- **m_throttle**, an `int` (throttle command to bring it into motion).

and will have the following public methods:
- **Default Constructor** – will set a default value of 0 to m_throttle and leave the rest uninitialized. Otherwise, it should behave the same as the Vehicle Default constructor. When it gets called, it should produce a debug output:
  "Car #vin: Default-ctor" (where vin the actual member value).
- **Parameterized Constructor** – will create a new object based on a desired value for the license plates passed as a C-string (pointer to `char` data), the VIN passed by-Value (same behavior as the Vehicle Parametrized), and a desired set of values for LLA passed by-Address (a pointer to `float` data). The default value of 0 should be set to m_throttle here as well. When it gets called, it should produce a debug output:
  "Car #vin: Parametrized-ctor" (where vin the actual member value).
- **Copy Constructor** – will create a new object based on the values of another Car object (same behavior as the Vehicle Copy). When it gets called, it should produce a debug output:
  "Car #vin: Copy-ctor" (where vin the actual member value).
- **Destructor** – called whenever an object gets destroyed. When it gets called, it should produce a debug output:
  "Car #vin: Dtor" (where vin the actual member value).
- **Assignment operator** – will assign member values to the calling object based on the values of another Car object. When it gets called, it should produce a debug output:
  "Car #vin: Assignment" (where vin the actual member value).
- **get/set methods** as appropriate for data members m_plates and m_throttle.

- A **drive** method which takes in a an `int` by-Value and uses it as a throttle value by setting it to m_throttle (begins driving at this throttle level).

You should also *override* the Base class':

- **move** method. The desired *overridden* behavior should:
  - produce a debug output:
    "Car #vin: DRIVE to destination, with throttle @ 75 " (where vin the actual value)member value.
  - call the Derived class' method **drive** with an argument value 75
  - finally updates m_lla with the passed values.

  Note: The Car is a more specialized child-Class. It inherits all the data and all behaviors from Vehicle, but it can also override behaviors such as move (the function that describes a moving behavior as it takes in a new LLA location), and implements it in its own specialized way, i.e. by Drive()-ing.

You should also provide an overload for this class for the:

- **Insertion operator.** When it gets called, it should produce an output:
  "Car #vin Plates: plates, Throttle: throttle @ [lat, lon, alt]" (where vin the actual member value, plates the actual license plates C-string, throttle the actual throttle member value, and lat, lon, alt the LLA[0-2] values).

The following minimum functionality and structure is required:

- Additionally to your class code, you are required to examine and explain the output of the provided proj5.cpp test driver for each Base and Derived class function call made.
  Your grade will be based on the **explanations** you provide in your documentation file (e.g. copy-paste the output from your terminal and explain what is happening line-by-line).
- You are free to use **pass by-Value, pass by-Reference**, **pass by-Address** for your function parameters.
- You are free to **return by-Value, return by-Reference**, **return by-Address** from your functions.
- **Pointers**, **References**, **Square Brackets**-based indexing are all allowed for array (or generally any other data) manipulation.
- Usage of all **built-in Operators** is freely allowed.
- **Const-correctness** (appropriate usage of the keyword **`const`**) is expected.
- You may use the **`<cstring>`** library functions for C-string manipulation. You are not allowed to use the `std::string` data type.

Automate the build process for this project using CMake (required!)

- Your code is required to follow the **file organization structure** demonstrated in your Labs, with subfolders for headers, source files, and a final build products location (generated during build).
- Your project's build should be based on a **CMakeLists.txt** script, which will be included with your deliverables.

The following are a list of restrictions:

- Your code may use the C++11 standard (or any standard higher or lower).

  Note: Usually, you would specify using the g++ command with some flags:
  `g++ -std=c++11 ...`

  But **CMake** provides the functionality of *autodetecting* your system's C++ compiler and generating the Makefiles to invoke the appropriate commands to be used when you eventually `make` your project.
  If you want to enforce configuring your project build to use a particular standard, you either do so everytime you run the cmake configuration command by:

  ```
  cmake -D CMAKE_CXX_STARDARD=11 ..
  ```

  Or you could put a line like the following inside your **CMakeLists.txt** script:

  ```
  set(CMAKE_CXX_STANDARD 11)
  ```

  You do not need to worry about either of these however, and for now just running the usual `cmake ..` for configuration should do the trick.
- No libraries except **`<iostream>`** and **`<fstream>`** and **`<cstring>`** / **`<string.h>`** allowed.
- No usage of the **`std::string`** libraries and data types.
- No global variables except `const` ones.
- No usage of dynamic memory.
- You are expected to employ code abstraction and reuse by implementing and using functions. The already provided code structure in the project description will be considered sufficient.
- You are expected to implement **`const`** correctness in your program design. This refers to class method qualifications, function parameter qualifications, etc.

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 or the provided Xubuntu VM image.
- 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 file structure (with .cpp and .h files, and your CMakeLists.txt). Also, your project documentation file.

Submission Instructions:

- You will submit your work via WebCampus
- Compress your:
  1. Code file structure (containing Source code files, Header files, CMakeLists.txt)
  2. Documentation

  Do not include executable or library files, nor any build, devel, or other non-required folders.
- Name the compressed folder:
  PA#_Lastname_Firstname.zip
  ([PA] stands for [ProjectAssignment], [#] is the Project number)
  Ex: PA5_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 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.