CS 202 - Computer Science II

Project 6

Due date (FIXED): Wednesday, 10/17/2018, 11:59 pm

Objectives: The main objectives of this project is to test your ability to create and use polymorphism with C++ classes. A review of your knowledge to manipulate classes with multiple constructors, static members/functions, operator overloading, inheritance, 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 functions (however the `std::string` type is still not allowed).

This project expands on a more simplified version of Project 5. 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 class (**Car**). You have to translate these specifications into class implementations (header and source files) and test them against a test driver (**proj6.cpp**) which is provided. You are also required to explain in your documentation the observed output from running the test driver.

The Vehicle 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).

and will have the following private methods:

- A **virtual** method **serialize**. Returns nothing. Takes in a std::ostream object by-Reference and uses it to insert into it (employing operator << as usual) the Vehicle's data members so that they form an eventual output:
  "Vehicle @ [lat, lon, alt]" (where lat, lon, alt the LLA[0-2] values).

  Note A1: In a quick view, this does the same thing as the insertion `operator<<` overload you have been using so far. It differs however by being a `virtual` class method. Therefore any polymorphic calls to it (e.g. a call to **serialize** via a Base class pointer which points to an object of a Derived class that overrides this method) will be able to use the Derived class overridden version of **serialize**.
  Note A2: By having **serialize** be a private method, no outside user of the class is able to call it. This means that it is no longer part of the class *Interface*, but it is part of the class *Implementation*. But one would say, not even any Derived class is able to call it directly! True, BUT Derived classes CAN override it – even if they CANNOT call it.
  This means that if a public method of Vehicle (part of its public *Interface*), e.g. **locate** like:

```
class Vehicle{
 public: …
   void locate(){  /*Base public interface, inherited by all Derived*/
     updateLLAfromGPS(); /*common Base method to get something done*/
     serialize(cout);       /*method overridden by each Derived*/
   }
 private: …
   void serialize(std::ostream& os){ os << … ; }
};
```

calls **`serialize`** among doing some other things too (as part of the implementation of a complete behavior), then a Derived class can choose to override only the **`serialize`** method (or anything else it needs to specialize according to its needs) and still maintain the same unifying *Interface* that is prescribed by **`locate`** (i.e. the sequence of calls and actions that in some point include a call to **`serialize`**), without having to override **`locate`** too. Then the behavior of **`locate`** when called on a Derived object will be specialized only for the specific methods it chooses to override (and being marked as virtual also gives it the ability to be called by Base class pointers as explained in *Note A1* above).
Note A3: This showed that even a private method can be overridden. Therefore the motivation to make a method (or generally a member) protected is only to be able to directly access it in Derived classes. So if you absolutely need to directly call **`serialize`** in a Derived class, it has to be protected – otherwise it should be private as described above.
Note A4: The above are a more involved description on how you can start to separate *Interface* from *Implementation* in Class Hierarchies (the **`locate`** method mentioned served as an example, you do not need to implement it for this project).

and finally the following public methods:

- **Default Constructor** – will leave everything uninitialized. When it gets called, it should produce a debug output: "Vehicle: Default-ctor".
- **Parameterized Constructor** – will create a new object based on 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: Parametrized-ctor".
- **Copy Constructor** – will create a new object based on the values of another Vehicle object. When it gets called, it should produce a debug output:
  "Vehicle: Copy-ctor".
- A **`virtual` Destructor** – called whenever an object gets destroyed. When it gets called, it should produce a debug output:
  "Vehicle: Dtor".
  Note B: Remember, the rule of thumb when working with polymorphic classes is for the Destructor to be virtual (formally either public and virtual –or– protected and non-virtual). This is in order to ensure that any `delete`-ion (a Destructor call to free object's resources) via a Base class pointer that is pointing to a Derived class object will first call the Derived class' Destructor and then proceed to the Base class' Destructor.
- **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: Assignment".
- **Get/Set methods** as appropriate for data member m_lla.
- A **Pure `virtual`** method **move** which takes in a new LLA location by-Address (a pointer to `float` data) in order for the Vehicle object to move there. There are no further prescriptions regarding this (for a reason).
  Note C: By having a pure `virtual` method, Vehicle is now an **Abstract** Class: It does have data and a number of associated behaviors, but no object of it can ever exist.

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

- **Insertion operator.** When it gets called, it should in turn call the **`serialize`** method of the passed object.
  Note D: Here is where the previous play their role: `operator<<` is not a Vehicle class method (it is a regular function, just overloaded to work with Vehicle class objects) therefore it *cannot* be `virtual`. But if instead of directly accessing private data members of Vehicle to output them, we can make a call to the private method **`serialize`** which is `virtual`.

This means that any insertion operator calls on (dereferenced) Base class pointers will then call the `virtual` **`serialize`** which will then call the appropriate Derived class' overridden specialized version.

The Car Class will inherit from Vehicle and will contain the following private data members:

- **m_throttle**, an `int` (throttle command to bring it into motion).

and will have the following private methods:

- A **`virtual`** method **`serialize`** (**overriding** the Vehicle `virtual` **`serialize`**). Returns nothing. Takes in a std::ostream object by-Reference and uses it to insert into it (employing operator << as usual) the Car's data members so that they form an eventual output:
  "Car: Throttle: throttle @ [lat, lon, alt]" (where throttle the actual throttle member value, and lat, lon, alt the LLA[0-2] values).
  Also see *Notes A14* explaining why the **`serialize`** method is declared to be `virtual` and why it is declared to be private (and how we can still override the Base class behavior without exposing it for direct calling in the Derived class).

and finally the following public methods:

- **Default Constructor** – will set a default value of 0 to m_throttle. Otherwise, it should behave the same as the Vehicle Default constructor (it should employ the Vehicle Default Constructor). When it gets called, it should produce a debug output:
  "Car: Default-ctor".
- **Parametrized Constructor** – will create a new object based on a desired value for LLA passed by-Address (a pointer to `float` data). It should do so by employing the Vehicle Parametrized Constructor. It should also set the default value of 0 for m_throttle as well. When it gets called, it should produce a debug output:
  "Car: Parametrized-ctor".
- **Copy Constructor** – will create a new object based on the values of another Car object. When it gets called, it should produce a debug output:
  "Car: Copy-ctor".
- A **`virtual`** **Destructor** – called whenever an object gets destroyed. When it gets called, it should produce a debug output:
  "Car: Dtor".
  Also see *Note B* explaining why the Destructor of the Base class needed to be `virtual`.
- **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: Assignment".
- **get/set methods** as appropriate for data member m_throttle.
- A **drive** method which takes in an `int` by-Value and uses it as a throttle value by setting it to m_throttle (begins driving at this throttle level).
- A **`virtual`** method **move** (**overriding** the Vehicle Pure `virtual` move) which takes in a new LLA location by-Address (a pointer to `float` data) in order for the Car object to move there. When it gets called, it should produce a debug output:
  "Car: DRIVE to destination, with throttle @ 75 " , then calls Drive with an argument value of 75 and finally updates m_lla with the passed `float` data values.
  Note C (continued): The Car is a **Concrete** Class. While Vehicle was Abstract (and *needed not* to provide an implementation of Pure `virtual` method move), Car *has to* implement an overriding version of `virtual` move (has to provide a specific –and specialized– implementation for this behavior).

Do NOT provide a corresponding overload for this class and for the:

- **Insertion operator.** It is not necessary: You can call the insertion operator and pass it a Car object, and even more, when it gets called, it will give the correct Car-specific output. Why? *Hint*: See *Note D* and refer back to *Notes A 1-4* to understand what is going on.

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 proj6.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 required to implement this Project's build process using **Makefile**(s). You may advise yourself from the respective samples provided on WebCampus to get started.
- 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.

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 using Makefile(s). 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
- The code file proj6.cpp is already provided and implements a test driver.
- If you have header file, name it proj6.h
- If you have class header and source files, name them as the respective class (Vehicle.h Vehicle.cpp Car.h Car.cpp) This source code structure is now mandatory.
- Compress your:
  1. Source code
  2. Makefile(s)
  3. Documentation

  Do not include executable
- Name the compressed folder:
  PA#_Lastname_Firstname.zip
  ([PA] stands for [ProjectAssignment], [#] is the Project number)
  Ex: PA6_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.