Winter Semester 2020, Final Project

Seneca Valet Application

Milestone 5 – The Car and Motorcycle Class

(V0.9)

Please watch the project repo for possible corrections

Milestone 5:

Seneca Valet Parking accepts two kinds of Vehicles; **Car** and **Motorcycle**.
Seneca Valet Parking also provides **Carwash** for the Cars, while they are parked in the parking.
It also allows Motorcycles with **sidecar** to be parked, but this option must be known when the Motorcycle is entering the Parking.
Inherit a **Car** and a **Motorcycle** class from the **Vehicle** class before the final stage of development of this project.

*Note that it is possible, (but unlikely) that you may have to go back to milestone 4 and do minor modifications to **Vehicle** to make these classes work properly.*

Milestone 5 Due Date:

This milestone is due by 2020.04.04 @ 23:59.

The Car module:

Derive a class called **Car** from the **Vehicle** Class in milestone 4.

The Car class implementation:

- As done before, when implementing the member functions of the class, you are responsible to recognize if a member function can change the state of the class or not. (i.e. if it is a constant function or not or if the arguments of a function are constants or not)

Properties: (member variables)

Carwash flag

In addition to what A **Vehicle** provides, a **Car** should be able to store having **carwash** during the park or not.

Public Constructor implementation:

- a **Car** can be created using a no-argument constructor that sets the **Car** (and therefore the Base class **Vehicle**) to a **safe Invalid empty** state. Also, a **Car** like its base class can be created using a **license plate** and a **make and model** value. In the latter case the values are used to set the properties of the Vehicle. If one of the **licence plate** or **make and model** are pointing to null or are invalid values, the **Car** is set into an **invalid empty state**.

- a **Car** can not get copied or assigned to another **Car.**

Member function implementations:

- There is no need for any additional mandatory implementation of member functions, but you can add your own if needed

Public operator overload implementations:

- **Read**

  This function overrides the **Read** of the **Vehicle** class.

  If the **Car** is set to **Comma Separated** mode it will read as follows:

  1. It calls the **read** of the Base class;
  2. It reads a Boolean value (integer value of 1 or 0) into the **carwash flag**.
  3. It ignores what ever character is left up to and including a **newline character**(`'\n'`).

  If the **Car** is not set to **Comma Separated** mode it will read as follows:

  1. It will prompt on the screen:
     `"Car information entry"` and prints a newline character.
  2. It will **read** the Base class.
  3. Then it will prompt:
     `"Carwash while parked? (Y)es/(N)o: "`
     Afterwards it will wait for user to enter a 'Y' or an 'N' (lowercase or upper case) and based on user's response it will set the carwash flag to true or false respectively.
     If the user responses with anything thing, other than a single character 'y' or 'n' it will keep printing:
     `"Invalid response, only (Y)es or (N)o are acceptable, retry: "`
     and waits for the user to try again.

  At the end the **istream** object is returned.

- **Write**

  If the **Car** is in an **invalid empty state**, this function will write the following message using the **ostream** object and returns the **ostream** object.
  `"Invalid Car Object"`

  When the **Car** is not in an **invalid empty state**:

  1- If the class is in **comma separated mode**, it will print
  `"C,"`
  Otherwise it will print:
  `"Vehicle type: Car"` and then goes to **newline**

2- It will **write** the base class.
3- If the class is in **comma separated mode**, it will print the **carwash flag** and goes to **newline**.
   Otherwise based the value of the **carwash flag** being **true** of **false** it will print `"With Carwash"` or `"Without Carwash"` respectively and then goes to **newline**.
4- It will return the **ostream** object at the end.

The Motorcycle module:

Derive a class called **Motorcycle** from the **Vehicle** Class in milestone 4.

The Motorcycle class implementation:

- As done before, when implementing the member functions of the class, you are responsible to recognize if a member function can change the state of the class or not. (i.e. if it is a constant function or not or if the arguments of a function are constants or not)

Properties: (member variables)

Has Sidecar flag
In addition to what A **Vehicle** provides, a **Motorcycle** should be able to store if it has a **sidecar** attached or not.

Public Constructor implementation:

- a **Motorcycle** can be created using a no-argument constructor that sets the **Motorcycle** (and therefore the Base class Vehicle) to a **safe Invalid empty** state. Also, a **Motorcycle** like its base class can be created using a **license plate** and a **make and model** value. In the latter case the values are used to set the properties of the **Vehicle**. If one of the **licence plate** or **make and model** are pointing to null or are invalid values, the **Motorcycle** is set into an **invalid empty state**.

- a **Motorcycle** can not get copied or assigned to another **Motorcycle.**

Member function implementations:

- There is no need for any additional mandatory implementation of member functions, but you can add your own if needed.

Public operator overload implementations:

- **Read**
  This function overrides the **Read** of the **Vehicle** class.
  If the **Motorcycle** is set to **Comma Separated** mode it will read as follows:
  4. It **reads** the Base class;
  5. It reads a Boolean value (integer value of 1 or 0) into the **Has Sidecar flag**.

6. It ignores what ever character is left up to and including a **newline** character(`'\n'`).

If the **Motorcycle** is not set to **Comma Separated** mode it will read as follows:

4. It will prompt on the screen:
   `"Motorcycle information entry"` and prints a **newline** character.
5. It will **read** the Base class.
6. It will prompt:
   `"Does the Motorcycle have a side car? (Y)es/(N)o: "`
   Afterwards it will wait for user to enter a 'Y' or an 'N' (lowercase or upper case) and based on user's response it will set the **Has Sidecar flag** to **true** or **false** respectively.
   If the user responses with anything thing, other than a single character **'y'** or **'n'** it will keep printing:
   `"Invalid response, only (Y)es or (N)o are acceptable, retry: "`
   and waits for the user to try again.

At the end the **istream** object is returned.

- **Write**
  If the **Motorcycle** is in an **invalid empty state**, this function will write the following message using the **ostream** object and returns the **ostream** object.
  `"Invalid Motorcycle Object"`

  When the **Motorcycle** is not in an **invalid empty state**:
  1- If the class is in **comma separated mode**, it will print
     `"M,"`
     Otherwise it will print:
     `"Vehicle type: Motorcycle"` and then goes to **newline**
  2- It will **write** the base class.
  3- If the class is in **comma separated mode**, it will print the **Has Sidecar flag** and goes to **newline**.
     Otherwise if the **Has Sidecar flag** is **true** it will print
     `"With Sidecar"` then goes to **newline**.
  4- It will return the **ostream** object at the end.

Milestone 5 submission:

To test and demonstrate execution of your program use the data provided in the execution sample below.

If not on matrix already, upload **Car.cpp, Car.h, Motorcycle.cpp, Motorcycle.h, Utils.cpp**, **Utils.h**, **ReadWritable.cpp**, **ReadWritable.h**, **Vehicle.cpp**, **Vehicle.h and ms5_CarMotorcycleTester.cpp** programs to your matrix account.
Compile and run your code and make sure that everything works properly.
Then, run the following command from your account (use your professor's Seneca userid to replace profname.proflastname, and your section ID to replace NXX, i.e., NAA, NBB, etc.):

```
~profname.proflastname/submit 200/NXX/MS5/CM <ENTER>
```

and follow the instructions generated by the command.

IMPORTANT: Please note that a successful submission does not guarantee full credit for this workshop. If the professor is not satisfied with your implementation, your professor may ask you to resubmit. Resubmissions will attract a penalty.