Summer Semester 2020, Final Project

Seneca Valet Parking Application

Milestone 4 – The Vehicle Bass Class
V1.0

Milestone 4:

To build a valet parking application we need to create a class that encapsulates a Vehicle to be parked in parking spot and can be retrieved when it is needed to be returned to the customer.

The Vehicle module:

Derive a class called **Vehicle** from the **ReadWritable** Class in milestone 3.

A **Vehicle** class should be able to store **a license plate** with maximum of **8** characters, a **make and model** with an unknown length more than two characters and a **parking spot number** that is an integer value.

The Vehicle class implementation:

- When implementing the member functions of the class, you are responsible to recognize if a member function can change the state of the Vehicle 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)

License plate
A license plate can be 1 to 8 characters.
Example: “ABC123”

Make and model
A make and model value that can not be a null address and can not be empty.
Example: “Bmw 320 M”

Parking spot number
A parking spot number can be zero or positive integer number. A **Vehicle** with a zero (0) parking spot number is considered as a Vehicle that is not parked.

Public Constructor implementation:

- a **Vehicle** can be created using a no-argument constructor that sets the **Vehicle** to a safe Invalid empty state. Also, a **Vehicle** 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 and the **parking spot** is set to zero. If one of the **licence plate** or **make and model** are pointing to null or are invalid values, the **Vehicle** is set into an **invalid empty state**.

- a **Vehicle** can not get copied or assigned to another **Vehicle**

Destructor implementation:

- Vehicle has a destructor that guarantees no memory is leaked when the object goes out of scope.

Protected Member function implementations:

- `setEmpty`

  sets the **Vehicle** to an **invalid empty state**

- `isEmpty`

  This function returns **true** if the **Vehicle** is in an **invalid empty state**, or else, it returns **false**.

- `getLicensePlate`

  This function returns a read only address of the license plate of the **Vehicle**.

- `getMakeModel`

  This function returns a read only address of the **make and model** of the Vehicle.

- `setMakeModel`

  This function resets the **make and model** of the **Vehicle** to a new value. If the new value is null or empty, the object is set to an **invalid empty state**.

Public Member function and operator overload implementations:

- **getParkingSpot**
  This function returns the **parking spot** number of the **Vehicle**.

- **setParkingSpot**

  Resets the **parking spot number** to a new value. If the value is invalid, it will set the vehicle to an Invalid empty state.

- **operator==**

  Compares **the license plate** of the **Vehicle** with a **license plate value** and returns **true** if the two license plates are identical or else it returns **false**. This comparison is **NOT** case sensitive (i.e. “123ABC” is the same as “123abc”).
  If any value is invalid, this function returns false;

  operator==
  Compares two **Vehicles** and if they have the same **license plate**, it will return **true**, or else it returns **false**. This comparison is NOT case sensitive.
  If any value is invalid, this function returns false;

- **Read**

This function overrides the **Read** of the **ReadWritable** class.
If the **Vehicle** is set to **Comma Separated** mode it will read as follows:

1. It will read an integer for **parking spot number** into the parking spot number
2. It will ignore one character for the delimiter (comma ‘,’)
3. It will read up to 8 characters or up to a comma character for the **license plate** and stores it in the **license plate** number in all UPPERCASE. Either way it will skip the comma afterwards.
4. It will read up to 60 characters or up to a comma character delimiter for **make and model** and dynamically stores it in the **make and model** of the **Vehicle**. Either way it will skip the comma afterwards.

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

It will prompt on the screen:
`"Enter Licence Plate Number: "`
Then it will read up to 8 characters from the console. If the user enters more than 8 Characters, it will print the following error message and tries again until a proper **license plate number** is entered.
`"Invalid Licence Plate, try again: "`

Then it will prompt:
`"Enter Make and Model: "`

Afterwards it will read 2 to 60 characters from the console. If the user enters a value with invalid length, it will print the following error message and tries again until a proper **make and model** is entered.
`"Invalid Make and model, try again: "`

Then in any mode (comma separated or not) the Read function will check if the **istream** object failed while reading. If this was true it will set the **Vehicle** object to an **invalid empty state**.

Also **license plates** are always stored as all **UPPERCASE** characters and the parking spot number is set to zero (0);

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

- **Write**

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

Otherwise, if the class is in comma separated mode, it will print the **values of parking spot**, **license plate** and **make and model**, separated by commas and ends by a comma without going to newline. (i.e **`12,ABC123,Bmw 320 M,`**)

If the class is not in **comma separated** mode, it will print the following using the **ostream** object:

1- `"Parking Spot Number: "`

2- Parking spot number or `"N/A"` if parking spot is zero

3- NEWLINE

4- `"Licence Plate: "`

5- Vehicle's license plate

6- NEWLINE

7- `"Make and Model: "`

8- Vehicle's Make and model

9- NEWLINE

write returns the ostream object at the end.

Other member functions:

- Add other member functions to the **Vehicle** class if needed.

Milestone 4 Duedate:

This milestone is due by Sunday July 26; 23:59;
`~profname.proflastname/submit 244/MS4/Vehicle -due`<ENTER>

Milestone 4 submission:

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

If not on matrix already, upload **Utils.cpp**, **Utils.h**, **ReadWritable.cpp**, **ReadWritable.h**, **Vehicle.cpp**, **Vehicle.h and ms4_VehicleTester.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 244/MS4/Vehicle` <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.

```
/* ----------------------------------------------------
Final Project Milestone 4
Module: ReadWritable
Filename: ms4_VehicleTester.cpp
Version 0.9
Author Fardad Soleimanloo
Revision History
-----------------------------------------------------------
Date      Reason
2020/7/4  Preliminary release
-----------------------------------------------------------*/


#include <iostream>
#include "Vehicle.h"
using namespace std;
using namespace sdds;
void TestValidations();
void TestOperatorEqualEqual(const Vehicle& A, const Vehicle& B);
void TestIO(Vehicle& V);
int main() {
   Vehicle V;
   Vehicle C("abcd", "C car");
   // you can comment each test to develop your Vehicle step by step:
   TestIO(V);
   TestOperatorEqualEqual(C, V);
   TestValidations();
   return 0;
}
void TestIO(Vehicle& V) {
   cout << "Invalid Vehicle Printing:" << endl;
   cout << V << endl;
   cout << "Reading vehicle from console: " << endl;
   cout << "Enter:" << endl << "abc<ENTER>" << endl << "abc<ENTER>" << endl;
   V.setCsv(false);
   cin >> V;
   cout << "Printing a Vehicle that is not parked:" << endl;
   cout << V << endl;
   V.setParkingSpot(12);
   cout << "Printing a Vehicle that is parked in the spot number 12:" << endl;
   cout << V << endl;
   cout << "Reading and Writing Comma Separated values: " << endl;
   cout << "Enter: " << endl;
   cout << "123,abcd,abcd,<ENTER>" << endl;
   V.setCsv(true);
   cin >> V;
   cin.ignore(1000, '\n');
   cout << V << endl;
}
```

```
void TestValidations() {
   Vehicle* V;
   cout << "Construction validations:" << endl;
   V = new Vehicle(nullptr, "abc");
   cout << *V;
   delete V;
   V = new Vehicle("123456789", "abc");
   cout << *V;
   delete V;
   V = new Vehicle("", "abc");
   cout << *V;
   delete V;
   V = new Vehicle("ABC", nullptr);
   cout << *V;
   delete V;
   V = new Vehicle("ABC", "");
   cout << *V;
   delete V;
   V = new Vehicle("ABC", "A");
   cout << *V;
   cout << "Input validations: " << endl;
   cout << "Enter: " << endl << "123456789<ENTER>" << endl << "abc<ENTER>" << endl <<
"abc<ENTER>" << endl;
   cin >> *V;
   cout << *V << endl;
   cout << "Enter: " << endl << "abc<ENTER>" << endl << "a<ENTER>" << endl << "ab<ENTER>"
<< endl;
   cin >> *V;
   cout << *V << endl;
   cout << "Testing setParkingSpot validation: " << endl <<  "Valid setting: " <<endl;
   V->setParkingSpot(20);
   cout << *V << endl;
   cout << "invalid setting: " << endl;
   V->setParkingSpot(-1);
   cout << *V << endl;
   delete V;
}
void TestOperatorEqualEqual(const Vehicle& A, const Vehicle& B) {
   cout << "opeator== (cstring):" << endl;
   if (A == "Abcd") {
      cout << "operator== with cstring works" << endl;
   }
   else {
      cout << "Bad Opertor==, fix your code" << endl;
   }
   cout << "opeator== (Vehicle):" << endl;
   if (A == B) {
      cout << "operator== with Vehicle works" << endl;
   }
   else {
      cout << "Bad Opertor==, fix your code" << endl;
   }
}

/*
Ouput:
Invalid Vehicle Printing:
Invalid Vehicle Object
```

```
Reading vehicle from console:
Enter:
abc<ENTER>
abc<ENTER>
Enter Licence Plate Number: abc
Enter Make and Model: abc
Printing a Vehicle that is not parked:
Parking Spot Number: N/A
Licence Plate: ABC
Make and Model: abc

Printing a Vehicle that is parked in the spot number 12:
Parking Spot Number: 12
Licence Plate: ABC
Make and Model: abc

Reading and Writing Comma Separated values:
Enter:
123,abcd,abcd,<ENTER>
123,abcd,abcd,
123,ABCD,abcd,
opeator== (cstring):
operator== with cstring works
opeator== (Vehicle):
operator== with Vehicle works
Construction validations:
Invalid Vehicle Object
Invalid Vehicle Object
Invalid Vehicle Object
Invalid Vehicle Object
Invalid Vehicle Object
Invalid Vehicle Object
Input validations:
Enter:
123456789<ENTER>
abc<ENTER>
abc<ENTER>
Enter Licence Plate Number: 123456789
Invalid Licence Plate, try again: abc
Enter Make and Model: abc
Parking Spot Number: N/A
Licence Plate: ABC
Make and Model: abc

Enter:
abc<ENTER>
a<ENTER>
ab<ENTER>
Enter Licence Plate Number: abc
Enter Make and Model: a
Invalid Make and model, try again: ab
Parking Spot Number: N/A
Licence Plate: ABC
Make and Model: ab

Testing setParkingSpot validation:
Valid setting:
```

```
Parking Spot Number: 20
Licence Plate: ABC
Make and Model: ab

invalid setting:
Invalid Vehicle Object



*/
```