Winter Semester 2020, Final Project

Seneca Valet Application

Milestone 6 – The Parking (V0.9)

Please watch the project repo for possible corrections

Milestone 6:

To complete your project for this semester, bring all the files in milestone 2 and 5 together into milestone 6.

You will modify and complete the Parking module to implement a fully functional valet parking system as stated in milestone 2.

Milestone 6 due date:

The due date for the project is Apr 14 @ 23:59.

Late penalties:

- Project complete before Apr 15, 23:59, maximum of 80%
- Project complete before Apr 16, 23:59, Maximum of 60%
- Project complete before Apr 17, 23:59, Maximum of 40%
- Project complete before Apr 18, 23:59, Maximum of 20%
- Project complete before Apr 19, 23:59, Maximum of 0%

All submissions are closed after Apr 19, 23:59. The project must be complete at this date in order to pass the course.

The penalties will be calculated based on the date the project becomes complete. A complete project is a project that has each milestone individually submitted. A project that is missing at least one milestone submission is considered incomplete.

The Parking module:

The Maximum Number of Parking Spots Constant value:

Add a constant integer value in the Parking module for **Maximum Number of Parking Spots** and initialize it to the value **100**.

Additional Mandatory Properties: (member variables)

Number of Spots.

Add an **integer** property to the Parking **class** for the **Number of Spots**. This value is always less than the **Maximum Number of Parking Spots** constant value.

Parking Spots:
Create an array of **Vehicle pointers** that act like the Parking **Spots** in the Parking. Use the **Maximum Number of Parking Spots** constant value for the size of the array.

Additional Recommended Properties: (member variables)

Number of Parked Vehicles

Add an integer property to the Parking for **Number of Parked Vehicles** in the Parking. This value is always less than the **Number of Spots** Property.

Constructor Modification:

`Parking( const char* datafile, int noOfSpots)`
- add an integer (**noOfSpots**) argument added the Parking constructor. Set the value of the **Number of Spots** in the Parking to this value. This is the maximum number of **Vehicles** the Parking can accept.
if this number is invalid (less than **10** or more than the **Maximum Number of Parking Spots** constant value) then the Parking is set an **Invalid Empty State**.

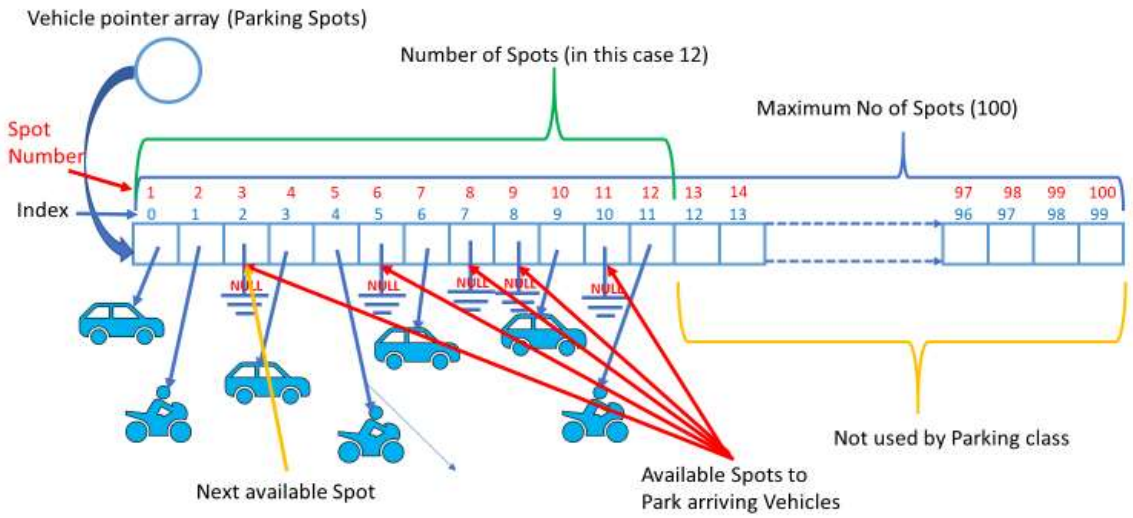
Also make sure that all the elements of **Parking Spots** array are initialized to **nullptr**.

The rest of the logic of the constructor remains the same as it was in milestone 2.

The Parking Module Logic:

Before starting to implement the member functions. Let us clarify how the Parking class manages the Parking **Sports** Vehicle pointers array.

The following illustration shows how the Parking **Spots** array is used to keep track of the parked **Vehicles** in the Parking.



As a **Vehicle** arrives, a dynamic object of its type (**Car** or **Motorcycle**) will get created and set to the **Vehicle's** specifications and the address of the object is kept in the next available (null) element of the Parking **Spots** array.

You can keep track of the number of parked **Vehicles** in a member variable (**Number of Parked Vehicles**) so you can stop when it reaches the **Number of Spots** in the Parking.

If a **Vehicle** leaves the Parking, it is deallocated from the memory and the Parking **Spot** element pointing to it, will be set back to **nullptr** which marks the Parking Spot empty again and ready for parking of the next arriving **Vehicle**.

Private Member function modifications and implementations:

- **isEmpty() function**
  Remains the same as MS2
- **Parking Status function**
  This function does not receive or return anything and prints the following:
  - `"****** Seneca Valet Parking ******"<NEWLINE>`
  - `"*****  Available spots: "`
  - In 4 spaces, left justified, it will print the **number of available Parking Spots**
  - `" *****"<NEWLINE>`
- **Park Vehicle function**
  This function does not receive or return anything.
  If there are no **Parking Spots** available, the function prints `"Parking is full"` and exits.
  Otherwise the function displays the submenu **Vehicle Selection menu** and then based on the user's selection performs the following:
  If user selects Cancel, it will print `"Parking Cencelled"<NEWLINE>` and exits. Otherwise it will dynamically create an instance of a **Car** or **Motorcycle** (based on user's selection) in a **Vehicle** pointer. Then it will set it NOT to be in **Comma Separated mode** and reads it from the console.

  BTP:
  After reading the Vehicle form the console, this function makes sure the license plate is not already in the Parking. If the license plate belongs to an already parked Vehicle, is prints:
  `"Can not park; license plate already in the system!"<NEWLINE>` and exits.

  After receiving the **Vehicle** information from the console, the function will search through the Parking Spots array and finds the first available (null) Parking Spot and sets it to the **Vehicle** pointer and also it will set the Parking **Spot** member variable of the **Vehicle** to the **spot number** it was parked in (**index + 1**) and prints the following:
  `"Parking Ticket"<NEWLINE>`
  and prints the **Vehicle**.

- **Return Vehicle function**

  This function does not receive or return anything and prints:
  `"Return Vehicle"<NEWLINE>`
  and then starts the process by performing the following:
  - Prompts the user for the **license plate** of the **Vehicle** that is to be returned:
  - `"Enter Licence Plate Number: "`
  - Receives the **license plate** value in a **C-style character string** between 1 to 8 characters. If the length if out of the **license plate** is less than 1 or more than 8 characters, it will print
    `"Invalid Licence Plate, try again: "`
    and receives the value again.
  - Searches through the **parked Vehicles** for a matching **licence plate**.
  - If not found it will print: `"License plate "` and then the actual **license plate** value and `" Not found"<NEWLINE>`
  - If found it will print `"Returning: "` and prints the **Vehicle**, then it will delete the **Vehicle** from the memory and set the Parking **Spot** element back to **nullptr**.
- **List Parked Vehicles function**

  This function does not receive or return anything and goes through all the Parking **Spot** elements of the Parking (obviously up to **Number of Spots**) and prints all that are not empty (**not null**) and separates them with the following line:
  `"------------------------------"<NEWLINE>`
- **Close Parking function**

  This function does not receive anything and returns a **Boolean**.
  If the Parking is empty this function will print:
  `"Closing Parking" <NEWLINE>`
  Then exits returning **true**.
  Otherwise this function first prints a confirmation message for the user as follows:
  `"This will Remove and tow all remaining Vehicles from the Parking!"<NEWNLINE>`
  `"Are you sure? (Y)es/(N)o: "`
  And waits for the user to enter either “Y” or “N” (lowercase or uppercase). If user enters an invalid response then prints:
  `"Invalid response, only (Y)es or (N)o are acceptable, retry: "`
  and repeats until proper response is entered.
  If the user response is **no**, then the function exits returning **false**.
  Otherwise, it will first print
  `"Closing Parking" <NEWLINE>`
  the it will go through all the parked Vehicles as follows:
  1- Prints a towing ticket:
  `"Towing request"<NEWLINE>`
  `"********************"<NEWLINE>`
  2- Prints the **Vehicle** and skips a line.
  3- Deletes the **Vehicle** and sets the Parking **Spot** to null until all the **Vehicles** are removed from the Parking.

then returns **true**.

- **Exit Parking App Function**
  This function remains the same as MS2.
- **Load Data File function**
  This function does not receive anything and returns a **Boolean**.
  Load Data reads **Vehicle** records from the datafile and saves them in the corresponding **Parking Spots** indices.

  Implementation guidelines: *(you may use your own logic or follow these guidelines)*
  If the Parking is not in an **invalid empty state**, using an instance of the **ifstream** class open the file named in the **Filename** member variable.

  If the opening of the file was not successful or the **Parking** was in an **empty state** (which means the **Parking** app is running for the first time) this function is in a **good state** (will return true when exits).
  Otherwise while the **ifstream** instance is in a **good state,** have a pointer of type Vehicle and do the following,
  1- Read one Character from the file and ignore the next. (this character should be either **'M'** or **'C'**)
  2- If the Character is '**M'** create a dynamic instance of a **Motorcycle** and keep the address in the **Vehicle** pointer.
  3- Otherwise If the Character is '**C'** create a dynamic instance of a **Car** and keep the address in the Vehicle pointer.
  4- If either of steps 2 or 3 was successfully done, then set the **Vehicle** to **Comma Separate Value** mode and read it from the **data file**.
  5- If the read was successful add the **Vehicle** to the **Parking** by saving the Vehicle pointer in the element of the **Parking Spot** array that corresponds to the **Parking Spot of the Vehicle**, otherwise if read was unsuccessful deallocate the **Vehicle**. In latter case the function goes in a **bad or failure state**.
     *Note that the index of the element of the **Parking Spot** array is always the **Parking Spot** number of the **Vehicle** minus **one**.*
  6- If the number of read records (i.e. **Number of Parked Vehicles**) is less than the **Number of spots** and the function is in **good state,** then go back to step one.

  At the end return the **state** of the function.

- **Save Data File function**
  Using an instance of **ofstream** class open the file named in the **Filename** member variable.
  If the file is opened successfully go through all the elements of the Parking Spots that are not null and save the **Vehicles** pointed by them in the **data file** using the **ofstream** object in **Comma Separated mode**.

Member function implementations:

- Add any additional member functions that you find necessary.

Execution Sample:

A fully functional executable program for **ms6** is provided below. Run the program on matrix to see how the program runs.

```
~sdds.submitter/244/ms6demo
```

*To have an already existing data file for the **parking** copy **ParkingData.csv** file to matrix, where you run the program. To restore the data back to original values after testing, overwrite **ParkingData.csv** file by **ParkingData.csv.bak**.*
*Make sure you set the transfer method of your FTP client to **TEXT**!*

Milestone 6 submission:

You have 3 ways to submit milestone 6:
Comprehensive submission, **Simple** submission and **Open** submission.

Comprehensive submission:
This submission goes through all the details of the application and makes sure everything works perfectly.
You can get up to 100% through this submission.

Simple submission:
Assumes that all the data entry is done correctly and does not do a complete check of the system.
You can get up to 75% through this submission.

Open Submission:
Your code must compile successfully with no warnings. You must demonstrate the functionality of your code to the best of your program's functionality. There is no guaranty that you will get a passing mark.
Your total mark depends on your output and your code. **This submission has a high risk of being rejected**; since the submitter program will do minimum checks on code; the code will be more thoroughly evaluated when marking.
You can get max 50% with this submission.

To test and demonstrate execution of your program use the data provided in the execution samples at the end of this document.

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 ,Menu.cpp, Menu.h, Parking.cpp, Parking.h, ParkingAppTester.cpp** programs and **ParkingData.csv** datafile to your matrix account.
Compile and run your code and make sure that everything works properly.
Then, run one of the following commands from your account based on the submission type you selected (replace **N??** with your section code: **NAA**, **NBB**, or **NCC**):

```
~cornel.barna/submit 200/N??/MS6/open <ENTER>
~cornel.barna/submit 200/N??/MS6/simple <ENTER>
~cornel.barna/submit 200/N??/MS6/complete <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, you may get a low or even a failing grade.