(source- google images)

List of Content

How to run the code?

1. Download the zip file from Git hub like attached
2. Extract the zip file
3. Copy your .txt input file in the same directory where submission.cpp file is present
4. Copy the name of your input test case file name, open the submission.cpp file, update the “Input_file” variable value.
5. Run the code in any installed software e.g., CodeBlocks, visual studio code, or any other compiler that supports file handling in C++ (CodeBlocks with C++14 is preferred)
6. You will get the output of all the queries in the file with the name stored in the Output_file variable; the file will be produced in the same directory.
7. Output_file will only contain outputs of the queries present in input_file only; any other function’s response will not be added to the output_file and will be printed to console only.
8. **Kindly contact me in case of any confusion in the program or if not able to run the program (contact at the 1st page)

Introduction

I have created a Parking lot system for using silent features of C++ language and concepts of Object-Oriented programming.

Our system provides us with the ability to find out:-

1. Vehicle Registration numbers for all cars which are parked by the driver of a certain age,
2. Slot number in which a car with a given vehicle registration plate is parked.
3. Slot numbers of all slots where cars of drivers of a particular age are parked.

All these three operations have O(1) time complexity.

We are using **an unordered_map** data structure, which is part of Standard Template Library in C++; unordered_map stores the key-value pair and facilitates searching, retrieval, and update in constant time.

(assuming count of output for each query << capacity of Parking Lot)

Park & Leave query

For the best empty slot allocation for the new car coming in, we find available slot using **priority_queue**, which is part of the Standard Template Library in C++. Priority_queues are implemented using heaps and provides O(log N) time complexity for both insertion and getting minimum element. So, the park query and Leave query has O(log N) time complexity where n is an element present in the priority_queue.

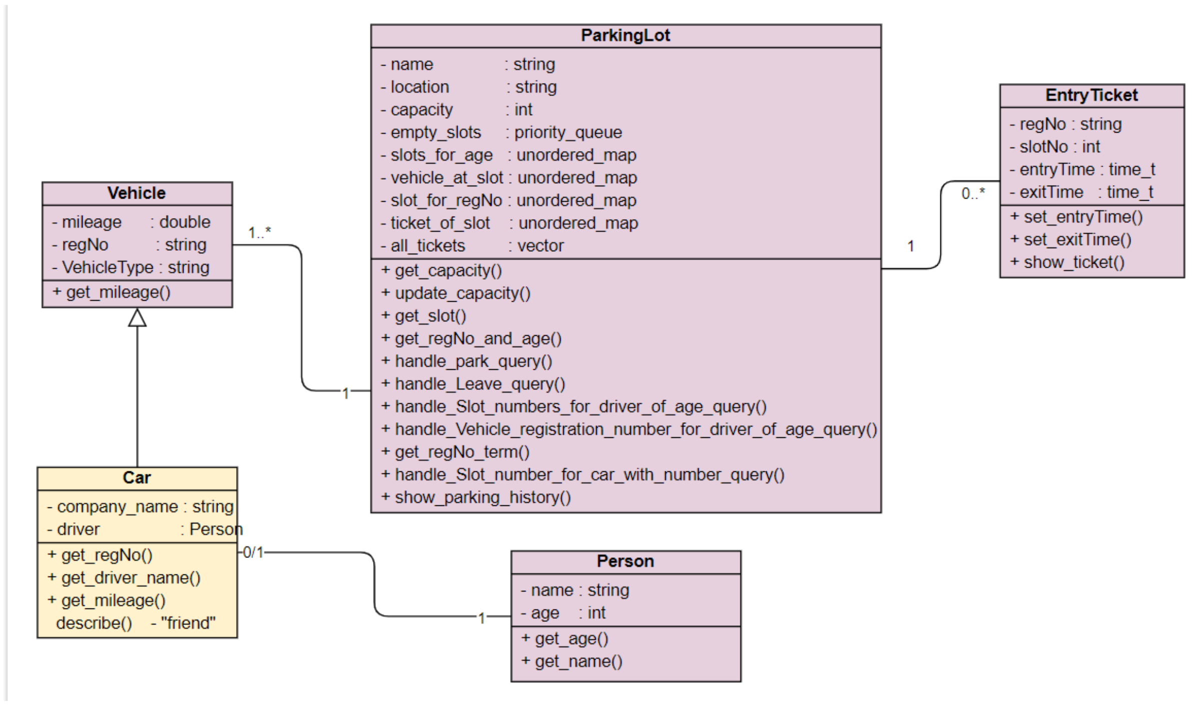
Usage of other OOP concepts are also demonstrated in the submission program file some they are:

- **Inherited** Car class from the Vehicle (base) class to inherit vehicle's properties.
- Describe function is **the friend** function for the class Car. This function can access all the members of the Car class.
- We have **a pure virtual function** in vehicle class, which makes it an **abstract class**.
- All the classes have members and member functions, which perfectly **encapsulates** the complexity of application to users.

Assumptions:

1. The first command is Create_parking_lot command with non-negative capacity.
2. Create_parking_lot command is entered only once for each compilation.
3. All the numeric values (e.g. capacity, age.e.t.c) are nonnegative and in the range of C++ int variable. (x<2^32)
4. Car drivers are co-operative and park cars in the assigned slot only.
5. The system should not allow more vehicles than the capacity of the system.
6. Capacity updates are made only when the slots getting removed are empty already.
7. Every new vehicle will be assigned with the neatest available free slot.
8. The parking lot has a single entry gate, which is the same as the exit gate. (justifies out best allocation assumption)
9. Only the Car object is used for parking as Car parking is asked in inputs, other classes like truck and bike can also be formed extending to the vehicle class.
10. The entry ticket will be saved for history information, only after a car leaves.
11. All the payment related classes, the status of the parking lot are not related to the scope of the problem statement provided.

Class Diagram –



(source – self developed using Visual paradigm online)

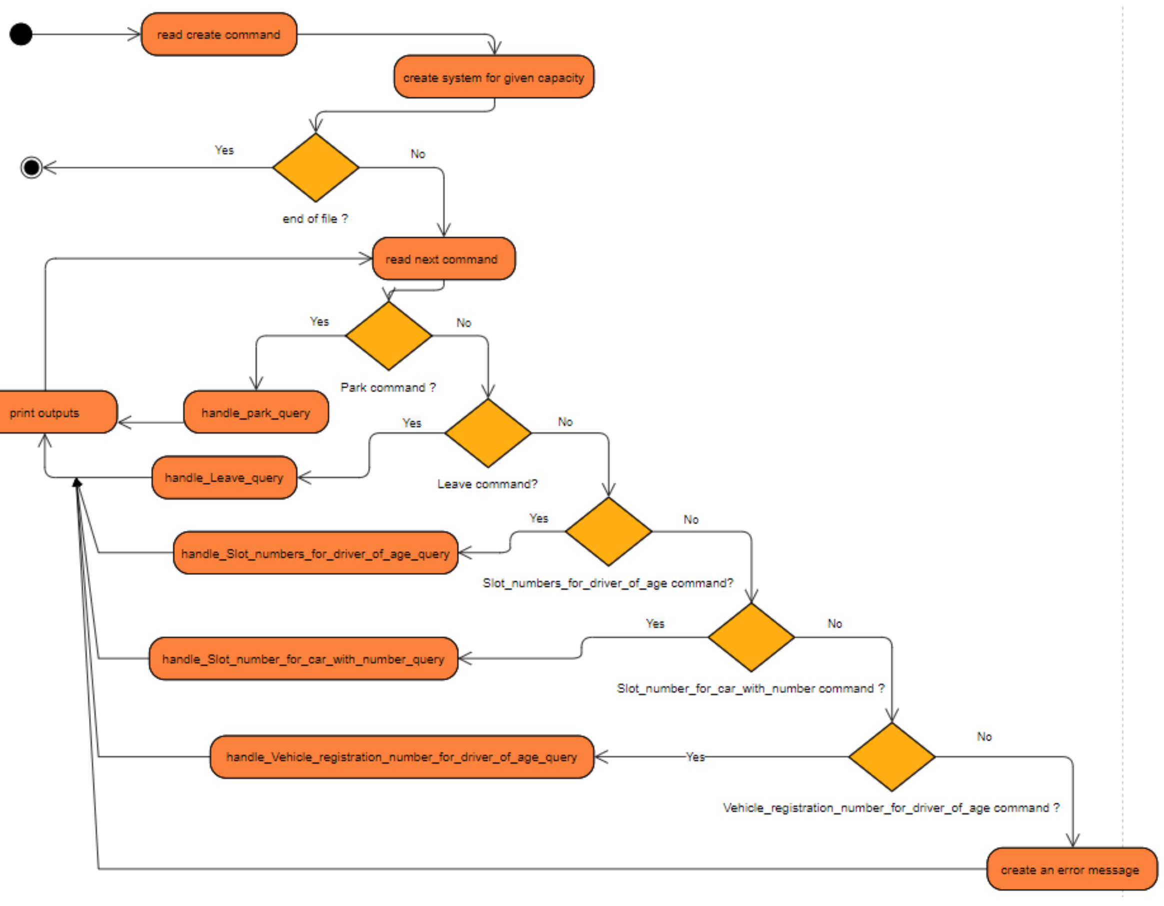
Relations between all the classes are expressed using a class diagram.

Here => + sign indicate member variable and – sign for member function.

Activity Diagram –

The flow of the program and some important function (which are not easy to understand)

1. Main Function –

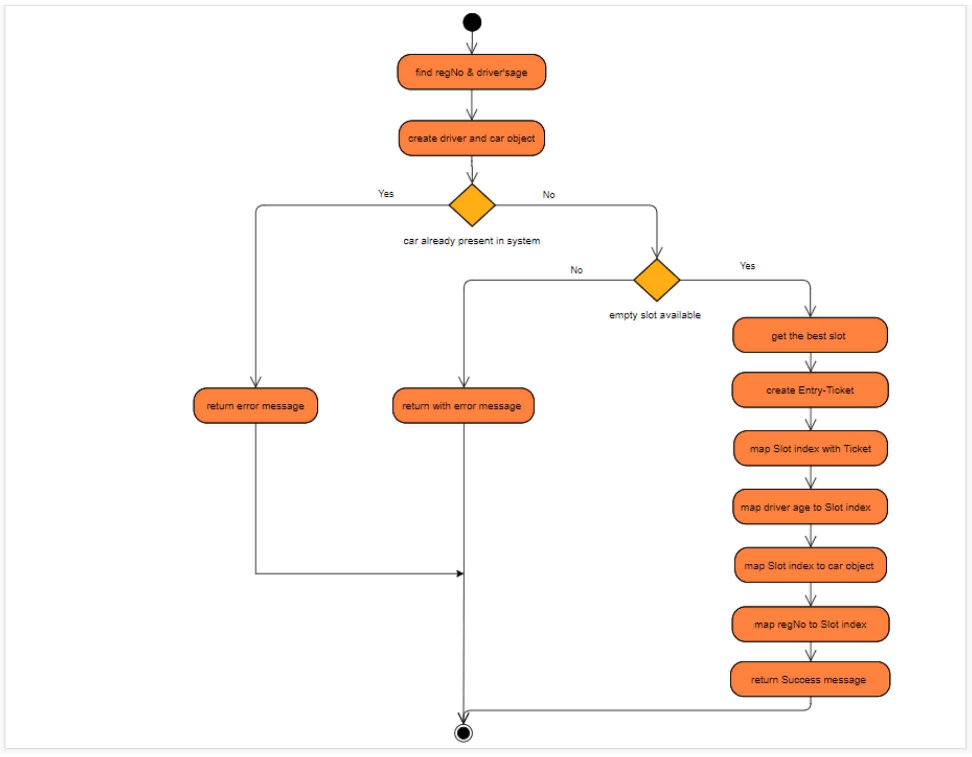

read create command
create system for given capacity
Yes
No
end of file ?
read next command
Yes
No
Park command ?
print outputs
handle_park_query
Yes
No
handle_Leave_query
Leave command?
Yes
No
handle_Slot_numbers_for_driver_of_age_query
Slot_numbers_for_driver_of_age command?
Yes
No
handle_Slot_number_for_car_with_number_query
Slot_number_for_car_with_number command ?
No
handle_Vehicle_registration_number_for_driver_of_age_query
Yes
Vehicle_registration_number_for_driver_of_age command ?
create an error message

(source – self developed using Visual paradigm online)

The main function directs every to a suitable function and performs appropriate read-write file operations.

In case of no matching operation of incorrect command, it prints an error message to the file.:

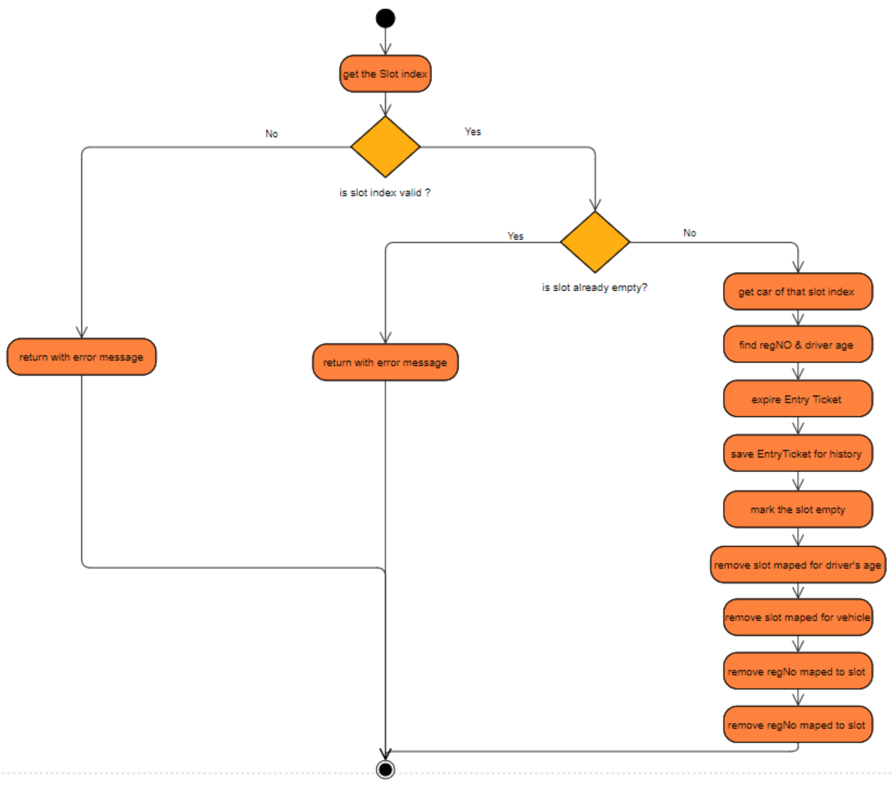
2. Handle_Park_query Function –



find regNo & driver'sage
create driver and car object
Yes
No
car already present in system
No
Yes
empty slot available
get the best slot
create Entry-Ticket
map Slot index with Ticket
map driver age to Slot index
map Slot index to car object
map regNo to Slot index
return Success message
return error message
return with error message

(source – self developed using Visual paradigm online)

3. Handle_leave_query Function –



get the Slot index
No
Yes
is slot index valid ?
Yes
No
is slot already empty?
get car of that slot index
find regNO & driver age
expire Entry Ticket
save EntryTicket for history
mark the slot empty
remove slot maped for driver's age
remove slot maped for vehicle
remove regNo maped to slot
remove regNo maped to slot
return with error message
return with error message

(source – self developed using Visual paradigm online)

Working of the three important queries:

1. Vehicle Registration numbers for all cars which are parked by the driver of a certain age.

Working :

I have created an unordered_map< driver_age, set< slot_index > > “slots_for_age”, which pairs a set of all the slot indexes where the driver has the same age, to the integer values of the age.

We can access the data of cars which are parked at a given slot, using another

unordered_map< slot_index, Car_parked > “vehicle_at_slot”, as we also know all the slots at which driver’s age is equal to the given age.

Now regNo can be simply found using get_regNo() function on the car object obtained.

2. Slot number in which a car with a given vehicle registration plate is parked.

Working :

I have created an unordered_map< regNo, slot_index > “slot_for_regNo” which pairs every registration number to the slot at which the corresponding car is parked.

So, we just put given registration no in the slot_for_regNo, and we get the slot no.

3. Slot numbers of all slots where cars of drivers of a particular age are parked.

Working :

I have created an unordered_map< driver_age, set< slot_index > >“slots_for_age”, which pairs a set of all the slot indexes where the driver has the same age, to the integer values of the age.

We can find the set of all the slot indexes having a driver with a given age just by putting driver age in the unordered_map