CSC190: Computer Algorithms and Data Structures
Lab 3
Assigned: Feb 6, 2017; Due: Feb 10, 2017 @ 10:00 a.m.

Please complete this lab individually

1 Objectives

In this lab, basic concepts behind object-oriented programming (OOP) are introduced. OOP promotes abstraction, modularity and portability of software systems. These concepts are very useful especially when applied to large software systems as good design reduces maintenance needs and promotes flexibility. Although C is not an object-oriented language, it is possible to mimic up to a certain degree several basic OOP concepts. Through this lab, you will be introduced to the notion of **object**, **class**, **interface** and **inheritance**.

We have provided files that you should use as a starting point to complete this lab. You will download content in the folder `Lab3` which contains two folders (`code` and `expOutput`) into your ECF workspace. Folder `code` contains a skeleton of function implementations and declarations. Your task is to expand these functions appropriately. `main.c` evokes all the functions you will have implemented and is similar to the file that will be used to test your implementations. Use `main.c` file to test all your implementations. Folder `expOutput` contains outputs expected for the supplied `main.c` file. Note that we will **NOT** use the same `main.c` file for grading your lab. Do **NOT** change the names of these files or functions.

2 Grading

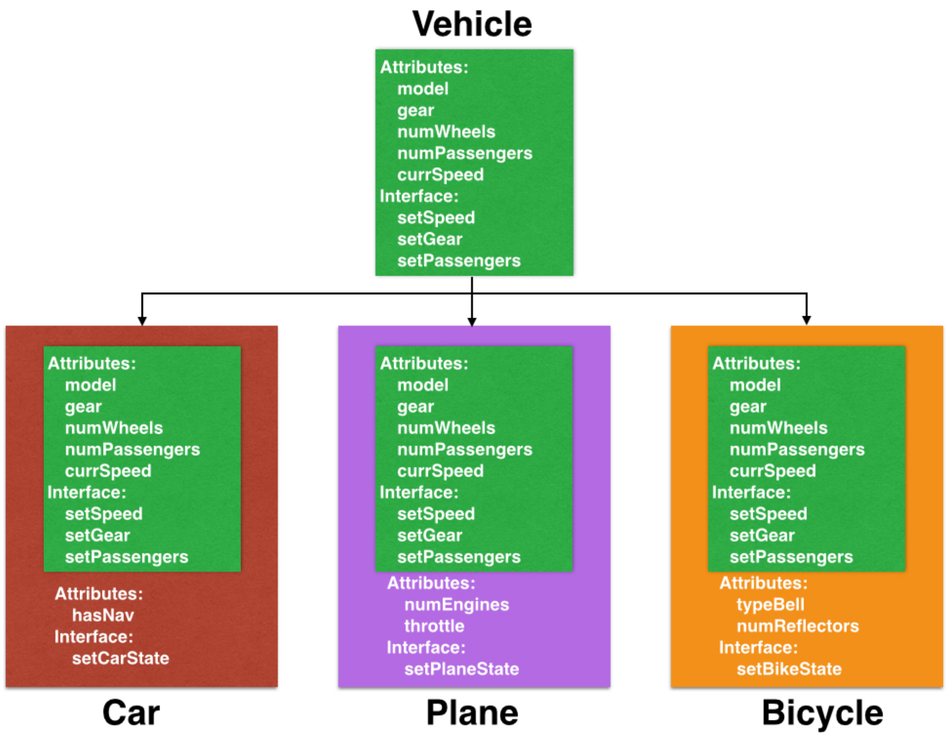
It is **IMPORTANT** that you follow all instructions provided in this lab very closely. Otherwise, you will lose a *significant* amount of marks as this lab is auto-marked and relies heavily on you accurately following the provided instructions. Following is the mark composition for this lab (total of 15 points):

- Successful compilation of all program files - i.e. the following commands result in no errors (1 point):
  ```
  gcc vehicle.c vInterface.c car.c expressLane.c main.c -o run
  valgrind --quiet --leak-check=full --track-origins=yes ./run 1
  valgrind --quiet --leak-check=full --track-origins=yes ./run 2
  ```
- Output from Part 1 exactly matches expected output (3 points)
- Output from Part 2 exactly matches expected output (5 points)
- Code content (6 points)

Sample expected outputs are provided in folder `expOutput`. **NO** late submissions are accepted. All submissions after the deadline will be assigned a grade of 0/15. **Please do not modify any of the header files or the `main.c` file as you will not be submitting these.**

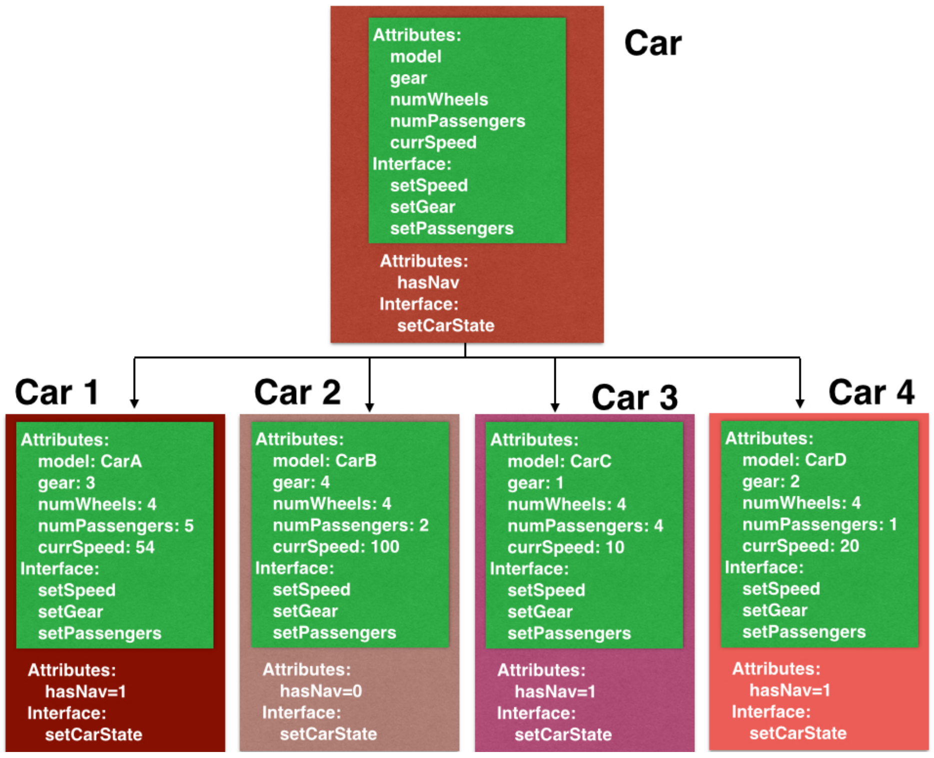
Introduction

OOP concepts are best demonstrated via examples taken from reality. For instance, consider vehicles. Most vehicles have some common *attributes*. Cars, bicycles and planes consist of common physical attributes such as wheels and gears. However, there are also distinguishing features between these vehicle types (e.g. cars may have navigation systems, planes have multiple engines, bicycles consist of bells). *Interactions* with these vehicles such as depressing a car's throttle or pedalling a bicycle will change the current state of the vehicle such as its speed. Various types of vehicles impose specific restrictions on the vehicular states (e.g. cars cannot go above 200 km/h, bicycles travel at 20 km/h, etc.). So far, we have presented a discussion on general *classes* of vehicles and how these differ or are similar. It is important to note that, specific *instances* of vehicles under the same class such as cars can easily differ from one another as well. For example, suppose



Vehicle
Attributes: model gear numWheels numPassengers currSpeed
Interface: setSpeed setGear setPassengers
Car
Attributes: hasNav
Interface: setCarState
Plane
Attributes: numEngines throttle
Interface: setPlaneState
Bicycle
Attributes: typeBell numReflectors
Interface: setBikeState

Figure 1: Classes, Interfaces and Inheritance



Car
Attributes: model gear numWheels numPassengers currSpeed
Interface: setSpeed setGear setPassengers
Attributes: hasNav
Interface: setCarState
Car 1
Attributes: model: CarA gear: 3 numWheels: 4 numPassengers: 5 currSpeed: 54
hasNav=1
Car 2
Attributes: model: CarB gear: 4 numWheels: 2 numPassengers: 2 currSpeed: 100
hasNav=0
Car 3
Attributes: model: CarC gear: 1 numWheels: 4 numPassengers: 4 currSpeed: 10
hasNav=1
Car 4
Attributes: model: CarD gear: 2 numWheels: 4 numPassengers: 1 currSpeed: 20
hasNav=1

Figure 2: Classes and Objects

that a car is travelling on the Highway 401. This car may travel at a different speed from another car ahead of it at the same time instance.

Hence, in the above discussion, it is possible to notice the narrowing of scope. We started with a general blueprint for all vehicles (i.e. wheels and gears are attributes; speed and gear position are states). There exists a set of common actions associated with all vehicles (changing the speed and setting the gear to various states) which are methods by which one can interact with vehicles. Then, we discussed specific types of vehicles and the restrictions imposed by these (number wheels/gears, speed limits, etc). We, then further narrowed our discussion to specific instances of vehicles on the road.

In OOP, there is a term associated with each one of these categories, attributes and states. General vehicle blueprint that was initially discussed is an example of a general **class**. A class contains a set of general features and actions. All interactions with vehicles are forged via **interfaces** (e.g. setting current speed, gear, etc.). Then, we further narrowed down the notion of vehicles to specific types of vehicles such as cars, bicycles and planes. We can define a class for cars, another class for bicycles and one for planes. However, as all these classes share common features and actions specified in the Vehicle class, these will **inherit** these features from the parent class Vehicle. Then, we discussed actual instances of cars on the road. These are referred to as **objects** as these are entities with actual set of states at a specific time instant. In Fig. 1 and 2, these concepts are summarized with respect to the vehicles example presented above.

Part 1: Classes, Objects, Inheritance and Interfaces

In this lab, you will use C to emulate some of the most basic concepts in OOP. First, let us begin with defining the general class `Vehicle`. All definitions and declarations associated with the Vehicle class can be found in the `vehicle.c` and `vehicle.h` files. In the `vehicle.h` file, you will notice that there is a structure definition and two function declarations. The structure is called `Vehicle` and consists of 6 members:

- 5 variables (attributes common to all vehicles): `numWheels`, `model`, `numPassengers`, `currSpeed`, `gear`
- 1 structure variable of type `vInterface`: consists of all the interface functions that change the value of the 5 attributes (e.g. changing the speed, gear, etc.). This will be discussed later.

The structure `Vehicle` consists of the blueprint of vehicles in general. The attributes represent common physical characteristics of all vehicles and the interface consists of a set of common functions that can induce change in these attributes. Hence, the structure `Vehicle` is similar to a general class in OOP. There are two function declarations in `vehicle.h` file. You will expand the definitions of these functions in `vehicle.c` as follows:

- `struct Vehicle * initVehicle(int wheels, char * model, struct vInterface vInt):` In this function you will dynamically allocate space for an instance of `struct Vehicle` and initialize the members of this structure. The members `numWheels`, `model`, `vehInt` are each set to the corresponding three arguments passed to the function and all other members are assigned the value 0. This function will return the memory address of this dynamically allocated structure. In OOP, this function is similar to a *constructor*. As the newly allocated structure is an instance of the class `Vehicle`, it is analogous to an *object*.
- `void cleanUpVehicle(struct Vehicle * v)`: This function is evoked when there is no longer a need for the object represented by `v`. This function deallocates any dynamically created space and is analogous to a *destructor* in OOP.

Next, lets examine the files `vInterface.h` and `vInterface.c` files. The interfaces or common actions that can be executed on all vehicles are setting the speed of the vehicle, setting the gear and setting the number of passengers in the vehicle. Although these interfaces are the same, the way these are implemented can vary from vehicle to vehicle. For instance, a car can consist of five passengers and a bicycle can carry at most two passengers. Hence, the implementations of these functions will be different between specific types of vehicles. In order to generalize this, we define an interface structure in `vInterface.h` called `vInterface` consisting of three members which are function pointers as follows:

- `setSpeed`: This is a function pointer to a function definition consisting of one integer argument and one integer return value. This function examines the passed value which is the speed and checks whether this is within the limits of the vehicle being considered.
- `setGear`: This is also a function pointer to a function definition consisting of one integer argument and one integer return value. This function examines the passed value which is the gear level and checks whether this is within the range of values the gear can take for the vehicle being considered.
- `setPassengers`: This is the final function pointer member that points to a function definition consisting of one integer argument and one integer return value. This function examines the passed value which is the number of passengers in the vehicles and checks whether this is within the acceptable limits of the vehicle being considered.

You will define the function declared in `vInterface.h` as follows:

- `struct vInterface initVInterface(int (*sS)(int), int (*sG)(int), int (*sP)(int))`: A structure variable of type `vInterface` is created within this function and all members are set to the values passed as arguments to this function (i.e. `sS` is assigned to `setSpeed`, `sG` is assigned to `setGear` and `sP` is assigned to `setPassengers`). This structure variable which is analogous to an *interface* instant in OOP. This is returned to the function call.

Now that you have defined the general `Vehicle` structure and interface, the next step is to define a subclass called `Cars`. All associated definitions and declarations are available in `car.h` and `car.c` files. A class consisting of only cars is a specific type of vehicle class. Hence, the `Car` class *inherits* the `Vehicle` class as per the definition of the structure `Car` in the `car.h` file as follows:

- The first member is `veh` which is a structure of type `Vehicle`. With this member, it is evident that the class `Car` inherits attributes and interfaces of the parent class `Vehicle`.
- The second member is `hasNav` which is an integer. This variable indicates whether an instance of `Car` has a navigation system or not.
- The final member is `setCarState` which is a function pointer to the function definition that sets various attributes of a `Car` instance.

You will implement the function definitions in `car.c` file as follows:

- First, you will define the three functions that form the interfaces to a `Car` instance: `setSpeed` (returns the argument if it is lesser than or equal to 260 otherwise -1 is returned), `setGear` (returns the argument if it is lesser than or equal to 5 otherwise -1 is returned), `setPassengers` (returns argument if it is lesser than or equal to 5 otherwise -1 is returned). These are interface functions that are specific to a car which check whether a state value satisfies the associated restrictions.
- Next, you will implement the function that initializes an instance of a car. `struct Car * initCar(int nav, char * Model)`: In this function, a structure of type `Car` is dynamically allocated and every member is initialized. The first member `veh` is initialized by evoking the `initVehicle` function that you had previously defined. You will pass in the values 4, `model` and the `vInterface` structure. You will create the `vInterface` structure by evoking `initVInterface` function and passing in the appropriate pointers to the interface functions you have defined above. The second member `hasNav` is set to `nav` which is passed as an argument and the third member is set to the function `setCarState` that you will define next. This function is analogous to a *constructor* in OOP.
- Next, you will define the function `setCarState` which assigns various values to the attributes of an instance of a `Car` as follows. `void setCarState(struct Car * c, int speed, int gear, int passengers, int nav)`: The first argument is a `Car` instance. To it you will assign state values to members of `veh` instance using the interface functions set in `veh->vehInt`. The value `nav` is regularly assigned to the `hasNav` member.
- Next, you will implement the function `void printCarState(struct Car * c)` that prints the current state of all members of the `Car` instance `c`. Please refer to the expected output file for the correct formatting. Finally, you will implement the function `void cleanUpCar(struct Car *`

c) that frees up all dynamically allocated space associated with the instance `c` that is passed as an argument to the function.

We will test this part by evoking the function `p1` in the `main.c` file. The output resulting from executing the following must match `p1.txt` in the `expCode` folder:

- `valgrind --quiet --leak-check=full --track-origins=yes ./run 1`

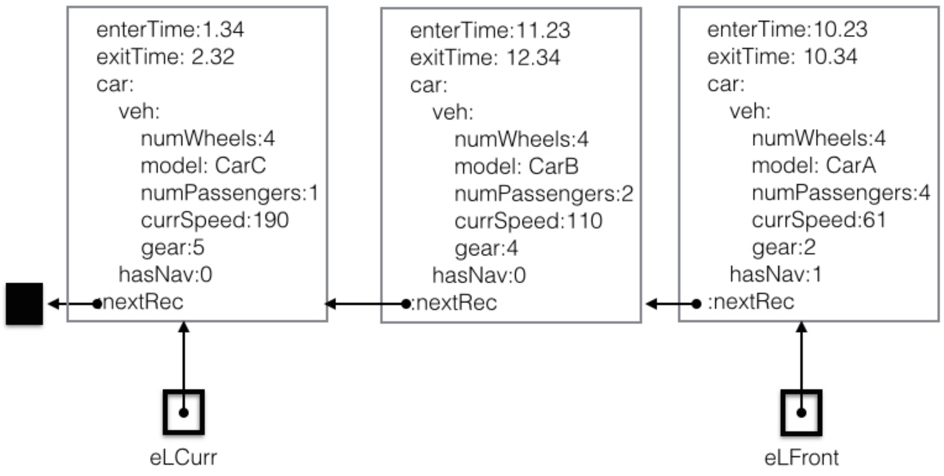


Figure 3: Example of Record List

Part 2: Modelling an Express Lane in a Highway

Files associated with this part are `expressLane.h` and `expressLane.c`. In `expressLane.h`, a structure called `ExpressLaneRec` is defined. This structure is used to keep track of every car that enters an express lane in a highway. The members of this structure is as follows:

- `enterTime`: This is a float variable that stores the time that a car has entered the express lane.
- `exitTime`: This is a float variable that stores the time at which the car leaves the lane
- `car`: This is a pointer variable storing the address of this car instance.
- `nextRec`: This is a pointer to the next record that keeps track of information about the next car that enters the express lane.

You will define the functions associated with this structure as follows:

- `struct ExpressLaneRec * initExpressLane(float enterTime, float exitTime)`: In this function, a new record is created via dynamic memory allocation. Each member of this record is initialized in this function. The members `enterTime` and `exitTime` are initalized to the two arguments passed to this function. The members `car` and `nextRec` are set to NULL. The pointer to the dynamically allocated structure is returned to the function call.
- `struct ExpressLaneRec * addCarRec(struct Car * c, float enterTime, float exitTime, struct ExpressLaneRec * eL)`: In this function, you will implement the function that creates a new record and assigns values to members of the new record. First, a new record is initialized using the `initExpressLane` function. Then, the member `car` is set to `c`. Then, this record is set as the `nextRec` of `eL`. Finally, the pointer to the newly created record is returned to the function call.

- `void printRecords(struct ExpressLaneRec * eLCurr)`: This function prints the content of all records starting from the first pointer `eLCurr`. For formatting of the output, please refer to p2.txt.
- `void cleanUpRec(struct ExpressLaneRec * eL)`: This function will free all dynamically allocated space starting from the first record `eL`.

Please refer to Fig. 3 that illustrates the set of records created via the statements (right before the call to `printRecords`) executed in the `p2` function of `main.c`. All implementations for this part will be checked via the function call to `p2` in main.c. The output resulting from executing:

- `valgrind --quiet --leak-check=full --track-origins=yes ./run 2`

must match `p2.txt` in the `expCode` folder.

3 Code Submission

- Log onto your ECF account
- Ensure that your completed code compiles
- Browse into the directory that contains your completed code (`vehicle.c`, `vInterface.c`, `car.c`, `expressLane.c`)
- Submit by issuing the command:
  `submitcsc190s 2 vehicle.c vInterface.c car.c expressLane.c`