Bikes and Cars

Use Cases

1. Sourabh has two vehicles. One is a honda accord car and another is Ducati Bike.
2. Honda Accord car runs on a fuel called diesel.
3. Ducati runs on a fuel called petrol
4. Ducati is a used and imported vehicle
5. Honda Accord is new and Made In India

Solution

1. From the Use-Cases, we can see a person named Sourabh has two vehicles. We can identify three different entities which are vehicles(of different types), the Person, and the Owner. Thus we create a class to represent a Person and a class to represent the owner of a vehicle. We also create a class for the vehicle which inherit from an abstract vehicle class.

```
#include <iostream>
#include <bits/stdc++.h>
using namespace std ;

struct Origin {
    string FromIndia = "From India" ;
    string Imported = "Imported" ;
} ;

struct FuelType{
    string Petrol = "Petrol" ;
    string Diesel = "Diesel" ;
};

struct History{
    string Used = "Used" ;
    string New = "New" ;
};


// Class to represent Person
class Person{
    protected:
        string name ;
    public:
```

```
        Person(string name) : name(name){}
        string getName(){
            return this -> name ;
        }
} ;

// abstract class to represent the Vehicle
class VehicleAbstract{
    protected:
        string type ;
        string name ;
        string fuelType ;
        string origin ;
        string history ;
    public:
        virtual string getType() = 0 ;
        virtual string getName() = 0 ;
        virtual string getFuelType() = 0 ;
        virtual string getOrigin() = 0 ;
        virtual string getHistory() = 0 ;
} ;

// Vehicle class that implements the VehicleAbstract class
class Vehicle: public VehicleAbstract{
    public:
        Vehicle(string name, string fuelType, string origin, string history){
            this -> name = name ;
            this -> fuelType = fuelType ;
            this -> origin = origin ;
            this -> history = history ;
        }
        string getType(){
            return this -> type ;
        }
        string getName(){
                return this -> name ;
        }
        string getFuelType(){
                return this -> fuelType ;
        }
        string getOrigin(){
                return this -> origin ;
        }
        string getHistory(){
            return this -> history ;
        }
} ;

// Class to represent the owner of a vehicle
class VehicleOwner: public Person{

    private:
        vector<Vehicle> owned_vehicles;
    public:
        VehicleOwner(string name) : Person(name){
```

```
            // add additional functionality when creating an owner
        }
        void addVehicle(Vehicle vehicle){
            // code to add new vehicle
            (this->owned_vehicles).push_back(vehicle) ;
        }
        void printVehicles(){
            cout<<"Vehicle owned by "<<this -> name<<": \n\n" ;
            for(int i = 0 ; i < (this -> owned_vehicles).size() ; i++){
                cout<<"Vehicle " << i + 1<< "\nName: "<<owned_vehicles[i].getName()<<
                ", Type: "<<owned_vehicles[i].getType()<<
                ", Fuel Type: "<<owned_vehicles[i].getFuelType()<<
                ", History: "<<owned_vehicles[i].getHistory()<<endl<<endl ;
            }
            cout<<endl ;
        }
} ;
```

2. We can identify there are two type of vehicles - **Car** and **Bike**. Hence we need to create the classes for each type. These class inherit from the Vehicle Class.

```
class Car: public Vehicle{
    public:
        Car(string name, string fuelType, string origin, string history) :
            Vehicle(name, fuelType, origin, history){
                this -> type = "Car" ;
        }
} ;

class Bike: public Vehicle{
    public:
        Bike(string name, string fuelType, string origin, string history) :
            Vehicle(name, fuelType, origin, history){
                this -> type = "Bike" ;
        }
} ;
```

3. Now we just need to implement the Use-Cases

```
int main() {
    Origin origin ;
    FuelType fuelType ;
    History history ;

    Vehicle accord = Car("Honda Accord", fuelType.Diesel, origin.FromIndia, history.New) ;
    //cout<<accord.getType()<<accord.getOrigin()<<endl;

    Vehicle ducati = Bike("Ducati", fuelType.Petrol, origin.Imported, history.Used) ;
    //cout<<ducati.getType()<<ducati.getOrigin()<<endl;
```

```
    VehicleOwner sourabh = VehicleOwner("Sourabh") ;
    // cout<<s.getName() ;
    // Adding Both the vehicle in the collection of owned vehicles of Saurabh
    sourabh.addVehicle(accord) ;
    sourabh.addVehicle(ducati) ;
    sourabh.printVehicles() ;
    return 0;
}
```

How OOPs implemented

There are main 4 pillars in any OOPs design

1. **Abstraction**: Data Abstraction is the property by virtue of which only the essential details are displayed to the user. We achieves Abstraction by Implementing the getters-setters methods in the respective classes to change the object attributes.
2. **Encapsulation**: It is defined as the wrapping up of data under a single unit. It is the mechanism that binds together the code and the data it manipulates. We have achieved encapsulation by defining classes with atomic behavior. Also, access modifiers are used to provide a level of data hiding.
3. **Inheritance**: It is the mechanism in which one class is allowed to inherit the features (fields and methods) of another class. In our code, The VehicleOwner class is inheriting the Person class. Car and Bike classes are inheriting the Vehicle class. This way we achieve Inheritance.
4. **Polymorphism:**
    a. **Runtime Polymorphism (Overriding)**: Overriding is a feature that allows a subclass or child class to provide a specific implementation of a method that is already provided by one of its super-classes or parent classes.

    In our code, we have overridden the methods inherited (virtual functions) from VehicleAbstract class. This way we achieve polymorphism.