Assignment No. 08

- **TITLE -** Design and develop a context for given case study and implement an interface for Vehicles Consider the example of vehicles like bicycle, car and bike. All Vehicles have common functionalities such as Gear Change, Speed up and apply brakes. Make an interface and put all these common functionalities. Bicycle, Bike, Car classes should be implemented for all these functionalities in their own class in their own way.

- **THEORY –**

This C++ program demonstrates the **concept of interfaces using pure virtual functions** and the **implementation of polymorphism** through different vehicle classes — `Bicycle`, `Bike`, and `Car`. The base class `Vehicle` acts as an **interface**, containing only pure virtual functions: `gear_change()`, `speed_up()`, `apply_brakes()`, and `display()`. These functions define a common structure or behavior that all types of vehicles must implement, but without providing any actual implementation. This ensures that every derived class provides its own specific logic for these actions.

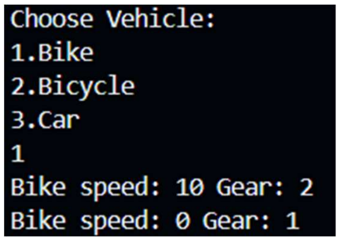
Each derived class — `Bike`, `Bicycle`, and `Car` — overrides the functions of the `Vehicle` interface according to its characteristics. For example, the `Bike` increases speed by 10 units and the `Car` by 20 units when accelerating, while the `Bicycle` increases speed by only 4 units. Similarly, the gear-changing logic and braking mechanisms differ slightly for each vehicle type, reflecting real-world differences in how each operates. This demonstrates how **abstraction and polymorphism** can model varied behaviors in an organized way using a single unified interface.

In the `main()` function, a pointer of type `Vehicle*` is used to invoke functions dynamically at runtime, based on the user's choice. This means that even though the pointer type is the same, it can refer to different objects (`Bike`, `Bicycle`, or `Car`), and the appropriate version of the methods will execute — showcasing **runtime polymorphism (dynamic binding)**. The program effectively highlights how interfaces provide a flexible and scalable design for systems where multiple entities share common functionalities but implement them differently.

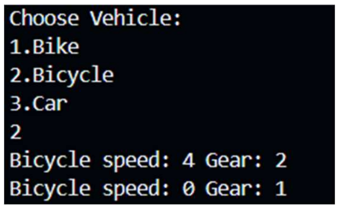
- **CODE –**

```
#include<iostream>
using namespace std;

class Vehicle{
    public:
    virtual void gear_change()=0;
    virtual void speed_up()=0;
    virtual void apply_brakes()=0;
    virtual void display()=0;
};

class Bike : public Vehicle{
    int speed;
    int gear;
    public:
    Bike(){
        speed=0;
        gear=1;
    }
    void gear_change(){
        gear++;
        if(gear>5){
            gear=5;
        }
    }
    void speed_up(){
        speed+=10;
    }
    void apply_brakes(){
        speed-=10;
        gear--;
    }
    void display(){
        cout<<"Bike speed: "<<speed<<" Gear: "<<gear<<endl;
    }
};

class Bicycle : public Vehicle{
    int speed;
    int gear;
    public:
    Bicycle(){
        speed=0;
        gear=1;
    }
    void gear_change(){
        gear++;
        if(gear>5){
            gear=5;
        }
    }
    void speed_up(){
        speed+=4;
    }
    void apply_brakes(){
        speed-=4;
        gear--;
    }
    void display(){
        cout<<"Bicycle speed: "<<speed<<" Gear: "<<gear<<endl;
    }
};

class Car : public Vehicle{
    int speed;
    int gear;
    public:
    Car(){
        speed=0;
        gear=1;
    }
    void gear_change(){
        gear++;
        if(gear>5){
            gear=5;
        }
    }
    void speed_up(){
        speed+=20;
    }
    void apply_brakes(){
        speed-=20;
        gear--;
    }
    void display(){
        cout<<"Car speed: "<<speed<<" Gear: "<<gear<<endl;
    }
};

int main(){
    int ch;
    while(true){
    cout<<"Choose Vehicle:\n1.Bike\n2.Bicycle\n3.Car\n";
    cin>>ch;
    Vehicle *v;
    switch(ch){
        case 1:
            v=new Bike();
            v->gear_change();
            v->speed_up();
            v->display();
            v->apply_brakes();
            v->display();
            break;
        case 2:
            v=new Bicycle();
            v->gear_change();
            v->speed_up();
            v->display();
            v->apply_brakes();
            v->display();
            break;
        case 3:
            v=new Car();
            v->gear_change();
            v->speed_up();
            v->display();
            v->apply_brakes();
            v->display();
            break;
        default:
            cout<<"Invalid choice"<<endl;
            return 0;
    }
    }
    return 0;
}
```

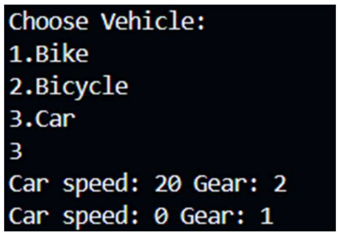
➢ **OUTPUT –**



```
Choose Vehicle:
1.Bike
2.Bicycle
3.Car
1
Bike speed: 10 Gear: 2
Bike speed: 0 Gear: 1
```



```
Choose Vehicle:
1.Bike
2.Bicycle
3.Car
2
Bicycle speed: 4 Gear: 2
Bicycle speed: 0 Gear: 1
```



```
Choose Vehicle:
1.Bike
2.Bicycle
3.Car
3
Car speed: 20 Gear: 2
Car speed: 0 Gear: 1
```

➢ **CONCLUSION –**

This program effectively demonstrates the concept of **interfaces and polymorphism** in object-oriented programming. By defining a common interface Vehicle and implementing it differently in Bike, Bicycle, and Car, the code achieves flexibility, reusability, and real-world modeling of diverse behaviors. It clearly shows how dynamic binding allows the same interface to control multiple object types efficiently.