Name	ADITI RAO
UID no.	202220003
Experiment No.	7

AIM:	**Program on Abstraction: Implement a Program to demonstrate Abstraction using abstract class**
Program 1	
PROBLEM STATEMENT:	Create a base class as a Vehicle. The Vehicle class has wheels and engine capacity as data members.and two pure virtual functions spec() to set the values for data members and display_stats() to display the values assigned. Create classes LMV(Light Motor Vehicle),HMV(Heavy Motor Vehicle) and TW(Two Wheeler) publicly derived from the Vehicle class. Include variables like speed,mileage and rpm in the derived classes and override the virtual methods in these classes.Also have constructor initializing the values to 0 as default and a virtual destructor for the classes.In main create an array of pointers of the base class and set them to the objects of the derived classes. Now make a call to the various methods for these objects using the base class pointer. Delete the objects created to show the appropriate destructor calls.
ALGORITHM:	CLASS Vehicle INTEGER wheels INTEGER engineCapacity CONSTRUCTOR Vehicle() SET wheels = 0 SET engineCapacity = 0 CONSTRUCTOR Vehicle(wheels, engineCapacity) SET this.wheels = wheels SET this.engineCapacity = engineCapacity ABSTRACT FUNCTION specs() ABSTRACT FUNCTION display_stats() DESTRUCTOR ~Vehicle() CLASS LMV INHERITS Vehicle

```
    PROTECTED INTEGER speed
    PROTECTED INTEGER mileage
    PROTECTED FLOAT rpm

    CONSTRUCTOR LMV()
      SET speed = 0
      SET mileage = 0
      SET rpm = 0.0

    FUNCTION specs()
      OUTPUT "Enter the number of wheels: "
      INPUT wheels
      OUTPUT "Enter the engine capacity: "
      INPUT engineCapacity
      OUTPUT "Enter the speed: "
      INPUT speed
      OUTPUT "Enter the mileage: "
      INPUT mileage
      OUTPUT "Enter the RPM: "
      INPUT rpm

    FUNCTION display_stats()
      OUTPUT "Number of wheels: " + wheels
      OUTPUT "Engine capacity: " + engineCapacity
      OUTPUT "Speed: " + speed
      OUTPUT "Mileage: " + mileage
      OUTPUT "RPM: " + rpm

    DESTRUCTOR ~LMV()

  CLASS HMV INHERITS Vehicle
    PROTECTED INTEGER speed
    PROTECTED INTEGER mileage
    PROTECTED FLOAT rpm

    CONSTRUCTOR HMV()
      SET speed = 0
      SET mileage = 0
      SET rpm = 0.0

    FUNCTION specs()
      OUTPUT "Enter the number of wheels: "
      INPUT wheels
```

```
        OUTPUT "Enter the engine capacity: "
        INPUT engineCapacity
        OUTPUT "Enter the speed: "
        INPUT speed
        OUTPUT "Enter the mileage: "
        INPUT mileage
        OUTPUT "Enter the RPM: "
        INPUT rpm

    FUNCTION display_stats()
        OUTPUT "Number of wheels: " + wheels
        OUTPUT "Engine capacity: " + engineCapacity
        OUTPUT "Speed: " + speed
        OUTPUT "Mileage: " + mileage
        OUTPUT "RPM: " + rpm

    DESTRUCTOR ~HMV()

CLASS TW INHERITS Vehicle
    PROTECTED INTEGER speed
    PROTECTED INTEGER mileage
    PROTECTED FLOAT rpm

    CONSTRUCTOR TW()
        SET speed = 0
        SET mileage = 0
        SET rpm = 0.0

    FUNCTION specs()
        OUTPUT "Enter the number of wheels: "
        INPUT wheels
        OUTPUT "Enter the engine capacity: "
        INPUT engineCapacity
        OUTPUT "Enter the speed: "
        INPUT speed
        OUTPUT "Enter the mileage: "
        INPUT mileage
        OUTPUT "Enter the RPM: "
        INPUT rpm

    FUNCTION display_stats()
        OUTPUT "Number of wheels: " + wheels
        OUTPUT "Engine capacity: " + engineCapacity
```

	OUTPUT "Speed: " + speed OUTPUT "Mileage: " + mileage OUTPUT "RPM: " + rpm DESTRUCTOR ~TW() FUNCTION main() CREATE Vehicle array vehicles with size 3 ASSIGN new LMV instance to vehicles[0] ASSIGN new HMV instance to vehicles[1] ASSIGN new TW instance to vehicles[2] FOR i = 0 to 2 CALL specs() on vehicles[i] IF wheels of vehicles[i] = 2 CREATE TW pointer tw and CAST vehicles[i] to TW IF tw is not null CALL display_stats() on tw ELSE CALL display_stats() on vehicles[i] FOR i = 0 to 2 DELETE vehicles[i] RETURN 0

PROGRAM:	#include <iostream>
	using namespace std;
	
	class Vehicle
	{
	public:
	int wheels;
	int engineCapacity;
	
	public:
	
	Vehicle(): wheels(0), engineCapacity(0) {}
	Vehicle(int wheels, int engineCapacity): wheels(wheels), engineCapacity(engineCapacity) {}
	virtual void specs() = 0;
	virtual void display_stats() = 0;

```
        virtual ~Vehicle() {}
};

class LMV : public Vehicle
{
   protected:
      int speed;
      int mileage;
      float rpm;

   public:
      LMV(): speed(0), mileage(0), rpm(0.0) {}
      void specs()
      {
         cout << "Enter the number of wheels: ";
         cin >> wheels;
         cout << "Enter the engine capacity: ";
         cin >> engineCapacity;
         cout << "Enter the speed: ";
         cin >> speed;
         cout << "Enter the mileage: ";
         cin >> mileage;
         cout << "Enter the RPM: ";
         cin >> rpm;
      }

      void display_stats()
      {
         cout << "Number of wheels: " << wheels << endl;
         cout << "Engine capacity: " << engineCapacity << endl;
         cout << "Speed: " << speed << endl;
         cout << "Mileage: " << mileage << endl;
         cout << "RPM: " << rpm << endl;
      }

      ~LMV() {}
};

class HMV : public Vehicle
{
   protected:
      int speed;
      int mileage;
```

```
    float rpm;

  public:
    HMV(): speed(0), mileage(0), rpm(0.0) {}
    void specs()
    {
      cout << "Enter the number of wheels: ";
      cin >> wheels;
      cout << "Enter the engine capacity: ";
      cin >> engineCapacity;
      cout << "Enter the speed: ";
      cin >> speed;
      cout << "Enter the mileage: ";
      cin >> mileage;
      cout << "Enter the RPM: ";
      cin >> rpm;
    }

    void display_stats()
    {
      cout << "Number of wheels: " << wheels << endl;
      cout << "Engine capacity: " << engineCapacity << endl;
      cout << "Speed: " << speed << endl;
      cout << "Mileage: " << mileage << endl;
      cout << "RPM: " << rpm << endl;
    }

    ~HMV() {}
};

class TW : public Vehicle
{
  protected:
    int speed;
    int mileage;
    float rpm;

  public:

    TW(): speed(0), mileage(0), rpm(0.0) {}
    void specs()
    {
      cout << "Enter the number of wheels: ";
```

```
        cin >> wheels;
        cout << "Enter the engine capacity: ";
        cin >> engineCapacity;
        cout << "Enter the speed: ";
        cin >> speed;
        cout << "Enter the mileage: ";
        cin >> mileage;
        cout << "Enter the RPM: ";
        cin >> rpm;
    }

    void display_stats()
    {
        cout << "Number of wheels: " << wheels << endl;
        cout << "Engine capacity: " << engineCapacity << endl;
        cout << "Speed: " << speed << endl;
        cout << "Mileage: " << mileage << endl;
        cout << "RPM: " << rpm << endl;
    }

    ~TW() {}
};


int main()
{
    Vehicle *vehicles[3];
    vehicles[0] = new LMV();
    vehicles[1] = new HMV();
    vehicles[2] = new TW();

    for (int i = 0; i < 3; i++)
    {
        vehicles[i]->specs();

        if (vehicles[i]->wheels == 2)
        {
            TW *tw = dynamic_cast<TW*>(vehicles[i]); //If the vehicle is a two-
wheeler, dynamically cast the value of specs() of LMV to TW
            if (tw)
            {
                tw->display_stats();
            }
```

```
        }
        else
        {
            vehicles[i]->display_stats();
        }
    }

    for (int i = 0; i < 3; i++)
    {
        delete vehicles[i];
    }

    return 0;
}
```

RESULT:

```
PS C:\Tesseract\Under Graduate Engineering Degree\SPIT\Semester II\PSOO
PL\Experiment 8\" ; if ($?) { g++ Program1_Vehicle.cpp -o Program1_Vehi
Enter the RPM: 23
Enter the number of wheels: 4
Enter the engine capacity: 214
Enter the speed: 53
Enter the mileage: 523
Enter the RPM: 43
Number of wheels: 4
Engine capacity: 214
Speed: 53
Mileage: 523
RPM: 43
Enter the number of wheels: 2
Enter the engine capacity: 123
Enter the number of wheels: 4
Enter the engine capacity: 123
Enter the speed: 421
Enter the mileage: 23
Enter the RPM: 123
Number of wheels: 4
Engine capacity: 123
Speed: 421
Mileage: 23
RPM: 123
Enter the number of wheels: 4
Enter the engine capacity: 421
Enter the speed: 231
Enter the mileage: 532
Enter the RPM: 64
Number of wheels: 4
Engine capacity: 421
Speed: 231
Mileage: 532
RPM: 64
Enter the number of wheels: 2
Enter the engine capacity: 352
Enter the speed: 124
Enter the mileage: 532
Enter the RPM: 124
Number of wheels: 2
Engine capacity: 352
Speed: 124
Mileage: 532
RPM: 124
```

Program 2

PROBLEM STATEMENT:

A liter is 0.264179 gallons. Write a program that will read in the number of liters of gasoline consumed by the user's car and the number of miles traveled by the car, and will then output the number of miles per gallon the car delivered. Your program should allow the user to repeat this calculation as often as the user wishes. Define a function to compute the number of miles per gallon. Your

	program should use a globally defined constant for the number of liters per gallon
ALGORITHM:	CLASS Car PRIVATE: litres: DOUBLE miles: DOUBLE PUBLIC: METHOD setFuelConsumption(litres: DOUBLE) this->litres = litres METHOD setMileage(miles: DOUBLE) this->miles = miles METHOD calculateMilesPerGallon() : DOUBLE gallons = litres / LITRES_PER_GALLON miles_per_gallon = miles / gallons RETURN miles_per_gallon ENDCLASS FUNCTION main() : INTEGER car: Car repeat: CHARACTER DO litres: DOUBLE miles: DOUBLE OUTPUT "Enter the number of litres of gasoline consumed: " INPUT litres car.setFuelConsumption(litres) OUTPUT "Enter the number of miles traveled: " INPUT miles car.setMileage(miles) miles_per_gallon = car.calculateMilesPerGallon() OUTPUT "The car delivered " + miles_per_gallon + " miles per gallon." OUTPUT "Do you want to calculate again? (Y/N): " INPUT repeat OUTPUT

	WHILE repeat == 'Y' OR repeat == 'y' RETURN 0 ENDFUNCTION

PROGRAM:	#include <iostream>
	using namespace std;
	
	const double LITRES_PER_GALLON = 0.264179;
	
	class Car {
	private:
	double litres;
	double miles;
	
	public:
	void setFuelConsumption(double litres) {
	this->litres = litres;
	}
	
	void setMileage(double miles) {
	this->miles = miles;
	}
	
	double calculateMilesPerGallon() {
	double gallons = litres / LITRES_PER_GALLON;
	double miles_per_gallon = miles / gallons;
	return miles_per_gallon;
	}
	};
	
	int main() {
	Car car;
	char repeat;
	
	do {
	double litres, miles;
	
	cout << "Enter the number of litres of gasoline consumed: ";
	cin >> litres;
	car.setFuelConsumption(litres);

```
        cout << "Enter the number of miles traveled: ";
        cin >> miles;
        car.setMileage(miles);

        double miles_per_gallon = car.calculateMilesPerGallon();
        cout << "The car delivered " << miles_per_gallon << " miles per gallon." <<
endl;

        cout << "Do you want to calculate again? (Y/N): ";
        cin >> repeat;
        cout << endl;
    }
    while (repeat == 'Y' || repeat == 'y');

    return 0;
}
```

RESULT:

```
PS C:\Tesseract\Under Graduate Engineering Degree\SPIT\Sem
PL\Experiment 8\" ; if ($?) { g++ Program2_LitresperGallon
Enter the number of litres of gasoline consumed: 50
Enter the number of miles traveled: 300
The car delivered 1.58507 miles per gallon.
Do you want to calculate again? (Y/N): y

Enter the number of litres of gasoline consumed: 30
Enter the number of miles traveled: 200
The car delivered 1.76119 miles per gallon.
Do you want to calculate again? (Y/N): n

PS C:\Tesseract\Under Graduate Engineering Degree\SPIT\Sem
```

CONCLUSION:

In the first program, abstraction is demonstrated through the use of abstract functions within the Vehicle class. Abstraction is a fundamental principle of object-oriented programming that allows us to define a common interface or contract for a group of related classes, without providing the implementation details.

The Vehicle class includes two abstract functions: specs() and display_stats(). These functions are declared with the VIRTUAL keyword, indicating that they must be implemented in the derived classes (LMV, HMV, TW) that inherit from the Vehicle class.

By defining these functions as abstract, the Vehicle class provides a common interface that all its derived classes must adhere to. However, the specific implementation details of these functions are left to the derived classes. This

	abstraction allows us to work with objects of the Vehicle class without worrying about the specific implementation details, promoting modularity and encapsulation. Also, in the second program, In this implementation, a Car class is defined with private member variables litres and miles to represent the fuel consumption and mileage of the car, respectively. The class provides public methods setFuelConsumption() and setMileage() to set the values of the fuel consumption and mileage, and a calculateMilesPerGallon() method to calculate the miles per gallon based on the provided values. By encapsulating the data and behavior related to the car's fuel consumption and mileage within the Car class, we achieve abstraction. The main function interacts with the Car object through the simplified public interface, without needing to know the underlying implementation details.