APCO
Date :
Page :

OBJECT ORIENTED PROGRAMMING:

The object oriented approach views a problem in terms of objects rather than procedure for doing it.

Object oriented programming allows the programming to be closer to real world and thereby making it less complex.

Classes And Objects:

A class is a way to bind the data describing entity and its associated functions together.
For instance, consider a user having characteristics username and password and some of its associated operations are sign up, sign in and logout.

Class is just a template, which declares and defines characteristics and behaviour, hence we need to declare objects of the class for it to be usable. In other words class represent a group of similar object.

Data members and member function:

Data members are the data-type properties that describe the characteristics of the class.

Member functions are the set of operation that

may be applied to Objects of class, i.e, the represent the behavioral aspect of the Object.

Syntax for defining class:

```
class classname{

    Access Modifier:
        Data members
        member functions
};
```

The class body contains the declaration of members (data & functions).

Declaring Objects of a class:

We can declare Objects of class either statically or dynamically just as we declare variable of primitive data types.

(i) Statically:

```
class_name Object_name;
```
] Syntax

(ii) Dynamically:

```
class_name *Object_name = new class_name
```
] Syntax

APCO
Date :
Page :

Example code for declaring Statically:

```
class Student {
  public:
    int rollno;
    char name[20];
};
int main (){
   Student S1;      // Declaring Objects.
   Student S2;
}
```

Example code for declaring dynamically:

```
class Student{
   public:
     int rollno;
     char names[20];
};
int main (){
Student *s1 = new Student  // Declaring of object
                           // of type student
                           // dynamically.
```

Access Modifiers:

- The **private** members can be accessed only from within the class. These members are hidden from the outside world. Hence they can be used only by member functions of the class in which it is declared.

- The **public** members can be accessed outside the class also. They can be directly accessible by any function, whether member function of class or non member function.

- By default members of class are private.

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

class student{          // Declaring class
    public:             // Access modifiers
      int rollno;
      int age;
};

int main(){

  student S1;           // Create objects statically
  student S2;
```

```
Student S3, S4, S5;          // Can be declared like this also

S1.age = 24;                 // Assigning values
S1.rollno= 101;

cout << S1.age << endl;      // Printing value
cout << S1.rollno << endl;

Student *S6 = new Student;   // Create objects dynamicly

(*S6).age = 23;              // Assigning values
(*S6).rollno = 104;

// Can also be done in a way given below:

S6 -> age = 23;
S6 -> rollno = 104;

}
```

Getter And Setters:

The private members of class are not accessible outside the class, although sometimes there is necessity to provide access to even private members, in these cases we need to create functions called getters and setters. They are also called as accessor and mutator function respectively.

Example Code:

```
class Student {

    int rollno;
    char name[20];
    float marks;
    char grade;

public:

int getRollno(){

        return rollno;}

int getMarks(){
        return marks;
        }

void setGrade(){

    if (marks >90) grade = 'A';
else if (marks >80) grade = 'B';
else if (marks >70) grade = 'C';
else if (marks >60) grade = 'D';
else grade = 'E';
    }
};
```

getRollno() & getMarks are getter & setGrade is setter.

If we need to define member function outside class:

```
int Student :: getMarks ()
{
    return marks;
}

int Student :: getRollno ()
{
    return rollno;
}
```

We can access member functions in similar manner via an object of class Student using dot operator.

APCO
Date :
Page :

Constructor:

A constructor in a class is means of initialisation or creating objects of a class. A constructor allocates memory to the data members when an object is created. It may also initialise the object with legal initial value.

Characteristics of Constructor:

- Constructor is a member function of a class and has same name as that of class.
- Constructor functions are invoked automatically when the objects are created.
- Constructor functions obeys the usual access rules. That is, private constructor are available only for member functions, however, public are available for all functions.
- Constructor has no return type, not even void.

APCO
Date :
Page :

T/default Constructor:

A constructor that accepts no parameter is called default constructor. The compiler automatically supplies a default constructor implicitly. This constructor is called public member of class.

Example code:

```
class Sum{
  int a, b
  Public:
   int getSum(){

    return a+b;
};
int main(){
   Sum obj; // Implicit default constructor invoked
}
```

- Creating your own default constructor:

When a user-defined default constructor is created, the compiler's implicit default constructor is overshadowed.

Example Code:

```
class Sum{

  int a, b;

public:
    Sum(){                    //User defined default constructor

    cout << "constructor invoked:";
        a = 10;
        b = 20;
    }
int getsum(){
    return a+b;
    }

}

int main(){

    Sum obj; //Explicitly defined default constructor called
}
```

Output = Constructor invoked:

APCO
Date :
Page :

Parameterized Constructor:

The constructor that can take arguments are called parameterized constructor.

Example Code:

```
class Sum{

int a,b;
public:

   Sum ( int num1, int num2 ) {   //Parameterised
                                  // constructor
       a= num1 ;
       b= num2 ;
   }
   int getSum (){
       return a+b ;}
};

int main (){

   Sum obj ( 4, 2 ); //Parameterised constructor is called

}
```

Declaring the constructor with arguments hides the default constructor. Hence, the Object declaring statment such as

Person Obj;

may not work.

- Any number of constructors may be declared and the constructor that would be invoked would be according to the parameters.

Eg:- Student S1; // constructor 1 will be called

Student S2(101); // Constructor 2 will be called

Student S3(20,102); // constructor 3 will be called

Student S4(S3); // Copy constructor

S1 = S2 // Copy assignment Operator

Student S5 = S4; // Copy constructor

Copy constructor is inbuilt.

APCO
Date :
Page :

Destructors:

A destructor as the name suggest is used to destroy the objects that have been created by a constructor. A destructor is also a member function whose name is the same as the class name but preceded by a tilde ('~').
For eg, destructor to class Sum is ~Sum().

- A destructor takes no argument and have no return type.
- It is automatically called by compiler when object is destroyed.
- But for dynamically declared Objects it does not call itself & we have to use keyword delete object_name;.
- As soon as object goes out of scope, destructor is called and object is destroyed releasing its occupied memory.
- Destructors are invoked in reverse order as constructors are invoked.

Example code:

```
class Sum {

    int a, b;

    Public:

Sum (int num1, int num2) {

    cout << "Constructor at work"; }endl;
    ~Sum (){
     cout << "Destructor at work ";
     }
    int getSum(){
        return a+b;
    }
};

int main(){

    Sum obj(4,6);
}
```

Output = Constructor at work
Destructor at work

APCO
Date :
Page :

this keyword:

C++ uses a unique keyword called this to represent an object that invokes member function. this is a pointer to the object for which the function was called. For example, the function calls for obj.getSum() will set the pointer this to the address of the object obj. This unique pointer is automatically passed to a member function when it is called. The pointer this acts as an implicit argument to all the member functions.

Example code:

```
class Sum{
    int a,b;
Public:
    Sum (int a,int b){
    this ~~>~~ -> a = a;
    this -> b = b;
    }
int getSum (){
    return a+b;
    }
};
```

In the constructor of the Sum class, since the data members and data members have same name, this keyword is used to differentiate between the

two. Here this->a refers to data member a of object obj.

Ques: Make a class naming fraction having a fraction number. Also do some operations.

```
Class Fraction{

Private :
int numerator;
int denominator;

Public :

Fraction( int numerator, int denominato)}

        this -> numerator = numerator;
        this -> denominator = denominator;
}

void print (){

    cout << numerator << " | " << denominator<<endl;
}
```

→ Or compass const fraction &

So that no copying takes place.

```
void add ( fraction &f2 ){

int lcm = denominator * f2.denominator;
```

APCO
Date :
Page :

```
int x = lcm / denominator ;
int y = lcm / b2.denominator ;
int num = x * numerator + (y * b2.numerator);
numerator = num;
denominator = lcm;
```
// Updating sum in b1 only.

```
int gcd = 1 ;
int j = min (numerator, denominator);
for ( int i = 1; i <= j ; i++)
    if ( numerator % i == 0 && denominator % i == 0)
        gcd = i ;
```

Finding HCF for simplifying fraction

```
numerator = numerator / gcd;
denominator = denominator / gcd;
}
```

```
int main(){

Fraction f1 (10,2);

Fraction f2 (15,4);

  f1.add(f2);

  f1.print();

  f2 print();

}
```

Output: 35|4
15|4

APCO
Date :
Page :

eg: Class with characters also:

```
#include <iostream>
using namespace std;
#include "Student.cpp"          //Adding class file

int main(){
    char name[] = "abcd";
    Student s1(20, name);
    s1.display();
    name[3] = 'e';
    Student s2(24, name);
    s2.display();
}
```

Class on next page

```
class Student {

    int age;
    char *name;

  Public :

  Student ( int age, char *name ) {

      this --> age = age;

Way:1 [   // Shallow copy

      this --> name = name;

Way:2 [   // deep copy

      this --> name = new char [ strlen(name)+1 ];

        strcpy ( this -> name, name );

  ]

  }

  void display ( ){

  cout << name << " " << age << endl;

  }

};
```

APCO
Date :
Page :

We have to use either method for copying. Output would be as follows:

Way:1 Output =

a b c d	20
a b c e	24
a b c e	20

Way:2 Output =

a b c d	20
a b c e	24
a b c d	20

In way1 all three name, S1 & S2 were pointing to same array. So when a b c d changes to a b c e it reflected not only in S2 but S1 also.

In way2 a new array is created that S2 a pointer in S1 is pointing. So any change in S2 name does not effect it.

APCO
Date :
Page :

Copy Constructor :

eg :- Using previous class Student.cpp with shallow copy

Just make name as public

```
#include<iostream>
using namespace std;
#include "Student.cpp"

int main(){
char name[] = "abcd";
Student S1(20, name);
S1.display();
Student S2(S1); // Default Copy Constructor
S2.name[0] = 'x';
S1.display();
S2.display();
}
```

Output: 20 abcd
20 Xbcd
20 xbcd

** Here even we have done deep copying but Xbcd is only printed for S1 as here when default constructor is called S2 also points to same pointer. So changes in one changes other.

APCO
Date :
Page :

So indirectly default constructor goes for shallow copying instead of deep copying.

So we can create our own copy constructor in that class which would be:

```
Student ( student s) {
    this -> age = s.age
    this -> name = new char[ strlen(s.name)+1];
    strcpy ( this -> new , s.name );
}.
```

~~Also by writing this in class no default constructor~~

Here argument of copy constructor needs copy constructor so as to pass by value.
So pass it by reference i.e

```
Student ( student &s) {
```

But we want so make it const. i.e dont want changes due to this so make

```
Student ( student const &s) {
```

APCO
Date :
Page :

INITIALISATION LIST:

- Whenever there are constant variable or reference variables in class we require initialisation list.

```
class Student {

  Public:
const int roll number;
    int age;
int & x;          // age reference variable

[Student (int r, int age) : rollNumber(r), age(age),
                              x (this->age)
{
};
```

Here while writing code as soon as compiler sees written any Object declaration generates error if ([) bracketed statement is not there.
As this means initialisation & so putting garbage values to all elements in class. Putting garbage value in const & or & variable generates error and so we must use statement.

APCO
Date :
Page :

Constant Functions:

- If a class constant object is created then it can only call constant functions.
- These are the functions that doesnot change any property of current object.
- To make a function constant add const keyword after function name.

Ques:- What will be output:

```
class Student{

  Public:
int rollnumber;
int age      ;
};

int main(){
  Student S1;

  Student const S2 = S1;

  S1.rollnumber = 101;
  S1.age        = 20;

cout << S2.rollnumber << " " << S2.age;
}
```

Output = Garbage Garbage

Ques:- What is the output:

```
class Student {
    int rollnumber;
public:
    int age;
    Student (int r) {
        rollnumber = r;
    }
    int getRollNumber () {
        return rollnumber;
    }
};
int main () {
    Student S1 (101);
    S1.age = 20;
    Student obj S2 = S1;
    cout << S2.getRollNumber ();
}
```

Output = Error [As we only calling non-def. function]

APCO
Date :
Page :

Static Properties of Class:

eg:- class Student{

```
    Public:
    int rollnumber;
    int age;
Static int totalStudents;
};
```

Here in the given class if we see different students S1, S2 ... as objects of class must have different rollnumbers and may have ~~same~~ different age but total number of students must be same. Such kind of properties of class which must be same for all objects of class are declared as static by using keyword static.

Now, if we make different objects S1, S2 they have age and rollnumber only. Static variable such as total students are only mentioned in class & copies of these are not formed in different objects.
We access static data members by:

```
int Student :: totalStudents = 101 ;
```

```
cout << Student :: totalStudents << endl;
```

Also can be accessed by S1.totalStudents;

If we need to ~~add~~ inc. total no of student acc to total objects made we can take help from constructor. For eg:

```
class Student {

Public:
  int rollnumber;
  int age;

  Student() {
    totalStudent++;
  }
```

So, as we declare objects total student value increases by 1

```
  int getRollnumber() {
      return rollnumber; }

  Static int getTotalStudents() {
      return totalStudents;
  }
};
```

Static function can also be made

APCO
Date :
Page :

- To access static function we use same scope resolution operator.

Eg:- `Student :: getTotalStudents();`

- We declare static those data members and functions which we want same for all objects.
- Static functions may only access static properties. i.e static datamembers or static functions.
- Static function does not have any this → pointer access.

Operator Overloading:

In previously created fraction class we have one function:

Fraction add (Fraction const& f2) }

3

If we want to overload this + operator we can rename it as

Fraction operator + (Fraction const& f2) }

So, f1.add(f2) = f1 + f2

~~eg:- Overload Operator ++ (Pre increment)~~

If there are both post & pre increment operator overloading then

~~Function Operator~~

Fraction operator ++ ( ) { || Pre

Fraction operator ++ (int) { || Post

eg:- Class Dynamic array:

```
class DynamicArray {

    int * data;
    int next_index;
    int capacity;

public:

DynamicArray(){                    //Constructor :initialise values

    data = new int [5];
next_index = 0;
    capacity = 5;
}

DynamicArray (DynamicArray const &d) {   //copy
                                         //constructor
this -> data = new int[d.capacity];      for deep
                                         copying
for (int i = 0; i < d.nextindex; i++){
    this -> data[i] = d.data[i];
}
this -> next_index = d.nextindex;
this -> capacity = d.capacity;

}
```

// Copy Assignment Operator
void operator= (DynamicArray const &d) {

    ——————
    —————— Same lines of
    —————— copy constructor

}

void add (int element) { // Adding element to dynamic array

    if (next_index == capacity) {

        int * newData = new int [2 * capacity];

        for (int i=0 ; i < capacity ; i++) {

            newData[i] = data[i];

        }

        delete [] data ;

        data = newData ;

        capacity = capacity * 2 ;

    }

    data [next_index] = element;
    next_index ++ ;

}

```
int get(int i){                    //To print value of certain
                                   //index i.
  if(i<next_index){
    return data[i];
  }
  else{
    return;
  }
}

void add(int i, int element){

  if(i<next_index){
    data[i] = element;
  }
  elseif(i==nexindex){
    add(element);
  }
  else{
    return;
  }
void print(){                      //To print whole array
  for(int i=0; i<next_index; i++){
    cout << data[i] << " ";
  }
}
};
```

APCO
Date :
Page :

*Remaining books after questions

APCO
Date :
Page :

#OOPS CONT. :

Abstraction And Encapsulation:

- Clubbing data and functions of any entity is called encapsulation. Class helps in achieving encapsulation.
- Abstraction is hiding unnecessary details from outside world.

INHERITANCE:

- Inheritance is the capability of one class of things to derive capabilities or properties from another class.
- Access modifier, protected, all the datamembers that are protected can only be used by child class.
- Inheritance allows the addition of additional features to an existing class without modifying it. One can derive a new class (sub class) from an existing one and add new feature to it.
- Only public and protected are accessible to child class from parent class.

APCO
Date :
Page :

- **Syntax:**

```
class Vehicle {

private:
int maxspeed;

protected:
int numtyres;

Public:
string colour;

};
```

Now, if we want to inherit some prop. of class car from vehicle we would write syntax as:

`class car : access-specifier Vehicle{};`

This access-specifier may be public, private or protected.

(i) If access-modifier is public i.e. inheriting publicly:

Private of vehicle → X never inherited
protected of vehicle → protected of car
Pubic of vehicle → public of car.

(ii) If access-modifier is protected:

Private ——→ Not accessible

Public & Protected of vehicle class becomes Protected of car class.

(iii) If access-modifier is Private:

Private ——→ Not accessible.

Public and Protected of vehicle class becomes private of car class.

- If access modifier is not specified then by default it is taken as private.

- If object of ~~car~~ class car is created then first constructor of parent class i.e vehicle is called and then constructor of car class is called.

- In destructor call is just opposite. First of child class then of parent class.

APCO
Date :
Page :

- Types of Inheritance:

(i) Single Inheritance:

(Parent) A
↓
(Child) B

OR

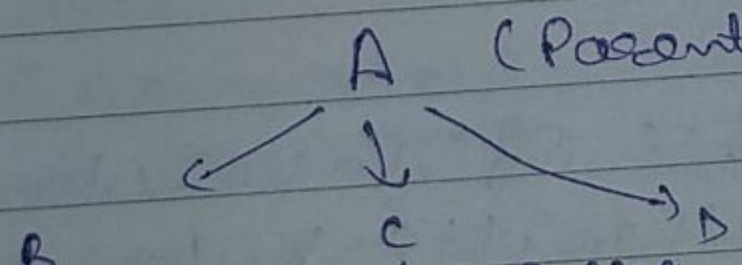
Vehical
↓
Car

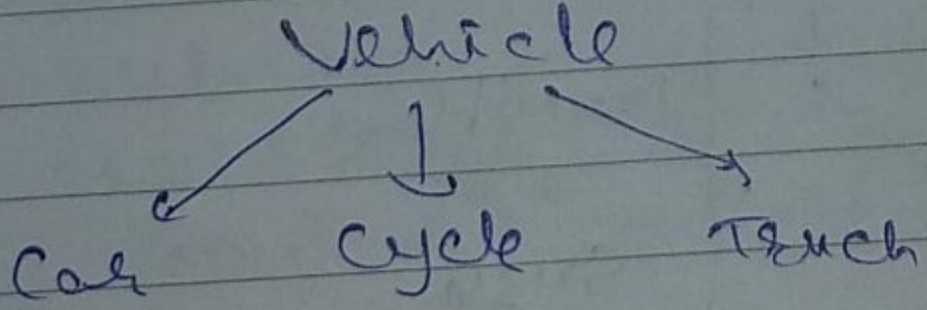
(ii) Multilevel Inheritance:

A
↓
B
↓
C

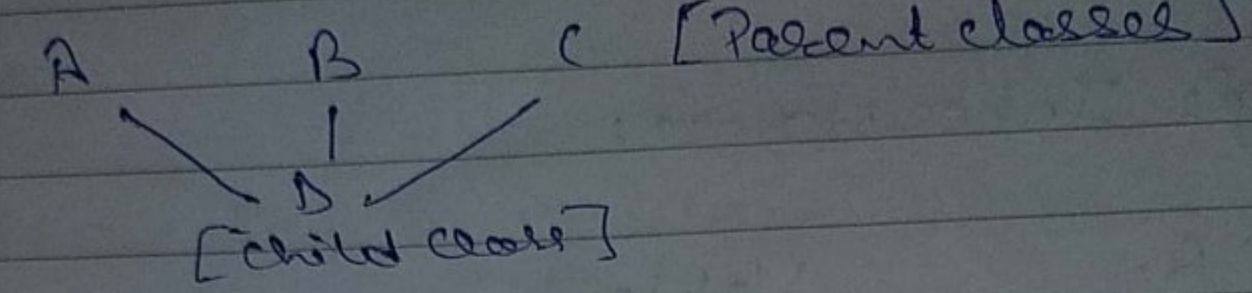
Vehicle
↓
Car
↓
Hondcity

(iii) Hierarchial Inheritance:





(iv) Multiple Inheritance:



class D : accessmodifier A, accessmodifier B, accessmodifier C {}

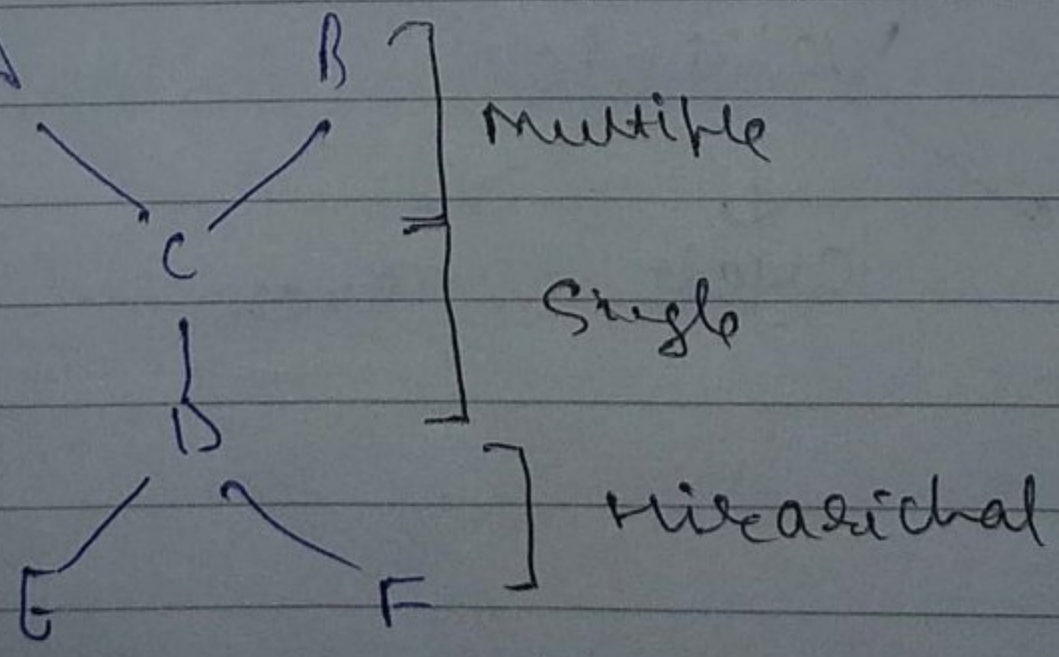
If two function let void print() be there in all A, B, C then

- A::print();

we have to mention which function it called using scope resolution operator.

(v) Hybrid Inheritance:

If two or more types of Inheritance are used together then it is hybrid inheritance.



APCO
Date :
Page :

POLYMORPHISM:

Polymorphism is a property by which the same message can be sent to objects of several different classes, and each object can respond in different way depending on its class.

(i) Compile Time: Polymorphism:

- Compile time polymorphism may be achieved by function overloading and operator overloading.
- Other method is function overiding, in this ib parent class and child class both have function with same name.

eg:- class vehicle {
public:
string colour;
void print () {
cout << "vehicle" << endl; } };

class car : public vehicle {
public:
void print () {
cout << "car" << endl; } };

```
int main(){

    vehicle v;
    car c;

    v.Print();      // Prints vehicle

    c.Print();      // Prints car

    vehicle *v1 = new Vehicle;

    vehicle *v2;

        v2 = &c;

    v1 ---> Print();   // Prints Vehicle

    v2 ---> Print();   // Prints vehicle

}
```

** Here, via V2 only those properties in C are accessible which are there in vehicle class also.

APCO
Date :
Page :

(ii) Run Time : Polymorphism :

- In previous case by V.2 → Point (), vehicle is printed. If we want car to be printed we make use of virtual functions.

- Virtual functions are those functions which are present in base class and they are overridden in derived class. We add keyword virtual before function name.

VIRTUAL FUNCTION AND ABSTRACT CLASS:

- Pure virtual function is one which does not have any definition.
  eg:- virtual void print() = 0;

- Any class that contains pure virtual functions are called abstract class.

- We cannot create object of abstract class.

- For any class to be inherited from abstract class have to either itself also become abstract class or implement all the pure virtual functions.

APCO
Date :
Page :

FRIEND FUNCTIONS AND CLASSES:

- Private members of a class are not accessible outside. But if function of other class wants to access private member of other class, this can be done by making that function friend function of that class. But definition of that function then is given outside the class.

eg:-
```
#include<iostream>
using namespace std;

class Bus{
  public:
  void print();
  };

class Truck{

  private:
    int x;

  Protected:
    int y;

  public;
    int z;

  friend void Bus::print();
};
```

```
void Bus::print(){

    Truck t;
    t.x = 10;
    t.y = 20;
  cout << t.x << " " << t.y << endl;
}

int main(){

  Bus b;

  b.print();

}
```

If we want to make friend a function which is not present in ~~outside~~ any class then we first declare it also we that class then inside class we write

friend void function_name();

then after class we define this function.

- These friend functions cannot use this → pointer.
- We can but these friend function in any access modifier it does not cause any effect.

- If we have two classes class Bus & class Truck and class Bus have large no. of functions which we have to make friend function then either we can do by making seperately each function as friend or we can make whole class as friend by writing in truck class the statement

friend class Bus;