Syntax for inheriting a class in C++:

08.00 class DerivedClass : accessSpecifier BaseClass {

09.00 // class members

10.00 };

11.00

12 00

01.00

02.00

03.00

04.00

05.00

06.00

EVE

A. Base class
{

}

Derive class

B

Notes

07 TUESDAY
Mar 2023 Week 10 / 066-299

Inheritance Constructor in C++.

08.00 → In c++, inheritance is a key notion in OOP. A derived class can inherit the traits and behaviors of
09.00 a base class thanks to the powerful inheritance capability in c++. Inheritance is a powerful
10.00 feature that enables developers to reuse code, improve efficiency, and organize code into
11.00 logical hierarchies.

12.00 → Along with inheritance, unstructors are also essential in c++. A constructor is a unique member
01.00 function that allows you to initialize the objects properties. An inheritance constructor is a constructor
02.00 used to initialize both the base class and derived class objects that the derived class has inherited.
03.00

→ The inheritance constructor is responsible for
04.00 initializing both the inherited base class members and the derived class members. To achieve
05.00 this, the constructor invokes the constructor of the base class, ensuring that all members of the
06.00 derived class are properly initialized, including those inherited from the base class.
EVE

→ By invoking the base class's constructor and passing
Notes all the necessary parameters, the inheritance constructor initializes the members of the base class. This is accomplished using the 'base class (args)' initialization list in the constructor. Further instructions particular to the derived class are provided in the constructor body.

The getdata() and putdata() functions in employee accept a name and number from the user and display a name and number. Functions also called getdata() and putdata() in the manager and scientist classes use the functions in employee and also do their own work. In manager, the getdata() function asks the user for a title and the amount of golf club dues, and putdata() displays these values. In scientist, these functions handle the number of publications.

08.00

09.00

10.00

11.00

12.00

01.00

02.00

03.00

04.00

05.00

06.00

EVE

Notes

→ "Abstract" Base Class

We didn't define any objects of the base class employee. We use this as a general class whose sole purpose is to act as a base from which other classes are derived.

The laborer class operates identically to the employee class, since it contains no additional data or functions. It may seem that the laborer class is unnecessary but by making it a separate class we emphasize that all classes are descended from the same source, employee. Also, if in the future we decided to modify the laborer class, we would not need to change the declaration for employee.

Classes used only for deriving other classes, as employee is in EMPLOY, are sometimes loosely called abstract classes, meaning that no actual instances (objects) of this class are created. However, the term abstract has a more precise definition. that we'll.

→ Constructors and Member Functions

Sunday 05

There are no constructors in either the base or derived classes, so the compiler creates objects of the various classes automatically when it encounters definitions like

```
manager m1, m2;
```

using the default constructor for manager calling the default constructor for employee.

Enter number of pubs: 999
Enter data for laborer 1
08.00 Enter last name: Jones
Enter number: 6546544
09.00

Then program then plays it back.
10.00

Data on manager 1
11.00 Name: Wadsworth
Number: 10
12.00 Title: President
Golf club dues: 1000000
01.00

Data on manager 2
02.00 Name: Bradley
Number: 124
03.00 Title: Vice-President
Golf club dues: 500000
04.00

Data on scientist 1
05.00 Name: Hauptman. Franglish
Number: 234234
06.00 Number of publications: 999
Data on laborer 1
EVE Name: Jones
Number: 6546544

Notes A more sophisticated program would use an array or some other containes to arrange the data so that a large number of employee objects could be accomodated.

```
    cout << "\nData on laborer 1";
    l1.putdata();
    cout << endl;
    return 0;
}
```

→ The main() part of the program declares four objects of different classes: two managers, a scientist, and a laborer. (Of course many more employees of each type could be defined, but the output would become rather large.) It then calls the getdata() member functions to ~~declare~~ obtain information about each employee, and the putdata() function to display this information

→ Here's a sample interaction with EMPLOY. First the user supplies the data.

```
Enter data for manager 1
    Enter last name: Blainsworth
    Enter number: 10
    Enter title: President
    Enter golf club dues: 1000000
Enter data on manager 2
    Enter last name: Bradley
    Enter number: 124
    Enter title: Vice-President
    Enter golf club dues: 500000
Enter data for scientist 1
    Enter last name: Hauptman-Frenglish
    Enter number: 234234
```

```
};
int main()
{
    manager m1, m2;
    scientist s1;
    laborer l1;

    cout<< endl;                    //get data for several employees
    cout << "\n Enter data for manager 1";
    m1.getdata();

    cout << "\n Enter data for manager 2";
    m2.getdata();

    cout <<"\n Enter data for scientist 1";
    s1.getdata();

    cout <<"\n Enter data for laborer 1";
    l1.getdata();

    cout <<"\n Data on manager 1";
    m1.putdata();

    cout <<"\n Data on manager 2";
    m2.putdata();

    cout <<"\n Data on scientist 1";
    s1.putdata();
```

```
void putdata() const
 {
   employee::putdata();
   cout << "\n Title: " << title;
   cout << "\n Golf club dues: " << dues;
 }
};

class scientist : public employee  //scientist class
 {
   private:
     int pubs;
   public:
     void getdata()
       {
         employee::getdata();
         cout << "Enter number of pubs: "; cin>>pubs;
       }

     void putdata() const
       {
         employee::putdata();
         cout << "\n Number of publications: " <<pubs;
       }
     };

class labord : public employee //labores class
  {
```

```
class employee
{
private:
    char name[LEN];              // employee name
    unsigned long number;        // employee number
public:
    void getdata()
    {
        cout<<"\n Enter last name: "; cin>>name;
        cout<<" Enter number: "; cin>>number;
    }

    void putdata() const
    {
        cout<<"\n Name: "<<name;
        cout<<"\n Number: "<<number;
    }
};

class manager: public employee   // management class
{
private:
    char title[LEN];             // "vice president" etc.
    double dues;                 // golf club dues
public:
    void getdata()
    {
        employee::getdata();
        cout<<" Enter title: "; cin>>title;
        cout<<" Enter golf club dues: "; cin>>dues;
    }
```

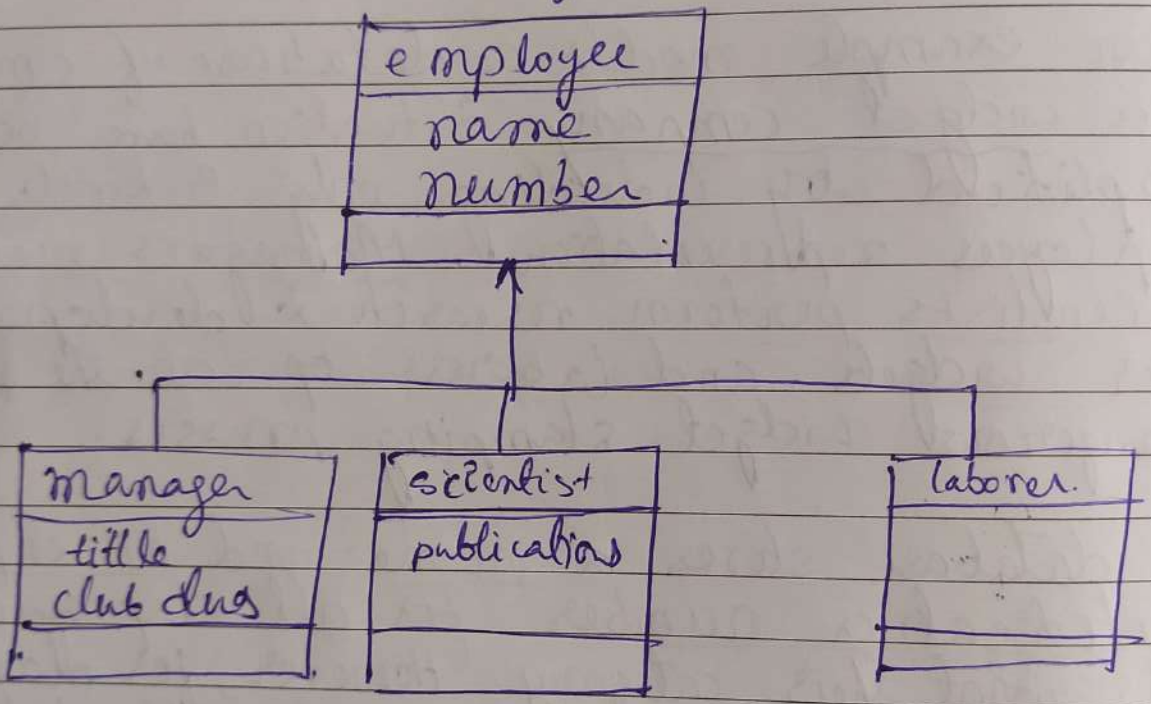
08.00 09.00 10.00 11.00 12.00 01.00 02.00 03.00 04.00 05.00 06.00 EVE

Sunday 26

MARCH MAY JUNE

Notes

Our example program starts with a base class employee. This class handles the employee's last name and employee number. From this class three other classes are derived: manager, scientist, and laborer. The manager and scientist classes contain additional information about these categories of employees and member functions to handle this information, as shown in below figure.



UML class diagram for EMPLOY

Here's the listing for EMPLOY:

```
// employ.cpp
// models employee database using inheritance
#include <iostream>
using namespace std;
const int LEN=80;          //maximum length of names
```

Lo

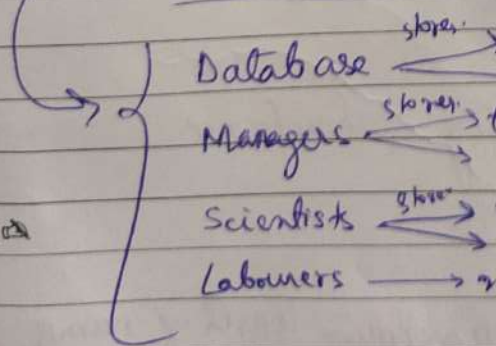
Class Hierarchies.

(R. Lafore - Ch-9, 388-Pg

In the examples, given previously inheritance has been used to add functionality to an existing class. Now, let's look at an example where inheritance is used for a diff. purpose: as a part of the original design of a program.

Our example models a database of employees of a widget company. Situation have been simplified by including only 3 kinds of employees representation. Managers manage, scientists perform research to develop better widgets, and laborers operate the dangerous widget-stamping presses.

The database stores a name and an employee identification number for all employees, no matter what their category. However, for managers, it also stores their titles and golf club dues. For scientists, it stores the number of scholarly articles they have published. Laborers need no additional data beyond their names and numbers.



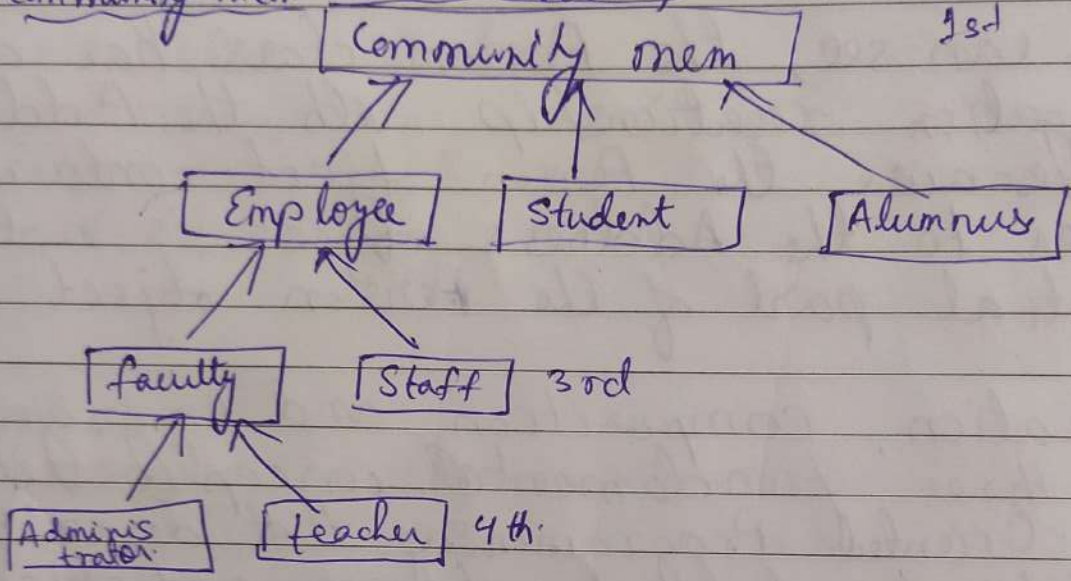
Notes

~~Constructor in Multiple inheritance in C++ program~~

08.00

Class hierarchy

09.00 E.g. Community mem class hierarchy.



10.00

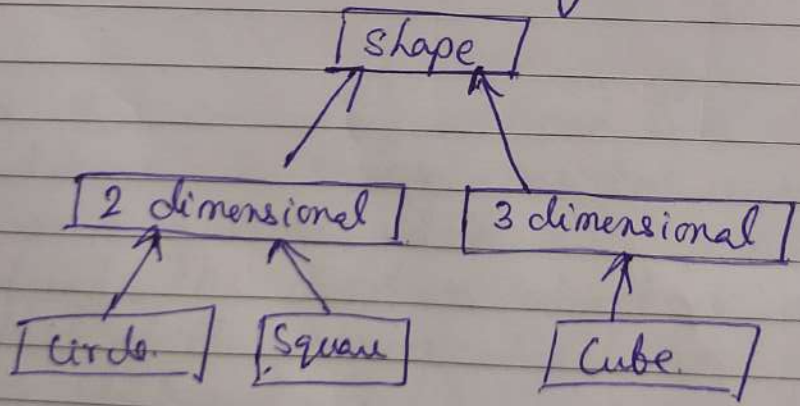
11.00

12.00

01.00

02.00

03.00 Eg:- Shape class hierarchy.



04.00

05.00

06.00

EVE

Notes

call, then call the setAddress() method of the Person class and pass it the address pointer. This sets the address of the Person object to the address object we created earlier.

As we can see, the Person class has an aggregation relationship with the Address class because the Person object contains a pointer to the Address object is not an essential part of the Person object.

Association, composition and Aggregation are three fundamental concepts in Object-Oriented Programming that describe the relationship between classes and objects.

Notes

```
public:
    Address(std::string street, std::string city, std::
            string state, std::string zip)
        :street(street), city(city), state(state), zip(zip) { }
private:
    std::string street;
    std::string city;
    std::string state;
    std::string zip;
};

int main() {
    Address* address = new Address("123 Main St.",
            "Anytown", "CA", "12345");
    Person person("John Doe");
    person.setAddress(address);
    return 0;
}
```

In this example, we have two classes: Person and Address. The Person class has a member variable named address, which is a pointer to an Address object. The Address object is not an essential part of the Person object, and if the Person object is destroyed, the Address object will not be destroyed.

In the main() function, we create a new Address object and store it in a pointer variable named address. We then create a new instance of the Person class and pass it the name "John Doe". We then

composed of an Engine object, and the startCar() method of the Car class uses the Engine object to start the car.

Aggregation:

Aggregation is a relationship between two classes in which one class, known as the aggregate class, contains a pointer or reference to an object of another class, known as the component class. The component class can exist independently of aggregate class, and it can be shared by multiple aggregate classes. In other words, the life time of the component class is not controlled by the aggregate class.

Example of Aggregation in C++:

```
class Person{
public:
    Person(std::string name) : name(name),
                               address(nullptr) { }
void setAddress(Address* address) {
        this->address = address;
    }
private:
    std::string name;
    Address* address;
};

class Address{
```

Sunday 19

```
Car() : engine(new Engine()) { }
void startCar() {
    engine->start();
}
private:
    Engine* engine;
};

int main() {
    Car car;
    car.startCar();
    return 0;
}
```

In this example, we have two classes: Engine and Car. The Car class is composed of an Engine object. The Engine object is an essential part of the Car object, and if the Car object is destroyed, the Engine object will also be destroyed.

In the Car class constructor, we create a new instance of the Engine class and store it in a pointer variable named engine. We then provide a method named startCar() in the Car class, which calls the start() method of the Engine object stored in the engine pointer.

In the main() function we create a new instance of the Car class and call the startCar() method to start the car. As we can see, the Car class is

and pass it the two Account objects and the amount of money to be transferred.

As we can see, the Bank class has a direct association with the Account class because it uses objects of the Account class as parameters in its method.

Composition:

Composition is a relationship between two classes in which one class, known as the composite class, contains an object of another class, known as the component class, as a member variable. The composite class owns the component class, and the component class cannot exist independently of the composite class. In other words, the lifetime of the component class is controlled by the composite class.

Example of composition in C++:

```
class Engine {
public:
    void start() {
        // Code to start the engine
    }
};

class Car {
public:
```

```
class Account {
public:
    Account(int id, double balance) : id(id),
                balance(balance) { }
    int getId() { return id; }
    double getBalance() { return balance; }
private:
    int id;
    double balance;
};

int main() {
    Account* account1 = new Account(123, 1000.00);
    Account* account2 = new Account(456, 500.00);
    Bank bank;
    bank.transferMoney(account1, account2, 250.00);
    return 0;
}
```

In this example, we have two classes: Bank and Account. Bank class has a transferMoney() method that takes two Account objects and a double value representing the amount of money to be transferred from one account to another. Bank class is associated with the Account class, meaning that it has a connection or a link to the Account class.

In the main() function, we create two Account objects with different IDs and balances, and then we create an instance of the Bank class. We then call the transferMoney() method of the Bank class

Association

08.00 It is a relationship between two objects. It's denoted by "has-a" relationship. In this relationship all
09.00 objects have their own lifecycle and there is no owner. Let's take an example of Teacher and Student.
10.00 Multiple students can associate with single teacher and single student can associate with multiple
11.00 teachers, but there is no ownership between the objects and both have their own lifecycle. Both
12.00 can create and delete independently.

01.00 ## Association in C++

Association in C++ is a relationship between
02.00 two classes where one class uses the functionalities provided by the other class. In other words, an
03.00 association represents the connection or link between two classes. In an association, one class
04.00 instance is connected to one or more instances of another class.

05.00

Example of Association in C++:

06.00

```
class Bank {
public:
    void transferMoney(Account* fromAccount,
        Account* toAccount, double amount) {
    // code to transfer money from one account to
       another
    }
};
```

EVE

Notes

```
public:
  int id;
  string name;

Employee (int id, string name, Address* address)
{
  this->id = id;
  this->name = name;
  this->address = address;
}

void display()
{
  cout << id << " " << name << " " << address->addressline
       << " " << address->city << " " <<
       address->state << endl;
}
};

int main (void){
  Address a1 = Address("C-146, Sec-15", "Noida", "UP");
  Employee e1 = Employee(101, "Nakul", &a1);
  e1.display();
  return 0;
}
```

Output: 101 Nakul C-146, Sec-15 Noida UP

Aggregation in C++

In C++, aggregation is a process in which one class defines another class as any entity reference. It is another way to reuse the class. It is a form of association that represents HAS-A relationship.

C++ Aggregation Example

Let's see an example of aggregation where Employee class has the reference of Address class as data member. In such way, it can reuse the members of Address class.

```
#include <iostream>
using namespace std;
class Address {
  public:
  string addressLine, city, state;
  Address(string addressLine, string city, string state)
  {
    this->addressline = addressline;
    this->city = city;
    this->state = state;
  }
};
class Employee
  {
  private:
  Address* address //Employee HAS-A Address
```

```
class Racing
{
public:
Racing()
{
cout << "This is for Racing\n";
}
};

class Ferrari: public Car, public Racing
{
public:
Ferrari()
{
cout << "Ferrari is a Racing Car\n";
}
};
int main(){
    Ferrari f;
    return 0;
}
```

Output: This is a Vehicle
This is a Car
This is for Racing
Ferrari is a Racing Car

In a real-world scenario, we will drive a Car. So Car is a class that comes under vehicle class. Thus, an instance of single inheritance.

If we talk about the Ferrari, that is a combination of the racing car and a normal car. So class Ferrari is derived from the class Car and Class Racing.

Hence, the above example is a single and multiple inheritance.

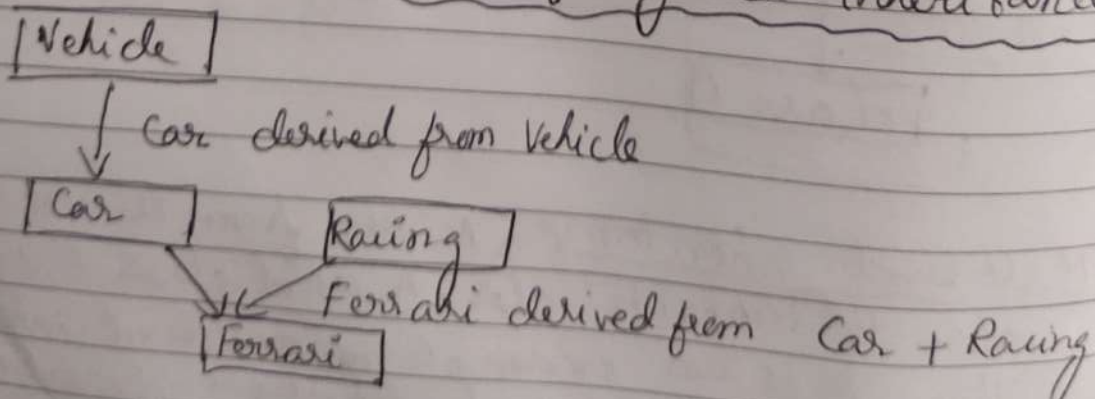
Code:

```
#include<iostream>
using namespace std;
class Vehicle
{
 public:
  Vehicle()
  {
   cout << "This is a vehicle\n";
  }
};
class Car:public Vehicle
 {
  public:
  Car()
  {
   cout<<"This is a Car\n";
  }
};
```

Syntax code

```
class A
{
  statement(s);
};

Class B: public A
{
 statement(s);
};

class C: public B
{
 statement(s);
};

class D: public B
{
 statement(s);
};
```

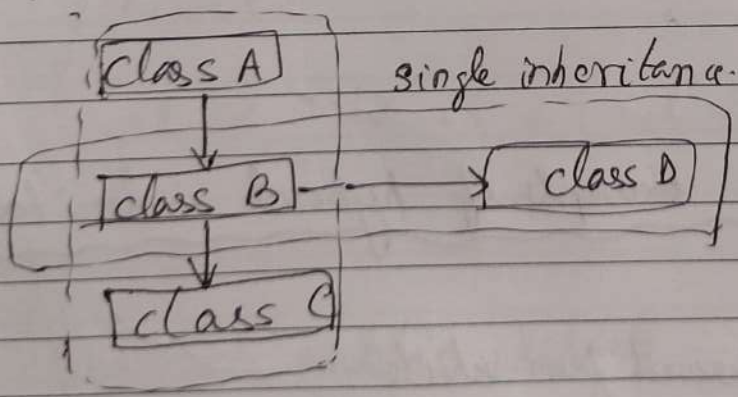
→ Now, a real life example of hybrid inheritance



```
class B : public A
{
 statement(s);
};

class C
{
 statement(s);
};

class D : public B, public C
{
 statement(s);
};
```

Multilevel inheritance, Single inheritance.

Multilevel inheritance.



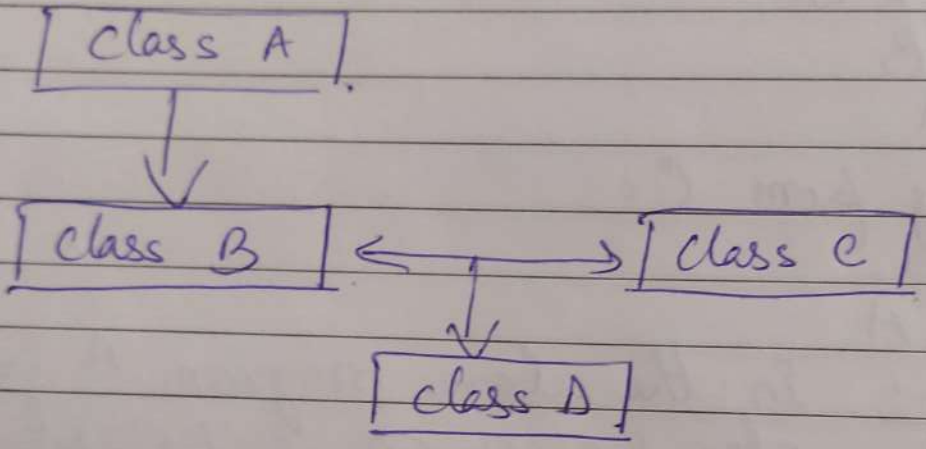
Multilevel inheritance → As seen from the above diagram, class B inherits class A and class C inherits class B. Thus, it is an example of multilevel inheritance.

Single inheritance → From the above diagram, class D inherits class B. Thus, it is an example of single inheritance.

Notes

Hybrid inheritance

In hybrid inheritance, there is a combination of one or more inheritance types. For instance, the combination of single and hierarchical inheritance. Therefore, hybrid inheritance is also known as multipath inheritance.



The diagram shows the hybrid inheritance that is a combination of single inheritance and multiple inheritance.

Single inheritance - Class B inherits class A. Thus an example of single inheritance.

Multiple inheritance - Class D is inherited from multiple classes (B and C shown above D). Thus, an example of multiple inheritance.

Syntax:

```
class A
{
statement(s)
};
```

```
class C
{
protected:
int c;
public:
void get_c()
 {
 cout<<" Enter the value of c is: ";
  cin>>c;
 }
};

class D : public B, public C        // multiple inheritance
{
  protected:
   int d;
  public:
  void mul()
  {
     get_a();
     get_b();
     get_c();
    cout<<" Multiplication of a,b,C is : "<<a*b*c;
  }
};
  int main()
  {
     D d;
     d.mul();
     return 0;
  }
```

Ex-2:

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

class A
{
  protected:
    int a;
  public:
  void get_a()
  {
    cout << "Enter the value of 'a' : ";
    cin >> a;
  }
};

class B : public A                    //single inheritance.
{
  protected:
  int b;
  public:
  void get_b()
  {
    cout << "Enter the value of 'b' : ";
    cin >> b;
  }
};
```

```
//base class
class Fare {
public:
  Fare() { cout<<"Fare of Vehicle\n"; }
};

// first sub class
class Car : public Vehicle {

};

//second sub class
class Bus : public Vehicle, public Fare {

};

//main Function
int main()
{
  // Creating object of sub class will
  //invoke the constructor of base class.
  Bus obj2;
  return 0;
}
```

O/P:

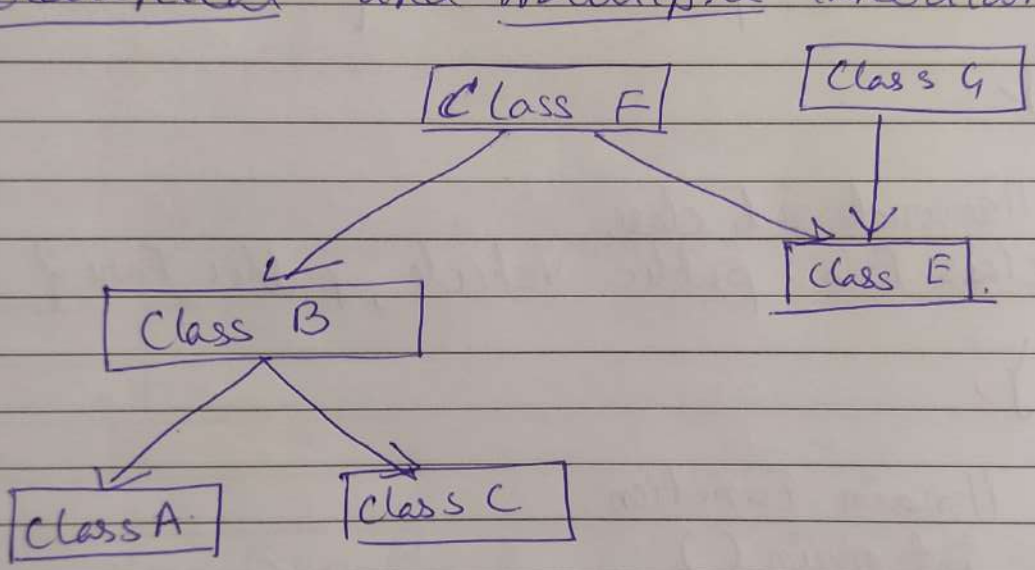
```
This is a Vehicle
Fare of Vehicle.
```

Sunday 29

5. ~~Hierarchical~~ Hybrid (Virtual) Inheritance:

Hybrid inheritance is implemented by combining more than one type of inheritance. For example: Combining Hierarchical inheritance and Multiple Inheritance.

Bllow image shows the combination of hierarchical and multiple inheritances:



Ex 1:- C++ program for Hybrid Inheritance.

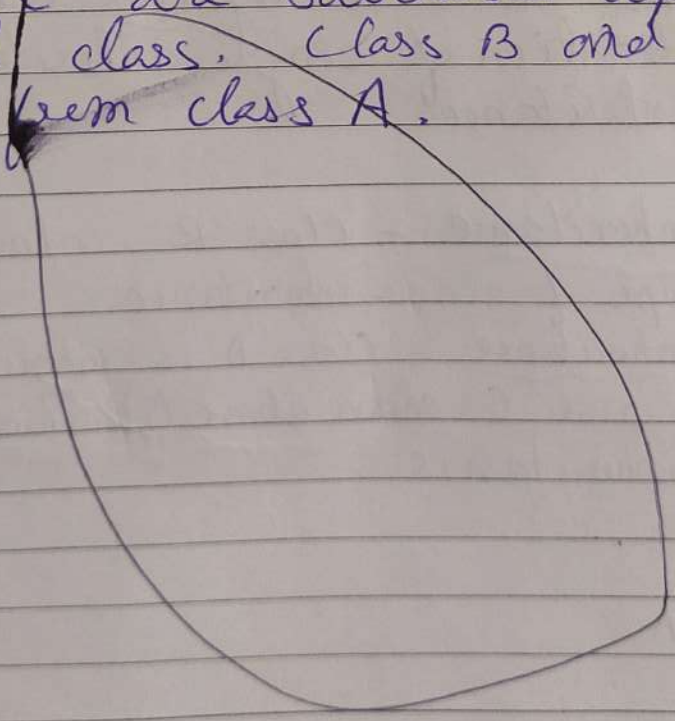
```
#include <iostream>
using namespace std;
//base class
class Vehicle{
  public:
    Vehicle() { cout << "This is a Vehicle\n"; }
};
```

```
cout<< "calling from C: "<<endl;
c.show_C();
c.show_A();
return 0;
}
```

Output:

```
calling from B:
class B
class A
calling from C:
class C
class A
```

Explanation: In the above program A is the superclass also known as a parent class and B and C are subclasses also known as the child class. Class B and class C inherit property from class A.



B Qt (5).

Ex-2: C++ program for hierarchical inheritance

```
class A
{
public:
void show_A(){
  cout<<"class A"<<endl;
}
};

class B: public A
{
public:
void show_B(){
  cout<<"class B"<<endl;
}
};
class C: public A
{
public:
void show_C(){
  cout<<"class C"<<endl;
}
};
int main(){
  B b;    //b is object of class B
cout<<"calling from B: "<<endl;
  b.show_B();
  b.show_A();

  C c;  //c is object of class C
```

Music is the medicine of a bredking heart

```
class D : public A.
{
    //body of class D.
};
```

08.00

09.00

10.00

Ex1:- C++ program to implement Hierarchical Inheritance.

11.00

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

//base class
class Vehicle {
public:
    Vehicle() { cout << "This is a Vehicle\n"; }
};

//first sub class
class Car : public Vehicle{
};

//second sub class
class Bus : public Vehicle{
};

//main function
int main()
{
    //creating object of sub class will
    //invoke the constructor of base class.
    Car obj1;
    Bus obj2;
    return 0;
}
```

12.00

01.00

02.00

03.00

04.00

05.00

06.00

EVE

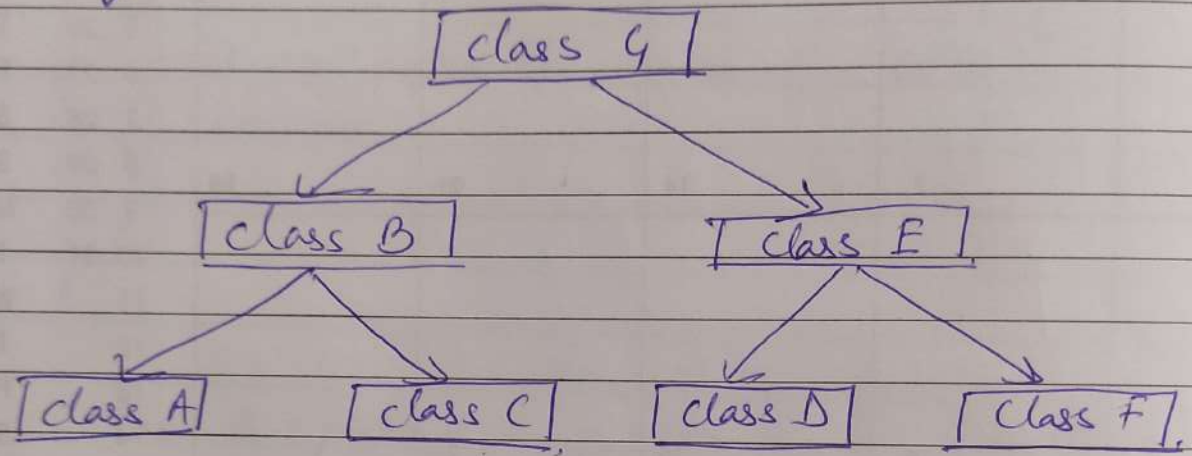
Notes

O/P: This is a Vehicle
This is a Vehicle

Nothing is so infectious as an example

4. Hierarchical Inheritance:

In this type of inheritance, more than one subclass is inherited from a single base class. i.e. more than one derived class is created from a single base class.



Syntax:

```
class A
{
   //body of the class A.
}
class B : public A.
{
   // body of class B.
}

class C : public A.
{
   // body of class C.
}
```

```
// first sub-class derived from class vehicle
class FourWheeler : public Vehicle {
public:
    FourWheeler()
    {
        cout << " objects with 4 wheels are vehicles \n";
    }
};

// sub class derived from the base class FourWheeler
class Car : public FourWheeler {

public:
    Car() { cout << " Car has 4 Wheels \n"; }
};

// main function
int main()
{
    // creating object of sub class will invoke
    // the constructor of base classes.
    Car obj;
    return 0;
}
```

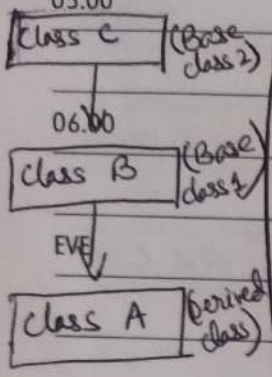
output:

- This is a Vehicle
- Objects with 4 wheels are vehicles
- Car has 4 wheels

}

(3) Multilevel Inheritance :

In this type of inheritance, a derived class is created from another derived class.



Example : C++ program to implement Multilevel Inheritance.

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

// base class
class Vehicle {

  public:
    Vehicle() { cout << "This is a Vehicle \n"; }
};
```

```
class TA : public Faculty, public Student {
  public:
    TA(int x) : Student(x), Faculty(x) {
    cout<<"TA :: TA(int) called"<< endl;
    }
};

int main() {

    TA ta1(30);
    return 0;
}
```

Output:

```
Person:: Person() called
Faculty:: Faculty(int) called
Student:: Student(int) called
TA:: TA(int) called
```

In the above program, constructor of 'Person' is called once. One important thing to note in the above output is, the default constructor of 'Person' is called. When we use 'virtual' keyword, the default constructor of grandparent class is called by default even if the parent classes explicitly call parameterized constructor.

In the above program, constructor of 'Person' is called two times. Destructor of 'Person' will also be called two times when object 'ta1' is destructed. So object 'ta1' has two copies of all members of 'Person' this causes ambiguities.

The solution to this problem is 'virtual' keyword. We make the classes 'Faculty' and 'Student' as virtual base classes to avoid two copies of 'Person' in 'TA' class.

For example, consider the following program.

```
#include<iostream>
using namespace std;
class Person {
public:
    Person(int x) { cout << "Person::Person(int) called" << endl; }
    Person() { cout << "Person::Person() called" << endl; }
};
class Faculty : virtual public Person {
public:
    Faculty(int x) : Person(x) {
        cout << "Faculty::Faculty(int) called" << endl;
    }
};
class Student : virtual public Person {
public:
    Student(int x) : Person(x) {
        cout << "Student::Student(int) called" << endl;
    }
};
```

```
int main() {
    TA ta1(30);
    return 0;
}
```

O/P:

```
Person::Person(int) called
Faculty::Faculty(int) called
Person::Person(int) called
Student::Student(int) called
TA::TA(int) called
```

Multiple + Multilevel is called Hybrid.

```
class Person {
    //Data members of person
public:
    Person(int x) { cout << "Person::Person(int)
        called" << endl; }
};

class Faculty : public Person {
    //data members of Faculty
public:
    Faculty(int x) : Person(x) {
        cout << "Faculty::Faculty(int) called" << endl;
    }
};

class Student : public Person {
    //data members of Student
public:
    Student(int x) : Person(x) {
        cout << "Student::Student(int) called" << endl;
    }
};

class TA : public Faculty, public Student {
public:
    TA(int x) : Student(x), Faculty(x) {
        cout << "TA::TA(int) called" << endl;
    }
};
```

Sunday 17

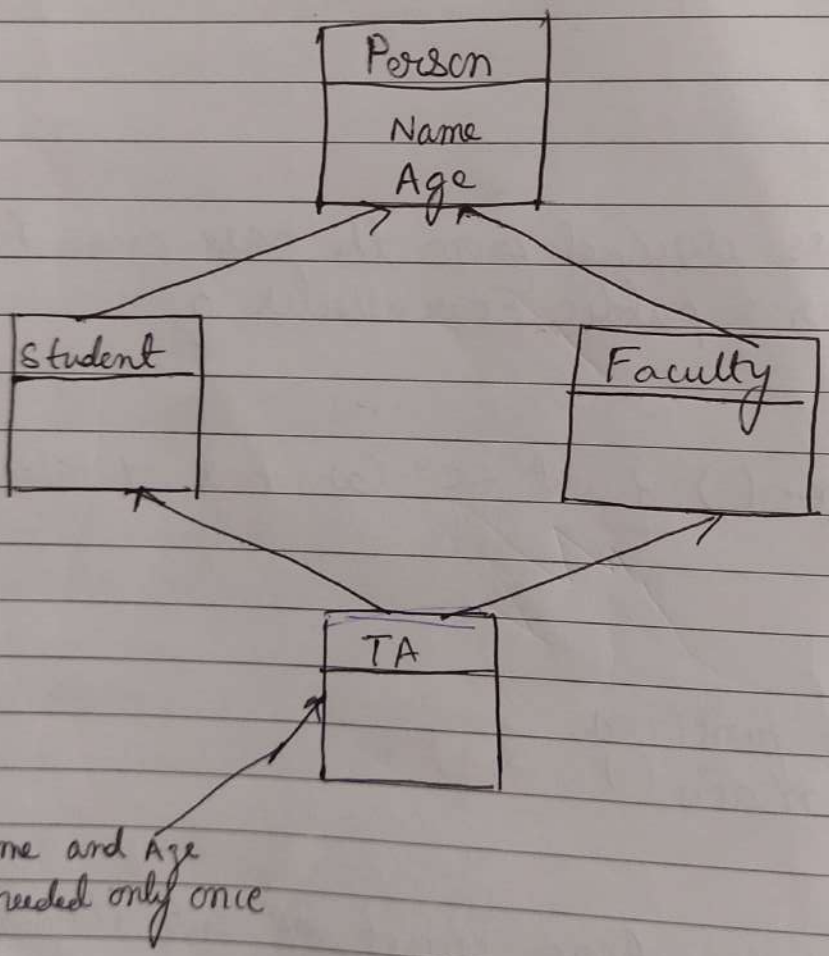
Notes

Keep your mind pure in the battlefield of life

Multiple Inheritance

The diamond problem

The diamond problem occurs when two superclasses of a class have a common base class. For example, in the following diagram, the TA class gets two copies of all attributes of Person class, this causes ambiguities



Notes

For example, consider the following program.

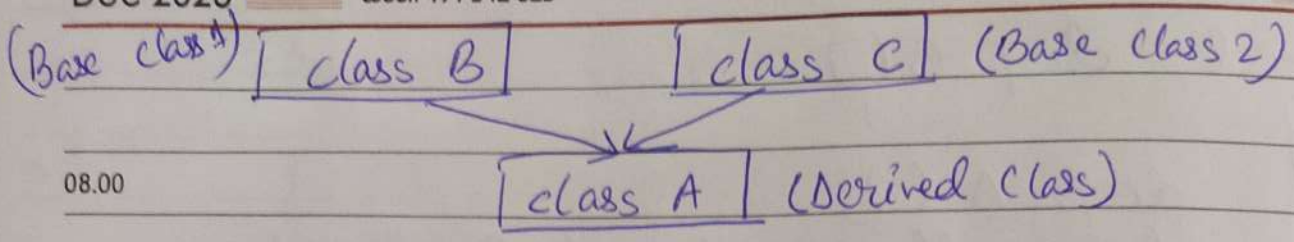
```
#include <iostream>
using namespace std;
```

Ex-1: C++ program to explain multiple inheritance.

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

// first base class
class Vehicle{
public:
    Vehicle() {cout<<"This is a Vehicle\n";}
};

//second base class
class FourWheeler{
public:
    FourWheeler()
    {
        cout<<"Thi is a 4wheeler Vehicle\n";
    }
};

//sub class derived from two base classes
class Car : public Vehicle, public FourWheeler{
};

// main function
int main()
{
    //Creating object of sub class will invoke the
    // constructor of base classes.
    Car obj;
    return 0;
}
```

O/P: This is a Vehicle.
This is a 4wheeler Vehicle.

Sunday 10



Syntax:

```
class subclass_name : access_mode base_class 1, access mode
                      base_class 2, .....
{
    //body of subclass
};
```

```
class B
{
    ....
};
class C
{
    ....
};
class A : public B, public C.
{
    ....
};
```

Here, the number of base classes will be separated by a comma (',') and the access mode for every base class must be specified.

```
void disp_B()
{
    disp_A();
    cout<<endl<<"Value of B = "<<b;
}

void cal_product()
{
    p= a *b;
    cout<<endl<<"Product of "<<a<<
                "*"<<b<<" = "<<p;
}
};

int main()
{
    B b;
    b.set_B(4,5);
    b.cal_product();

    return 0;
}
```

O/P:
Product of 4*5=20

(2.) Multiple Inheritance

It is a feature of C++ where a class can inherit from more than one class. i.e. one subclass is inherited from more than one base class.

Eg: (i) A CHILD class is derived from FATHER and MOTHER class.
(ii) A PETROL class is derived from LIQUID and FUEL class.

Ex: 3 (with arguments)

```
class A
{
protected:
int a;

public:
void set_A(int x)
{
    a = x;
}

void disp_A()
{
    cout<< endl<<"Value of A = "<<a;
}
};

class B: public A
{
    int b, p;

public:
void set_B(int x, int y)
{
    set_A(x);
    b=y;
}
```

```
public:
    void set_B()
    {
        set_A();
        cout<<"Enter the Value of B=";
        cin>>b;
    }
    void disp_B()
    {
        disp_A()
        cout<<endl<<"Value of B= "<<b;
    }
    void cal_product()
    {
        p= a*b;
        cout<<endl<<"Product of "<<a<<" * "<<b
                  <<" = "<<p;
    }
};
int main()
{
    B _b;
    _b.set_B();
    _b.cal_product();
    return 0;
}
```

O/P:
Enter the Value of A=3
Enter the Value of B=5
Product of 3*5=15

C++

Need to execute

// Example: 2

08.00

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

class A
{
  protected:
  int a;

  public:
    void set_A()
    {
      cout<<" Enter the Value of A = ";
      cin>>a;
    }

    void disp_A()
    {
      cout<<endl<<"Value of A = "<<a;
    }
};

class B:public A
{
  int b,p;
```

09.00

10.00

11.00

12.00

01.00

02.00

03.00

04.00

05.00

06.00

EVE

Notes

Ex: ① C++ program to explain single Inheritance

```
#include<iostream>
using namespace std;
//base class
class Vehicle{
    public:
      Vehicle()
      {
      cout << "This is a Vehicle \n";
      }
};
// sub class derived from a single base classes
class Car : public Vehicle{
};
//main function
int main()
{
    // creating object of sub class will invoke the
    the constructor of base classes
    Car obj;
    return 0;
}
```

Output: This is a Velicle

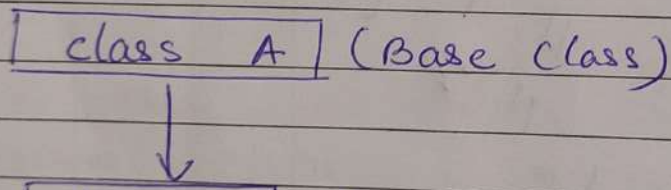
Types of Inheritance:-

(1) Single Inheritance
(2) Multilevel Inheritance
(3) Multiple Inheritance
(4) Hierarchical Inheritance
(5) Hybrid Inheritance

Type of Inheritance in C++

(1) Single Inheritance:

In single inheritance, a class is allowed to inherit from only one class. i.e one subclass is inherited by one base class only



Syntax:

```
class subclass_name: access_mode base_class
{
    //body of subclass
};
```

OR

```
class A
{
    ... ... ...
};
```

```
    //z is not accessible from B
};
class C : protected A {
    //x is protected
    //y is protected
    //z is not accessible from C
};
class D: private A //'private' is default for classes
{
    //x is private
    //y is private
    //z is not accessible from D
};
```

The below table summarizes the above three modes and shows the access specifier of the members of the base class in the subclass when derived in public, protected and private modes:

Base class member access specifier	Modes / ~~Type~~ of inheritance		
	Public	Protected	Private
Public	Public	Protected	Private
Protected	Protected	Protected	Private
Private	Not accessible (Hidden)	Not accessible (Hidden)	Not accessible (Hidden)

3. Private Mode: If we derive a subclass from a private base class, then both public members and protected members of the base class will became private in the derived class.

NOTE: The private members in the base class cannot be directly accessed in the derived class, while protected members can be directly accessed.

For ex → Classes B, C and D all contain the variables x, y, and z in the below example. It is just a question of access.

Example :- // C++ implementation to show that a derived class doesn't inherit access to private data members. However, it does inherit a full parent object.

```
class A {
    public:
        int x;
    protected:
        int y;
    private:
        int z;
};
class B: public A {
    // x is public
    // y is protected
```

```
// An object of class child has all data members
// and member functions of class parent
    obj1.id_c=7;
    obj1.id_p=91;
    cout<<"Child id is: "<<obj1.id_c<<'\n';
    cout<<"Parent id is: "<<obj1.id_p<<'\n';

    return 0;
}
```

Output:

```
Child id is: 7
Parent id is: 91
```

In the above program, the 'child' class is publicly inherited from the 'Parent' class so the public data members of the class 'Parent' will also be inherited by the class 'child'.

Modes of Inheritance: There are 3 modes of inheritance:

1. Public Mode: If we derive a subclass from a public base class, then the public member of the base class will become public in the derived class and protected members of the base class will become protected in the derived class.

2. Protected Mode: If we derive a subclass from a Protected base class, then both public member and protected members of the base class will become Protected in the derived class.

```
int main()
{
    Student S;
    s.set-s(1001, "Ram", "B.Tech", 2000);
    s.display-s();
    return 0;
}
```

Ex-4- C++ program to demonstrate implementation of inheritance.

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

//Base class
class Parent{
  public:
    int id-p;
};

// Sub class inheriting from Base Class (Parent)
class Child: public Parent {
  public:
    int id-c;
};

// main function
int main()
{ Child obj1;
```

```
    this -> id = id
    strcpy ( this -> name, n);
}

void Person :: display-p()
 {
   cout << endl << id << "\t" << name;
 }

class Student : private Person
 {
    char course[50];
    int fee;
    public:
    void set_s (int, char[], char[], int);
    void display_s();
 };

void Student :: set_s(int id, char n[],
                      char c[], int f)
 {
    set_p(id, n);
    strcpy(course, c);
    fee = f;
 }

void Student :: display_s()
 {
    display_p()
    cout << "\t" << course << "\t" << fee;
 }
```

Output:

```
Enter the Id : 101
Enter the Name : Dev
Enter the Course Name : CCS
Enter the Course Fee : 70000
Id : 101
Name : Dev
Course : CCS
Fee : 70000
```

// Ex : define member function with argument outside the class.

```
#include<iostream>
#include<string.h>
using namespace std;

class Person
{
    int id;
    char name[100];

    public:
        void set_p(int, char[]);
        void display_p();
};

void Person :: set_p(int id, char n[])
{
```

```
class Student: private Person
{
    char course[50];
    int fee;
public:
    void set_s();
    void display_s();
};
void Student::set_s()
{
    set_p();
    cout<<"Enter the Course Name: ";
    cin>>course;
    cout<<"Enter the Course Fee: ";
    cin>>fee;
}
void Student::display_s()
{
    display_p();
    cout<<"\nCourse: "<<course<<"\nFee: "
        <<fee<<endl;
}
int main()
{
    Student s;
    s.set_s();
    s.display_s();
    return 0;
}
```

Course : GCS
Fee : 70000,

// Example: define member function without argument outside the class.

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

class Person
{
    int id;
    char name[100];
public:
    void set_p();
    void display_p();
};

void Person::set_p()
{
    cout << "Enter the Id: ";
    cin >> id;
    cout << "Enter the Name: ";
    cin >> name;
}

void Person::display_p()
{
    cout << endl << "id: " << id << "\nName: " << name;
}
```

```
set_p();
cout<<"Enter the Course Name: ";
cin >> course;
cout <<"Enter the Course Fee : ";
cin >> fee;
}

Void display_s()
{
display_p();
cout<<"Course: "<< course<<"\nFee: "<< fee<<endl;
}
};

int main()
{
Student s;
s.set_s()
s.display_s()
return 0;
}
```

op: Enter the Id: 101
Enter the Name: Dev
Enter the Course Name: CCS
Enter the Course Fee: 70000

Id: 101
Name: Dev

Ex → Define member function without argument within the class.

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

class Person {
    int id;
    char name[100];

public:
    void set_p()
    {
        cout << " Enter the Id: ";
        cin >> id;
        cout << " Enter the Name: ";
        cin >> name;
    }

    void display_p()
    {
        cout << endl << " Id: " << id << "\nName: "
             << name << endl;
    }
};

class Student : private Person {
    char course[50];
    int fee;

public:
    void set_s()
    {
```

Example:

1. class ABC : private XYZ //private derivation
{ }

2. class ABC : public XYZ //public derivation
{ }

3. class ABC : protected XYZ
// protected derivation
{ }

4. class ABC : XYZ
derivation by default.
{ }

NOTE: • When a base class is privately inherited by the derived class, public members of the base class becomes the private members of the derived class and therefore, the public members of the base class can only accessed by the member functions of the derived class. They are inaccessible to the objects of the derived class.

• On the other hand, when the base class is publicly inherited by the derived class, public members of the base class also become the public members of the derived class.

Therefore, the public members of the base class are accessible by the objects of the derived class as well as by the member functions of the derived class.

→ Using inheritance, we've to write the functions only one time instead of 3 times as we're inherited the rest of the three classes from the base class (Vehicle)

Implementing inheritance in C++:

For creating a sub-class that is inherited from the base class we've to follow the syntax below.

Syntax

```
class <derived_class_name> : <access-specifier> <base_class_name>
{
    //body
};
```

where

class → keyword to create a new class.

derived_class_name → name of the new class, which will inherit the base class.

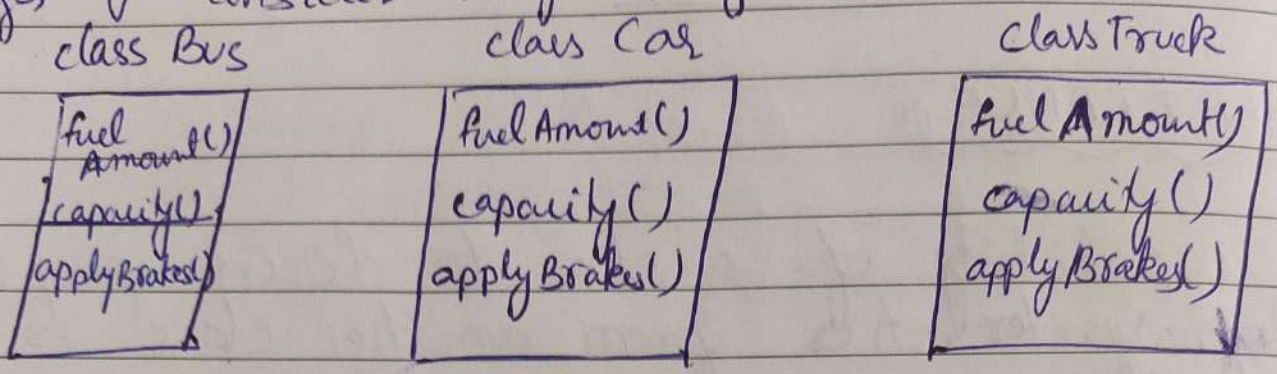
access-specifier → either of private, public or protected. If neither is specified PRIVATE is taken as default.

base_class_name → name of the base class.

Notes NOTE: A derived class doesn't inherit access to private data members. However, it does inherit a full parent object, which contains any private members which that class declares.

→ Why and when to use inheritance?

e.g → consider a group of vehicles



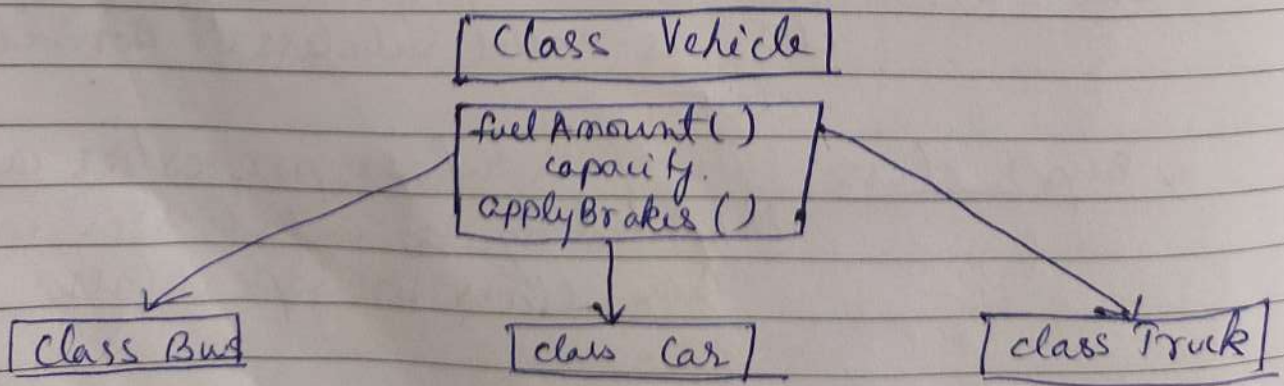
→ Methods are same for all three classes.

→ We can see clearly that the above process results in duplication of the same code 3 times.

→ This increases the chances of error and data redundancy. To avoid this type of situation, inheritance is used.

→ If we create a class Vehicle and write these three functions in it and inherit the rest of the classes from the vehicle class then we can simply avoid the duplication of data and increase re-usability.

→ Look at the below diagram in which 3 classes are inherited from vehicle class



UNIT-3

INHERITANCE

→ The capability of a class to derive properties and characteristics from another class is called Inheritance.

→ It is one of the most imp. features of OOPS.

→ It is a feature or a process in which new classes are created from the existing classes. ~~The new~~

→ The new class created is called "derived class" or "child class" and the existing class is known as the "base class" or "parent class".

→ The derived class inherits all the properties of the base class, without changing the properties of base class and may add new features to its own.

→ These new features in the derived class will not affect the base class. The derived class is the specialized class for the base class.

* Sub class: That inherits properties from another class is called Subclass or Derived class.

* super class: The class whose properties are inherited by a subclass is called Base class or Superclass.

Notes