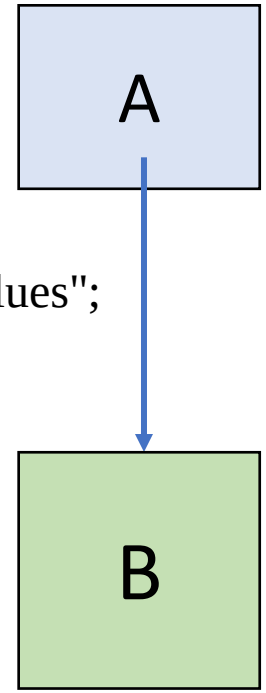
Inheritance

- The mechanism of **deriving a new class from an existing one is called inheritance**.
- The old class is referred to as the **base class** and the **new one** is called the **derived class** or **sub class.**
- The derived class inherits (acquires) some or all of the traits from the base class.
- **Reusability** is an important feature of Inheritance

class derived-class-name : access-specifier base-class-name
{
……
……
}

The **colon** indicates that the derived class name is derived from the base-class-name.

the access specifies is optional and if present, may be either **private or protected or public.**




```
class A
{
public:
int a,b;
void get() {
cout<<"Enter any two Integer values";
cin>>a>>b;
}
};
class B:public A
{  int c;
public:
void add() {
   c=a+b;
    cout<<a<<"+"<<b<<"="<<c;
}
};
int main() {
B b;
b.get();
b.add();
}
```

Modes of inheritance in C++

Public mode:

- public member(base class) will become public(child class) also,
- protected - > protected in the child class,
- private members are not accessible in the derived class.

```
class ABC:public XYZ
{
members of ABC;
};
```

Protected mode:

- public and protected members (base) → protected in the derived class
- private members (base) are again not accessible in the derived class.

```
class ABC:protected XYZ {
// protected derivation;
};
```

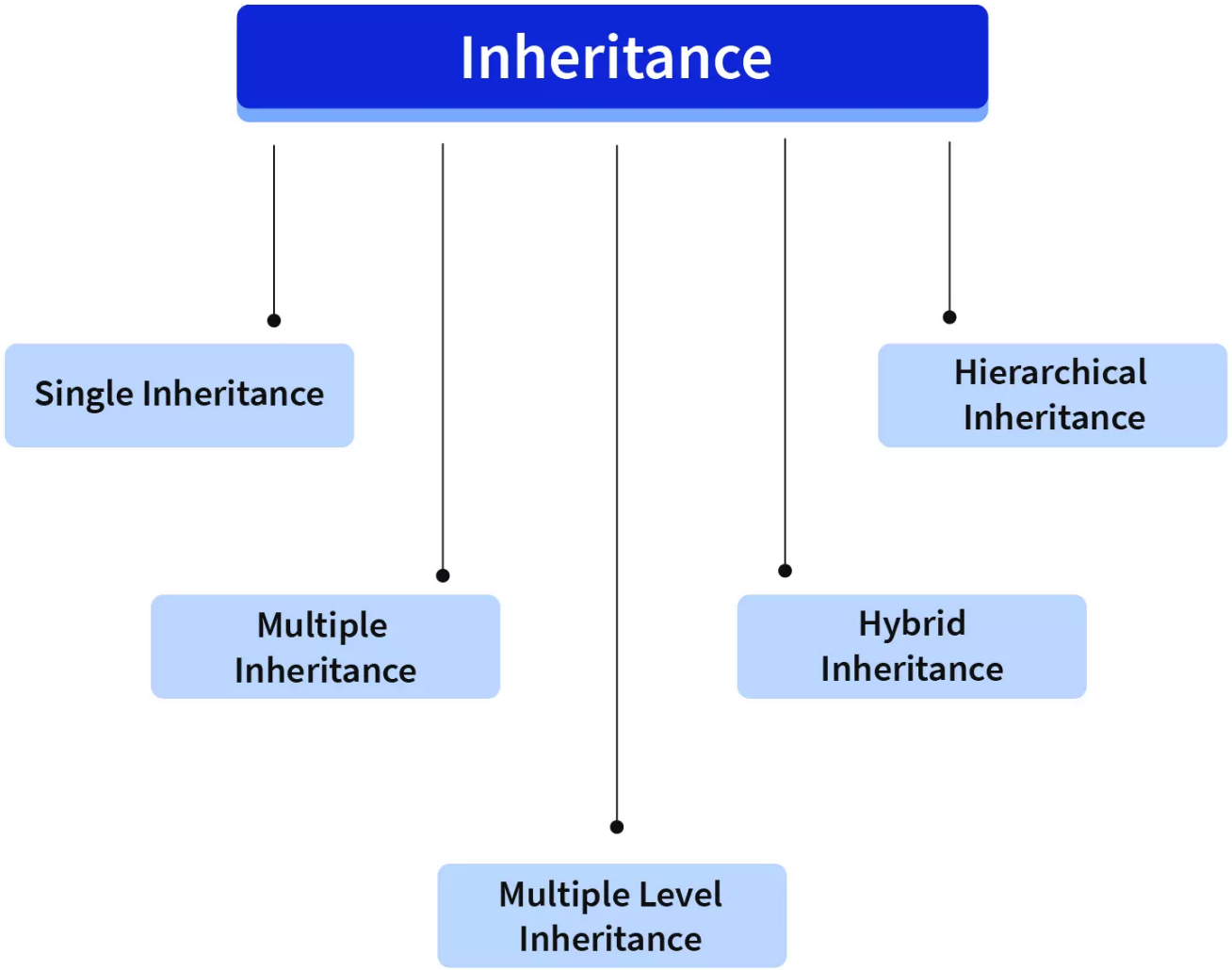
Private mode:

- public and protected members(base)→ private in the derived class,
- private members of the base class are again not accessible in the derived class

```
class ABC : private XYZ
{
members of ABC;
};
```

Base class member access specifier	Type of Inheritence		
	Public	Protected	Private
Public	Public	Protected	Private
Protected	Protected	Protected	Private
Private	Not accessible(Hidden)	Not accessible(Hidden)	Not accessible(Hidden)

Types of inheritance in C++



Single Inheritance

- When the derived class **inherits only one base class,** it is known as Single Inheritance.


```
class A
{

};
class B: visibility_mode A
{

}
```

```
class Base {
  public:
    float salary = 900;
};
class Derived: public Base {
  public:
    float bonus = 100;
    void sum() {
      cout << "Your Total Salary is: " << (salary + bonus) << endl;
    }
};
int main() {
Derived x; // Creating an object of the derived class.

  // Gets the salary variable of Base class.
  cout << "Your Salary is:" << x.salary << endl;
  // Gets the bonus variable of the Derived class.
  cout << "Your Bonus is:" << x.bonus << endl;
  x.sum();
  return 0;
}
```

University Student Database System

You're developing a university student database system in C++. The program includes a **Person class** for basic information, a **Student class** that inherits from Person and adds student-specific details

Implement inheritance in the Student class affects the accessibility of Person class members, design functions to add, display.

Input:

Enter the Id: 1001
Enter the Name: Alice Smith
Enter the Course Name: Computer Science
Enter the Course Fee: 5000

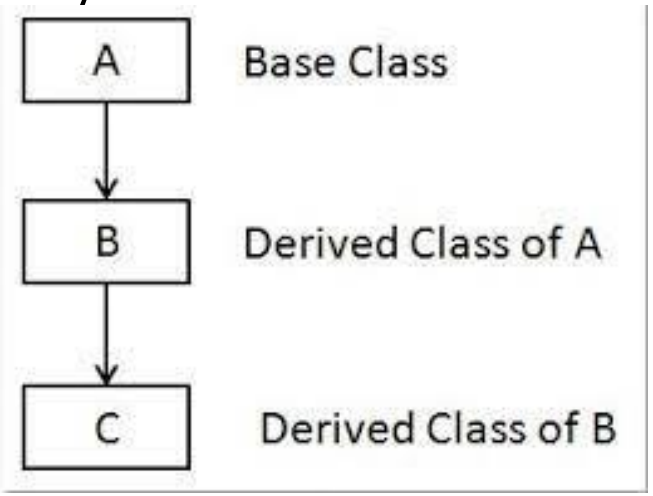
Output:

Id: 1001
Name: Alice Smith
Course: Computer Science
Fee: 5000

```
#include <iostream>
using namespace std;
class Person {
   int id;
   char name[100];
public:
   void setPerson() {
      cout << "Enter the Id:";
      cin >> id;
      cout << "Enter the Name:";
      cin >> name;
   }

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

class Student : public Person {
char course[50];
   int fee;
public:
   void setStudent() {
      setPerson();
      cout << "Enter the Course Name:";
      cin >> course;
      cout << "Enter the Course Fee:";
      cin >> fee;
   }

   void displayStudent() {
      displayPerson();
      cout << "Course: " << course << "\nFee: " << fee << endl;
   }
};

int main() {
   Student s;
   s.setStudent();
   s.displayStudent();
   return 0;
}
```

Multilevel Inheritance

- Deriving a class from another derived class.
- When one class inherits another class, it is further inherited by another class.




```
class A  //base class
{   …
};
class B: public A   //intermediate class derived and base
{   …
};
class C: public B //derived class
{
}
```

```
class Airline
{           public:
  void Print1() {
cout<<"Airway is fastest means of transport.\n"; }
};
class Indigo: public Airline
{ public:   void Print2() {
  cout<<"Indigo is the largest airline company in India.\n"; }
};
class Boeing747: public Indigo
{ public:
            void Print3(){
cout<<"Boeing747 of Indigo can have a max speed of
660mph.\n"; }
};
int main()
{           Boeing747 plane;
            plane.Print1();
            plane.Print2();
            plane.Print3();
            return 0;
}
```

Create a multilevel inheritance hierarchy for vehicles, consisting of a base class "Vehicle," a subclass "Car" inheriting from "Vehicle," and a further subclass "SportsCar" inheriting from "Car." The "Vehicle" class should have properties for "speed" (in mph) and "color," with methods for starting, stopping, accelerating, and braking. The "Car" class inherits these properties and methods and adds a "model" property, along with methods for honking the horn and changing gears. The "SportsCar" class, which inherits from "Car," introduces a "turbo boost" method for significant speed enhancement, primarily designed for a car racing game simulation.

INPUT
Enter car color and model
red
Audi

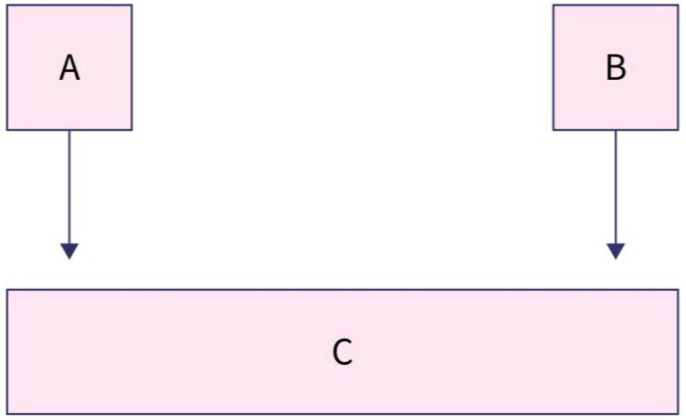
OUTPUT:
The red Car is starting.
The red Car is accelerating to 60 kmph.
The red audi car is changing to gear 3.
The red audi sports car is turbo boosting!
The red Car is breaking to 40 kmph.
The red Car is stopping.
The red audi car is honking.

```
class Vehicle {
protected:    string type;  string color; int speed;
public:
   Vehicle(string type, string color) :type(type), color(color), speed(0) {}
   void Start() {  cout << "The " << color << " " << type << " is starting.“;  }
   void Stop() {   cout << "The " << color << " " << type << " is stopping.";  }
void Accelerate(int vspeed) {
  speed += vspeed;
 cout <<"The"<< color << " " << type << " is accelerating to " << speed << " kmph“;
   }
void Brake(int vspeed) {
      speed -= vspeed;
  cout << "The " << color << " " << type << " is breaking to " << speed << " kmph“;
   }
};


class Car:public Vehicle{
public: string model;
   Car(string color,string model):Vehicle("Car",color),model(model){}

 void Honk() {
       cout << "The " << color << " " << model << " car is honking." << endl;
   }
 void ChangeGear(int gear)
 cout <<"The "<<color << " " << model << "car is changing to gear " << gear;
   }
};
```

```
class SportsCar : public Car {
public:
   SportsCar(string color,string model) : Car(color, model) {}

   void TurboBoost() {
      cout << "The "<< color <<" "<< model <<"sports car is turbo
boosting!" << endl;
   }
};
int main() {

                string car_color, car_model;
                cout<<"Enter car color and model \n";
                cin>>car_color>>car_model;
   SportsCar scar(car_color, car_model);
   scar.Start();
   scar.Accelerate(60);
   scar.ChangeGear(3);
   scar.TurboBoost();
   scar.Brake(20);
   scar.Stop();
   scar.Honk();

   return 0;
}
```

Multiple Inheritance

- When the derived class **inherits from more than one base class,** it is known as Multiple Inheritance.




```
class A
{   …
};
class B
{   …
};
class C: visibility_mode A, visibility_mode B;
{

}
```

```
class A {
  public:     int a;
void getA() { cout<<"Enter an Integer value";cin>>a; }
};
class B {
  public:   int b;
    void getB() { cout<<"Enter an Integer value";cin>>b; }
};
class C: public A, public B {
  public: int c;
    void sum() {  c=a+b
      cout << "Total is: " << c << endl;
    }
};
int main()
{
  C x; x.getA(); x.getB();
  x.sum();
  return 0;
}
```

```
class Person {   string name;   int age;
public:
   void readPerson()
    {
      cout << "Enter Name: "; cin >> name;
      cout << "Enter Age: ";   cin >> age;
   }
};

class Employee {
public:  string emID;   double salary;
   void readEmp()
    {
      cout << "Enter Employee ID: ";  cin >> employeeID;
      cout << "Enter Salary: Rs";   cin >> salary;
   }
};
class Manager : public Person, public Employee {
public:
   void readManager() {
      readPerson();
      readEmp();
      cout << "Enter Department: ";       cin >> department;
   }
   void dispManager()
   {
      cout << "Manager Details:" << endl;
      cout << "Name: " << name << ", Age: " << age << endl;
      cout << "Emp ID: " << empID << ", Salary: Rs." << salary ;
      cout << "Department: " << department << endl;
    }

private:
   string department;
};

int main() {
   Manager m;

   m.readManager();
   m.dispManager();

   return 0;
}
```

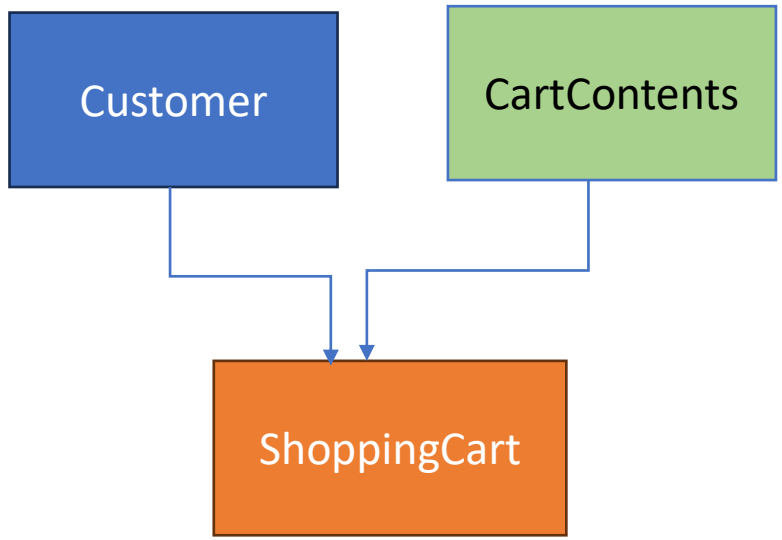
Streamlined E-Commerce: Utilizing Multiple Inheritance for Shopping Carts

You're developing an e-commerce app in C++ with a **ShoppingCart** class inheriting from **Customer** and **CartContents**.
The class represents a shopper's cart, **combining customer info and item selection**.
Implement this with Multiple inheritance streamlines development, simplifying cart and customer data management, making the shopping experience more intuitive and organized.

INPUT:

Enter Customer Name: Virat kohli
Enter Customer Email: viratk@bcci.com

Customer Information:
Customer Name: Virat kohli
Customer Email: viratk@bcci.com
Cart Contents:
- Item 1
- Item 2

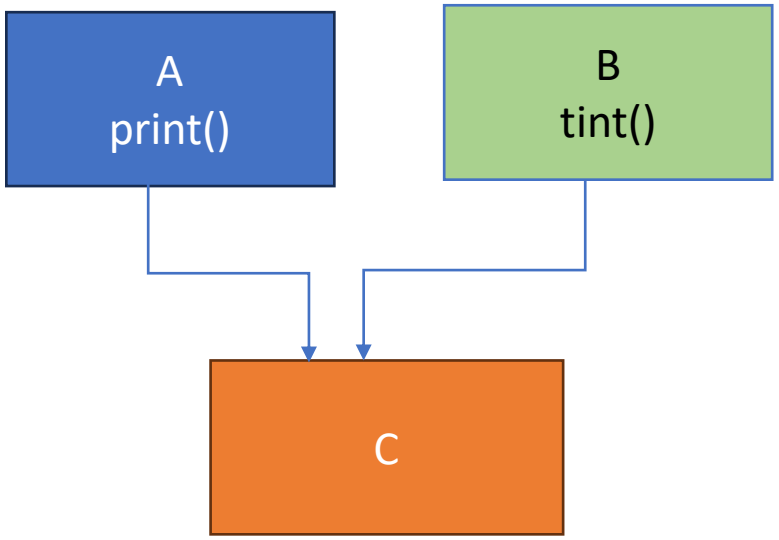


```
class Customer {
public:
   string name;    string email;
   void readCust() {
      cout << "Enter Customer Name: ";       cin >> name;
      cout << "Enter Customer Email: ";      cin >> email;
   }
   void dispCust() {
      cout << "Customer Name: " << name << endl;
      cout << "Customer Email: " << email << endl;
   }
};

class CartContents {
public:
   string items[100]; // Assuming a maximum of 100 items
   int itemCount = 0;
 void addItem(string& item) {
      if(itemCount < 100) {
         items[itemCount] = item;
         itemCount++;
      }
   }
};
```

```
class ShoppingCart : public Customer, public CartContents {
public:
   void dispCart() {
      cout << "Cart Contents added:" << endl;
       for (int i = 0; i < itemCount; i++)  {
       cout << "- " << items[i] << endl; }
      }
   void checkout(){ cout<<“Items checked out”;}
};
int main()
{
   ShoppingCart cart;
   cart.readCust();
   cart.addItem("Item 1");
   cart.addItem("Item 2");
   cart.dispCust();
   cart.dispCart();
   cart.checkOut();

return 0;
}
```

Ambiguity in Multiple Inheritance

- When the two base classes of a derived class contains same function name.



C obj;
obj.print(); //calls A's print() or B's
Ambiguity occurs here

Use :: operator to avoid ambiguity
obj.A::print()
obj.B::print()

Inside derived class to use print of use A::print()

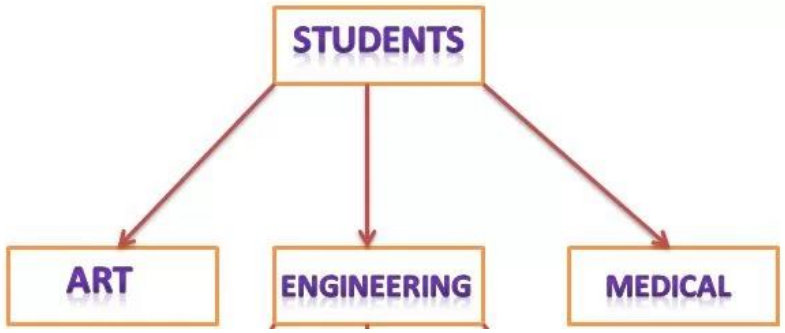
```
class A {
  public:
void print() { cout << " I am in class A" << endl;}
};
class B {
  public:  ;
    void print()  { cout<<" I am in class B ";cin>>b; }
};
class C: public A, public B {
  public:
  void show()  { cout<<" I am in c";}
};
int main()
{
  C x;
  x.print(); //which print() is called?
    return 0;
}
```

Use :: operator to call specific class function

x.A::print(); // Call A's printn method for the object 'x'
x.B::print(); // Call A's printn method for the object 'x'

Hierarchical Inheritance

- When one or more classes derived **from more one base class,** it is known as Hi Inheritance.




```
class A
{   …
};
class B: access specifier A
{   …
};
class C: access specifier A,
{

}
```

```
class A //superclass A
{
public:  void show_A() {  cout<<"class A"<<endl; }
};
class B : public A  //subclass B
{  public:  void show_B() {  cout<<"class B"<<endl; }
};
class C : public A //subclass C
{
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
cout<<"calling from C: "<<endl;
c.show_C();
c.show_A();
return 0;
}
```

```
// hierarchical inheritance
#include <iostream>
using namespace std;
// base class
class Animal {
  public:
   void info() {
      cout << "I am an animal." << endl;
   }
};
// derived class 1
class Dog : public Animal {
  public:
   void bark() {
      cout << "I am a Dog. Woof woof." << endl;
   }
};
// derived class 2
class Cat : public Animal {
  public:
   void meow() {
      cout << "I am a Cat. Meow." << endl;
   }
};
```

```
int main() {
   // Create object of Dog class
   Dog dog1;
   cout << "Dog Class:" << endl;
   dog1.info();  // Parent Class function
   dog1.bark();

   // Create object of Cat class
   Cat cat1;
   cout << "\nCat Class:" << endl;
   cat1.info();  // Parent Class function
   cat1.meow();

   return 0;
}
```

Educational Management System with Hierarchical Inheritance

In a simplified educational management system, you are tasked with implementing a **hierarchy of classes** using C++. The base **class Person**, holds common attributes like **name** and **age** along with methods for **setting and displaying** personal details.

Derived from Person, the **Student class** adds a **studentID** attribute, specific methods for **setting** student details,

Similarly, the **Teacher class**, also derived from **Person**, includes a subject attribute and tailored methods for teacher details,

Student Information:
Name: Shubhman
Age: 20 years
Student ID: 101

Teacher Information:
Name: Sachin
Age: 35 years
Subject: Mathematics

```
class Person {
protected:
   string name;     int age;
public:
   void setPerson(string n, int a) {
      name = n;   age = a;
   }
   void dispPerson() {
      cout << "Name: " << name << endl;
      cout << "Age: " << age << " years" << endl;
   }
};
class Student : public Person {
private:     int studentID;
public:
   void setStudent(string n, int a, int id) {
      setPerson(n, a);
     studentID = id;
   }
   void dispStudent() {
      dispPerson();
      cout << "Student ID: " << studentID << endl;
   }
};
```

```
class Teacher : public Person {
private:     string subject;
public:
   void setTeacher(string n, int a,string sub) {
      setPerson(n, a);
     subject = sub;
   }
   void dispTeacher() {
      displayInfo();
      cout << "Subject: " << subject << endl;
   }
};
int main() {
   Student s
   s.setStudent(“Shubhman", 20, 101);
   cout << "Student Information:" << endl;
   s.dispStud();
   Teacher t;
   t.setTeacher(“Virat", 35, "Mathematics");
   cout << "\nTeacher Information:" << endl;
   t.dispTeacher();
   return 0;
}
```

Creating a Hierarchical inheritance for Electronic Devices

You are developing a program to model different types of electronic devices.
You have identified three key classes: Device, Phone, and Tablet.
The Device class contains common attributes and methods for all devices, the Phone class represents mobile phones with specific function for making calls , and the Tablet class represents tablet devices and specific function for Playing games.
Implement Hierarchical inheritance to for these electronic devices

Output:

Device: iPhone
iPhone is turning on.
Making a phone call with iPhone.
iPhone is turning off.

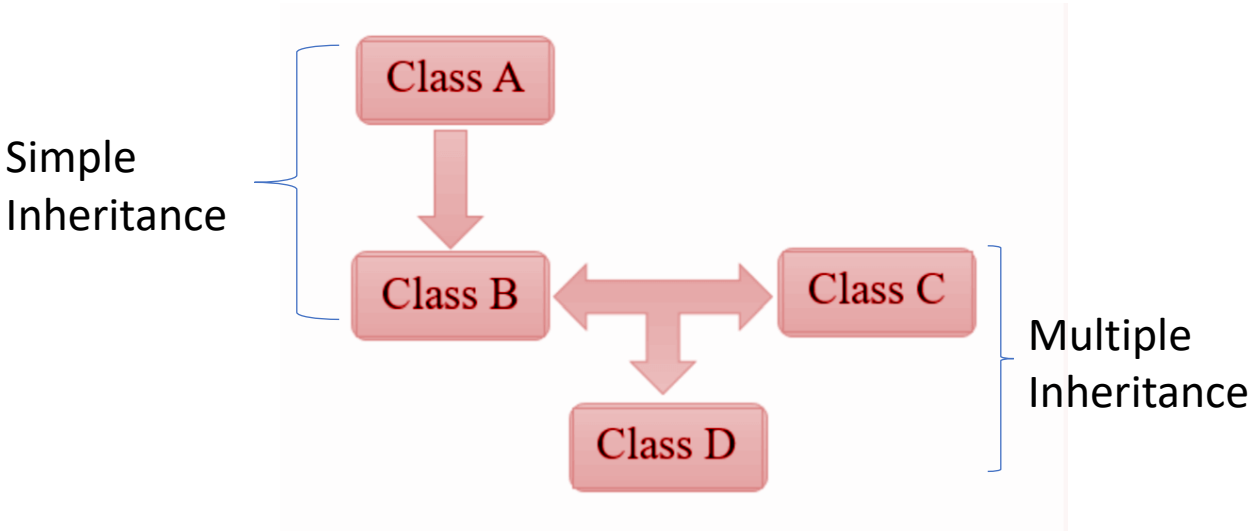
Device: iPad
iPad is turning on.
Playing a game on iPad.
iPad is turning off.

```
class Device {
string name;
public:
   void SetName(const string& name) {
      this->name = name;
   }
   void TurnOn() {
      cout << name << " is turning on." << endl;
   }
   void TurnOff() {
      cout << name << " is turning off." << endl;
   }
   void PrintInfo() {
      cout << "Device: " << name << endl;
   }
};

class Phone : public Device {
public:
   void MakeCall() {
      cout << "Making a phone call with " << name << "." <<
endl;
   }
};
```

```
class Tablet : public Device {
public:
      void playGame() {
       cout << "Playing a game on " << name << "." << endl;
   }
}

int main() {
   Phone phone;
   phone.SetName("My Smartphone");
   phone.TurnOn();
   phone.PrintInfo();
   phone.MakeCall();
   phone.TurnOff();

   Tab mytab;
   mytab.SetName("My Tablet");
   mytab.TurnOn();
   mytab.PrintInfo();
   mytab.playGame();
   mytab.TurnOff();

   return 0;
}
```

Hybrid Inheritance

- Combining various types of inheritance
- Ex. Hybrid = simple + hierarchical



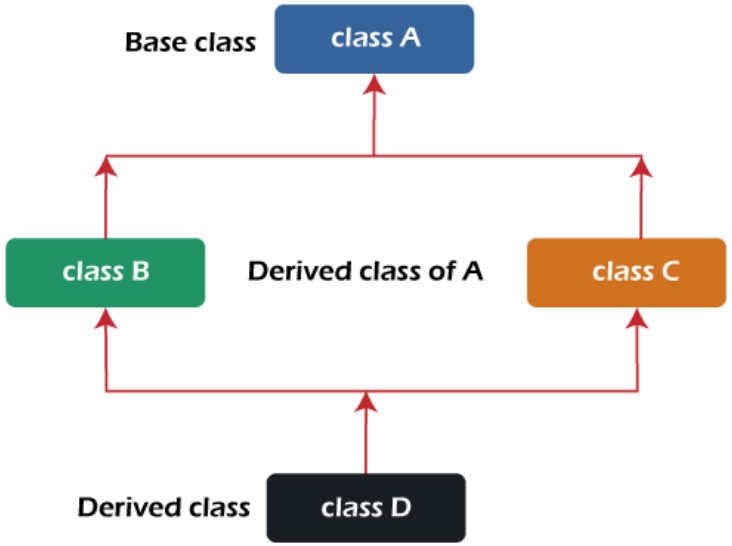

```
class A
{   …
};
class B: access specifier A {   …
};
class C:  {  …
}
class D: access specifier B, access specifier  C {
}
```

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

```
//HYBRID INHERITANCE= SINGLE + MULTIPLE
class A
{
  public: int a;
    void get_a()
    {
      cout << "Enter the value of 'a' : " << sendl;
      cin>>a;
    }
};
class B : public A
{   public:  int b;
    void get_b()
    {
      cout << "Enter the value of 'b' : " << endl;
      cin>>b;
    }
};
class C
{   public:  int c;
    void get_c()
    {
       cout << "Enter the value of c is : " << endl;
       cin>>c;
    } };
class D : public B, public C
{   public:  int d;
    void mul()
    {
        get_a();
        get_b();
        get_c();
        std::cout << "Multiplication of a,b,c is : " <<a*b*c<<
std::endl;
    }
};
int main()
{   D d;
    d.mul();
    return 0;
}
```

Hybrid Inheritance

- Combining various types of inheritance like multiple, simple, and hierarchical inheritance is known as hybrid inheritance.

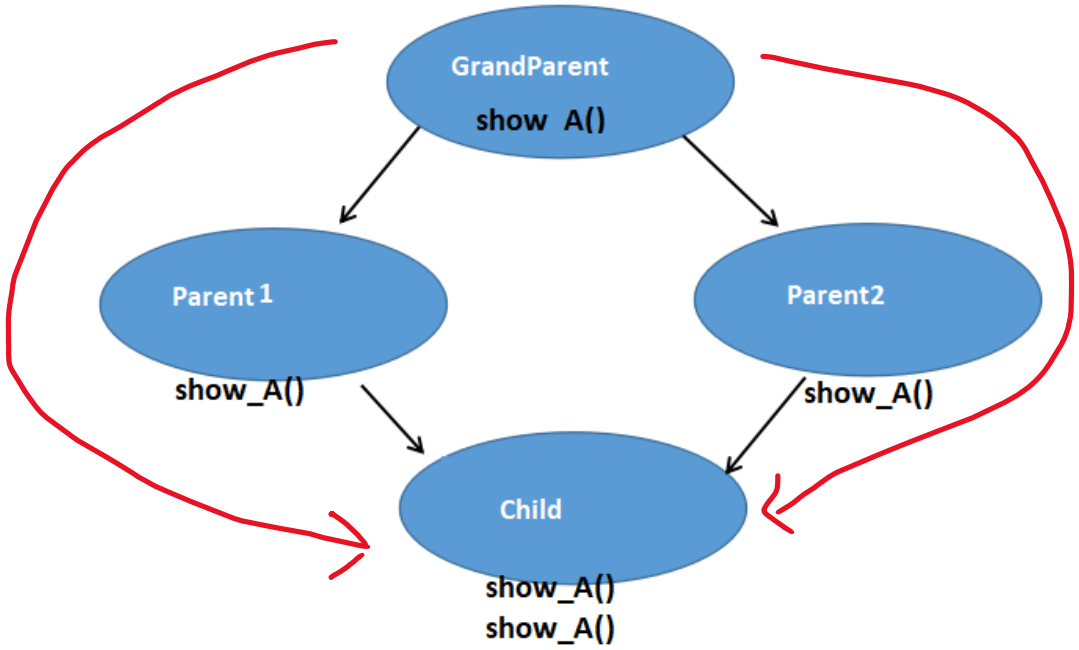



```
class A
{   …
};
class B: access specifier A {   …
};
class C: access specifier A {  …
}
class D: access specifier B, access specifier  C {
}
```

```
class A //superclass A
{
public:  void show_A() {  cout<<"class A"<<endl; }
};
class B : public A  //subclass B
{  public:  void show_B() {  cout<<"class B"<<endl; }
};
class C : public A //subclass C
{
public:  void show_C() {  cout<<"class C"<<endl;  }
};
Class D: public B, public c
{
}
int main() {
B b; // b is object of class B
cut<<"calling from B: "<<endl;
b.show_B();
b.show_A();
C c; // c is object of class C
cout<<"calling from C: "<<endl;
c.show_C();
c.show_A();
return 0;
}
```

Diamond Problem/ Ambiguity problem in Hybrid Inheritance

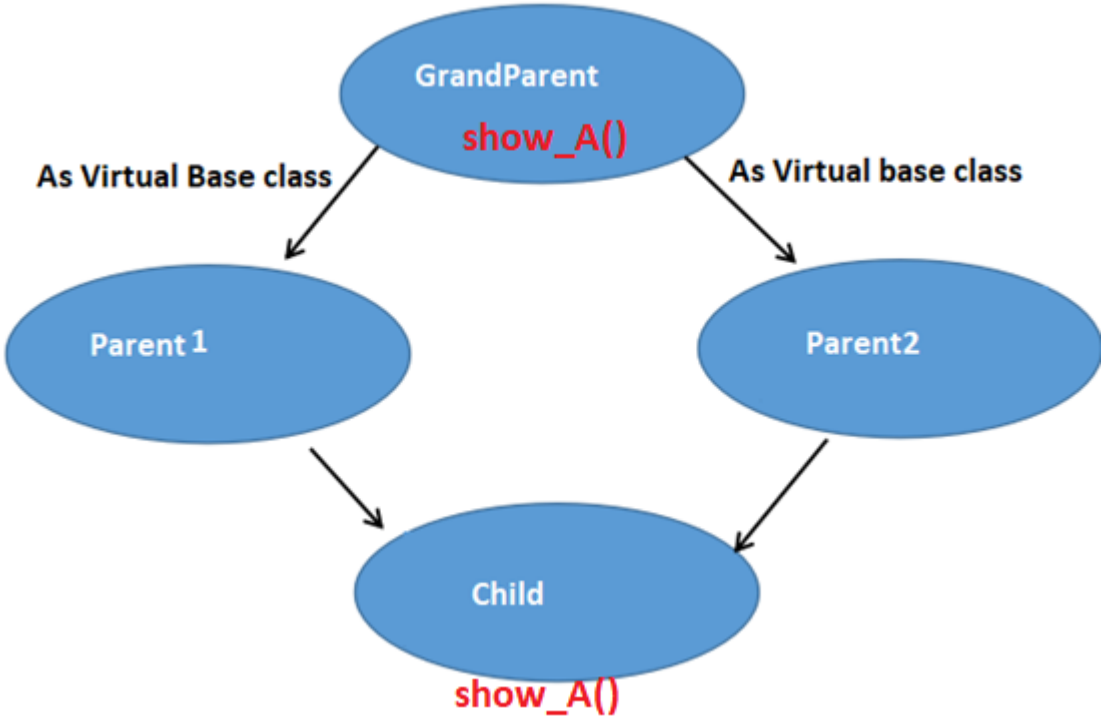
- All the public and protected members of 'grandparent' are inherited into 'child' twice, first via 'parent1' and second via 'parent2'. This means child would have duplicate sets of members inherited from 'grandparents'.
- This introduces ambiguity and should be avoided




```
class A {
public:     void show() {
                        cout << "Hello form A \n";
            }
};
class B : public A {
};
class C : public A {
};
class D : public B, public C {
};
int main()
{
            D d;
            d.show(); //  show from A or B?
}
```

Child can access grand parent in two ways: Though Parent1 and Parent2
Whim makes ambiguity

Virtual base class




```
class A {
public:     void show() {
                        cout << "Hello form A \n";
            }
};
class B : virtual public A {
};
class C : virtual public A {
};
class D : public B, public C {
};
int main()
{
            D d;
            d.show(); //  show from A or B?
}
```

To avoid ambiguity in Hybrid inheritance we can use of Virtual Base class, so that only one copy of the grand parent is available to child

Need for Virtual Base Class

- To **prevent the error** and let the compiler work efficiently, we've to use a virtual base class **when HYBRID inheritances occur**. It saves space and **avoids ambiguity/ diamond problem**
- When a class is specified as a virtual base class, it **prevents duplication of its data members**.
- Only one copy of its data members is shared by all the base classes that use the virtual base class.
- If a virtual base class is not used, **all the derived classes will get duplicated data members**. In this case, the compiler cannot decide which one to execute.

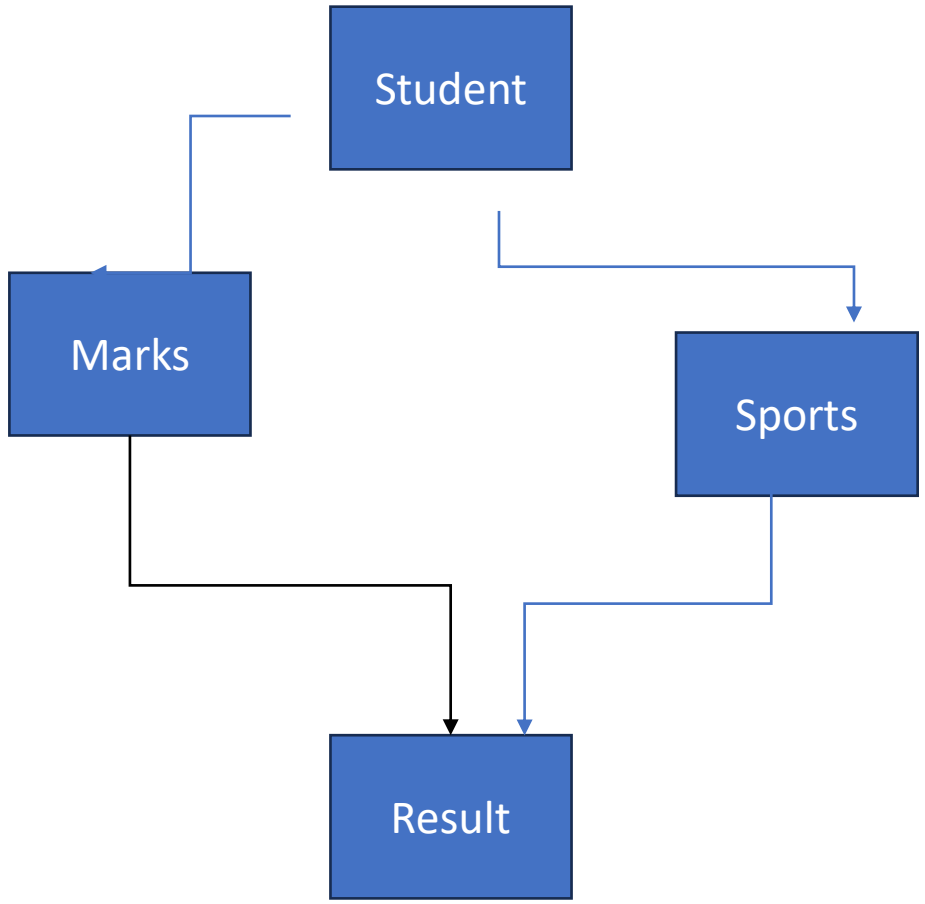
```
//AMBIQUITY
class A
{
  public:    A()
  {
        cout << "Constructor A\n";
  }
  void display()
  {
    cout << "Hello form Class A \n";
    }
};
class B: public A {};
class C: public A {};
class D: public B, public C {};
int main()
{
  D obj;
  obj.display();//member 'display' is
ambiguous
}
```

Student Record Management with Hybrid Inheritance

You are developing a program to manage student records that include **academic marks** and **sports achievements.** Implement a C++ program using **hybrid inheritance** to represent and display a student's information, marks, sports details, and overall **result**. The program should allow you to input student data and display the **result**.

INPUT:
Student Name: Sachin
Roll Number: 102
Math Marks: 95
Science Marks: 88
Favorite Sport: Cricket
Sport Score: 80

OUTPUT:
Name: Sachin
Roll Number: 102
Math Marks: 95
Science Marks: 88
Sport: Cricket
Sport Score: 80
Total Marks: 183
Overall Result: 263



```
//creating hybrid inheritance (hierarchial + multiple)
#include<iostream>
using namespace std;

class Student
{  public: int rno; string name;
   void readStudent(){ cout<<"Enter rno and name";
  cin>>rno>>name;  }
};

class Marks: virtual public Student
{
public: int m,p;
 void readMarks()
{
cout<<"Enter marks in maths and phisics \n";cin>>m>>p; }
};

class Sports: virtual public Student
{
   public: int score;
  void readSport(){ cout<<"Enter score in sports  \n";cin>>score; }

};
```

```
class Result:public Marks,public Sports
{
public:
 int tot_score; float avg;
   void showResult(){  tot_score=m+p+score; cout<<"Total
score="<<tot_score;
     avg=(float)tot_score/3; if(avg<40) cout<<" Failed \n"; else
cout<<"passed \n";}
};

int main()
{
                Result r;
                r.readStudent();
                r.readMarks();
                 r.readSport();
               r.showResult();
               return 0;
}
```

"Hospital Patient Information System Using Hybrid Inheritance"

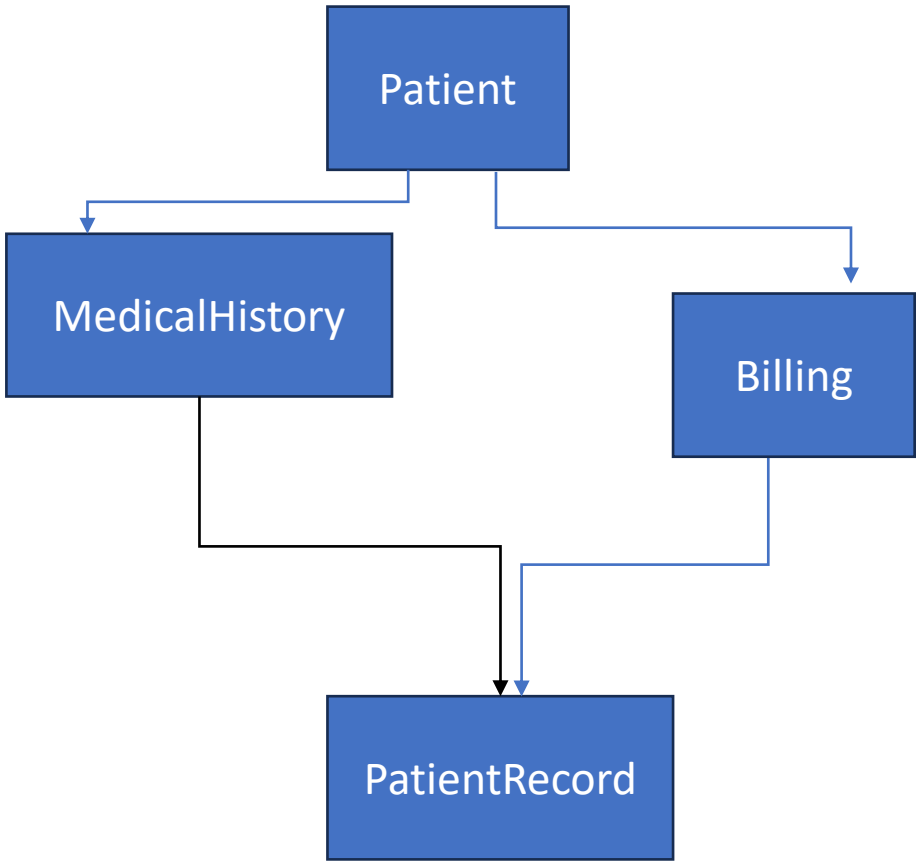
Build a C++ program for a hospital to manage patient records.
Employ hybrid inheritance to store **patient details, medical history**, and **billing information**.
Include the functionality to generate itemized bills for patients.

INPUT
Patient Name: John Smith
Patient ID: 101
Diagnosis: Fever
Treatment: Prescribed medication
Total Charges: Rs. 7500.00

OUTPUT:
Patient Record:
Patient Name: John Smith
Patient ID: 101
Diagnosis: Fever
Treatment: Prescribed medication
Total Charges: Rs. 7500.00

Itemized Bill:

Total Amount Due: Rs. 7500.00



```
class Patient {
protected:
   string name;    int patientID;
public:
   void setPatientInfo(const std::string& _name, int _patientID) {
      name = _name;       patientID = _patientID;
   }
   void displayPatientInfo() {
      cout << "Patient Name: " << name << "\n";
      cout << "Patient ID: " << patientID << "\n";
   }
};
class MedicalHistory : virtual public Patient {
protected:    string diagnosis;    string treatment;
public:
   void setMedicalHistory(string _diagnosis,string  _treatment) {
      diagnosis = _diagnosis;
      treatment = _treatment;
   }
   void displayMedicalHistory() {
      cout << "Diagnosis: " << diagnosis << "\n";
      cout << "Treatment: " << treatment << "\n";
   }
};
class BillingInformation : virtual  public Patient {
protected:    double totalCharges;
public:
   void setBillingInformation(double _totalCharges) {
      totalCharges = _totalCharges;
   }
   void displayBillingInformation() {
      cout << "Total Charges: $" << totalCharges << "\n";
   }
   double getBillingAmount() {
      return totalCharges;
   }
};
class PatientRecord : public MedicalHistory, public BillingInformation {
public:
   void displayPatientRecord() {
      displayPatientInfo();
      displayMedicalHistory();
      displayBillingInformation();
   }
   double generateItemizedBill() {
      return getBillingAmount();
   }
};
int main() {
   PatientRecord pt;
   pt.setPatientInfo("Rakesh Patel", 101);
   pt.setMedicalHistory("Fever", "Prescribed medication");
   pt.setBillingInformation(750.0);

   cout << "Patient Record:\n";
   pt.displayPatientRecord();
   cout << "\nItemized Bill:\n";
   cout << "----------------\n";
   double billAmount = pt.generateItemizedBill();
   cout << "Total Amount Due: Rs." << billAmount << "\n";
   return 0;
}
```

```
//virtual base class

class Animal {
public:
    void eat() {
        cout << "Animal is eating" << std::endl;
    }
};

class Mammal : public virtual Animal {
public:
    void breathe() {
        cout << "Mammal is breathing" << std::endl;
    }
};

class Bird : public virtual Animal {
public:
    void fly() {
        cout << "Bird is flying" << std::endl;
    }
};

class Bat : public Mammal, public Bird {
public:
    // Bat can access eat(), breathe(), and fly() without ambiguity
};

int main() {
    Bat bat;
    bat.eat();
    bat.breathe();
    bat.fly();

    return 0;
}
```

Need for Virtual Base Class

- To **prevent the error** and let the compiler work efficiently, we've to use a virtual base class **when HYBRID inheritances occur**. It saves space and **avoids ambiguity.**
- When a class is specified as a virtual base class, it **prevents duplication of its data members**.
- Only one copy of its data members is shared by all the base classes that use the virtual base class.
- If a virtual base class is not used, **all the derived classes will get duplicated data members**. In this case, the compiler cannot decide which one to execute.

```
//AMBIQUITY
class A
{
  public:    A()
  {
        cout << "Constructor A\n";
  }
  void display()
  {
    cout << "Hello form Class A \n";
    }
};
class B: public A {};
class C: public A {};
class D: public B, public C {};
int main()
{
  D obj;
  obj.display();//member 'display' is
ambiguous
}
```

Student Performance Analysis

In a program that models student information, you have the **Result** class inheriting from both the **Test** and **Sports** classes, which in turn inherit from the **Student** class.

The **Result** class calculates the total score by adding the math and physics marks to the PT score.

If you were tasked with extending this program to include additional subjects like chemistry and biology in the total score calculation, how would you modify the code to accommodate these subjects while maintaining the existing class hierarchy?

INPUT:

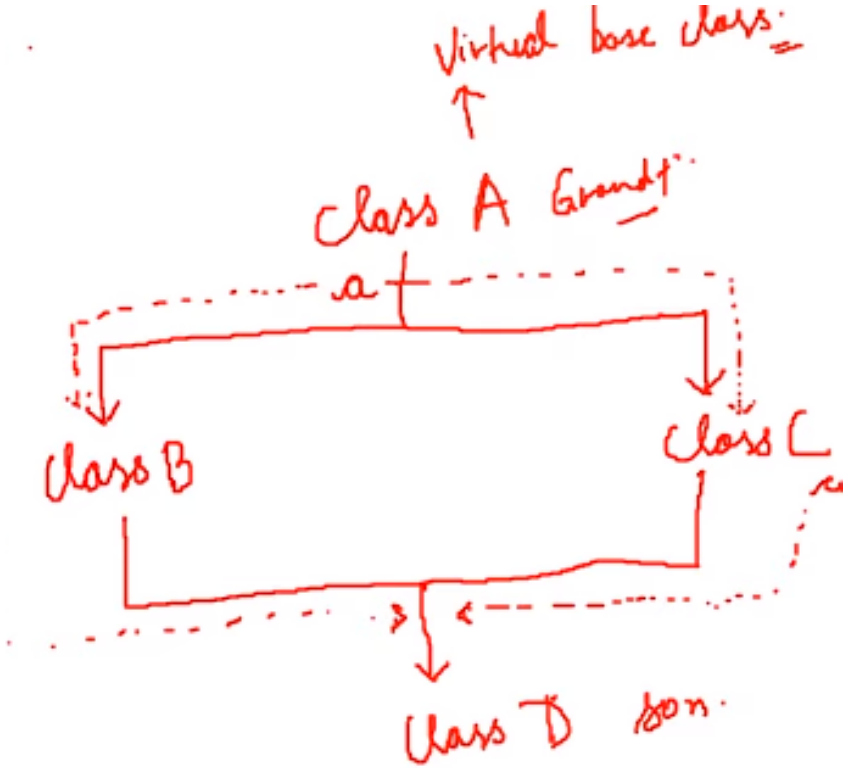
Enter student's roll number: 4200
Enter math marks: 78.9
Enter physics marks: 99.5

Enter PT score: 9

OUTPUT:

Student Information:
Roll Number: 4200
Math Marks: 78.9
Physics Marks: 99.5

PT Score: 9
Total Score: 369.3



```
class Student{
   protected:       int roll_no;
   public:
      void set_number(int a){
         roll_no = a;
      }
      void print_number(void){
         cout<<"Your roll no is "<< roll_no<<endl;
      }
};
class Test : virtual public Student{
   protected:       float maths, physics;
      public:
         void set_marks(float m1, float m2){
            maths = m1;
            physics = m2;
         }
         void print_marks(void){
            cout << "You result is here: "<<endl
                 << "Maths: "<< maths<<endl
                 << "Physics: "<< physics<<endl;
         }
};
```

```
class Sports: virtual public Student{
   protected:       float score;
   public:       void set_score(float sc){    score = sc;      }
      void print_score(void){
         cout<<"Your PT score is "<<score<<endl;
      }
};
class Result : public Test, public Sports{
   private:       float total;
   public:
      void display(void){
         total = maths + physics + score;
         print_number();
         print_marks();
         print_score();
         cout<< "Your total score is: "<<total<<endl;
      }};
int main(){
   Result res;
   res.set_number(4200); // ambiguity occurs if no virtual base
class
   res.set_marks(78.9, 99.5);
   res.set_score(9);
   res.display();     return 0;}
```

Resolving the Diamond Problem in a Graphics Framework

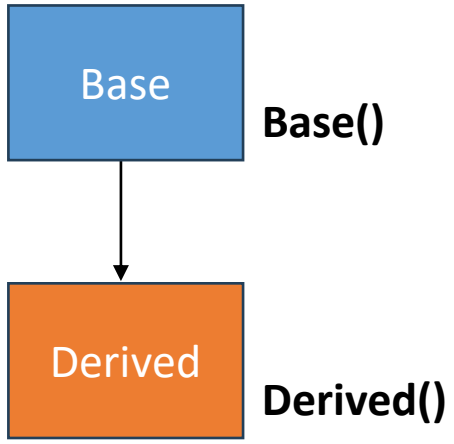
In developing a graphics framework, when incorporating specific shapes like **Circle** and **Square** from a common **Shape** class, the challenge of the "diamond problem" in multiple inheritance arises.
To prevent ambiguity and ensure a clear hierarchy, modifications to the **Circle**, **Square**, and **Diamond** classes involve implementing virtual inheritance.

This strategic use of the **virtual** keyword guarantees the correct invocation of the **displayInfo()** function from the **Shape** class when called by a **Diamond** object. The approach emphasizes scalability, anticipates future shape additions, and necessitates effective communication within the development team for collaborative implementation in the graphics framework.

```
#include <iostream>

class Shape {
public:
    Shape(int sides) : numSides(sides) {}
    virtual void displayInfo() const {
        cout << "Shape with " << numSides << " sides.“;
    }
private:
    int numSides;
};

class Circle : public virtual Shape {
public:
    Circle() : Shape(0) {}
    // No additional code needed
};

class Square : public virtual Shape {
public:
    Square() : Shape(4) {}
    // No additional code needed
};

class Diamond : public Circle, public Square {
public:
    // No additional code needed
};

int main() {
    // Create an object of the Diamond class
    Diamond diamond;

    // Call the displayInfo() function on the Diamond object
    diamond.displayInfo();

    return 0;
}
```

Constructors in Inheritance

- We can use **constructors** in derived classes in C++
- When we create an object of the **Derived class**, the default constructors are called in the order of **inheritance**, from the **base class to the derived class.**



Derived d;
First constructor Base() is called then Derived() called.

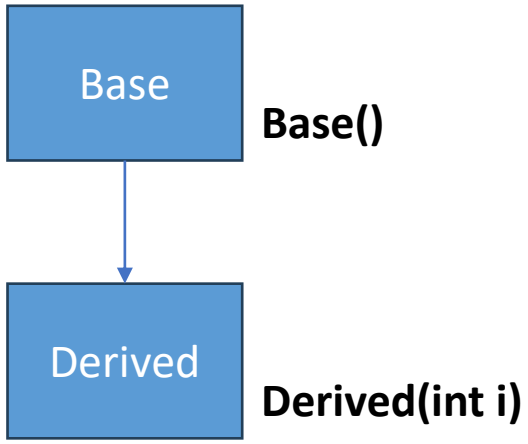
```
class A {   //base class
public:
   A()  //base class constructor
   {
      cout << "Base Class Default Constructor";
   }
};
class B : public A {  //derived class
public:
   B() { //derived class constructor
      cout << "Derived Class Default Constructor“;
   }
};
int main()
{
   B b;  // Creating an object of the derived class
  return 0;
}
```

OUTPUT:
Base Class Default Constructor
Derived Class Default Constructor

Default Constructor in base class & Parameterized Constructor in Derived class Constructors

- If the **base class constructor had default constructor and derived class have default and parameterized constructors**

- When we **create an object of derived class** with parameters then base class default constructor is called first then derived class **parameterized constructor is called**




```
class Base {
public: int x;
   Base()
    {
      cout << "Base Class Default Constructor" << endl;
    }
};
class Derived : public Base {
public: int y;
Derived(int i)
  {  y= i;
     cout << "Derived class parameterized constructor\n";
     cout<<“Value of y in Derived class is”<<y;
  }
};
int main() {
Derived d2(10); //calls Base() and Derived(int i)
 return 0;
}
```

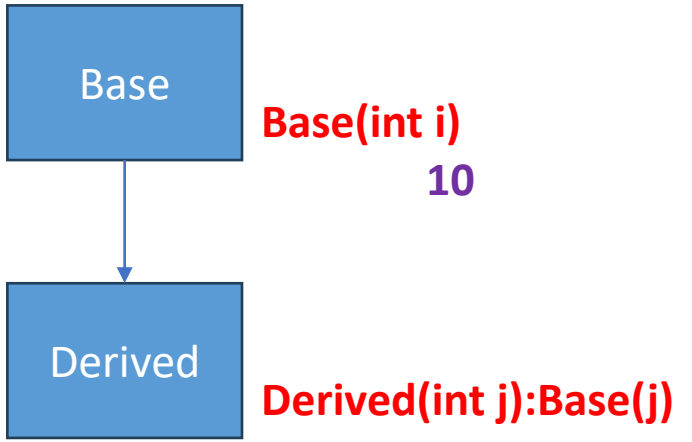
OUTPUT:

Base class default constructor
Derived parameterized constructor
Value of y in Derived class is: 10

Parameterized Constructor in Base and Derived classes

- If there are **one or more arguments in the base class constructor**, **derived class need to pass argument to the base class constructor.**

```
//when one parameter in base and derived
Derived-Const(arg): Base -Constr(arg)
{
}
```



Base(int i)

10

Derived(int j):Base(j)

10 10

OUTPUT:

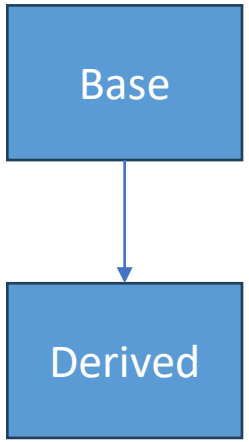
```
Base Parameterized Constructor
Value of x in base class: 10
Derived Parameterized Constructor
Value of x in base class: 10
```

```
class Base
{   int x;
  public:
  Base(int i) // parameterized constructor
  {   x = i;
      cout << "Base Parameterized Constructor\n";
      cout << “Value of x in base class: “<<x;
  }
};
class Derived: public Base
{    int y;
  public:
  Derived(int j):Base(j) // parameterized constructor
  {     y = j;
      cout << "Derived Parameterized Constructor\n";
      cout << “Value of y in Derived class: “<<y;
  }
};
int main()
{
  Derived d(10) ;
}
```

Parameterized Constructors in Base and Derived classes

- If there are **one or more arguments in the base class constructor, derived class need to pass argument to the base class constructor.**

```
//when more than one parameters
Derived-Const(arg1, arg2, arg3): Base Constr (arg1,arg2)
{
}
```



Base(int i)
10

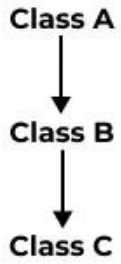
Derived(int j,int k):Base(j)
10 20 10

OUTPUT:

```
Base Parameterized Constructor
Value of x in base class: 10
Derived Parameterized Constructor
Value of y in base class: 20
```

```
class Base {
private:    int x;
public:
   Base(int i) {       x = i;
      cout << "Base Parameterized Constructor: " << x << endl;
      cout << “Value of x in base class: “<<x;
   }
};
class Derived : public Base {
private:    int y;
public:
   Derived(int j, int k) : Base(j) {
      y = k;
      cout << "Derived Parameterized Constructor: " << y ;
      cout << “Value of y in base class: “<<y;
   }
};

int main() {
   Derived d(10, 20);
   return 0;
}
```

order of execution of constructors and destructors.



Order of Constructor Call	Order of Destructor Call
A() - Class A Constructor	C() - Class C Destructor
B() - Class B Constructor	B() - Class B Destructor
C() - Class C Constructor	A() - Class A Destructor

```
class A
{    public:
    A() {
        cout << “class  A Constructor\n";
    }
};
class B : public A
{     public:
    B() {
    cout << “class B  Constructor\n";
    }
};
Class C: public B
{ public:
   C() { cout<<“class  C Constructor \n”;
}
int main()
{
    B c;
    return 0;
}
```

```
Class A Constructor
Class B Constructor
Class C Constructor
```