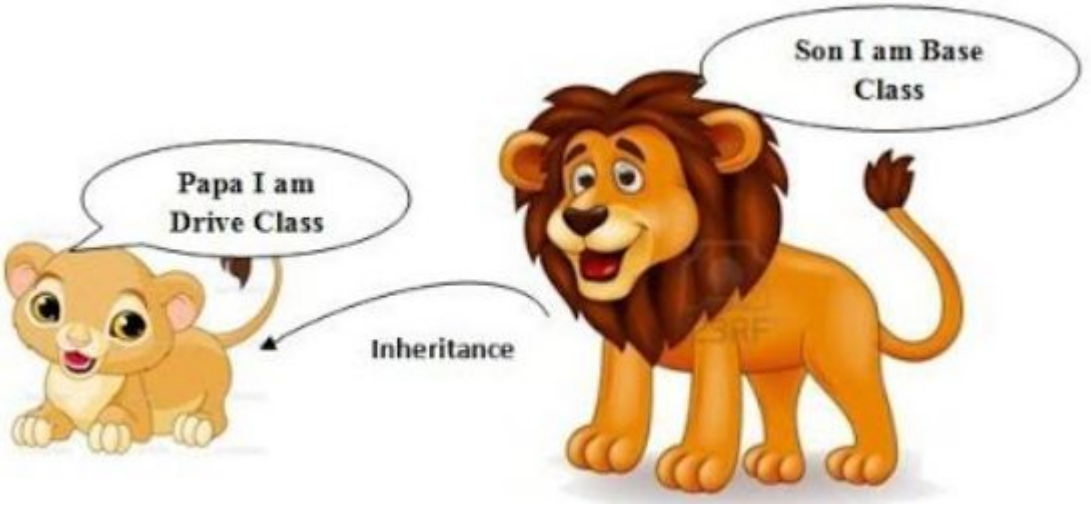
Inheritance

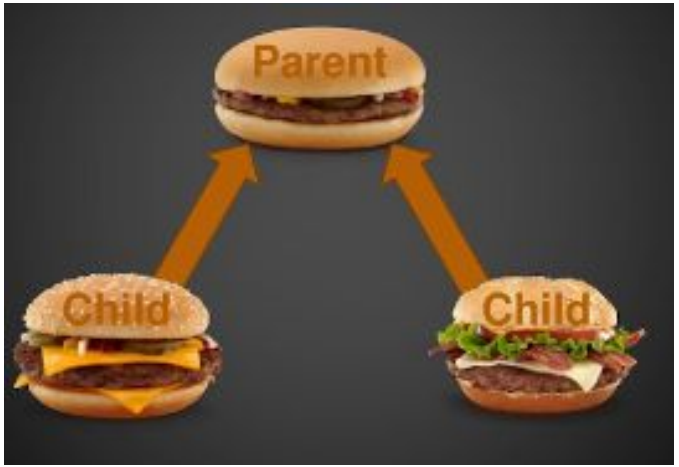


Is A Relationship

Sometimes, one class is an extension of another class

A car is a vehicle

Cricket is a sport



Is A Relationship

The extended (or child) class contains all the features of its base (or parent) class, and may additionally have some unique features of its own.

The key idea behind `inheritance`.

Inheritance

Capability of a class to derive properties and characteristics from another class.

The existing class is called base class (or sometimes super class) and the new class is referred to as derived class (or sometimes sub class).

The car is a vehicle, so any attributes and behaviors of a vehicle are also attributes and behaviors of a car.

Base & Derived Classes

The class that inherits properties from another class is called Sub class or Derived Class or Child Class.

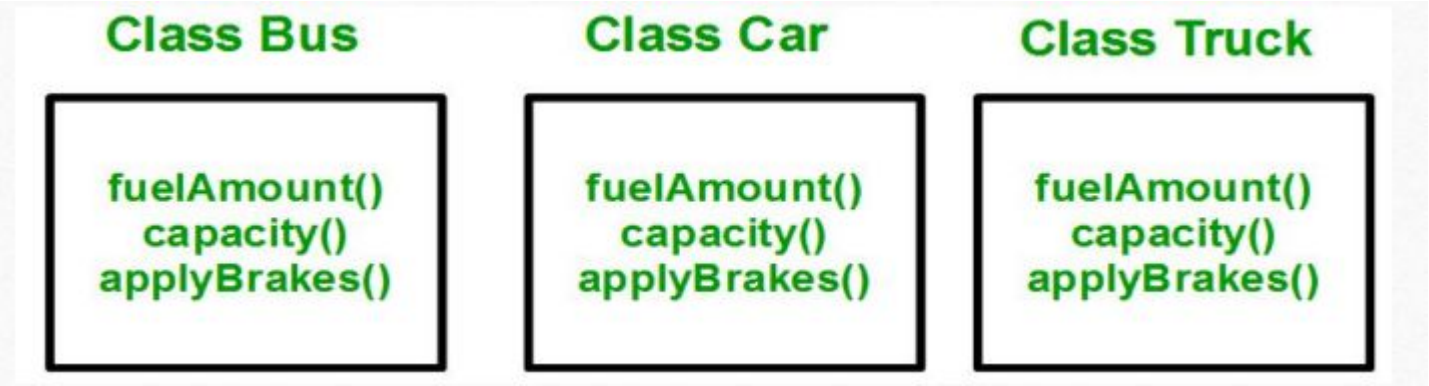
Super Class: The class whose properties are inherited by sub class is called Base Class or Super class or Parent Class.

Every derived-class object is also an object of its base class, and one base class can have many derived classes.

A derived class can access all non-private members of its base class.

Why and when to use inheritance?

Consider a group of vehicles. You need to create classes for Bus, Car and Truck. The methods fuelAmount(), capacity(), applyBrakes() will be same for all of the three classes.

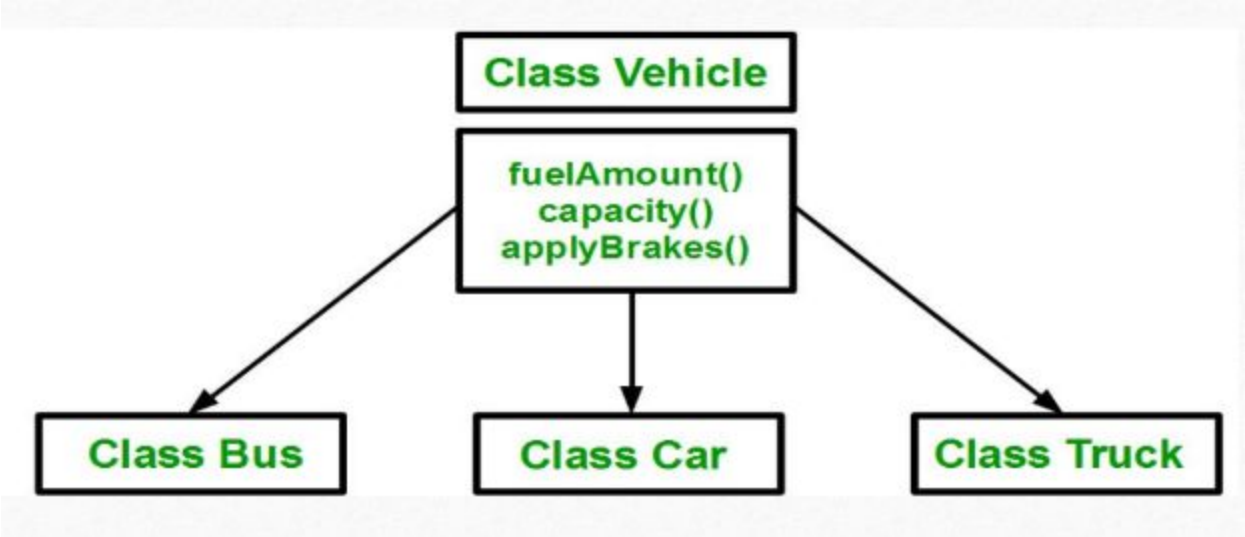


Why and when to use inheritance?

Above process results in duplication of 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.

Why and when to use inheritance?



Implementing inheritance in C++

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

Inheritance

```
class Vehicle
{
 // data members of base class
}

class Car: public Vehicle
{
    //data members of derived class
}
```

Modes of Inheritance

Public mode: If we derive a sub class 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 derived class.

Protected mode: If we derive a sub class from a Protected base class. Then both public member and protected members of the base class will become protected in derived class.

Private mode: If we derive a sub class from a Private base class. Then both public member and protected members of the base class will become Private in derived class.

Modes of Inheritance

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)

Modes of Inheritance

Base-class member-access specifier	Type of inheritance		
	`public` inheritance	`protected` inheritance	`private` inheritance
`public`	`public` in derived class. Can be accessed directly by member functions, `friend` functions and nonmember functions.	`protected` in derived class. Can be accessed directly by member functions and `friend` functions.	`private` in derived class. Can be accessed directly by member functions and `friend` functions.
`protected`	`protected` in derived class. Can be accessed directly by member functions and `friend` functions.	`protected` in derived class. Can be accessed directly by member functions and `friend` functions.	`private` in derived class. Can be accessed directly by member functions and `friend` functions.
`private`	Hidden in derived class. Can be accessed by member functions and `friend` functions through `public` or `protected` member functions of the base class.	Hidden in derived class. Can be accessed by member functions and `friend` functions through `public` or `protected` member functions of the base class.	Hidden in derived class. Can be accessed by member functions and `friend` functions through `public` or `protected` member functions of the base class.

Public Inheritance

The **public** members of a base class are treated as **public** members of the derived class by other classes further down the hierarchy.

Example:

```
class myDerived: public myBase
{
    // derived class members
}
```

The **protected** members of a base class are treated as **protected** members of the derived class by other classes further down the hierarchy.

Public Inheritance

```
 class Parent
{
      private:   int a;
 public:     int b;
      protected:    int c;
}


 class Child: public Parent
{
    // can never access a directly
   // can access b & c directly
}
```

```
class GrandChild: public Child
{
  // can never access a directly
  // can access b directly
   // can access c directly
}
```

Protected Inheritance

The **public** members of a base class are treated as **protected** members of the derived class by other classes further down the hierarchy.

Example:

```
class myDerived: protected myBase
{
    // derived class members
}
```

The **protected** members of a base class are treated as **protected** members of the derived class by other classes further down the hierarchy.

Protected Inheritance

```
class Parent
{
      private:    int a;
 public:      int b;
      protected:    int c;
}

class Child: protected Parent
{
   // can never access a directly
   // can access b & c directly
}
```

```
class GrandChild: public Child
{
   // can never access a directly
   // can access b directly
    // can access c directly
}
```

Private Inheritance

All **public** & **protected** members of a base class are treated as **private** members of the derived class by other classes further down the hierarchy.

In other words, these members can be seen as locked and cannot be accessed further down the hierarchy.

Example:

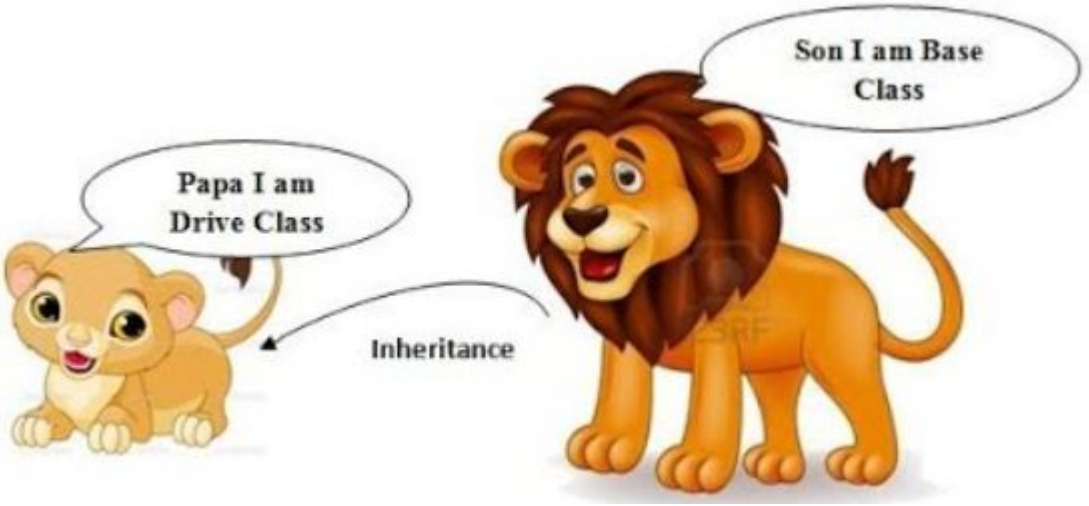
```
class myDerived: private myBase
{
    // derived class members
}
```

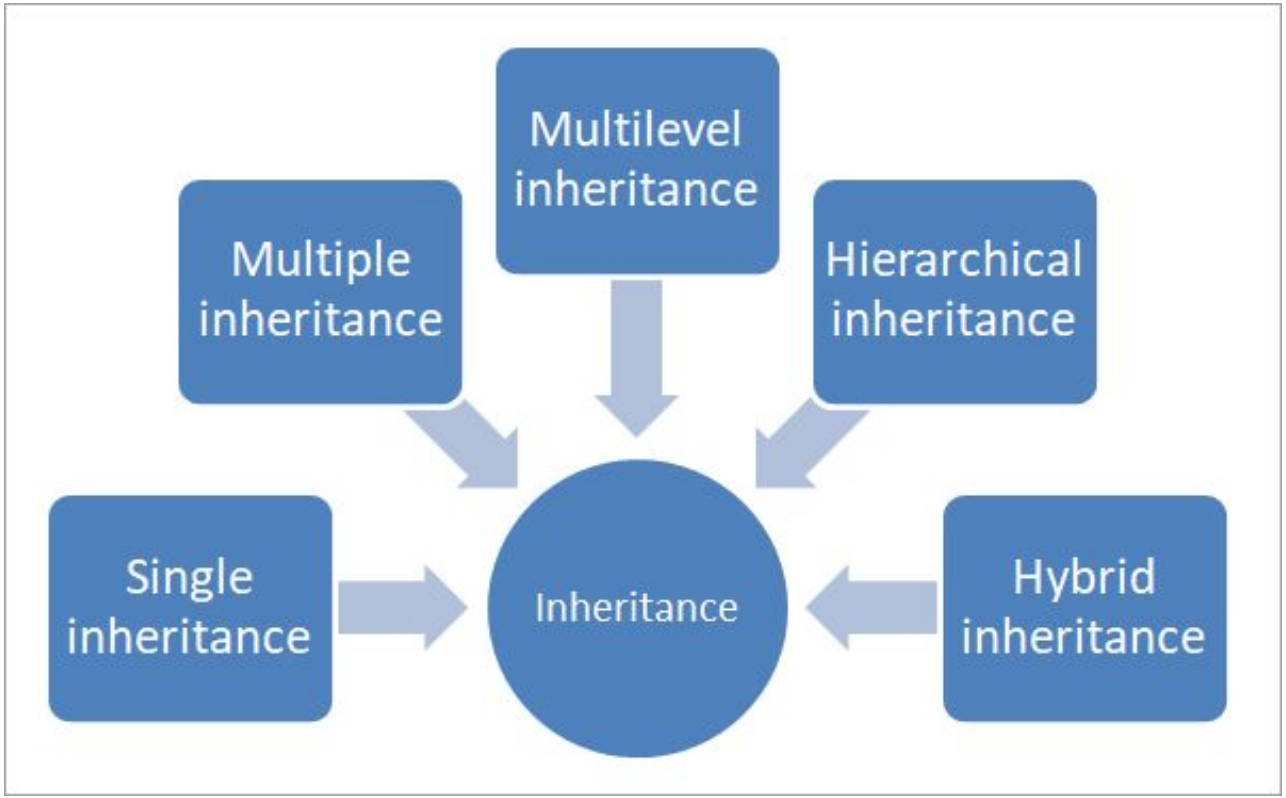
Private Inheritance

```
class Parent
{
      private:    int a;
 public:      int b;
      protected:    int c;
}


 class Child: private Parent
{
     // can never access a directly
    // can access b & c directly
}
```

```
class GrandChild: public Child
{
   // can never access a directly
   // cannot access b directly
    // cannot access c directly
}
```

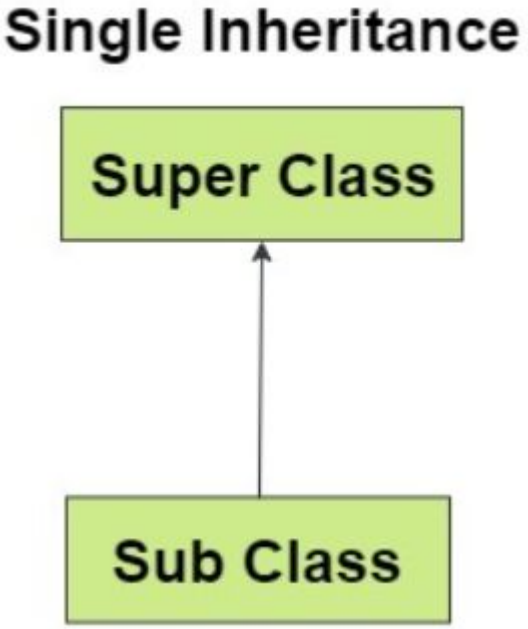
Types of Inheritance




Multilevel inheritance
Multiple inheritance
Hierarchical inheritance
Single inheritance
Inheritance
Hybrid inheritance

Single Inheritance

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




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

Single Inheritance

```
#include <iostream>
using namespace std;
class Account {
public:
float salary = 60000; };

class Programmer: public Account {
public:
float bonus = 5000; };

int main(void) {
Programmer p1;
cout<<"Salary: "<<p1.salary<<endl;
cout<<"Bonus : "<<p1.bonus<<endl;
return 0; }
```

Output????

```
#include <iostream>
#include <string>
using namespace std;
class Animal
{
   string name="";
   public:
   int tail=1;
   int legs=4;

};
class Dog : public Animal
{
   public:
   void voiceAction()
   {
      cout<<"Barks!!!";
   }
};
int main()
{
   Dog dog;
   cout<<"Dog has "<<dog.legs<<" legs"<<endl;
   cout<<"Dog has "<<dog.tail<<" tail"<<endl;
   cout<<"Dog ";
   dog.voiceAction();
}
```

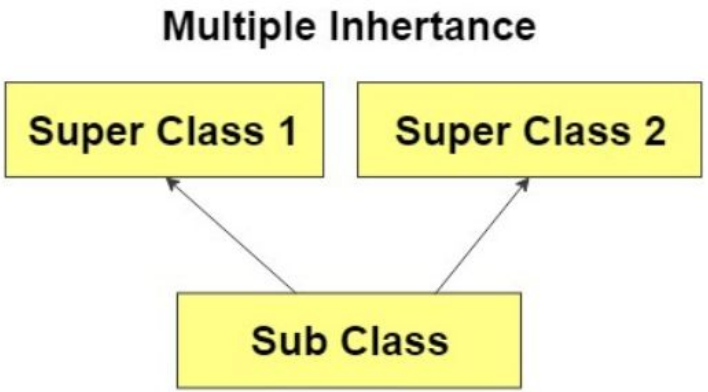
Multiple Inheritance

In Multiple Inheritance a class can inherit from more than one classes.

i.e one sub class is inherited from more than one base classes.

```
class subclass_name : access_mode base_class1,
access_mode base_class2, ....
{
//body of subclass
};
```

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



Multiple Inheritance

```
#include <iostream>
using namespace std;
class Account {
public:
float salary = 60000; };

class Languages {
public:
string language1 = "C++";
};

class Programmer: public Account, public Languages {
public:
float bonus = 5000; };
```

```
int main(void) {
Programmer p1;
cout<<"Salary: "<<p1.salary<<endl;
cout<<"Language:
"<<p1.language1<<endl;
cout<<"Bonus : "<<p1.bonus<<endl;
return 0; }
```

Multiple Inheritance

```
#include <iostream>
using namespace std;
//multiple inheritance example
class student_marks {
protected:
int rollNo, marks1, marks2;
public:
void get() {
cout << "Enter the Roll No.: "; cin >> rollNo;
cout << "Enter the two highest marks: "; cin >> marks1 >> marks2;
   }
};
class cocurricular_marks {
protected:
int comarks;
public:
void getsm() {
cout << "Enter the mark for CoCurricular Activities: "; cin >> comarks;
   }
};
```

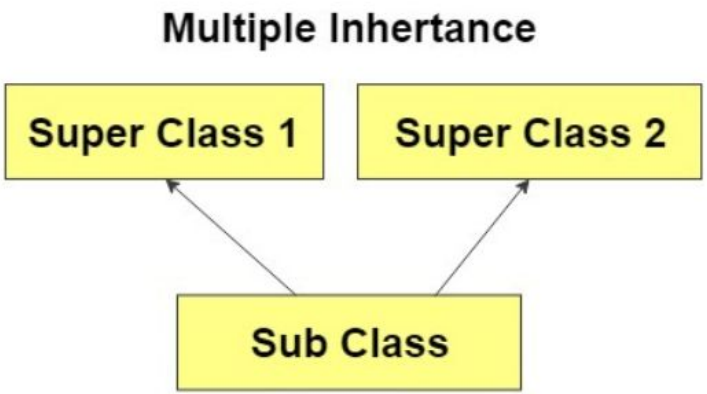
Multiple Inheritance

```
//Result is a combination of subject_marks and cocurricular activities marks
class Result : public student_marks, public cocurricular_marks {
   int total_marks, avg_marks;
   public:
   void display()
   {
      total_marks = (marks1 + marks2 + comarks);
      avg_marks = total_marks / 3;
      cout << "\nRoll No: " << rollNo << "\nTotal marks: " << total_marks;
      cout << "\nAverage marks: " << avg_marks;
   }
};
int main()
{
   Result res;
res.get(); //read subject marks
res.getsm(); //read cocurricular activities marks
res.display(); //display the total marks and average marks
}
```

Multiple Inheritance

Ambiguity can be occurred in using the multiple inheritance when a function with the same name occurs in more than one base class.

The above issue can be resolved by using the class resolution operator
with the function.



Multiple Inheritance

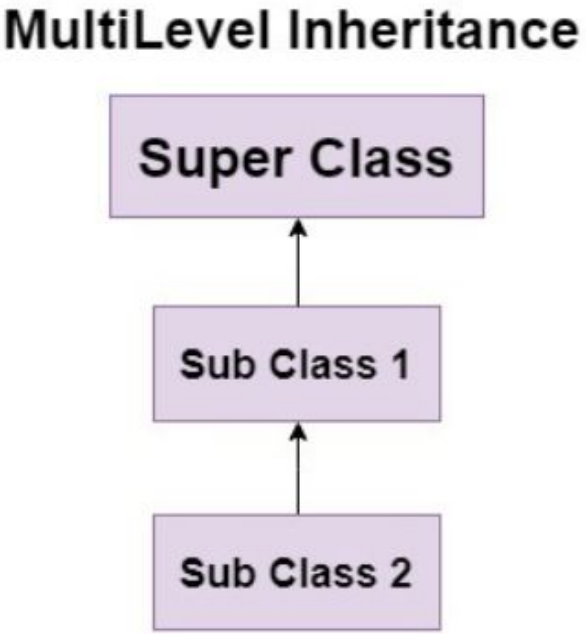
```
#include <iostream>
using namespace std;
class Account {
public:
float salary = 60000;
void display(){
    cout<<salary; }
};

class Languages {
public:
string language1 = "C++";
void display(){
    cout<<language1; }
};

class Programmer: public Account, public Languages {
public:
float bonus = 5000; };
```

```
int main(void) {
Programmer p1;
p1.Languages :: display();
return 0; }
```

MultiLevel Inheritance

Multilevel inheritance is a process of deriving a class from another derived class.

When one class inherits another class which is further inherited by another class, it is known as multi level inheritance in C++. Inheritance is transitive so the last derived class acquires all the members of all its base classes.



Multilevel Inheritance

```
#include <iostream>
using namespace std;
class Account {
public:
float salary = 60000; };

class Programmer: public Account {
public:
float bonus = 5000; };

class FrontendDeveloper: public Programmer {
public:
float extra = 15000; };

int main(void) {
FrontendDeveloper p1;
cout<<"Total Salary: "<<p1.salary + p1.bonus+ p1.extra<<endl;
return 0; }
```

MultiLevel Inheritance

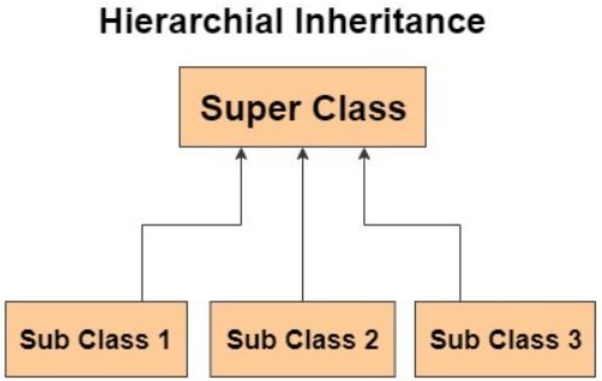
```
#include <iostream>
#include <string>
using namespace std;
class Animal
{
   string name="";
   public:
   int tail=1;
   int legs=4;

};
class Dog : public Animal
{
   public:
   void voiceAction()
   {
      cout<<"Barks!!!";
   }
};
class Puppy:public Dog{
   public:
   void weeping()
   {
      cout<<"Weeps!!";
   }
};
```

```
int main()
{
   Puppy puppy;
cout<<"Puppy has "<<puppy.legs<<" legs"<<endl;
cout<<"Puppy has "<<puppy.tail<<" tail"<<endl;
cout<<"Puppy ";
puppy.voiceAction();
cout<<" Puppy ";
puppy.weeping();
}
```

Hierarchical Inheritance

Hierarchical inheritance is defined as the process of deriving more than one class from a base class.

```
class A
{ // body of the class A.};

class B : public A
{ // body of class B. };

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

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



Hierarchical Inheritance

```
#include <iostream>
using namespace std;
class Account {
public:
float salary = 60000; };

class Programmer: public Account {
public:
float bonus = 5000; };

class HR: public Account {
public:
float bonus = 1000; };

int main(void) {
Programmer p1;
HR h1;
cout<<"Programmer: "<<p1.salary + p1.bonus<<endl;
cout<<"HR : "<<h1.salary +h1.bonus<<endl;
return 0; }
```

Hierarchical Inheritance

```
#include <iostream>
using namespace std;
//hierarchical inheritance example
class Shape                    // shape class -> base class
{
public:
int x,y;

void get_data(int n,int m) {
      x= n;
      y = m;
   }
};
class Rectangle : public Shape // inherit Shape class
{
public:
int area_rect() {
int area = x*y;
return area;
   }
};
```

Hierarchical Inheritance

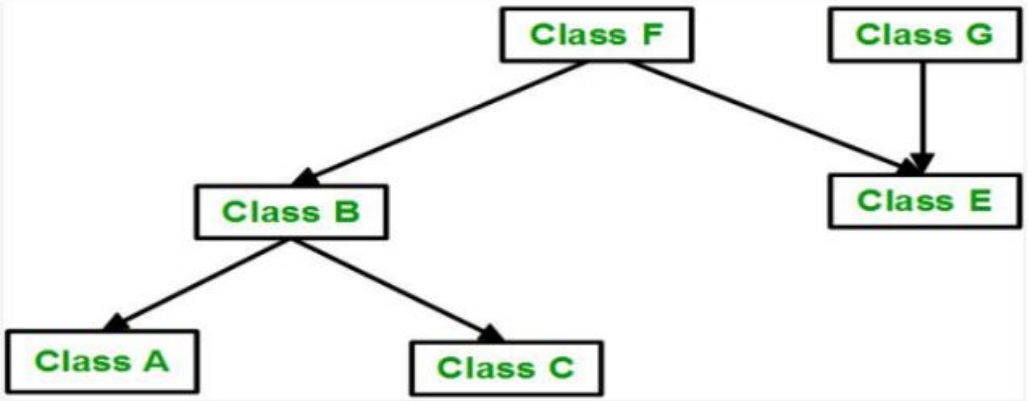
```
class Triangle : public Shape // inherit Shape class
{
public:
int triangle_area() {
float area = 0.5*x*y;
return area;
    }
};
class Square : public Shape // inherit Shape class
{
public:
int square_area() {
float area = 4*x;
return area;
    }
};
```

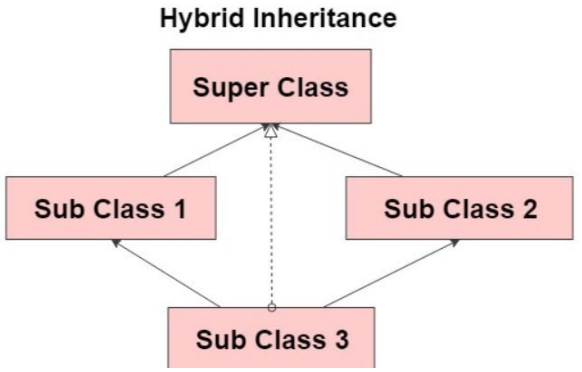
```
int main()
{ Rectangle r;
   Triangle t;
   Square s;
   int length,breadth,base,height,side;
   //area of a Rectangle
   std::cout << "Enter the length and breadth of a rectangle: "; cin>>length>>breadth;
   r.get_data(length,breadth);
   int rect_area = r.area_rect();
   std::cout << "Area of the rectangle = " <<rect_area<< std::endl;
   //area of a triangle
   std::cout << "Enter the base and height of the triangle: "; cin>>base>>height;
   t.get_data(base,height);
   float tri_area = t.triangle_area();
   std::cout <<"Area of the triangle = " << tri_area<<std::endl;
   //area of a Square
   std::cout << "Enter the length of one side of the square: "; cin>>side;
   s.get_data(side,side);
   int sq_area = s.square_area();
   std::cout <<"Area of the square = " << sq_area<<std::endl;
   return 0;
}
```

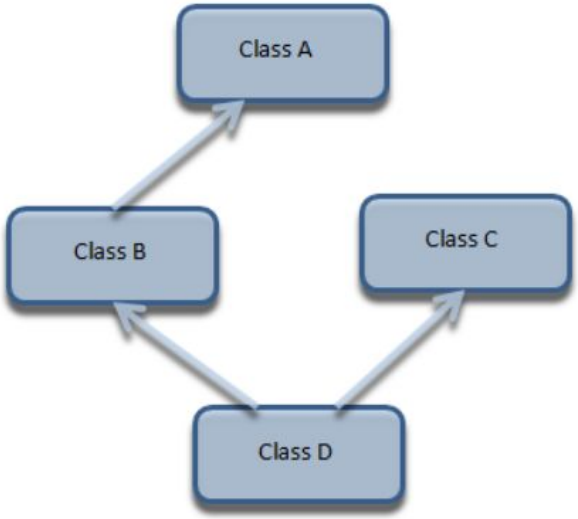
Hybrid (Virtual) Inheritance

Hybrid inheritance is a combination of more than one type of inheritance.

For example: Combining Hierarchical inheritance and Multiple Inheritance.







Hybrid Inheritance

```
#include <iostream>
#include <string>
using namespace std;
//Hybrid inheritance = multilevel + multilpe
class student{ //First base Class
                int id;
                string name;
                public:
                void getstudent(){
                                cout << "Enter student Id and student name"; cin >> id >> name;
    }
};
class marks: public student{ //derived from student
                protected:
                int marks_math,marks_phy,marks_chem;
                public:
                void getmarks(){
                                cout << "Enter 3 subject marks:"; cin >>marks_math>>marks_phy>>mar
    }
};
```

Hybrid Inheritance

```
class sports{
              protected:
              int spmarks;
              public:
              void getsports(){
                              cout << "Enter sports marks:"; cin >> spmarks;
   }
};
class result : public marks, public sports{//Derived class by multiple inheritance//
               int total_marks;
               float avg_marks;
               public :
              void display(){
                            total_marks=marks_math+marks_phy+marks_chem;
                            avg_marks=total_marks/3.0;

                          cout << "Total marks =" << total_marks << endl;
                          cout << "Average marks =" << avg_marks << endl;
                          cout << "Average + Sports marks =" << avg_marks+spmarks;
   }
};
```

Hybrid Inheritance

```
int main(){
                result res;//object//
                res.getstudent();
                res.getmarks();
                res.getsports();
                res.display();
               return 0;
}
```

```
Enter student Id and student name 25 Ved
Enter 3 subject marks:89 88 87
Enter sports marks:40
Total marks =264
Average marks =88
Average + Sports marks =128
```

Class Activity

Create a class named MusicalComposition that contains fields for title, composer, and year written(attributea only accessible by its subclass). Include a setter function that requires all three values and an appropriate display function. The child class NationalAnthem contains an additional field that holds the name of the anthem’s nation. The child class setter requires a value for this additional field. The child class also contains a display function.

Write a main() function that instantiates object of subclass and call both display functions .

Class Activity

Make a class named Fruit with a data member to calculate the number of fruits in a basket. Create two other class named Apples and Mangoes to calculate the number of apples and mangoes in the basket. Print the number of fruits of each type and the total number of fruits in the basket.

Class Activity

Create two classes named Mammals and MarineAnimals. Create another class named BlueWhale which inherits both the above classes. Now, create a function in each of these classes which prints "I am mammal", "I am a marine animal" and "I belong to both the categories: Mammals as well as Marine Animals" respectively. Now, create an object for each of the above class and try calling

1 - function of Mammals by the object of Mammal

2 - function of MarineAnimal by the object of MarineAnimal

3 - function of BlueWhale by the object of BlueWhale

4 - function of each of its parent by the object of BlueWhale

Order Of Constructor

When we construct a derived class object, the base object must be created first.

If we do not specify any base-constructor, it calls a default base-constructor. This is because the base-constructor does initialization of derived object's the inherited base-class member. The members of derived object are initialized by the derived-constructor.

Order Of Constructor

Instantiating a derived-class object begins a **chain of constructor calls** in which the derived-class constructor, before performing its own tasks, invokes its direct base class's constructor either explicitly (via a base-class member initializer) or implicitly (calling the base class's default constructor).

Similarly, if the base class is derived from another class, the base-class constructor is required to invoke the constructor of the next class up in the hierarchy, and so on.

The last constructor called in this chain is the constructor of the class at the base of the hierarchy, whose body actually finishes executing first. The original derived-class constructor's body finishes executing last. Each base-class constructor initializes the base-class data members that derived-class object inherits.

DESTRUCTOR CALLS

When an object is destroyed, the destructor of the derived class is first called, followed by the destructor of the base class.

The reverse order of the constructor calls applies.

You need to define a destructor for a derived class if actions performed by the constructor need to be reversed. The base class destructor need not be called explicitly as it is executed implicitly.

Order Of Constructor & Destructor

Derived-class constructor always calls a base-class constructor.

The base-class object should be constructed before the code enters the body of the derived-class constructor.

Order Of Constructor & Destructor

```
#include <iostream>
using namespace std;
class A{
public:
A() { cout << "A()" << endl; }};
class B : public A{
public:
B() { cout << "B()" << endl; }}; // BY DEFAULT B() : A()

int main(){
B b;
return 0;
}
```

Important Points

Whenever the derived class’s default constructor is called, the base class’s default constructor is called automatically.

To call the parameterized constructor of base class inside theparameterized constructor of sub class, we have to mention it explicitly.

The parameterized constructor of base class cannot be called in default constructor of sub class, it should be called in the parameterized constructor of sub class.

Constructor & Destructor

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

class A{
protected:
int ia;
public:
A(int n) : ia(n) { cout << "A()" << endl; }};

class B : public A{
int ib;
public:
B(int n) : ib(n)
{ cout << "B()" << endl; } //[Error] no matching function for call to 'A::A()'
};
int main(){
B b(2);
return 0;}
```

Parameterized Base Constructor

If we need to initialize inherited the base-class member with different value form a default value, we can use base-class constructor in the initializer list of the derived-class constructor.

```
Derived class constructor(arg1,arg2) : base(arg1){

derived= arg2;}

// order of arguments doesn't matter
```

OR

```
Derived class constructor(arg1,arg2) : base(arg1) , derived(arg2){}
```

OR

```
Derived class constructor(arg1) : base(any value) , derived(arg1){}
```

Parameterized Base Constructor

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

class A{
public:
A(int n = 1) : ia(n)
{ cout << "A() ia = " << ia << endl; }
protected:
int ia;};

class B : public A{
int ib;
public:
B(int n , int a) : ib(n), A(a)
{ cout << "B()" << endl; } };

int main(){
B b(2 ,4);
return 0;}
```

Parameterized Base Constructor

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

class A{
public:
A(int n = 1) : ia(n)
{ cout << "A() ia = " << ia << endl; }
protected:
int ia;};

class B : public A{
int ib;
public:
B(int n ) : ib(n), A(n)
{ cout << "B()" << endl; } };

int main(){
B b(2);
return 0;}
```

Constructor and destructor in Hierarchical inheritance

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

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

class B : public A{
public:
B() { cout << "B()" << endl; } };

class C : public A{
public:
C() { cout << "C()" << endl; } };

int main(){
B b;
C c;
return 0;}
```

Constructor and destructor in multiple inheritance

Constructors from all base class are invoked first and
the derived class constructor is called.

Order of constructor invocation depends on the order of
how the base is inherited.

For example:
class C :public A , public B

constructor of class A is called first and then constructor of class B is called.

However, the destructor of derived class is called first and then destructor of the base class which is mentioned in the derived class declaration is called from last towards first in sequentially.

Constructor and destructor in multiple inheritance

```
Derived class constructor(arg1,arg2,arg3) : base1(arg1) ,
                                            base2(arg2){
                                          derived= arg3;}
```

OR

```
Derived class constructor(arg1,arg2,arg3) : base1(arg1) ,
                                            base2(arg2),
                                          derived(arg3){}
```

Constructor and destructor in Multiple inheritance

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

class A{
public: A(int a) { cout << "A()" << a << endl; }};

class B{
public: B(int b ){ cout << "B()" << b << endl; } };

class C :public A , public B{
public:
int m;
C(int x, int y, int z): A(x) , B(y) ,m(z){
cout<< "C()" << m << endl;}};

int main(){
C c(2,4,6);
return 0;}
```

Constructor and destructor in Multiple inheritance

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

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

class B : public A{
public:
B() { cout << "B()" << endl; } };

class C : public B{
public:
C() { cout << "C()" << endl; } };

int main(){
C c;
return 0;}
```

Task:

Create a class named A include a constructor that accepts one parameter(any type) , create another class named B that inherits from A also include a constructor that accepts one parameter(any type) , create a class named C that inherits from B having a constructor that accepts one parameter now initialize all base class constructors through the child classes.

Remember if the base class is derived from another class, the base-class constructor is required to invoke the constructor of the next class up in the hierarchy.

Constructor and destructor in Multiple inheritance

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

class A{
public:
A(int a=1) { cout << "A()" << a << endl; }};

class B : public A{
public:
B(int b , int valA ):A(valA) { cout << "B()" << b << endl; } };

class C : public B{
public:
C(int c ,int valB):B(valB ,6) { cout << "C()" <<c << endl; } };

int main(){
C c(2,4);
return 0;}
```

Class ACTIVITY

Create an Investment class that contains fields to hold the initial value of an investment, the current value,the profit (calculated as the difference between current value and initial value), and the percent profit (the profit divided by the initial value). Include a constructor that requires initial and current values and a display function.

Create a House class that includes fields for street address and square feet, a constructor that requires values for both fields, and a display function.

Create a HouseThatIsAnInvestment class that inherits from Investment and House. It includes a constructor and a display function that calls the display functions of the parents.

Write a main()function that declares a HouseThatIsAnInvestment and displays its values.

References

https://order66.medium.com/oop-series-what-is-an-object-b22fa34933f3

https://www.geeksforgeeks.org/copy-constructor-in-cpp/

https://www.softwaretestinghelp.com/types-of-inheritance-in-cpp/