# Inheritance

## Introduction

* Inheritance is the one of the most powerful feature of object-oriented programming.
* Inheritance or derivation is the process of creating a new class, called **derived class** from existing class, called **base class.**
* The derived class inherits some or all the properties from base class. The base class is unchanged by this.
* **A derived class has direct access to both its own members and the public, and protected members of the base class.**
* The idea of inheritance implements the “is a”relationship.

**Person**

name

address

**Base Class**

**HOD**

department

**Teacher**

major

level

**Student**

roll

subject

**Derived Classes**

**Fig: Inheritance**

## Benefits of inheritance

The benefits of inheritance are listed below:

* Inheritance provides the concept of reusability. This means additional feature can be added to an existing class without modifying the original class. This helps to save development time and reduce cost of maintenance.
* It also reduces coding effort. Code sharing can occur at several places.
* It will permit the construction of reusable software components. The new software system can be generated more quickly and conveniently.

## Visibility Modifier (Access Specifier)

**Three visibility labels:** Private, protected, and public. **private** members of a class are accessible only from within the same class using member functions. private visibility label implements the concept of data hiding. We cannot access them outside of the class.

**protected** members are accessible by the member functions of the same class and also from any class immediately derived from it (i.e. Subclasses), and cannot be accessed by the functions outside these two classes. Thus, it forms the basis for inheritance.

Finally, **public** members are accessible from anywhere where the object is visible. It provide interface to interact with class members.

|  |  |  |  |
| --- | --- | --- | --- |
| **Visibility Modifier**  **(Access Specifier)** | **Accessible from own class** | **Accessible from**  **derived class** | **Accessible from**  **objects outside class** |
| Public | Yes | Yes | Yes |
| Private | Yes | No | No |
| Protected | Yes | Yes | No |

## Defining Derived Class (specifying Derived Class)/Defining inheritance

A derived class can be defined by specifying its relationship with the base class in addition to its own detail.

**The general syntax is:**

class derived\_class\_name : visibility\_mode base\_class\_name

{

**//members of derived class.**

} ;

Where, the colon (:) indicates that the derived\_class\_name is derived from the base\_class\_name. The visibility\_mode is optional, if present, may be **private, protected or public**. The default visibility mode is private. Visibility mode controls the visibility and availability of inherited base class members in the derived class.

**Four Important Points (regardless of access specifier):**

1. The constructors and destructors of a base class are not inherited.

2. The friend functions and friend classes of the base class are not inherited.

3. The derived class does not have access to the base class's private members.

4. The derived class has access to all public and protected members of the base class.

##### Based on these three visibility labels, there are three forms of inheritance.

1. public Derivation or public inheritance
2. private Derivation or private inheritance
3. protected Derivation or protected inheritance

Public inheritance expresses an “is-a” relationship: a B is a particular type of an A, as a car is a type of vehicle, a manager is a type of employee, and a square is a type of shape.

Protected and private inheritance serve different purposes from public inheritance. Protected inheritance makes the public and protected members of the base class protected in the derived class. Private inheritance makes the public and protected members of the base class private in the derived class.

### 1. Public Derivation or public inheritance

class ABC : public XYZ // public derivation

{

//members of ABC.

} ;

**When the base class is publicly inherited by derived class,** the public and protected members are inherited. The public members of base class become public member in derived class. So, they can be accessed through the objects of the derived class. Whereas protected members of base class become protected in derived class. And the private members cannot be inherited.

**Example:**

class D : public B **//publicly derived class D from base class B.**

{

private :

int k;

public :

void getk( )

{

cout<< "Enter k= ";

cin>>k ;

}

void sum( )

{

int s=y+z+k ;

cout<< "y+z+k= "<<s<<endl ;

}

} ;

int main( )

{

D d1;

d1.getdata( ) ; **//why?**

d1.getk( ) ;

d1.showdata( ) ; **//why?**

d1.sum( );

}

#include <iostream>

using namespace std;

class B

{

private:

int x;

protected:

int y;

public:

int z;

void getdata( )

{

cout<< "Enter 3 numbers= " ;

cin>>x>>y>>z;

}

void showdata( )

{

cout<< "x="<<x<<endl ;

cout<< "y="<<y<<endl ;

cout<< "z="<<z<<endl ;

}

} ;

### 2. Private Derivation or private inheritance

class ABC : private XYZ **// private derivation**

{

//members of ABC.

} ;

When the base class is privately inherited, all the public and protected members of the base class become private members of the derived class. So, these cannot be accessed outside the class directly through the derived class object, but can be accessed by public functions in the derived class. Like general private members, these members can be used freely within the derived class. With a private inheritance, subsequent derived classes cannot inherit properties of base class. If class B is derived privately from class A, and class C is derived from class B, then class C cannot have access to any members of class A.

class A **Base class(Grandfather)**

class B: private A **Intermediate base class(Father)**

Class C: public B **Derived class (Child)**

int main( )

{

D d1;

//d1.getdata();

//d1.showdata();

d1.setdata(4,6);

d1.sum( );

}

Class D: private B

**//Private inheritance**

**{**

private :

int z ;

public :

void setdata(int a, int b)

{

z = a;

}

void sum( )

{

cout<< "Sum: "<<z+y ;

}

} ;

#include <iostream>

using namespace std;

class B

{

private :

int x ;

protected :

int y ;

public :

void getdata( )

{

cout<< "enter x="; cin>>x ;

cout<< "enter y=" ; cin>>y ;

}

void showdata( )

{

cout<< "x="<<x<<endl ;

cout<< "y="<<y<<endl ;

}

};

### 3. Protected Derivation or protected inheritance

class ABC : protected XYZ **// protected derivation**

{

//members of ABC.

};

**When base class is derived using protected mode**, all the protected and public members of the base class become protected members of the derived class. This means, like a private inheritance, these members cannot be directly accessed through object of the derived class. But can be used freely within the derived class. Whereas, unlike a private inheritance, they can still be inherited and accessed by subsequent derived classes. In other words, protected inheritance does not end a hierarchy of classes, as private inheritance does.

**Example:**

#include <iostream>

class D : protected B **//Protected inheritance.**

{

private:

int z ;

public:

void setdata(int a, int b )

{

z = a;

y = b;

}

void showdata( )

{

cout<< "z: "<<z<<endl<<"y: "<<y ;

}

} ;

int main( )

{

D d1 ;

d1.getdata(); **//won't work, it is protected member of D.**

d1.display(); **//won't work, it is protected member of D.**

d1.setdata();

d1.showdata( );

}

using namespace std;

class B

{

private :

int x ;

protected :

int y ;

public :

void getdata( )

{

cout<< "enter x="; cin>>x ;

cout<< "enter y=" ; cin>>y ;

}

void display( )

{

cout<< "x="<<x<<endl ;

cout<< "y="<<y<<endl ;

}

};

**Summary:**

|  |  |  |  |
| --- | --- | --- | --- |
| **Base Class Visibility** | **Derived Class Visibility** | | |
| **Public derivation** | **Private derivation** | **Protected derivation** |
| Private | Not inherited | Not inherited | Not inherited |
| Protected | protected | private | protected |
| Public | Public | private | protected |

## Types of Inheritance

A class can inherit properties from one or more classes and from one or more levels. On the basis of this concept, there are five types of inheritance.

1. Single Inheritance
2. Multiple Inheritance
3. Hierarchical Inheritance
4. Multilevel Inheritance
5. Hybrid Inheritance

### 1 Single inheritance

If a class is derived from only one base class, then that is called single inheritance. The figure below show this inheritance.

**Example**

class A **//base class.**

{

A

B

members of A Base Class

};

Derived Class

class B : public A **//publicly derived class B.**

{

members of B

};

### 2 Multiple Inheritance

If a class is derived from more than one base class then inheritance is called as multiple inheritance. Multiple inheritance allows us to combine the features of several existing classes into one single class. It is like a child inheriting the physical features of one parent and the intelligence of another.

**The syntax of multiple inheritance is:**

class D: visibility B1, visibility B2 ........visibility Bn

{

member of class D.

} ;

The visibility can be private, public or protected. Note this is also possible that one visibility is public and another one is protected or private, etc.

Bn

B1

B2

B3

**For example:**

(1) class D : public B1, public B2

{private : int a ;} ;

(2) class D : public B1, protected B2

{private : int a;} ;

**D**

(3) class D : private B1, protected B2, public B3

**Fig: Multiple Inheritance**

{private : int a ;} ;

##### EXAMPLE:

#include<iostream>

using namespace std;

class biodata

{

protected:

char name[20] ;

char semester[20] ;

int age ;

int rn ;

public:

void getbiodata( )

{

cout<< "Enter name: "; cin>>name ;

cout<< "Enter semester: "; cin>>semester ;

cout<< "Enter age: " ; cin>>age ;

cout<< "Enter rn: " ; cin>>rn ;

}

void showbiodata( )

{

cout<< "Name:"<<name<<endl ;

cout<< "Semester:"<<semester<<endl ;

cout<< "Age:"<<age<<endl ;

cout<< "Rn:"<<rn<<endl ;

}

} ;

class marks

{

protected:

char sub[10] ;

float total ;

public:

void getrm( )

{

cout<< "Enter subject name:" ; cin>>sub ;

cout<< "Enter marks:" ; cin>>total ;

}

void showm( )

{

cout<< "Subject name:"<<sub<<endl ;

cout<< "Marks are:"<<total<<endl ;

}

} ;

class final: public biodata, public marks

{

char steacher[20] ;

public:

void gets( )

{

cout<< "Enter your subject teacher:" ;

cin>>steacher ;

}

void shows( )

{

cout<< "Subject teacher:"<<steacher<<endl ;

}

} ;

int main( )

{

final f ;

f.getbiodata( ) ;

f.getrm( ) ;

f.gets( ) ;

f.showbiodata( ) ;

f.showm( ) ;

f.shows( ) ;

}

### 3 Hierarchical Inheritance

When two or more than two classes are derived from one base class, it is called hierarchical inheritance. With the help of hierarchical inheritance, we can distribute the property of one class into many classes. The diagram for hierarchical inheritance is given below.

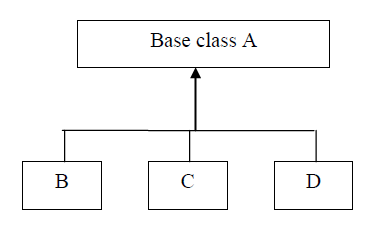


Fig: Hierarchical Inheritance

**General Format:**

class B { ------ }

class D1 : derivation B { ----- } ;

class D2 : derivation B { ----- } ;

class D3 : derivation B { ------} ; where derivation can be either public, protected or private type.

**Example**:

class A

{

……….

……….

};

class B : public A

{

………

...……

};

class C : public A

{

………

………

};

Here two classes B and C are derived from same base class A.

#### EXAMPLE:

#include<iostream>

using namespace std;

class B

{

protected:

int x, y ;

public:

void assign( )

{

x=10;

y=20;

}

}; **//end of class B.**

class D1: public B

{

int s ;

public:

void add( )

{

s=x+y;

cout<< "x+y: "<<s<<endl ;

}

};**//end of class D1.**

class D2: public B

{

int t ;

public:

void sub( )

{

int main()

{

D1 d1 ;

D2 d2 ;

D3 d3 ;

d1.assign( ) ;

d1.add( ) ;

d2.assign( ) ;

d2.sub( ) ;

d3.assign( ) ;

d3.mul( ) ;

}

t=x-y ;

cout<< "x-y: "<<t<<endl ;

}

};**//end of class D2.**

class D3: public B

{

int m ;

public:

void mul( )

{

m=x\*y;

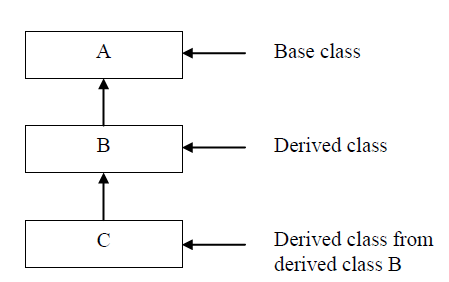
cout<< "x\*y: " <<m<<endl ;

}

};

### 4 Multilevel Inheritance

The mechanism of deriving a class from another derived class is called multilevel inheritance. If A, B, and C are three classes. And if A is a base class, B is derived from A, again C is derived from B then derived class B is called intermediate base class. Since class B provides a link for the inheritance between A and C. The chain ABC is known as inheritance path.



**Fig: multilevel inheritance**

**Base Class**

**Intermediate**

**Base Class**

**Derived Class**

**For example:**

class A {………..}; **//Base Class**

class B : public A{………..}; **//B derived from A**

class C : public B{………..}; **//C derived from B**

In this example class B is derived from base class A and class C is derived from derived class B.

##### Complete Example:

class student

{

protected:

char name[20] ;

int rn ;

public :

void getdata( )

{

cout<< "Student= "; cin>>name ;

cout<< "Roll no.= "; cin>>rn ;

}

void showdata( )

{

cout<< "Student= "<<name<<endl ;

cout<< "Roll no="<<rn<<endl ;

}

}; **// end of base class student.**

class marks : public student

{

protected:

int m1, m2 ;

public:

void getm( )

{

cout<< "enter marks in Maths:" ;

cin>>m1 ;

cout<< "enter marks in English=" ; cin>>m2 ;

}

void showm( )

{

cout<< "Maths: "<<m1<<endl ;

cout<< "English= "<<m2<<endl ;

}

}; **//end of intermediate base class marks.**

class result : public marks

{

int total ;

public:

void calculate( )

{

total=m1+m2 ;

}

void show( )

{

cout<< "Total marks= "<<total ;

}

}; **//end of derived class result.**

int main( )

{

result s1;

s1.getdata();

s1.getm();

s1.calculate();

s1.showdata();

s1.showm();

s1.show( );

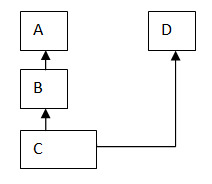
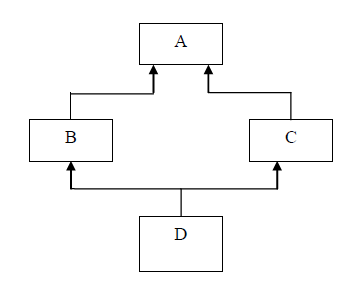
return 0;

}

In this example, class marks is derived from base class student and class result is derived from derived class marks.

### 5 Hybrid Inheritance

This inheritance is the combination of two or more types of inheritance.



**Fig: Hybrid Inheritance**

#include <iostream>

using namespace std;

class B1

{

protected :

int x ;

public:

void assignx( )

{

x=2;

}

}; **//end of class B1.**

class B2

{

protected:

int k ;

public:

void assignk( )

{

k=80;

}

};

class D1: public B1

{

protected :

int y ;

public :

void assigny( )

{

y=40;

}

}; **//end of class D1.**

class D2 : public D1

{

protected :

int z ;

public :

void assigz( )

{

z=60;

}

} ;

class D3 : public B2, public D2

{

private :

int z;

int total ;

public :

void assignz(int p)

{

z=p;

}

void output( )

{

total=x+y+z+k ;

cout<< "x+y+z+k=" <<total<<endl ;

}

};

int main( )

{

D3 s;

s.assignx();

s.assigny();

s.assignz(8);

s.assignk();

s.output( );

}

## Function Overriding

Defining a function in the derived class with same name as in the base class is called overriding. In this case, both base and derived class functions have same name, same number of arguments and similar type of arguments. Overriding is an object-oriented programming feature that enables a derived class to provide different implementation for a function that is already defined in its base class.

If base class and derived class have member functions with same name and arguments. If you create an object of derived class and write code to access that member function then, the member function in derived class is only invoked, i.e., the member function of derived class overrides the member function of base class. This feature in C++ programming is known as function overriding.

**Accessing the Overridden Function in Base Class From Derived Class**

If both derived class and base class contain same function name, then derived class object always accesses the derived class function. To access the overridden function of base class from derived class, scope resolution operator (::) is used.

**Syntax:**

base\_class\_name::function\_name(); **// Calling function function\_name () of base class base\_class\_name.**

**Example:**

class A

{

public:

void show()

{

cout<<"This is class A";

}

};

class B : public A

{

public:

void show() **//overridden function.**

{

cout<<"This is class B"<<endl;

}

};

int main()

{

B b;

b.show(); **//invokes the member function from class B.**

b.A :: show(); **//invokes the member function from class A.**

}

## Ambiguity in Multiple Inheritance

In multiple inheritance, when a member function with the same name appears in more than one base class, and if derived class does not have this member, and if we try to access this member using the objects of the derived class, it will be ambiguous. The ambiguity is that which base class function should be invoked by the compiler when we inherit those classes.

##### Example:

**class A**

{

public:

**void display()**

**{**

**cout<< “This is base class A ”;**

**}**

};

**class B**

{

public:

**void display()**

**{**

**cout<< “This is base class B ”;**

**}**

};

**class C: public A, public B**

{

//here class C does not contain a functionnamed **display();**

};

**int main()**

{

C c; **//creating object of derived class C.**

**c.display();** // ambiguous**- will not compile, *which function to call either of class A or class B.***

}

Two ways to solve this problem

#### 1. Using scope resolution operator.

To resolve this ambiguity we can call each of them using scope resolution operator.

**int main()**

{

C c; **//creating object of derived class C.**

**c.A::display();** // now not ambiguous, **calls display() function of class A.**

**c.B::display();** // now not ambiguous, **calls display() function of class B.**

}

The problem is solved using the scope resolution operator to specify the class in which the function lies. Thus **c.A::display();** refers to the version of display() that’s in the class A, while **c.B::display();** refers to the function in the class B.

#### 2. Function Overriding

We can solve this problem by defining a same name function in the derived class. Simply we override those base class functions.

**class A**

{

public:

**void display()**

**{**

**cout<< “This is base class A ”;**

**}**

};

**class B**

{

public:

**void display()**

**{**

**cout<< “This is base class B ”;**

**}**

};

**class C: public A, public B**

{

public:

void display() **//function overriding, here class C contain a function named display().**

**{**

A:: display(); **//calls display() function of class A.**

B:: display(); **//calls display() function of class B.**

**}**

};

**int main()**

{

C c; **//creating object of derived class C.**

**c.display();** // now not ambiguous, **calls display() function of derived class C.**

}

## Ambiguity in Hybrid Inheritance and Virtual Base classes

Another kind of ambiguity arises if we derive a class from two classes that are each derived from the same base class. This creates a diamond shaped inheritance, shown in the figure. This ambiguity is due to several paths exist to a base class from the derived class. The ambiguity is that the derived class has multiple copies of the same base class i.e., duplicate sets of members inherited from a single base class, hence compiler can’t decide which copy to use and signals an error.

A

C

B

D

**Fig: Multi-path Inheritance**

Classes B and C are derived from same base class A, and class D is derived by multiple inheritance from both B and C. Ambiguity arises when we try to access a public members in class A from an object of class D. In this case both B and C contain a copy of that function, inherited from A. The compiler can’t decide which copy to use, and signals an error.

**Consider following example.**

#include<iostream>

using namespace std;

class A

{

protected:

int adata;

public:

void getdata(int a)

{

adata = a;

}

void display()

{

cout<<" adtaa:"<<adata;

}

};

class B : public A

{

};

class C : public A

{

};

class D : public B, public C

{};

int main()

{

D d;

d.getdata(4); //ambiguous call to **getdata()** function, since two copies are available.

d.display(); //ambiguous call to **display()** function.

}

To resolve this ambiguity we need to use the concept of virtual base classes.

#### Virtual Base Classes

The duplication of inherited members due to multiple paths can be avoided by making the common base class as virtual base class. This can be achieved by declaring the base class as virtual when it is being inherited. Such a base class is known as virtual base class. This can be achieved by preceding the base class name with the word virtual.

When a class is made virtual, necessary care is taken by the compiler so that the duplication is avoided regardless of the number of paths that exist to the base class.

**Example:**

class A

{

protected:

int adata;

public:

void display()

{

cout<<" adtaa:"<<adata;

}

};

class B : virtual public A **//parent1**

{

};

class C : public virtual A **//parent2**

{

};

class D : public B, public C

{

};

int main()

{

D d;

d.getdata(4); //unambiguous, **since only one copy of is inherited.**

d.display(); //unambiguous, **since only one copy of is inherited.**

}

As we can see, the keyword virtual is used while deriving B, and C. Now that both B and C have inherited base class A as virtual, any multiple inheritance involving them will cause only one copy of base to be present. Therefore, in derived class D, there is only one copy of base A and hence the statements d.getdata(4), d.display() are perfectly valid and unambiguous.

Here, base class A is inherited as virtual base class to both B and C class, so derived class D has only one copy of the members of the base class A.

## Constructor in derived classes

It is possible for the base class, the derived class or both to have constructor and / or destructor. There are several cases.

#### Case I: base class contains no constructor or contains only default constructor

If base class contains no constructor or contains only default constructor, the derived class does not need a constructor function.

#### Case II: If the base class contains constructor with arguments:

However, if any base class contains a constructor with one or more arguments, then it is mandatory for the derived class to have a constructor and pass the arguments to the base class constructors.

##### Supplying values to base class constructor:

When applying inheritance we usually create objects using the derived class. If the base class contains a constructor it should be called from the initializer list in the derived class constructor as follows:

**Syntax:**

derived\_class\_constructor(arglist1, arglist2,…,arglistn):base1(arglist1), base2(arglist2), basen(arglistn)

{

**//body of derived constructor.**

}

When both the derived and base classes contain constructors, the base constructor is executed first and then the constructor in the derived class is executed.

**Example:**

class A

{

protected:

int adata;

public:

A(int a)

{

adata = a;

cout<<"A is initialized"<<endl;

}

};

class B : public A

{

int bdata;

public:

B(int x, int y) : A(x)

{

Bdat a = y;

cout<<"B is initialized"<<endl;

}

void showdata()

{

cout<<"adata = "<<adata<<endl <<"bdata = "<<bdata;

}

};

int main()

{

B b(5, 6);

b. showdata();

}

#### Case III: Case of multiple and multi-level inheritance:

In case of multiple inheritance, the base classes are constructed in order in which they appear in the declaration of the derived class. Similarly, in multilevel inheritance, the constructor will be executed in the order of inheritance.

**Example1: For Multiple Inheritance**

class A

{

protected:

int adata;

public:

A(int a)

{

adata = a;

cout<<"A is initialized"<<endl;

}

};

class B

{

protected:

int bdata;

public:

B(int b)

{

bdata = b;

cout<<"B is initialized"<<endl;

}

};

class C: public B, public A **//constructor of base class B is executed first then of base class A.**

{

int cdata;

public:

C(int a, int b, int c) : A(a), B(b)

{

cdata = c;

cout<<"C is initialized"<<endl;

}

void display()

{

cout<< "adata, bdata, cdata: " <<adata<<bdata<<cdata;

}

};

int main()

{

C v(4, 5, 6);

v.display();

}

**Example2: For Multilevel Inheritance**

class derived2 : public derived1

{

public:

derived2( )

{

cout<< " Constructing derived2 \n" ;

}

};

int main( )

{

derived2 obj;

return 0;

}

**OUTPUT:**

Constructing base

Constructing derived1

Constructing derived2

class base

{

public:

base( )

{

cout<< " Constructing base \n" ;

}

};

class derived1: public base

{

public:

derived1( )

{

cout<< " Constructing derived1 \n" ;

}

};

**Another Example:**

class Base1:public Base

{

int y;

public:

Base1 ( int d, int f):Base(f)

{

y=d;

cout << "Inside Base1 constructor" << endl;

}

~Base1 ( )

{

cout << "Inside Base1 destructor" << endl;

}

};

class Derived : public Base1

{

int k;

public:

Derived (int a, int b, int c ): Base1(b, a)

{

k=c;

cout << "Inside Derived constructor" << endl;

}

~Derived ( )

{

cout << "Inside Derived destructor" << endl;

}

};

int main( )

{

Derived x(4, 5, 8);

}

#include<iostream>

using namespace std;

class Base

{

int i;

public:

Base ( int f)

{

i=f;

cout << "Inside Base constructor" << endl;

}

~Base ( )

{

cout << "Inside Base destructor" << endl;

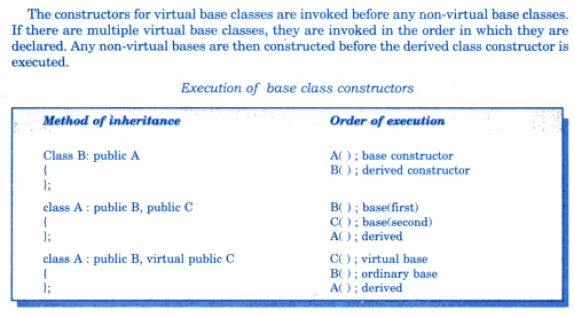
}

};

## Execution order of constructors & destructors

The base class constructor is executed first and then the constructor in the derived class is executed. In case of multiple inheritance, the base class constructors are executed in the order in which they appear in the definition of the derived class.

Similarly, in a multilevel inheritance, the constructors will be executed in the order of inheritance. Furthermore, the constructors for virtual base classes are invoked before any non-virtual base classes. If there are multiple virtual base classes, they are invoked in the order in which they are declared in the derived class.



The derived class destructor is executed first and then the destructor in the base class is executed. In case of multiple inheritance, the derived class destructors are executed in the order in which they appear in the definition of the derived class. Similarly, in a multilevel inheritance, the destructors will be executed in the order of inheritance.

**Example:**

class base

{

public:

base()

{

cout<< "Constructing base \n" ;

}

~ base() **//base class destructor**

{

cout<< "Destructing base \n" ;

}

};

class derived : public base

{

public:

derived()

{

cout<< " Constructing derived \n" ;

}

~ derived( ) **//derived class destructor**

{

**Output:**

Constructing base

Constructing derived

Destructing derived

Destructing base

cout<< "Destructing derived \n" ;

}

};

int main( )

{

derived obj;

return 0;

}

## Containership: (Aggregation)

A class can contain objects of another class as its members, which is called **containership** or **aggregation** or **nesting**. The class which contains the object is called container class. Containership implements a “has a” relationship. We say that a library has a book, meaning that each library includes an instance of a book. In object oriented programming, has a relationship occurs when object of one class is contained in another class. Containership is useful with classes that act like a data type. The object of these classes can be used almost like other variables in the class.

class A

{

……

……

};

class B

{

……

……

};

class C **//Here, class C is a container class.**

{

…..

A obj1; **//object of class A as a member.**

**B obj2; //object of class B as a member.**

…..

};

**Example:**

class Employee

{

int eid, sal;

public:

void getdata()

{

cout<< "Enter id and salary of employee"<<endl;

cin>>eid>>sal;

}

void display()

{

cout<< "Emp ID:"<<eid<<endl<<"Salary:"<<sal;

}

};

class Company **//Company is a container class.**

{

int cid;

char cname[45];

Employee e; **//Company contains object of class Employee.**

public:

int main()

{

Company c;

c.getdata();

c.display();

}

void getdata()

{

cout<< "Enter id and name of the company:"<<endl;

cin>>cid>>cname;

e.getdata();

}

void display()

{

cout<<"Comp ID: "<<cid<<endl<<"Comp Name:"<<cname;

e.display();

}

};

#### Think: If Company contains 10 employees, what modification is needed in above program?

**Solution:**

class Employee

{

int eid, sal;

public:

void getdata()

{

cout<< "Enter id and salary of employee"<<endl;

cin>>eid>>sal;

}

void display()

{

cout<< "Emp ID:"<<eid<<endl<<"Salary:"<<sal;

}

};

class Company

{

int cid;

char cname[45];

Employee e[10]; **//array of 10 Employees.**

public:

void getdata()

{

cout<< "Enter id and name of the company:"<<endl;

cin>>cid>>cname;

for(int i=0;i<10;i++)

{

e[i].getdata();

}

}

void display()

{

cout<<"Comp ID: "<<cid<<endl<<"Comp Name:"<<cname;

for(int i=0;i<10;i++)

{

e[i].display();

}

}

};

int main()

{

Company c;

c.getdata();

c.display();

}

**Above Program Using Constructor**

class B

{

float salary;

A d; **//here, class B contains object of class A, as a member.**

public:

B(int a, char c[ ], float r): d(a, c) //provides data to class A.

{

salary= r;

}

void display()

{

cout<<" Salary: " <<salary<<endl;

d.display();

}

};

int main()

{

B c(48,"Ram",87882.36);

c.display();

}

class A

{

int age;

char name[40 ];

public:

A(int a, char s[40])

{

age=a;

strcpy(name, s);

}

void display()

{

cout<<" name: "<<name<<endl;

cout<<" age: " <<age;

}

};

#### Above program using standard C++ string

#include<iostream>

#include<string>

using namespace std;

class A

{

int age;

string name;

public:

A(int a, string s)

{

age=a;

name=s;

}

void display()

{

cout<<" name: "<<name<<endl;

cout<<" age: " <<age;

}

};

class B

{

float salary;

A d;

public:

B(int a, string c, float r): d(a, c)

{

salary= r;

}

void display()

{

cout<<" Salary: " <<salary<<endl;

d.display();

}

};

int main()

{

B c(48,"Ram",87882.36);

c.display();

}

### Exercise

1. Define a student class (with necessary constructors and member functions). Derive a Computer Science and Mathematics class from student class adding necessary attributes (at least three subjects). Use these classes in a main function and display the average marks of computer science and mathematics students.

2. Define a shape class (with necessary constructors and member functions). Derive triangle and rectangle classes from shape class adding necessary attributes. Use these classes in main function and display the area of triangle and rectangle.

3. Make a book class with member variables name, price, author\_name and functions to get and show values of member variables. Make another class named grade with member variables level, 5 book objects, number\_of\_students, and get and show functions to get and show values of member variables. Also make main function to show uses of objects of class grade.

## Local Classes

Classes can also be defined and used inside a function or a block. Such classes are called local classes.

**Example**:

void test (int a)

{

………

………

class A

{

………

………

};

………

………

A a; **//create object of type A.**

………

}

Local classes can use global variables and static variables declared inside the function but cannot use automatic local variables. The global variables should be used with the scope resolution operator.