**Inheritance:**

**==>Inheritance is one of the most important features of Object-Oriented Programming.**

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

**==>Inheritance is a feature or a process in which, new classes are created from the existing classes.**

**==>The new class created is called "derived class" or "child class" and the existing class is known as the "base class” or “parent class”.**

**The derived class now is said to be inherited from the base class.**

**==>When we say derived class inherits the base class, it means, the derived class inherits all the properties of the base class, without changing the properties of base class it may add new features to its own. These new features in the derived class will not affect the base class.The derived class is the specialized class for the base class.**

**==>It not only helps to reuse the old code but adds in extending the functionality. i.e Reusability + Extensibility**

**Reusability:**

**Once a class is written and tested,it can be further used for creating new classes. These derived classes not only inherit the features of their base class,**

**but also have their own individualist features.**

**it means that if the derived class wants to use its base class properties,it can do so because those properties are also available to the derived class by the virtue of inheritance.**

**Now you can reuse the members of your parent class. So, there is no need to define the member again. So less code is required in the class.**

**Extensibility:**

**It is the mechanism of being able to derive classes from existing classes that provides extensibility of adding and removing classes in a hierarchy**

**as and when required.Any changes to data or functionalities contained within a base class are immediately inherited by all derived classes.**

**==>each child has "is a " relationship with its parent.**

**==>Sub Class: The class that inherits properties from another class is called Subclass or Derived Class.**

**==>Super Class: The class whose properties are inherited by a subclass is called Base Class or Superclass.**

**Syntax:**

**class <derived\_class\_name> : <access-specifier> <base\_class\_name>**

**{**

**//body**

**}**

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

**There are three Access specifiers in C++. These are:**

**Access Modifiers or Access Specifiers in a class are used to assign the accessibility to the class members.**

**Note: If we do not specify any access modifiers for the members inside the class, then by default the access modifier for the members will be Private.**

**==========================================================================**

**public – members are accessible from outside the class, and members can be accessed from anywhere. The data members and member functions declared as public can be accessed by other classes and functions too.The public members of a class can be accessed from anywhere in the program using the dot operator with the object of that class.**

**// C++ program to demonstrate public access modifier**

**#include<iostream>**

**using namespace std;**

**class Circle**

**{**

**public:**

**double radius;**

**double compute\_area()**

**{**

**return 3.14\*radius\*radius;**

**}**

**};**

**int main()**

**{**

**Circle obj;**

**// accessing public datamember outside class**

**obj.radius = 5.5;**

**cout << "Radius is: " << obj.radius << "\n";**

**cout << "Area is: " << obj.compute\_area();**

**return 0;**

**}**

**Output:**

**Radius is: 5.5**

**Area is: 94.985**

**In the above program, the data member radius is declared as public so it could be accessed outside the class and thus was allowed access from inside main().**

**===========================================================================private – members cannot be accessed (or viewed) from outside the class, i.e members are private to that class only.**

**The class members declared as private can be accessed only by the member functions inside the class. They are not allowed to be accessed directly by any**

**object or function outside the class. Only the member functions or the friend functions are allowed to access the private data members of the class.**

**// C++ program to demonstrate private access modifier**

**#include<iostream>**

**using namespace std;**

**class Circle**

**{**

**// private data member**

**private:**

**double radius;**

**// public member function**

**public:**

**double compute\_area()**

**{ // member function can access private**

**// data member radius**

**return 3.14\*radius\*radius;**

**}**

**};**

**int main()**

**{**

**Circle obj;**

**obj.radius = 1.5;**

**cout << "Area is:" << obj.compute\_area();**

**return 0;**

**}**

**The output of the above program is a compile time error because we are not allowed to access the private data members of a class directly from outside the class.**

**Yet an access to obj.radius is attempted, but radius being a private data member, we obtained the above compilation error.**

**However, we can access the private data members of a class indirectly using the public member functions of the class.**

**===========================================================================**

**How to make a Private Member Inheritable**

**The private member is not inheritable. If we modify the visibility mode by making it public, but this takes away the advantage of data hiding.**

**C++ introduces a third visibility modifier, i.e., protected. The member which is declared as protected will be accessible to all the member functions within the class as well as the class immediately derived from it.**

**protected – members cannot be accessed from outside the class, but, they can be accessed in inherited classes or derived classes.**

**The protected access modifier is similar to the private access modifier in the sense that it can’t be accessed outside of its class unless with the help of a friend class.**

**The difference is that the class members declared as Protected can be accessed by any subclass (derived class) of that class as well.**

**// C++ program to demonstrate protected access modifier**

**#include <iostream>**

**using namespace std;**

**class Parent**

**{**

**protected:**

**int id\_protected;**

**};**

**class Child : public Parent**

**{**

**public:**

**void setId(int id)**

**{**

**// Child class is able to access the inherited protected data members of base class**

**id\_protected = id;**

**}**

**void displayId()**

**{**

**cout << "id\_protected is: " << id\_protected << endl;**

**}**

**};**

**int main() {**

**Child obj1;**

**// member function of the derived class can access the protected data members of the base class**

**obj1.setId(81);**

**obj1.displayId();**

**Parent p1;**

**// p1.id\_protected=10;**

**// p1.show();**

**}**

**Output:**

**id\_protected is: 81**

**=============================================================================================**

**Note:**

**When derived class obj is created memory is allocated for base class datamembers as well as for derived class data members.**

**Through derived class obj, we can access base class members except private as well as derived class members.**

**#include <bits/stdc++.h>**

**using namespace std;**

**class Parent {**

**public:**

**int id\_p;**

**void set\_display1()**

**{**

**}**

**protected:**

**int id\_pro;**

**};**

**class Child : public Parent {**

**public:**

**int id\_c;**

**void set\_display()**

**{**

**id\_pro=10;**

**cout<<id\_pro;**

**}**

**};**

**int main()**

**{**

**Child obj1;**

**// An object of class child has all data members and member functions of class parent**

**obj1.id\_c = 7;**

**obj1.id\_p = 91;**

**// obj1.id\_pro=7;//ERROR**

**//THRU DERIVED CLASS OBJ WE CAN ACCESS BASE CVLASS AS WELL AS DERIVED CLASS DATAS MEMBERS**

**cout << "Child id is: " << obj1.id\_c << '\n';**

**cout << "Parent id is: " << obj1.id\_p << '\n';**

**obj1.set\_display1();**

**obj1.set\_display();**

**cout<<sizeof(obj1);**

**}**

**==============================================================**

**Note 2:**

**When base class obj is created memory is allocated for base class datamembers only. Through base class obj, we can access base class members only ,we cannot access derived class members.**

**#include <bits/stdc++.h>**

**using namespace std;**

**class Parent {**

**public:**

**int id\_p;**

**};**

**class Child : public Parent {**

**public:**

**int id\_c;**

**void display()**

**{**

**}**

**};**

**int main()**

**{**

**Parent obj1;**

**// An object of class child has all data members and member functions of class parent**

**// obj1.id\_c = 7;//error**

**obj1.id\_p = 91;**

**// cout << "Child id is: " << obj1.id\_c << '\n';//error**

**cout << "Parent id is: " << obj1.id\_p << '\n';**

**cout<<sizeof(obj1);**

**//obj1.display();**

**}**

**==============================================================================================**

**Modes of Inheritance: There are 3 modes of inheritance.**

**Modes of inheritance specifies the way in which a class is derived. It tells about the access rights given to the derived class to access its base class attributes and functions. These mode are:**

**private(default)**

**public**

**protected**

**ex:**

**class derived\_class\_name :: visibility-mode base\_class\_name**

**{**

**// body of the derived class.**

**}**

**Where,**

**derived\_class\_name: It is the name of the derived class.**

**visibility mode: The visibility mode specifies whether the features of the base class are publicly , protected inherited or privately inherited. It can be public or private.**

**Example:**

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

**{ }**

**2. class ABC : public XYZ //public derivation**

**{ }**

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

**{ }**

**4. class ABC: XYZ //private derivation by default**

**{ }**

**=======================================================================**

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

**the base class will become protected in the derived class.**

**1. C++ public Inheritance**

**In this example, public inheritance is demonstrated. Since private and protected members will not be directly accessed from main( )**

**so we have to create functions name getPVT( ) to access the private variable and getProt( ) to access the protected variable from the inherited class.**

**Example:**

**// C++ program to demonstrate the working of public inheritance**

**#include <iostream>**

**using namespace std;**

**class Base {**

**private:**

**int pvt = 1;**

**protected:**

**int prot = 2;**

**public:**

**int pub = 3;**

**// function to access private member**

**int getPVT() { return pvt; }**

**};**

**class PublicDerived : public Base {**

**public:**

**// function to access protected member from Base**

**int getProt() { return prot; }**

**};**

**int main()**

**{**

**PublicDerived object1;**

**cout << "Private = " << object1.getPVT() << endl;**

**cout << "Protected = " << object1.getProt() << endl;**

**cout << "Public = " << object1.pub << endl;**

**return 0;**

**}**

**Output**

**Private = 1**

**Protected = 2**

**Public = 3**

**==================================================================**

**Protected Mode: If we derive a subclass from a Protected base class. Then both public members and protected members of the base class will become protected in the derived class.**

**We know that protected members can only be accessed from the Derived class. These members cannot be directly accessed from outside the class. So we cannot use getPVT() from ProtectedDerived.This is also why we need to create getPub() function in the Derived class in order to access the pub variable.**

**Example:**

**// C++ program to demonstrate the working of protected inheritance**

**#include <iostream>**

**using namespace std;**

**class Base {**

**private:**

**int pvt = 1;**

**protected:**

**int prot = 2;**

**public:**

**int pub = 3;**

**// function to access private member**

**int getPVT() { return pvt; }**

**};**

**class ProtectedDerived : protected Base {**

**public:**

**// function to access protected member from Base**

**int getProt() { return prot; }**

**// function to access public member from Base**

**int getPub() { return pub; }**

**};**

**int main()**

**{**

**ProtectedDerived object1;**

**cout << "Private cannot be accessed." << endl;**

**cout << "Protected = " << object1.getProt() << endl;**

**cout << "Public = " << object1.getPub() << endl;**

**//        cout<<object1.getPVT() ;**

**//cout<<object1.pub;**

**return 0;**

**}**

**============================================================================**

**Private Mode: If we derive a subclass from a Private base class. Then both public members and protected members of the base class will become Private in the derived class.**

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

**#include <iostream>**

**using namespace std;**

**class Base {**

**private:**

**int pvt = 1;**

**protected:**

**int prot = 2;**

**public:**

**int pub = 3;**

**// function to access private member**

**int getPVT() { return pvt; }**

**// int getPub() { return pub; }**

**};**

**class PrivateDerived : private Base {**

**public:**

**// function to access protected member from Base**

**int getProt()**

**{**

**//prot=10;**

**return prot;**

**}**

**};**

**int main()**

**{**

**PrivateDerived object1;**

**cout << "Private cannot be accessed." << endl;**

**cout << "Protected = " << object1.getProt() << endl;**

**//cout << "Public = " << object1.getPub() << endl;**

**return 0;**

**}**

**===================================================================================================**

**When the base class is privately inherited by the derived class, public members of the base class becomes the private members of the derived class. Therefore, the public members of the base class are not accessible by the objects of the derived class only by the member functions of the derived class.**

**ex:**

**#include<iostream>**

**using namespace std;**

**class A**

**{**

**int c;**

**public:**

**int a;**

**protected:**

**int b;**

**public:**

**void display()**

**{**

**cout<<"in display of A\n";**

**}**

**};**

**class B:private A**

**{**

**public:**

**int c;**

**public:**

**void show()**

**{**

**display();//allowed**

**cout<<a<<b;**

**}**

**};**

**class C:private B**

**{**

**public:**

**void show()**

**{**

**// display(); not allowed**

**//cout<< a<<b; not allowed**

**cout<<c;**

**}**

**};**

**int main()**

**{**

**B bobj;**

**bobj.show();**

**//        bobj.display(); not allowed**

**}**

**In the above example, class A is privately inherited. Therefore, the display() function of class 'A' cannot be accessed by the object of class B. It can only be accessed by the member function of class B.**

**=========================================================================**

**====================================================================================**

**When the base class is publicly inherited by the derived class, public members of the base class also become the public members of the derived class. Therefore, the public members of the base class are accessible by the objects of the derived class as well as by the member functions of the derived class.**

**#include<iostream>**

**using namespace std;**

**class A**

**{**

**int c;**

**public:**

**int a;**

**protected:**

**int b;**

**public:**

**void display()**

**{**

**cout<<"in display of A\n";**

**}**

**};**

**class B:public A**

**{**

**public:**

**public:**

**void show()**

**{**

**display();**

**cout<<a<<b;**

**}**

**};**

**class C:public B**

**{**

**public:**

**void show()**

**{**

**cout<< a<<b;**

**}**

**};**

**int main()**

**{**

**B bobj;**

**bobj.show();**

**bobj.display();**

**}**

**=====================================================================**

**When the base class is protectedly inherited by the derived class, the public members of the base class are not accessible by the objects of the derived class**

**(because public members of base class becomes protected in derived class and protected datamember cannot be accessed outside the class) only by the member functions of the derived class.**

**#include<iostream>**

**using namespace std;**

**class A**

**{**

**int c;**

**public:**

**int a;**

**protected:**

**int b;**

**protected:**

**void display()**

**{**

**cout<<"in display of A\n";**

**}**

**};**

**class B:protected A**

**{**

**public:**

**int d;**

**public:**

**void show()**

**{**

**B bobj;**

**bobj.display();**

**cout<<a<<b;**

**}**

**};**

**int main()**

**{**

**B bobj;**

**//bobj.display();**

**}**

**==================================================================================**

**Types Of Inheritance**

**C++ supports five types of inheritance:**

**Single inheritance**

**Multiple inheritance**

**Hierarchical inheritance**

**Multilevel inheritance**

**Hybrid inheritance**

**C++ Single Inheritance**

**Single inheritance is defined as the inheritance in which a derived class is inherited from the only one base class.**

**.#include <iostream>**

**using namespace std;**

**class Account {**

**protected:**

**float bonus = 2000;**

**};**

**class Programmer: public Account {**

**float monthly\_salary = 5000;**

**public:**

**void CalculateSalary()**

**{**

**cout<<"Total Salary: "<<monthly\_salary+bonus<<endl;**

**}**

**};**

**int main() {**

**Programmer p1;**

**p1.CalculateSalary();**

**return 0;**

**}**

**=========================================================================**

**While creating an object of the derived class, the base class constructor is called first and then the derived class constructor is called. The base class constructor is responsible for initializing the inherited data members and the derived class constructor is responsible for initializing data members of derived class.**

**The member initializer list is used to indicate which base class constructor to call in the derived class constructor.otherwise , the no argument constructor of base class is used.**

**Derived object has two parts: a base part and a derived part.**

**The base part of derived object is constructed first. then the derived part is constructed. therefore the constructor calls are in the order of base->derived.**

**==========================================================**

**Why the base class’s constructor is called on creating an object of derived class?**

**What happens when a class is inherited from other? The data members and member functions of base class comes automatically in derived class based on the access specifier but the definition of these members exists in base class only. So when we create an object of derived class, all of the members of derived class must be initialized but the inherited members in derived class can only be initialized by the base class’s constructor as the definition of these members exists in base class only. This is why the constructor of base class is called first to initialize all the inherited members.**

**/\*#include<iostream>**

**using namespace std;**

**class A**

**{**

**int a;**

**public:**

**A();**

**void display();**

**};**

**A::A()**

**{**

**cout<<"in default of A\n";**

**a=10;**

**}**

**void A::display()**

**{**

**cout<<a<<endl;**

**}**

**class B:public A**

**{**

**int b;**

**public:**

**B();**

**void display();**

**};**

**B::B()**

**{**

**cout<<"in default of B\n";**

**b=20;**

**}**

**void B::display()**

**{**

**A::display();**

**cout<<b<<endl;**

**}**

**int main()**

**{**

**B bobj;**

**bobj.display();**

**}**

**==========================================================**

**#include<iostream>**

**using namespace std;**

**class A**

**{**

**int a;**

**public:**

**A(int);**

**void display();**

**};**

**A::A(int p)**

**{**

**cout<<"in para of A\n";**

**a=p;**

**}**

**void A::display()**

**{**

**cout<<a<<endl;**

**}**

**class B:public A**

**{**

**int b;**

**public:**

**B(int,int);**

**void display();**

**};**

**B::B(int p,int q):A(p)//base class initilization list**

**{**

**cout<<"in para of B\n";**

**b=q;**

**}**

**void B::display()**

**{**

**A::display();**

**cout<<b<<endl;**

**}**

**int main()**

**{**

**B bobj(10,20);**

**bobj.display();**

**}**

**===========================================================**

**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 the parameterized 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.**

**=============================================================================================**

**Multilevel Inheritance**

**Multilevel Inheritance in C++ is the process of deriving a class from another derived class. When one class inherits another class it is further inherited by another class. It is known as multi-level inheritance.**

**For example, if we take Grandfather as a base class then Father is the derived class that has features of Grandfather and then Child is the also derived class that is derived from the sub-class Father which inherits all the features of Father.**

**class A // base class**

**{**

**...........**

**};**

**class B : access\_specifier A // derived class**

**{**

**...........**

**} ;**

**class C : access\_specifier B // derived from derived class B**

**{**

**...........**

**} ;**

**// C++ program to implement constructor in multilevel Inheritance**

**#include<iostream>**

**using namespace std;**

**// Base class**

**class A**

**{**

**public:**

**A()**

**{**

**cout << "Base class A constructor \n";**

**}**

**};**

**// Derived class B**

**class B: public A**

**{**

**public:**

**B()**

**{**

**cout << "Class B constructor \n";**

**}**

**};**

**// Derived class C**

**class C: public B**

**{**

**public:**

**C()**

**{**

**cout << "Class C constructor \n";**

**}**

**};**

**// Driver code**

**int main()**

**{**

**C obj;**

**return 0;**

**}**

**==================================================================**

**class employee**

**{**

**int id;**

**public:**

**employee();**

**employee(int);**

**void display();**

**int findsalary()**

**{**

**return 0;**

**}**

**};**

**employee::employee()**

**{**

**cout<<"in default of emp\n";**

**id=0;**

**}**

**employee::employee(int i)**

**{**

**cout<<"in para of emp\n";**

**id=i;**

**}**

**void employee::display()**

**{**

**cout<<"id of an emp is "<<id<<endl;**

**}**

**class wageemployee:public employee**

**{**

**int hrs,rate;**

**public:**

**wageemployee();**

**wageemployee(int,int,int);**

**void display();**

**int findsalary();**

**};**

**wageemployee::wageemployee()**

**{**

**cout<<"in default of wage\n";**

**hrs=0;**

**rate=0;**

**}**

**wageemployee::wageemployee(int i,int h,int r)**

**:employee(i)**

**{**

**cout<<"in para of wage\n";**

**hrs=h;**

**rate=r;**

**}**

**int wageemployee::findsalary()**

**{**

**return hrs \* rate;**

**}**

**void wageemployee::display()**

**{**

**employee::display();**

**cout<<hrs<<endl;**

**cout<<rate<<endl;**

**}**

**class salesmanager:public wageemployee**

**{**

**int sales,comm;**

**public:**

**salesmanager();**

**salesmanager(int,int,int,int,int);**

**void display();**

**int findsalary();**

**void show();**

**};**

**salesmanager::salesmanager()**

**{**

**cout<<"in default of sales\n";**

**sales=comm=0;**

**}**

**salesmanager::salesmanager(int i,int h,int r,int s,**

**int c):wageemployee(i,h,r)**

**{**

**cout<<"in para of sales\n";**

**sales=s;**

**comm=c;**

**}**

**void salesmanager::display()**

**{**

**wageemployee::display();**

**cout<<"sales of an emp is "<<sales<<endl;**

**cout<<"comm of an emp is "<<comm<<endl;**

**}**

**int salesmanager::findsalary()**

**{**

**return wageemployee::findsalary() + sales \* comm;**

**}**

**void salesmanager::show()**

**{**

**cout<<"in show fun\n";**

**}**

**int main()**

**{**

**salesmanager \*ptr=new salesmanager();**

**cout<<"salary is "<<ptr->findsalary();**

**ptr->display();**

**}**

**=================================================================**

**============================================================================================**

**Function Overriding**

**1)two or more function having same function name and same signature and same return type is called as function overriding**

**2)overrided function must be present in inherited classes i.e scopes are different**

**==============================================================**

**Multiple Inheritance in C++:**

**1 Multiple Inheritance is a feature of C++ where a class can inherit from more than one classes.**

**2 The constructors of inherited classes are called in the same order in which they are inherited.**

**==>The list of base classes mentioned in the derived class declaration must be seperated by commas and each of the base classes should use its own specifier . If the class specifier is not specified, default is private.**

**==>The order in which base class constructors are invoked is the same as the order in which they appear in the derived class constructor's member initialization list.**

**Syntax:**

**class A**

**{**

**... .. ...**

**};**

**class B**

**{**

**... .. ...**

**};**

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

**{**

**... ... ...**

**};**

**=======================================================================**

**#include<iostream>**

**using namespace std;**

**class A**

**{**

**public:**

**A() { cout << "A's constructor called" << endl; }**

**~A() { cout << "A's constructor called" << endl; }**

**};**

**class B**

**{**

**public:**

**B() { cout << "B's constructor called" << endl; }**

**~B() { cout << "B's constructor called" << endl; }**

**};**

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

**{**

**public:**

**C() { cout << "C's constructor called" << endl; }**

**~C() { cout << "C's constructor called" << endl; }**

**};**

**int main()**

**{**

**C c;**

**return 0;**

**}**

**Output:**

**B's constructor called**

**A's constructor called**

**C's constructor called**

**The destructors are called in reverse order of constructors.**

**=======================================================================**

**Ambiguity Problem in Multiple Inheritance**

**In Multiple Inheritance, when a single class is derived from two or more base or parent classes. So, it might be possible that both the parent class have the same named member functions, and it shows ambiguity when the child class object invokes one of the same-named member functions. Hence, we can say, the C++ compiler is confused in selecting the member function of a class for the execution of a program.**

**ex:**

**Base classes can contain a function with the same name,signature and returntype.**

**class A**

**{**

**func()**

**};**

**class C**

**{**

**... .. ...**

**};**

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

**{**

**};**

**B bobj**

**bobj.func();**

**==>Here,the function call is ambiguous and doesnot compile,because it is not clear whether it refers to A::func() or C::func().This ambiguity can be resolved by any one of the following 2 ways.**

**1)Making the call explicit by resolving the function call with the classname i.e specifying either bobj.A::func() or bobj.c::func()**

**2)overriding the func() in class B . then the statement bobj.func() will call func() of class B and the call will not be ambiguous.**

**#include <iostream>**

**using namespace std;**

**// create class A**

**class A**

**{**

**public:**

**void show()**

**{**

**cout << " It is the member function of class A " << endl;**

**}**

**};**

**// create class B**

**class B**

**{**

**public:**

**void show()**

**{**

**cout << " It is the member function of class B " << endl;**

**}**

**};**

**// create a child class to inherit the member function of class A and class B**

**class child: public A, public B**

**{**

**public:**

**void disp()**

**{**

**cout << " It is the member function of the child class " << endl;**

**}**

**};**

**int main ()**

**{**

**// create an object of the child class to access the member function**

**child ch;**

**ch.show(); // It causes ambiguity**

**ch.disp();**

**return 0;**

**}**

**==================================================================**

**Diamond Inheritance:**

**In diamond inheritance,one child has two parents and both the parents share a common parent. When in a hierarchy, a class is derived from two classes and both the classes are derived from a common base class,it is termed as diamond inheritance.**

**The diamond problem The diamond problem occurs when two superclasses of a class have a common base class.**

**#include<iostream>**

**using namespace std;**

**class Person {**

**// Data members of person**

**public:**

**Person(int x) { cout << "Person::Person(int ) called" << endl; }**

**};**

**class Faculty : public Person {**

**// data members of Faculty**

**public:**

**Faculty(int x):Person(x) {**

**cout<<"Faculty::Faculty(int ) called"<< endl;**

**}**

**};**

**class Student : public Person {**

**// data members of Student**

**public:**

**Student(int x):Person(x) {**

**cout<<"Student::Student(int ) called"<< endl;**

**}**

**};**

**class TA : public Faculty, public Student {**

**public:**

**TA(int x):Student(x), Faculty(x) {**

**cout<<"TA::TA(int ) called"<< endl;**

**}**

**};**

**int main() {**

**TA ta1(30);**

**}**

**Person::Person(int ) called**

**Faculty::Faculty(int ) called**

**Person::Person(int ) called**

**Student::Student(int ) called**

**TA::TA(int ) called**

**In the above program, constructor of ‘Person’ is called two times. Destructor of ‘Person’ will also be called two times when object ‘ta1’ is destructed. So object ‘ta1’ has two copies of all members of ‘Person’, this causes ambiguities. The solution to this problem is ‘virtual’ keyword. We make the classes ‘Faculty’ and ‘Student’ as virtual base classes to avoid two copies of ‘Person’ in ‘TA’ class.**

**==========================================================================================**

**#include<iostream>**

**using namespace std;**

**class Person {**

**public:**

**Person(int x) { cout << "Person::Person(int ) called" << endl; }**

**Person() { cout << "Person::Person() called" << endl; }**

**};**

**class Faculty : virtual public Person {**

**public:**

**Faculty(int x):Person(x) {**

**cout<<"Faculty::Faculty(int ) called"<< endl;**

**}**

**};**

**class Student : virtual public Person {**

**public:**

**Student(int x):Person(x) {**

**cout<<"Student::Student(int ) called"<< endl;**

**}**

**};**

**class TA : public Faculty, public Student {**

**public:**

**TA(int x):Student(x), Faculty(x) {**

**cout<<"TA::TA(int ) called"<< endl;**

**}**

**};**

**int main() {**

**TA ta1(30);**

**}**

**Output:**

**Person::Person() called**

**Faculty::Faculty(int ) called**

**Student::Student(int ) called**

**TA::TA(int ) called**

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

**=======================================================================================**

**How to call the parameterized constructor of the ‘Person’ class?**

**The constructor has to be called in ‘TA’ class.**

**For example, see the following program.**

**#include<iostream>**

**using namespace std;**

**class Person {**

**public:**

**Person(int x) { cout << "Person::Person(int ) called" << endl; }**

**Person() { cout << "Person::Person() called" << endl; }**

**};**

**class Faculty : virtual public Person {**

**public:**

**Faculty(int x):Person(x) {**

**cout<<"Faculty::Faculty(int ) called"<< endl;**

**}**

**};**

**class Student : virtual public Person {**

**public:**

**Student(int x):Person(x) {**

**cout<<"Student::Student(int ) called"<< endl;**

**}**

**};**

**class TA : public Faculty, public Student {**

**public:**

**TA(int x):Student(x), Faculty(x), Person(x) {**

**cout<<"TA::TA(int ) called"<< endl;**

**}**

**};**

**int main() {**

**TA ta1(30);**

**}**

**Output:**

**Person::Person(int ) called**

**Faculty::Faculty(int ) called**

**Student::Student(int ) called**

**TA::TA(int ) called**

**In general, it is not allowed to call the grandparent’s constructor directly, it has to be called through parent class. It is allowed only when ‘virtual’ keyword is used.**

**=====================================================================================**

**FAQ:**

**class S: public A1, virtual A2**

**{**

**….**

**};**

**Here,**

**A2(): virtual base constructor**

**A1(): base constructor**

**S(): derived constructor**

**Example 1: Below is the C++ program to show the concept of Constructor in Multiple Inheritance.**

**// C++ program to implement**

**// constructor in multiple**

**// inheritance**

**#include<iostream>**

**using namespace std;**

**class A1**

**{**

**public:**

**A1()**

**{**

**cout << "Constructor of the base class A1 \n";**

**}**

**};**

**class A2**

**{**

**public:**

**A2()**

**{**

**cout << "Constructor of the base class A2 \n";**

**}**

**};**

**class S: public A1, virtual A2**

**{**

**public:**

**S(): A1(), A2()**

**{**

**cout << "Constructor of the derived class S \n";**

**}**

**};**

**// Driver code**

**int main()**

**{**

**S obj;**

**return 0;**

**}**

**Output**

**Constructor of the base class A2**

**Constructor of the base class A1**

**Constructor of the derived class S**

**====================================================================**

**FAQ ON MULTIPLE INHERITANCE:**

**1)**

**#include<iostream>**

**using namespace std;**

**class Base1 {**

**public:**

**Base1()**

**{ cout << " Base1's constructor called" << endl; }**

**};**

**class Base2 {**

**public:**

**Base2()**

**{ cout << "Base2's constructor called" << endl; }**

**};**

**class Derived: public Base1, public Base2 {**

**public:**

**Derived()**

**{ cout << "Derived's constructor called" << endl; }**

**};**

**int main()**

**{**

**Derived d;**

**return 0;**

**}**

**(A) Compiler Dependent**

**(B) Base1′s constructor called**

**Base2′s constructor called**

**Derived’s constructor called**

**(C) Base2′s constructor called**

**Base1′s constructor called**

**Derived’s constructor called**

**(D) Compiler Error**

**Answer: (B)**

**Explanation: When a class inherits from multiple classes, constructors of base classes are called in the same order as they are specified in inheritance.**

**===================================================================================================**

**2)**

**#include <iostream>**

**using namespace std;**

**class Base1 {**

**public:**

**~Base1() { cout << " Base1's destructor" << endl; }**

**};**

**class Base2 {**

**public:**

**~Base2() { cout << " Base2's destructor" << endl; }**

**};**

**class Derived: public Base1, public Base2 {**

**public:**

**~Derived() { cout << " Derived's destructor" << endl; }**

**};**

**int main()**

**{**

**Derived d;**

**return 0;**

**}**

**(A)**

**Base1's destructor**

**Base2's destructor**

**Derived's destructor**

**(B)**

**Derived's destructor**

**Base2's destructor**

**Base1's destructor**

**(C)**

**Derived's destructor**

**(D) Compiler Dependent**

**Answer: (B)**

**Explanation: Destructors are always called in reverse order of constructors.**

**==================================================================================**

**3)**

**#include<iostream>**

**using namespace std;**

**class P {**

**public:**

**void print() { cout <<" Inside P"; }**

**};**

**class Q : public P {**

**public:**

**void print() { cout <<" Inside Q"; }**

**};**

**class R: public Q { };**

**int main(void)**

**{**

**R r;**

**r.print();**

**return 0;**

**}**

**(A) Inside P**

**(B) Inside Q**

**(C) Compiler Error: Ambiguous call to print()**

**Answer: (B)**

**Explanation: The print function is not present in class R. So it is looked up in the inheritance hierarchy. print() is present in both classes P and Q, which of them should be called? The idea is, if there is multilevel inheritance, then function is linearly searched up in the inheritance hierarchy until a matching function is found.**

**======================================================================================**

**Virtual base class in C++**

**Virtual base classes are used in virtual inheritance in a way of preventing multiple “instances” of a given class appearing in an inheritance hierarchy when using multiple inheritances.**

**Need for Virtual Base Classes: Consider the situation where we have one class A . This class A is inherited by two other classes B and C. Both these class are inherited into another in a new class D**

**data members/function of class A are inherited twice to class D. One through class B and second through class C. When any data / function member of class A is accessed by an object of class D, ambiguity arises as to which data/function member would be called? One inherited through B or the other inherited through C. This confuses compiler and it displays error.**

**Example: To show the need of Virtual Base Class in C++**

**#include <iostream>**

**using namespace std;**

**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 object;**

**object.show();**

**}**

**Compile Errors:**

**prog.cpp: In function 'int main()':**

**prog.cpp:29:9: error: request for member 'show' is ambiguous**

**object.show();**

**^**

**prog.cpp:8:8: note: candidates are: void A::show()**

**void show()**

**^**

**prog.cpp:8:8: note: void A::show()**

**How to resolve this issue?**

**To resolve this ambiguity when class A is inherited in both class B and class C, it is declared as virtual base class by placing a keyword virtual as :**

**Syntax for Virtual Base Classes:**

**Syntax 1:**

**class B : virtual public A**

**{**

**};**

**Syntax 2:**

**class C : public virtual A**

**{**

**};**

**Note:**

**virtual can be written before or after the public. Now only one copy of data/function member will be copied to class C and class B and class A becomes the virtual base class. Virtual base classes offer a way to save space and avoid ambiguities in class hierarchies that use multiple inheritances. When a base class is specified as a virtual base, it can act as an indirect base more than once without duplication of its data members. A single copy of its data members is shared by all the base classes that use virtual base.**

**#include <iostream>**

**using namespace std;**

**class A {**

**public:**

**int a;**

**A() // constructor**

**{**

**a = 10;**

**}**

**};**

**class B : public virtual A {**

**};**

**class C : public virtual A {**

**};**

**class D : public B, public C {**

**};**

**int main()**

**{**

**D object; // object creation of class d**

**cout << "a = " << object.a << endl;**

**return 0;**

**}**

**Output**

**a = 10**

**Explanation :**

**To solve this ambiguity we will make class “A” as a virtual base class. To make a virtual base class “virtual” keyword is used.**

**When one class is made virtual then only one copy of its data member and member function is passed to the classes inheriting it. So in our example when we will make class “A” a virtual class then only one copy of the data member and member function will be passed to the classes “B” and “C” which will be shared between all classes. This will help to solve the 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.**

**NOTE:**

**In this case, we are using a virtual base class in C++, so only one copy of data from Class A was inherited to Class D; hence, the compiler will be able to print the output.**

**When we mention the base class as virtual, we avoid the situation of duplication and let the derived classes get only one copy of the data.**

**There are a few details that one needs to remember.**

**1)Virtual base classes are always created before non-virtual base classes. This ensures all bases are created before their derived classes.**

**2)Note that classes B and C still have calls to class A, but they are simply ignored when creating an object of class D. If we are creating an object of class B or C, then the constructor of A will be called.**

**3)If a class inherits one or more classes with virtual parents, the most derived class is responsible for constructing the virtual base class. Here, class D is responsible for creating class A object.**

**========================================================================================**

**book:**

**problem 1:**

**If the base class contain a function with same name then while calling this function in the derived class or through an object of derived class, the compiler flashes an error of ambiguous call.**

**problem 2:**

**data duplication occurs when the derived class has multiple copies of the same base class.**

**==>To avoid 2 major problems in case of diamond inheritance,c++ introduces a concept of a virtual base class. A virtual base class is a class that is virtually present in its derived classes(i.e in the form of pointer).**

**==>A base class is made virtual by placing the keyword virtual before its name in the derived class declaration.**

**==Virtual base classes offer a way to save spacce and aoid ambiguities in class hierarchies that use diamond inheritance.**

**==>When a base class is specified as virtual,it can act as an indirect base class more than once without duplicating its data members.A single copy of its data members is shared by all the base classes that use it as virtual base.**

**Employee: id**

**Manager:petrolallow,foodallow,virtual base pointer**

**Salesperson:sales,comm,virtual base pointer**

**Salesmanager:petrolallow,foodallow,virtual base pointer + sales,comm,virtual base pointer**

**class Salesperson:virtual public Employee**

**{}**

**class Manager:virtual public Employee**

**{}**

**Class Employee becomes a virtual base class.The data members of the virtual base class are not directly present in the derived class. Instead, the derived class Salesperson and Manager acquire a pointer each. This pointer is called virtual base pointer.It causes the size of their objects to increase by 4 bytes each. This virtual base pointer points to common data members of the virtual base class.**

**Constructors in multiple inheritance:**

**If the virtual base class has constructors, then the most derived class constructor is responsible for initializing the virtual base class.**

**==>Employee is the virtual base class.The most derived class constructor is responsible for invoking the Employee constructor.**

**ex:SalesManager constructor must be as follow:**

**SalesManager::SalesManager(--):Employee(--),Salesperson(-),Manager(-)**

**{**

**}**

**Now on instantiating an object sm1 of class Salesmanager**

**1)memory is alloacted for the salesmanager object**

**2)constructor of the class salesmanager is invoked**

**3)constructor of the class Employee is invoked and executed directly**

**4)constructor of the class that appears first when the salesmanager class is derived(i.e salesperson) is invoked and executed.**

**5)constructor of the class that appears next is invoked and executed**

**6)constructor of the class salesmanager is executed.**

**==>As Employee class is made a virtual base class, the code that was written previously for accept(),where the compiler was giving error as ambiguous call for the statement Employee:accept() will work fine.**

**=====================================================================================**