OOPS

- **It is a programming methodology to organize complex program in to simple program in terms of classes and object such methodology is called oops.**
- **It is a programming methodology to organized complex program into simple program by using concept of abstraction , encapsulation , polymorphism and inheritance.**
- **Languages which support abstraction , encapsulation polymorphism and inheritance are called oop language.**
- **4 major pillars of oops**
  - **Abstraction**
  - **Encapsulation**
  - **Modularity and Hierarchy**

- **Minor pillars of oops**
  - **Polymorphism**
  - **Concurrency**
  - **Persistence**
- **Abstraction : getting only essential things and hiding unnecessary details is called as abstraction.**
- **Abstraction always describe outer behavior of object.**
- **In console application when we give call to function in to the main function it represents the abstraction**

- **Encapsulation :binding of data and code together is called as encapsulation. Implementation of abstraction is called encapsulation.**
- **Encapsulation always describe inner behavior of object.**
- **Function call is abstraction**
- **Function definition is encapsulation.**
- **Abstraction always changes from user to user.**

- Information hiding
- Data : unprocessed raw material is called as data.
- Process data is called as information.
- Hiding information from user is called information hiding.
- In c++ we used access specifier to provide information hiding.

- Modularity
- Dividing programs into small modules for the purpose of simplicity is called modularity.
- There are two types of modularity
  - Physical modularity
  - Logical modularity
- Physical modularity
  - Dividing a classes into multiple files is nothing but physical modularity
- Logical modularity
  - Dividing classes into namespaces is called logical modularity

- **Polymorphism (Typing)**
- **One interface having multiple forms is called as polymorphism.**
- **Polymorphism have two types**
- **Compile time polymorphism and runtime polymorphism.**

Compile time	**Run time**
Static polymorphism	**Dynamic polymorphism**
Static binding	**Dynamic binding**
Early binding	**Late binding**
Weak typing	**Strong typing**
False polymorphism	**True polymorphism**

- **Compile time polymorphism: when the call to the function resolved at compile time it is called as compile time polymorphism. And it is achieved by using function overloading and operator overloading**
- **Run time polymorphism : when the call to the function resolved at run time it is called as run time polymorphism. And it is achieved by using function overriding.**

- **Hierarchy**
- **Order/level of abstraction is called as hierarchy.**
- **Types of Hierarchy**
  - **has a Hierarchy (Composition) / has a relationship**
  - **is a Hierarchy (inheritance) / is a relationship**
  - **use a Hierarchy (dependency) / use a relationship**

Composition : when object is made from other small-small objects it is called as composition.
When object is composed of other objects it is called as composition
eg: room has a wall
room has a chair
system unit has motherboard
system unit has modem.
Types of composition :
1. Association 2. Aggregation 3. Containment

1. Association
Removal of small object do not affect big object it is called as association
eg. Room has chair . Association is having "loose coupling"

2. Aggregation :
Removal of small object affects big object it is called as Aggregation .
eg. Room has wall Aggregation is having "tight coupling"

3. Containment : stack , queue, linked list, array , vectors these are collectively called collections.
When class contain object of collection it is called as Containment.
Eg: room has number of chairs
company has number of employees.
In composition we always declared object of one class as a data member of another class.

```
class Date
{
        int dd;    int mm;        int yy;
};
class Address
{
      char  addressinfo[20];   char city[20]; int pincode;
};
class Person
{
      char name[30];   Date  birthdate;        Address address;
};
```

```
#include<iostream>
#include<string.h>
class Date
{
          int dd;  int mm;  int yy;
    public:
          Date(){
                this->dd=1;
                this->mm=1;
                this→yy=1900;
             }
             Date(int dd,int mm,int yy){
                   this->dd=dd;
                   this->mm=mm;
                   this->yy=yy;
             }
             void Display()
             {
               cout<<this->dd<<"/"<<this->mm<<"/"<<this->yy<<endl;
             }
             };
```

```
class Address
{           char addressinfo[20];
            char city[20];
            int pincode;
public:
          Address(){
            strcpy(this->addressinfo,"");
             strcpy(this->city,"");
             this->pincode=0;
            }
            Address(char* ad,char* ct,int pin)
            {       strcpy(this->addressinfo,ad);
                    strcpy(this->city,ct);
                    this->pincode=pin;
             }
             void Display()
             {      cout<<"address :"<<this->addressinfo<<endl;
                    cout<<"city :"<<this->city<<endl;
                    cout<<"Pin Code :"<<this->pincode<<endl;
             }
};
```

```
class Person
{         char _name[30];   Date  birthdate;  Address address;
                    public:
          Person() {
                    strcpy(this->name,"");
          }
          Person(char* nm,int dd,int mm,int yy,char*ad,char* ct,int pin)
                              :birthdate(dd,mm,yy),address(ad,ct,pin) {
                                        strcpy(this->name,nm);
          }
          void Display(){
                    cout<<"Name :"<<this->name<<endl;
                    birthdate.Display();
                    address.Display();
          }
};
void main()
{         Person p("ABC",11,1,1975,"markeyard","pune",12345);
                    p.Display();
}
```

Inheritance: acquiring all properties(all data members) and behavior of one class (base class) by another class (derived class) this concept is called as inheritance.
Ex: class Employee : public Person (Is-a repletionship)
Derived Class : public Base Class

at the time of inheritance when name of members of base class and name of members of derived class are same at that time explicitly mention scope resolution operator is a job of programmer.

When you create a object of derived class at that time constructor of base class will call first and then constructor of derived class will called. Destructor calling sequence is exactly opposite that is destructor of derived class will called first and then the destructor of base class will called.

At the time of inheritance all the data members and member functions of base class are inherited into derived class but there are some functions which are not inherited into derived class.
1. constructor 2. copy constructor 3. destructor 4. friend function
5. assignment operator function

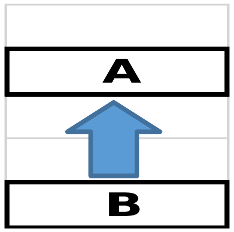
When we create a object of derived class at that time size of that object is size of all non static data members declared in base class plus size of all non static data members of declared in derived class.

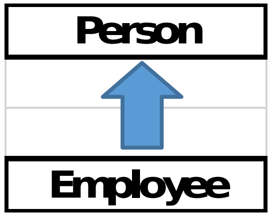
```
#include<iostream>
class A
{
                  int a;
                  public:
                  A() {
                            this->a=10;
                   }
                  A(int a) {
                               this->a=a;
                  }
                  void Print()  {
                           cout<< "a ::"<<this->a<<endl;
                  }
};
```

```
class B: public A
{
        int b;
        public:
          B() {
               this->b=20;
          }
          B(int b) {
               this->b=b;
          }
          void Print(){
                    A::Print();
                    cout<< "b ::"<<this->b<<endl;
          }
};
int main()
{
        B obj;
        obj.Print();
        cout<<"size of b ::"<<sizeof(obj)<<endl;
        A obj1;
        obj1.Print();
        cout<<"size of a ::"<<sizeof(obj1)<<endl;
        return 0;
        }
```

Types of inheritance:

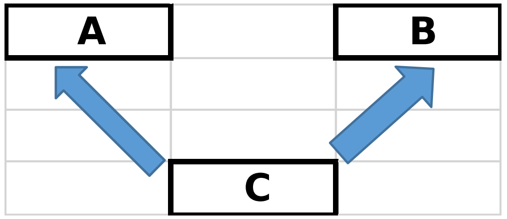
1.Single Inheritance: B is derived from A.



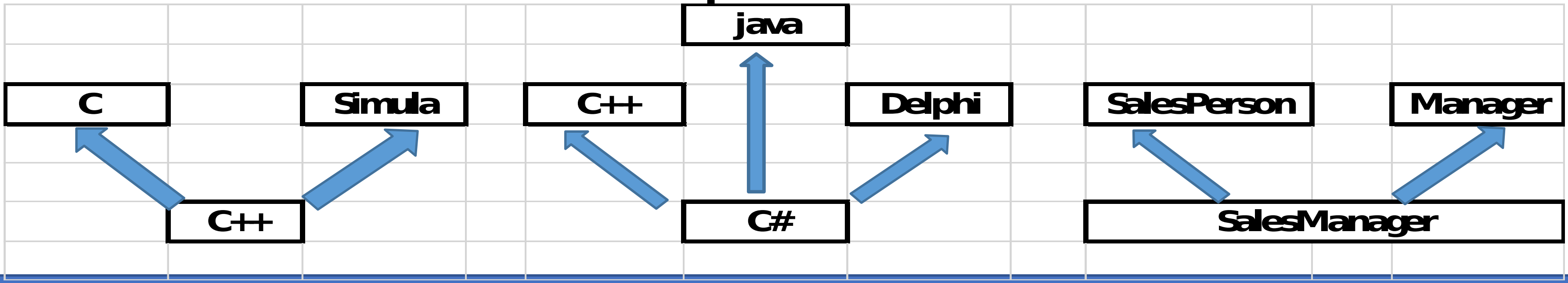


one base class having only one derived class. such inheritance is called as single inheritance. Eg Employee is a person

2.2. Multiple Inheritance:
C is derived from A and B.

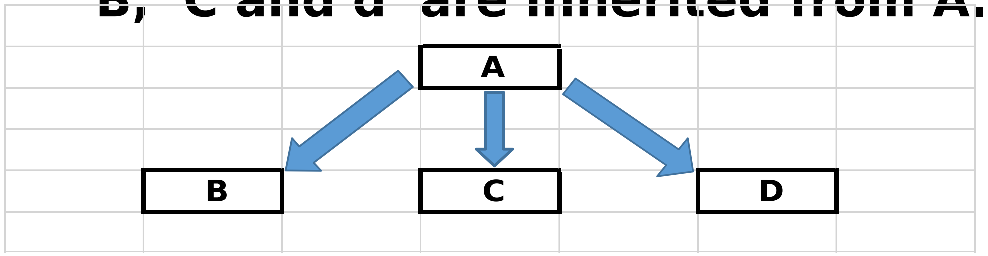


multiple base classes having only one derived class such type of inheritance is called as multiple inheritance

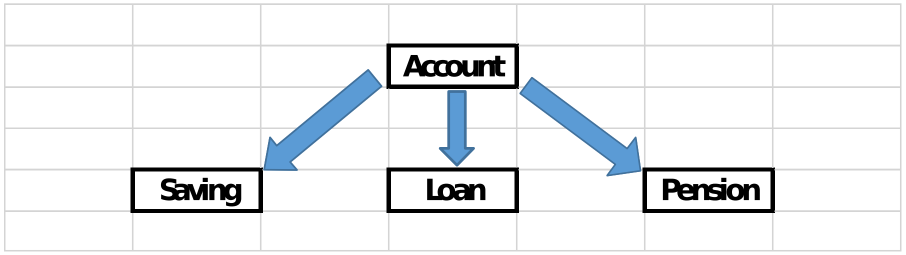


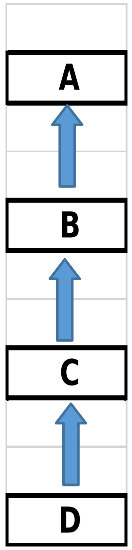
3. hierarchical inheritance:

B, C and d are inherited from A.



one Base class having multiple derived classes such type of inheritance is called hierarchical inheritance.





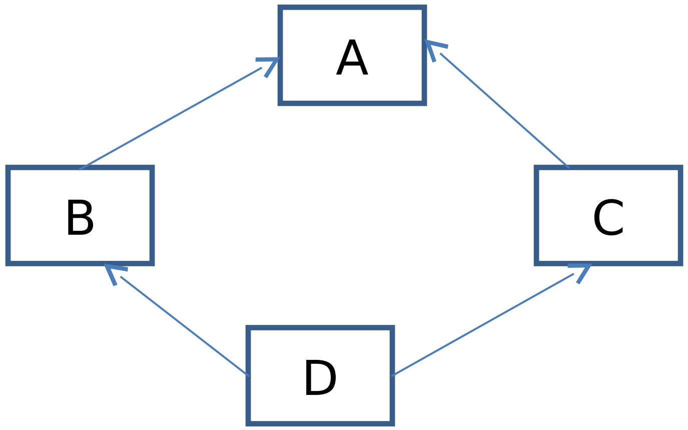
4. Multilevel inheritance :

B is derived from A , C is derived from B and D is derived from C.

When single inheritance has multiple levels is called as multilevel inheritance.

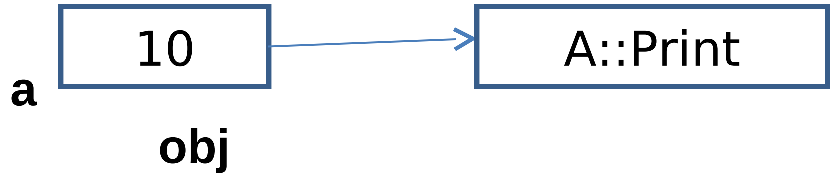
Hybrid Inheritance :
Combination of all types of inheritance is called as hybrid inheritance.
Diamond Problem:
What do u mean by diamond problem.?
What do u mean by virtual base class ?
What do u mean by virtual inheritance ?

Constructor calling sequence A->B-> A->C->D
destructor calling sequence D->C->A->B->A



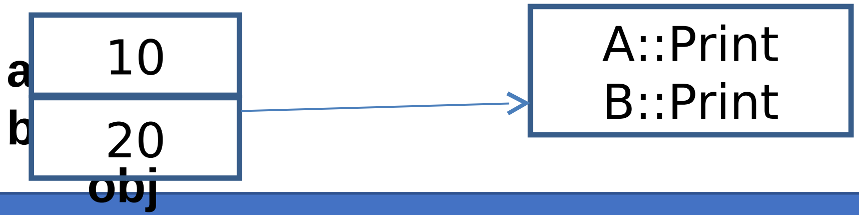
- Class A is direct base class for class B and C.
- Class B and class C are direct base classes for class D
- Class A is indirect base class for class D

- when we create object for class A it will get single copy of all the data members declared in class.
- A obj;

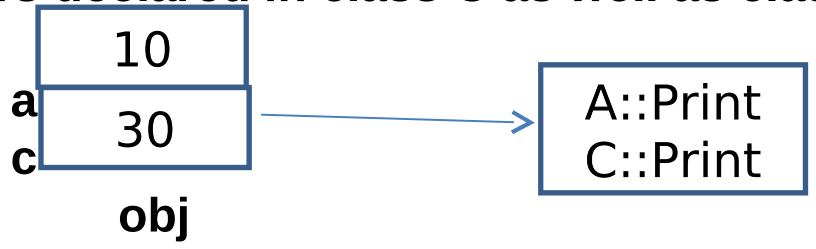


- A having only one member function. A:: Print()

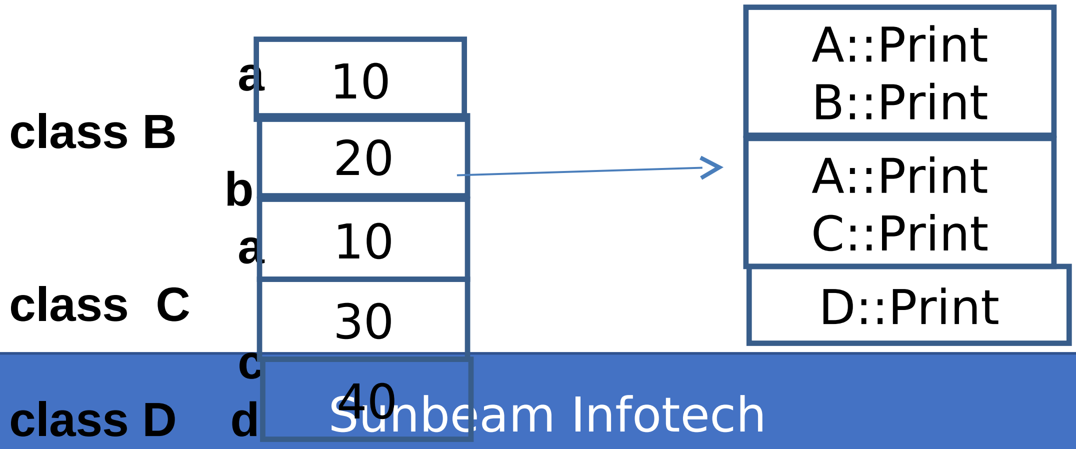
- Class B is derived from class A.
- When we cerate a object of class B. it will get single copy of all the non static data members declared in class B as well as class A



•Class C is derived from class A.
•When we cerate a object of class c. it will get single copy of all the non static data members declared in class C as well as class A



•Class D is derived class which is derived from class B and class C. This is called as multiple inheritance. When we crate a object of class D. it will get a single copy of all the data members of data members of declared in class B as well as class C and class D.



In above diagram class A is indirect base class for class D. that why all the data members and member functions of class A are available twice in class D.

When we try to access these members of class A by using object of class D Compiler will confuse. (which members to be access members available from class B and class C)
Such problem created by hybrid inheritance is called diamond problem.

Solution for hybrid inheritance :
1.Explicitly mention name of the class which of which data member and member function do u want to access.
•D obj;
•cout<<"BA::"<< obj.B::a <<endl;
•cout<<"CA::"<< obj.C::a <<endl;
•obj.print();

•Declared base class as a virtual i.e. derived class B from class A virtually and derived class C from class A virtually
•class B : virtual public A
•class C : virtual public A
•Such type of inheritance is called as virtual inheritance.
•Now in this case (class D)will get single copy of all the data members of indirect base class .

Mode of inheritance

when we use private ,public , protected at the time of inheritance it is called as mode of inheritance.

class B : public A

here is mode inheritance is public.

In c++ by default mode of inheritance is private.

Mode of inheritance public:

	Base (same class)	Derived	Indriect derived class	Out side class
Private	A	NA	NA	NA
Protected	A	A	A	NA
Public	A	A	A	A

A means Accessible

NA means Not Accessible

•In private mode of inheritance,

•public member of base becomes private members of derived.

•protected members of base become private members of derived.

•private members of base become private members of derived [not accessible in derived].

	Base (Same class)	Derived	Indirect derived class	Out side class
Private	A	NA	NA	NA
Protected	A	A	NA	NA
Public	A	A	NA	A using base class object NA using Derived class Object

In protected mode of inheritance,

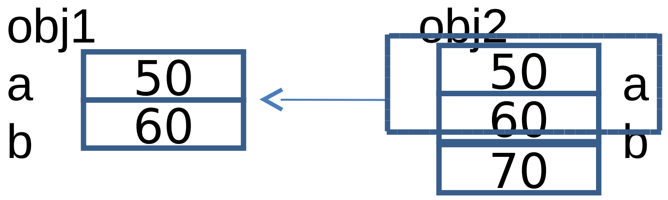
public member of base becomes protected members of derived.
protected members of base become protected members of derived.
private members of base become private members of derived [not accessible in derived].

	Base	Derived	Indirect derievd class	Out side class
Private	A	NA	NA	NA
Protected	A	A	A	NA
Public	A	A	A	A using base class object NA using Derived class Object

Object slicing : when we assign derived class object to the base class object at that time base class portion which is available in derived class object is assign to the base class object. Such slicing (cutting) of base class portion from derived class object is called object slicing.

A obj1;

B obj2(50, 60, 70);



Up casting : Storing address of derived class object into bas class pointer . Such concept is called as up casting.

A * ptr ; B obj; ptr= &obj; or A*ptr=new B();

Down casting : storing address of base class object into derived class pointer is called as downcasting.

B *ptr=new A();

•Virtual functions: function which gets called depending on type of object rather than type of pointer such type of function is called as virtual function.

•Class which contains at least one virtual function such type of class is called as polymorphic class. This is late binding.

Late Binding :
When call to the virtual function is given by either by using pointer or reference the it is late binding. In rest of the cases it is Early binding

•Function overriding :Virtual function define in base class is once again redefine in derived class is called function overriding.

•Functions which take part in to function overriding such functions are called overrided functions.
For function overriding function in base class must be virtual.

• *Function overloading*	• *Function overriding*
• **Function overloading is compile time polymorphism**	• **Function overriding is run time polymorphism**
• **Signature of the function must be different**	• **Signature of the function must be same**
• **For function overloading no keyword is required**	• **For function overriding virtual keyword is required in base class**
• **For function overloading functions must be in same scope . ie either function must be global or it must be inside the same class only.**	• **Function must be in base class and derived class.**
• **Function gets call by looking toward the mangled name**	• **Function gets call by looking toward the vtable**

Sunbeam Infotech

- **virtual function table**
- **When class contain at least one virtual function at that time complier internally creates one table which stores address of virtual function declared in side that class. Such table is called virtual function table or vftable or vtable**

- **Virtual function pointer**
- **When class contains virtual functions internally vtable is created by the compiler and to store address of the vtable compiler implicitly adds one hidden member inside a class which stores address of vtable such hidden member is called virtual function pointer / vfptr / vptr**

- **Vtable stores addresses of virtual function and vptr stores address of the vtable**

•Pure virtual function
•Virtual function which is equated to zero such virtual function is called pure virtual function.

•Generally pure virtual functions do not have body.
•Class which contains at least one pure virtual function such type of clas
is called as called abstract class .

•If class is abstract we can not create object of that class. But we can create pointer or reference of that class.
•It is not compulsory to override virtual function but it is compulsory to override pure virtual function in derived class.
•If we not override pure virtual function in derived class at that time derived class can be treated as abstract class

•Abstract class can have non virtual member function, virtual functions as well as pure virtual functions

RTTI

C++ provides feature of Run Time Type Information, which enables to find type of the object at runtime. For this feature, language contains “typeid” operator that returns reference to constant object of “type_info” class.

This class is declared in header <typeinfo> and contains information about the data type i.e. name of type.

For example, assuming that class circle and rectangle are inherited from shape class, following code explains use of RTTI:

```
int main()
 {
        Shape *ptrshape=NULL;
        int choice;
        cin >> choice;
        if(choice==1)
            ptrshape = new Circle;
        else
            ptrshape = new Rectangle;
        const type_info& info =    typeid(*ptrshape);
        cout << info.name() << endl;
        delete ptrshape;
        ptrshape=NULL;
        return 0;
}
```