OOPs

OOPs:- OOPs stands for Object-Oriented Programming language, which is a programming paradigm that bind togather the data and the functions that can operate on them so that no other part of the code can access this data except the function.

Class: It is a user defined data type, which holds its own data members and member functions which can be accessed and used by creating instance of that class.
A class is like a blue print for an object. eg:-
There may be many cars with different brand and names but all of them will share common properties that all car have.
- Data members are the data variables and member functions are the functions used to manipulate these variables togather there data member.

Object: An object is an identifiable entity with some characteristics and behaviour. An Object is an instance of a class. When a class is defined, no memory is allocated but when it is instantiated (i.e, an object is created) memory is allocated.

OOPs

① Encapsulation ④ Abstraction ③ Polymorphism ② Inheritance

Encapsulation: or Access modifier.

* **Access Specifiers:** in C++ are keywords used to define the accessibility of class members (variables and functions) from different parts of a program. There are three types

i) **Public**: The Public access specifier allows the members to be accessed from anywhere in the program. Public members are accessible by objects of the class, derived classes and even outside the class.

ii) **Private**: The private access specifier restrict the access of members to within the class only. Private members cannot be accessed or modified directly by objects of the class or derived classes. They are hidden from outside the classes.

iii) **Protected**: The protected access specifier is similar to the private access, but with an additional feature that allow access to derived classes. Protected member are not accessible from outside the class but can be accessed by derived classes. [~~protected member can be~~ or

NOTE The private a s can only be accessed through public member functions or friend function

eg: indirectly access by using function.

	private	protected	Public
inside class	✓	✓	✓
inside derived class	✗	✓	✓
on object	✗	✗	✓

Encapsulation:- It is defined as the wrapping up of data and information in a single unit. or it can be defined as binding together the data and the functions that manipulate them.

eg: in a company there are different sections like the accounts section, finance section, sales section etc.

• The finance section handles all the financial transaction and keeps record of all the data related to finance.
• Similarly the sales section handles all the sales-related activities and keeps records of all the sales.

Now there may arise a situation when for some reason an official from the finance section needs all the data about sales in a particular month.

In this case he is not allowed to directly access the data of the sales section. He will first contact some other officer in the sales section and then request him to give the particular data.

This is called encapsulation

* Features of encapsulation

F1) Data member and variables are declared private so to provide security.

F2) Member function should be declared public so that anyone can't change and work according to that function.

F3) We cannot access any function from the class directly we need an object to access that function that is using member variable of that class.

F4) The function which we are making inside the class must use only member variables, only then it is called encapsulation.

eg:-

```
#include <iostream>
#include <string>
using namespace std;
class person {
    private:
        string name;
        int age;
    public:
        person (string name, int age) {
            this->name = name;
            this->age = age;
        }
```

This is the way of initializing private member of the class with the help of constructor

```
void setname (string name) {
    this -> name = name;
}
```

to indicate that name is of private member

This is the way of initializing private member of the class by using setaram function.

```
string getname () {
    return name;
}
void setage(int age) {
    this -> age = age;
}
int getage() {
    return age;
};
```

How encapsulation is working here

→ There are some data of the class which are private (name, age) thus can only be changed by member function of the same class hence we encapsulate the data and member fn in a class which is called encapsulation.

```
int main() {
    person Person1("Yash Verma", 21);
    cout << "name" << person1.getname() << endl;
    cout << "age" << person1.getage() << endl;

    person1.setname("Yash Verma1");
    person1.setage(20);
    cout << "name" << person1.getname() << endl;
    cout << "age" << person1.getage() << endl;
    return 0;
}
```

F3

This is done by set name and set age i.e. without using constructor

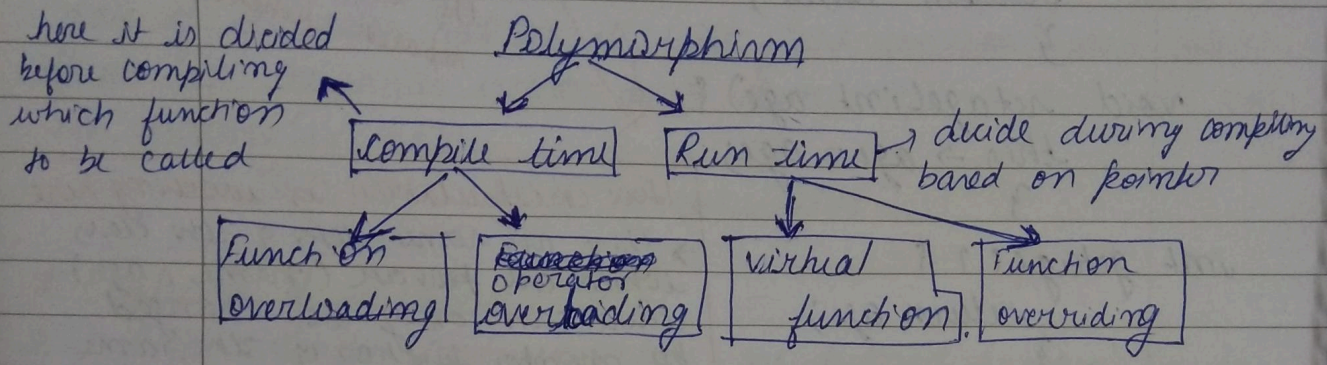
This is done by using constructor

Output

Name: Yash Verma
age: 21

Name: Yash Verma1
age: 20

Polymorphism: It is defined as the ability of a message to be displayed in more than one form.
eg: A man who at the same time can have different characteristics like he is a father, a husband, and an employee. So the same person exhibit the different behaviour in different situation. This is called polymorphism.



1) Compile time Polymorphism
This type of poly. is achieved by function overloading and operator overloading.

A) Function overloading: When there are multiple functions with the same name but different parameters, then the function is said to be overloaded.

```
eg:- #include <bits/stdc++.h>
     using namespace std;
     class Geeks {
       Public:
           void func(int x)  → Function 1
             { cout << x << end; }

           void func(double x) → Function 2
             { cout << x << end; }

           void func(int x, int y) → Function 3
           { cout << x << " " << y << endl;
           }
     };
```

```
int main()
{
    Rectangle obj1;
    // Function being called depends on the parameter
    // passed func() is called with int value.
    obj1.func(7);
    obj1.func(9.132);
    obj2.func(85, 64);
    return 0;
}
```

} respective function will automatically called acc. to data type

OUTPUT

7

9.132

85 64

3. Operator overloading

C++ has the ability to provide the operators with special symbol meaning for a data type, this ability is known as operator overloading. For example, we can make use of addition operator (+) for string class to concatenate two string. So a single operator (+) when placed b/w integer operands, add them and when placed between string operands, concatenates them.

2. Run time Polymorphism

* Virtual Function

Function overriding

Function overriding in C++ is a feature that allows a derived class to provide different implementation for a function that is already defined in its base class. When a derived class overrides a function, it provide its own implementation that is used instead of the base class.

In Own language :- If the base class and the derived class have the same function and if we call that function by using object of derived class than function

of derived class will be called. This is called function overriding.

eg:-
```
class Base1
{
  Public:
  void display(){
    cout<<"I am in Base1"; }
}
class Base2
{
  Public:
  void display(){
    cout<<"I am in Base2"; }
}

class derived : Public Base1, Public Base.
{
  Public:
  void display(){
    cout<<"I am in derived class"; }
};

int main(){
 derived d;
 d.display(); => This will print ("I am in derived class".)
```
Scope resolution

d.Base1::display(); ⇒ This will print("I am in Base1 class")

d.Base2::display(); ⇒ This will print("I am in Base2 class").

Base1 P;

return 0; P.display(); ⇒ This will display("I am in Base1 class").

function of derived class override the function of base class

Run time Polymorphism

It allows different object to respond differently to the same function call based on their actual runtime type.
Runtime polymorphism is achieved through two mechanism: inheritance and keywords.

* Virtual Function

In c++, a virtual function is a member function declared in a base class that can be overridden by derived classes. It enables runtime polymorphism by allowing different objects to provid their own implementation of the virtual function.
When a base class is declared as virtual, it serves as a blueprint for derived classes. Each derived class can chose to override the virtual function and provide its own implementation.

We will write two code without virtual and with virtual then we will automatically know what is virtual function is.

Virtual function

```
#include <iostream>
#include <bits/stdc++.h>
using namespace std;
class BaseClass {
        Public :
          int var_base = 45;
            void display() {
            cout<<" Display base class variable var_base"<<
                                    var_base<<endl;
              }
        };

class DerivedClass : public BaseClass {
          Public :
          int var_derived = 11;
        void display() {
```

NOTE: function is called is based on pointer

```
cout<<"Display Base class variable var_derived "<<var_base<<endl;
cout<<"Display Derived class variable var_derived"<<var_derived<<endl;
    }
};
```

{ Abdul Bari → Base class ptr and derived class ptr

Code with Harry }

```
int main(){
    BaseClass * base_class_pointer;
    BaseClass obj_base;
    derived class obj_derived;
    base_class_pointer = &obj_derived; // pointing base class pointer to derived class.

    base_class_pointer->display();
    base_class_pointer->var_derived; // This will show error as the pointer is of base class and we want to access member of derived class.

    base_class_pointer->var_base;  // no error
    base_class_pointer->var_derived;  // This will show error
    return 0;
}
```

This will not show any error

OUTPUT

→ Display Base Class variable var_base 45

Explanation - Base class ke pointer se derived class ka member ko access nhi kr skte. If we want to do so run the function of derived class we will use virtual keyword. See the below code and the difference.

: same

```
class BaseClass {
    Public:
        int var_base = 45;
        virtual void display() {
        cout<<"Display Base class variable var_base"<<var_base<<endl;
```

Same

```
; same }
int main(){ };
class Derivedclass : public BaseClass {
    public:
        int var_derived = 11;
    void display(){
        cout << "Display Derived class variable var_derived" << var_
                                                    derived << endl;
int main(){
    BaseClass * base_class_pointer;
    BaseClass obj_base;
    DerivedClass obj_derived;
    base_class_pointer = & obj_base;
    base_class_pointer -> display();
    base_class_pointer = & obj_derived;
    base_class_pointer -> display();
    return 0;
}
```

OUTPUT

→ Base class variable var_base 45
→ Base class variable var_derived 11

** Base car, Advance car
Base car * p = & advancecar object → ✓

Advancecar * p = & Basecar object → ✗

** ** ** C^mI Baseclass b;
Derivedclass * derived_class_pointer;
derived_class_point = & b ⇒ ✗
Wrong we can do this but can be done reversibly.

Think by Advance and Basic car.

It is a member function in a base class that we can override in derived class. It allows for runtime polymorphism ; decision of which function to call is made during runtime based on the type of the object being pointed to, rather than the type of pointer.

Here we can able to use the display function of both the class.

Base class ka pointer derived class ka object ko point kr skta hai lekin derived class ka pointer base class ka object ko point nhi kr skta

#Friend Function and Friend class

A friend function can be granted special access to private and protected members of a class in C++. They are non member function that can access and manipulate the private and protected members of the class for they are declared as friend.

They are of two types
i) A global function
ii) A member function of another class.

i) Global Function as friend function.

eg:-
```
#include <iostream>
class base{
    private:
        int private_variable;

    protected:
        int protected_variable;

    public:
        base()
        {   private_variable = 10;
            protected_variable = 20;
        }
    // friend f^n declaration.
    friend void friendFunction( base & obj);
    };
void friendFunction( base & obj){
  cout << "private variable:" << obj.private_variable << end;
  cout << "protected variable:" << obj.protected_variable << end;
    }
int main()
  { base object1
    friendFunction(object1);
    return 0;
  }
```

OUTPUT

private variable : 10
private variable : 20

ii) Member Function of another class as friend Function.

```
#include <iostream>
using namespace std;
class base; // forword defention needed
class anotherClass {
    Public:
        void memberFunction(base & obj);
};
class base {
    Public Private:
        int private_variable;

    protected;
        int protected_variable;

    public:
        base() {
            private_variable = 10;
            protected_variable = 20; }
    friend void anotherclass::memberFunction(base & obj);
};
void anotherclass::memberFunction(base & obj)
{
cout<<"Private variable "<< obj.private_variable <<endl;
cout<<"protected variable "<< obj.protected_variable <<endl;
}
int main(){
 base object1;
 anotherclass object2;
 object2.memberFunction(object1);
 return 0;
}
```

OUTPUT

```
private variable = 10
protected variable = 20
```

{ dusre class ka member fn ko bhi friend function

anotherclass ka memberFunction ko friend function bna.

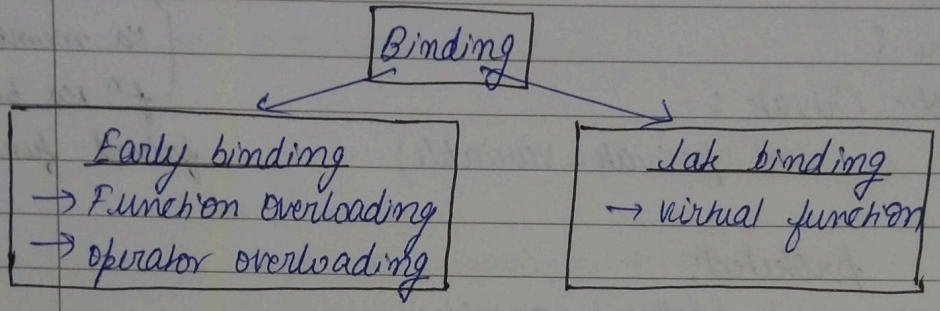
REASON

→ memberfunction another class me hai aur class ke function or member ko call krne ke liye object bnana hoga aur call krne ke baad base class ka member/variable lene ke liye base class ka object input me dalna hoga

Q) What are bindings ?

Ans: Binding refers to the process of converting identifiers (such as variables and performance names) into addresses.

Binding is done for each variable and functions. For functions, it means that matching the call with the right function definition by the compiler.



In Early binding compiler directly associate an address to the function call. It replaces the call with a machine language instruction that tells the mainframe to leap to the address of the function.

In Late binding the compiler adds the code that identifies the kind of object at runtime then matches the call with the right function definition.

Q) # Friend class

A friend class can access private and protected members of other class in which it is declared as friend. It is sometimes useful to allow a particular class to access private and protected members of other classes.

eg:

```
#include<iostream>
using namespace std;
class ABC {
  private:
      int private_variable;
protected:
   int protected_variable;
```

```
Public:
  GFG()
  { private- variable = 10;
    protected- variable = 99;
  }

  friend class F;
    };
  class F {
    public:
      void display (GFG & t)
       {
         cout << "t. private_variable << endl;
         cout << t. protected- variable << endl;
         }
       };
      int main() {
       GFG g;
       F Friend;
       Friend. display g);
        return 0;
       }
```

An exampu Virtual function

```
      class base {
            Public:
                virtual void print()
                  { cout << "This is base print" << endl;
                  }
               void show()
                  { cout << "Base show fun" << endl;
                  }
        }
      class derived {
            Public:
                Void print()
          { cout << "derived print" << endl; }
```

```
void show()
  { cout << "derived show fun" << endl;
  }
int main()
  { base *bptr;
    derived der;
    bptr = &der;

    bptr -> print();    // Run time
    bptr -> show();     // compile time
  }
```

output: derived print // Late binding
Base show fun // Early binding.

Constructors

Constructors is an special member function of the class. It is automatically invoked when an object is created.

→ It has no return type
→ Constructor has same name as class itself.
→ If we do not specify, then C++ compilers generates a default constructor for us.

```
eg:- #include<iostream>
     using namespace std;
     class rectangle;
      { private:
          int length;
          int breadth;
        Public:
          int area()
            { return length * breadth; }
          int parameter()
            { return 2*(length + breadth); }
        void setlength(int l)
         { if(l>=0)
             length=l;
           else length=0;
         }
```

```
void setbreadth(int b)
{ if(b>=0)
    breadth = b;
  else
  breadth=0;
}

rectangle()   // This is non Paramatrized constructor
  { length=0;
    breadth=0; }

rectangle(int l=0, int b=0) //This is paramatrized constructor.
 { setlength(l);
   setbreadth(b);
 }

rectangle(rectangle &rect) //This is copy constructor.
   { length = rect.length;
     breadth = rect.breadth;
   }

int getlength()
     { return length; }
int getbreadth()
     { return breadth; }
};
int main()   r1
{rectangle(4, 6); //constructor get automatically called
                  when we create object of the class.
cout << r.getlength() << endl;
cout << r.getbreadth() << endl;

rectangle r2(r1);
cout << r2.getlength();
cout << r2.getbreadth();
}

OUTPUT
→ 4, 6        4, 6
```

Destructor

Destructor is an instance member function which is invoked automatically whenever an object is going to be destroyed. / Meaning, a destructor is the last function that is going to be called before an object is destroyed.

* Destructor is also a special member function like constructor. Destructor destroy the class object created by constructor.
* It has same name as class name.
* It is not possible to define more than one constructor.
* It doesn't requires any argument not return any value.
* It is automatically called when object goes out of scope.

e.g:-

```
# include <iostream>
    using namespace std;
    // destructor never takes an argument nor does it return value.
int count =0;
class num{
    Public:
    num() {
        count++;
    cout<<" constructor is called for object number << count<<endl;
    }
~num() {
    cout<<" Destructor is called for object number << count << endl;
        count--;
        }
    };
int main(){
    cout<<" we are inside the main fn"<<endl;
    cout<<" Creating first object n1"<<endl;
    num n1;
        { cout<<"Creating two more objects" <<endl;
    num n2, n3;
    cout<<"exiting this block" <<endl;
     }
    cout<<" back to main" <<endl;
    return 0;
    }
```

OUTPUT

we are inside main function.
Creating first object n°1.
constructor is called for object number 1.
Creating two more objects.
constructor is called for object number 2.
constructor is called for object number 3.
Exiting this block
destructor is called for object number 3.
destructor is called for object number 2.
back to main
destructor is called for object number 1.

INHERITANCE

The capability of a class to derive properties and characteristics from another class is called inheritance.

- Subclass :- The class that inherits properties from another class is called Subclass or Derived class.
- Superclass :- The class whose properties are inherited by a Subclass is called base class or Superclass.

NOTE

```
class ABC : private XYZ    // private derivation.
class ABC : public XYZ     // public derivation
class ABC : protected XYZ  // protected derivation.
class ABC : XYZ            // private derivation by default.
```

* When a base class is privately inherited by derived class, public members of the base class becomes the private members of the derived class and therefore, the public members of the base class can only be accessed by the member functions of the derived class.

NOTE There are two ways of calling a class

(i) by derived class

(ii) Class Rectangle

```
{ =
}
```

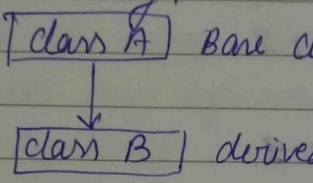
Class cuboid : Public Rectangle

```
{ =
}
```

```
class Table {
   Rectangle top;
   ...
};
```

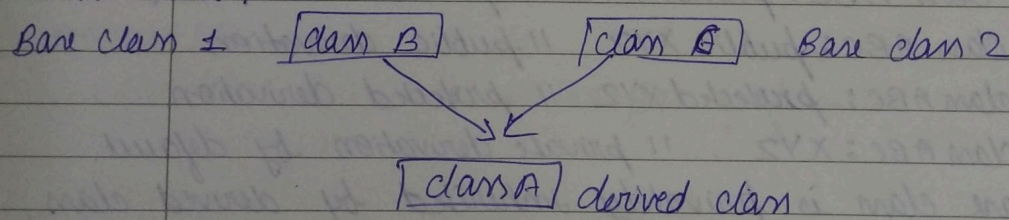
Types of Inheritance

1) **Single Inheritance**: In single inheritance, a class is allowed to inherit from only one class i.e, one subclass is inherited by one base class only.



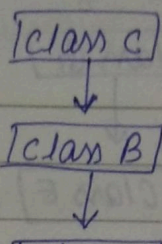
Syntax eg:-
```
class A
{
  ....
};
class B : public A
{
  ....
};
```

2) **Multiple Inheritance**: A class can inherit from more than one class.



Syntax eg:-
```
class B
{...
};
class C
{...
};
class A : public B, public C.
{
  ....
};
```

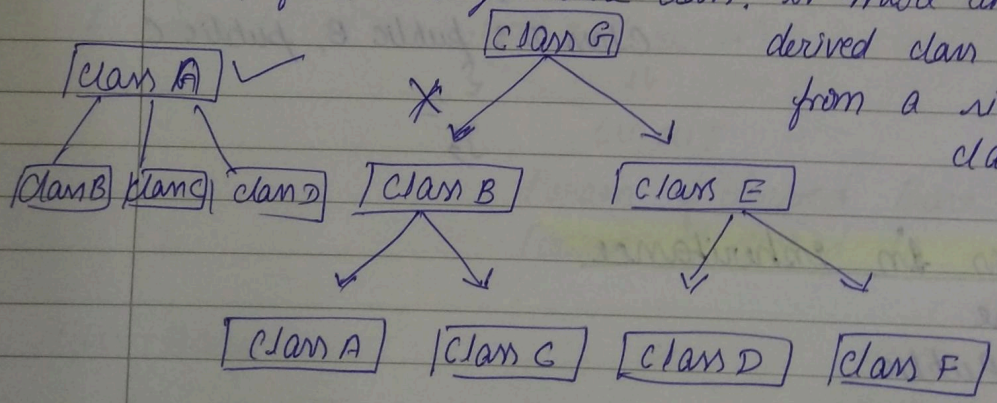
3) **Multilevel Inheritance**: a derived class is inherited from another derived class.



Syntax :-

```
Class C
  {
  ....
  };
Class B : public C
  {
  ....
  };
Class A : public B
  {
  ....
  };
```

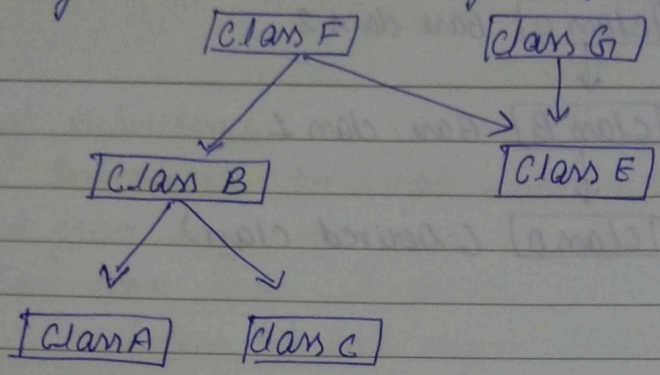
4) **Hierarchial Inheritance**: In this, more than one subclass is inherited from a single base class. or more than one derived class is created from a single base class.



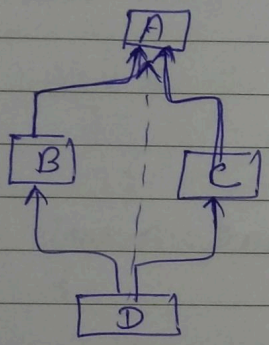
Syntax :

```
class A :
  { ...
  };
class B : public A
  {...
  };
class C : public A
  {...
  };
class D : public A
  {...
  };
```

5) Hybrid (virtual) Inheritance: Hybrid inheritance is implemented by combining more than one type of inheritance.



* Multipath Inheritance




```
Class A
{
};
Class B : virtual public A
{
};
Class C : virtual public A
{
};
Class D : public B, public C
{
};
```

Constructors in Inheritance

```
Class Base
{ Public
    Base()
    { cout<<"Default of base"<< endl;
    }
  Base (int x)
   { cout<<"Param of Base "<< end;
   }
};
Class derived : public Base
{
```

```
Public:
    Derived()
        { cout<<"Default of derived";
        }
    Derived(int a)
        { cout <<"Param of Derived"<<a;
        }
        };

int main()
  { derived d
    return 0;
  }
```

OUTPUT
⇒ default of base
- default of derived

```
int main()
    { derived d(10);
      return 0;
    }
```

OUTPUT
⇒ Default of base
Param of derived 10

```
int main()
  { derived(int x, int a) : Base(x);
  }
```

OUTPUT
Param of ~~derived~~ Base 20
Param of derived 10

```
class Base
    {
    =
    =
    };
class derived : public base
        {
            public:
                Derived()
                  { cout << "Default of derived";
                  }
```

```
Derived(int a)
    {
      cout << "param of derived << a << endl; //
    }

Derived(int x, int a): Base(x)
    {
        cout << "param of derived " << a << endl;
    }
};
int main()
{
    Derived d(5,10);
    return 0;
}
```

OUTPUT

param of base 5
param of derived 10.

Access modifier through example

```
class Base
{   private : int a;
    protected : int b;
    public : int c;
    void function()
    {   a=10;
        b=20;
        c=30;
    }
};
int main()
{   Base x;
    x.a = 15; X cant be accers
    x.b = 30; X cant be accers
    x.c = 90; ✓ can be accers.
}
```

```
class Base
{
    == Same as previous
};

class derived : Base class
    public:
        funderived()
        {   a=1; ✗ can't be access
            b=2; ✓ can be access
            c=3; ✓ can be access
        }
};
```

Ways of Inheritance

```
class parent
{
  private : int a;
  protected : int b;
  public : int c;
  void funparent()
  { a=10; ✓
    b=5; ✓
    c=15; ✓
  }
};
class child : public parent
{ public :
  void funchild()
  { a=10; ✗
    b=5; ✓ protected
    c=15; ✓ public
  }
};
class grandchild : public child
{ public :
  void fungrandchild()
  {
```

:: it is inside the class so all are accessible

```
class parent
{
  private : int a;
  protected : int b;
  public : int c;
  void funparent()
  { a=10; ✓
    b=5; ✓
    c=15; ✓
  }
};
class child : protected parent
{ public :
  void funchild()
  { a=10; ✗
    protected b=5; ✓
    protected c=15; ✓
  }
};
class grandchild : public child
{ public :
  void fungrandchild()
  {
```

depends on

```
class parent
{
  private : int a;
  protected : int b;
  public : int c;
  void funparent()
  { a=10; ✓
    b=5; ✓
    c=15; ✓
  }
};
class child : private parent
{ public :
  void funchild()
  { a=10; ✗
    b=5; ✗ ✓ private
    c=15; ✗ ✓ private
  }
};
class grandchild : public child
{ public :
  void fungrandchild()
  {
```

```
a=10; ✗
b=5; ✓ protected
c=20; ✓ public
};
int main()
{ child c
  c.a=10; ✗
  c.b=5; ✗
  c.c=20; ✓
}
```
public

depend on

```
a=10; ✗
b=5; ✓ protected
c=20; ✓ protected
};
int main()
{ child c
  c.a=10; ✗
  c.b=5; ✗
  c.c=20; ✗
}
```

```
a=10; ✗
b=5; ✗
c=20; ✗
};
int main()
{ child c
  c.a=10; ✗
  c.b=5; ✗
  c.c=20; ✗
}
```

Pure virtual function

First we learn what is abstract class.

An abstract class is a class that cannot be instantiated directly and is designed to be used as a base class for other classes.

eg:-

```
class AbstractClass {
public:
    virtual void someFunction()=0 // pure virtual function
    void concreteFunction() {
        // implementation
    }
};
class DerivedClass : public AbstractClass {
public:
    void someFunction() {
    // implementation specific to the derived class
    }
};
```

Base with all concrete fn	Base with some concrete fn and some pure virtual fn	Base with only pure virtual fn
↓	↓	↓
reusability	reusability and polymorphism	polymorphism

Abstraction

Abstraction refers to the providing only essential information about the data to the outside world, hiding the background detail

Type of Abstraction

1) Data Abstraction: This type only shows the required information about the data and hides unnecessary data.

2) Control Abstraction: This type only shows required information about the implementation and hides unnecessary information.

Abstraction in header file

The pow() method is present in math.h header file. When we need it we simply call the pow() function without knowing the underlying algorithm according to which the function is actually calculating the power of numbers.

Abstraction using Access Specifiers

Members declared as public in a class can be accessed from anywhere in the program.

Members declare as private in a class, can be accessed only from within the class

Static member

* It is declared using static keyword
* only one copy of that member is created for the entire class and is shared by all the objects of that class, no matter how many objects are created.
* It is initialized before any object of this class is created, even before the main start.
* It is visible only within the class, but its lifetime is the entire program.

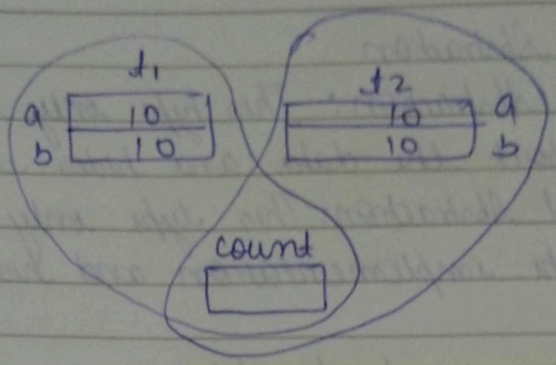
Syntax

```
static data_type data_member_name;
```

eg:

```
class Test
{
```

Private:
int a;
int b;
public:
static int count;
Test()
{ a=10;
b=10;
count++;
}
};



int Test::count=0; → this is to make accessible only in the class to the object of the respective class.

main()
{ Test t1;
Test t2;
cout << t1.count; => 2
cout << t2.count; => 2
cout << Test::count => 2
} } all are same

* if we declare a static member function in the class.

static int getCount()
{
a++; ✗ => Can't be written as it can't
return count; ✓ cannot access non static member like a and b here.
}

Eg: 2 class student
{ Public:
int rollno;
static int adminNO;
Student()
{ adminNO++;
rollno = adminNO;
}
};
int Student::adminNO=0;

```
int main()
{ student S1;
  Student S2;
}
```