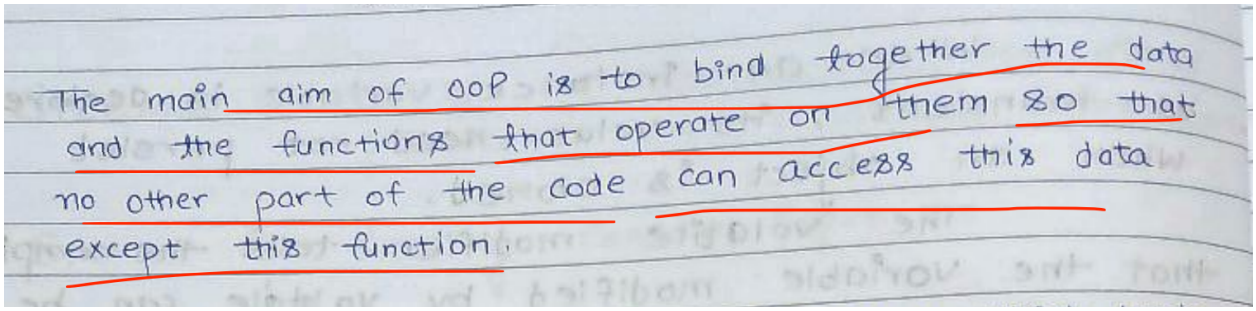
OOPS

OOPS- It is methodology or paradigm that is used to design a program using class and objects.



Classes-

Class is a **user-defined data type which defines its properties and its functions**. Class is the only **logical representation of the data**. For example, Human being is a class. The body parts of a human being are its properties, and the actions performed by the body parts are known as functions. The class does not occupy any memory space till the time an object is instantiated.

C++ Syntax (for class) :

```
class student{
    public:
      int id; // data member
      int mobile;
      string name;

      int add(int x, int y){ // member functions
          return x + y;
      }
};
```

Objects-

Object is a **run-time entity**. It is **an instance of the class**. An object can represent a person, place or any other item. An object can operate on both data members and member functions.

C++ Syntax (for object):

```
student s = new student();
```

Note : When an object is created **using** a new keyword, then space is allocated for the variable in a heap, and the starting address is stored in the stack memory. When an object is created **without** a new keyword, then space is not allocated in the heap memory, and the object contains the null value in the stack.

Structure vs class

Here, we are going to discuss the main differences between the structure and class. Some of them are as follows:

- By default, all the members of the structure are public. In contrast, all members of the class are private.
- The structure will automatically initialize its members. In contrast, constructors and destructors are used to initialize the class members.
- When a structure is implemented, memory allocates on a stack. In contrast, memory is allocated on the heap in class.
- Variables in a structure cannot be initialized during the declaration, but they can be done in a class.
- There can be no null values in any structure member. On the other hand, the class variables may have null values.
- A structure is a value type, while a class is a reference type.
- Operators to work on the new data form can be described using a special method.

Inheritance-

Inheritance is a process in which one object acquires all the properties and behaviors of its parent object automatically. In such a way, you can **reuse, extend or modify** the attributes and behaviors which are defined in other classes.

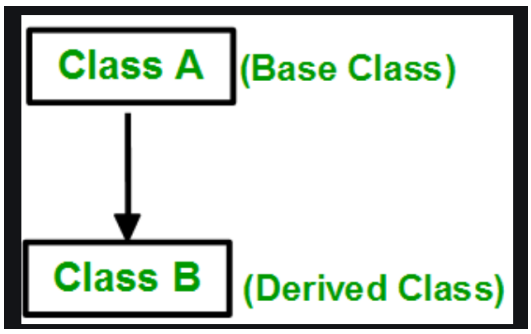
In C++, the class which inherits the members of another class is called derived class and the class whose members are inherited is called base class. The derived class is the specialized class for the base class.

C++ Syntax :

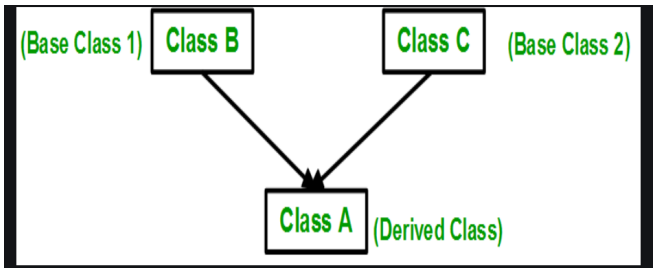
class derived_class :: visibility-mode base_class;
visibility-modes = {private, protected, public}

Types of Inheritance :

1. Single inheritance : When one class inherits another class, it is known as single level inheritance
2. Multiple inheritance : Multiple inheritance is the process of deriving a new class that inherits the attributes from two or more classes.
3. Hierarchical inheritance : Hierarchical inheritance is defined as the process of deriving more than one class from a base class.
4. Multilevel inheritance : Multilevel inheritance is a process of deriving a class from another derived class.
5. Hybrid inheritance : Hybrid inheritance is a combination of simple, multiple inheritance and hierarchical inheritance.



Single



Multiple

– Java does not support multiple inheritance due to diamond problem

-> Diamond problem - What Java does not allow is multiple inheritance where one class can inherit properties from more than one class. It is known as the diamond problem.

```java
import java.io.*;

class GrandParent {
void fun() {
    System.out.println("Grandparent");
}
}

class Parent1 extends GrandParent {
void fun() {
    System.out.println("Parent1");
}
}
```

```java
class Parent2 extends GrandParent {
void fun() {
    System.out.println("Parent2");
}
}

class Test extends Parent1, Parent2 {
public static void main(String args[]) {
    Test t = new Test();

    t.fun();
}
}
```

This program shows that there exists a diamond problem as test.fun is confused between two fun() functions which to call this occurs in java.

Multiple inheritance: In java to achieve multiple inheritance we use the concept of interfaces.

Interface-

An **Interface in Java** programming language is defined as an abstract type used to specify the behavior of a class. An interface in Java is a blueprint of a class. A Java interface contains static constants and abstract methods.

The interface in Java is *a* mechanism to achieve abstraction. There can be only abstract methods in the Java interface, not the method body. It is used to achieve abstraction and multiple inheritance in Java. In other words, you can say that interfaces can have abstract methods and variables. It cannot have a method body. Java Interface also **represents the IS-A relationship**.

Abstract method means the method inside an interface is only declared, its body is not implemented inside it.

```java
interface Mummy {
    void age();
}
interface Papa {
    void age();
    void sex();
}
```

Here age and sex are both abstract methods.
The body is initialized in the class that implements the interface.

```java
// Class implements multiple interfaces
// Multiple inheritance in JAVA using interface
class Baccha2 implements Mummy, Papa {
    public void age(){
        System.out.println("20+");
    }
    public void sex(){
        System.out.println("F");
    }
}
```

CODE - diamond_problem(solution).java
In This as you can see bacha2 implements both mummy and papa interfaces and inside we have implemented both the two methods.

Now, the above figures also show us how we have tackled the diamond problem.
As you can see we have initialized two interfaces and declare two same functions inside them (if we use different functions too there occurs a diamond problem too in java) after this we have implemented a bacha2 class in which we overrides the two functions which help compiler from the confusion as it was occurring before which function to select.

Some extra info about Interfaces:

I. Interface can extend many other interfaces.

```
interface Mummy {
    void age();
}
interface Papa {
    void age();
    void sex();
}

// One interface extends another interface not implements it
// One interface can implement multiple interfaces
interface Baccha1 extends Mummy, Papa {
    void name();
}
```

Ii. we can also achieve the multiple inheritance by doing this too

```
class Grandbaccha implements Baccha1 {
    public void sex(){
        System.out.println("M");
    }
    public void age(){
        System.out.println("18+");
    }
    public void name(){
        System.out.println("land");
    }
}
```

(THE MAIN THING TO ACHIEVE MULTIPLE INHERITANCE IS TO IMPLEMENT A ABSTRACT METHODS INTO ITS SUBCLASS)

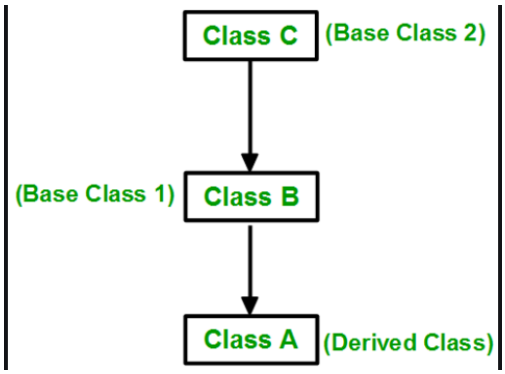
Iii. The data members of the interface are always public,static and final.

Iv. Interface helps us in achieving 100% abstraction.

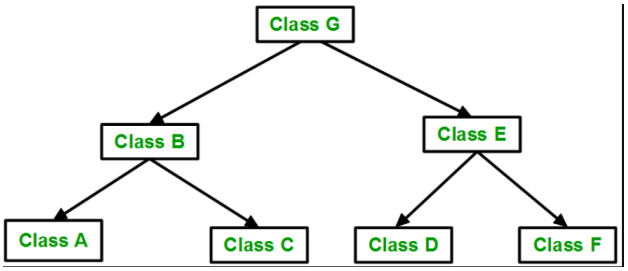
Multiple inheritance in cpp

It is simple like any other inheritance but if we do not override in cpp too we can get an error.

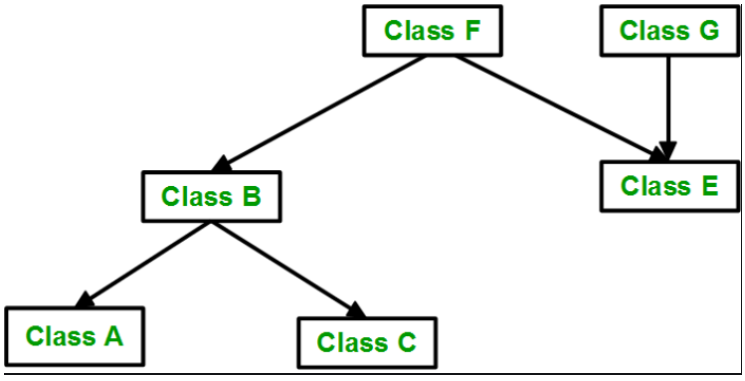
(iska example k photo)



Multilevel



Hierarchical



Hybrid

Modes of Inheritance: There are 3 modes of inheritance.

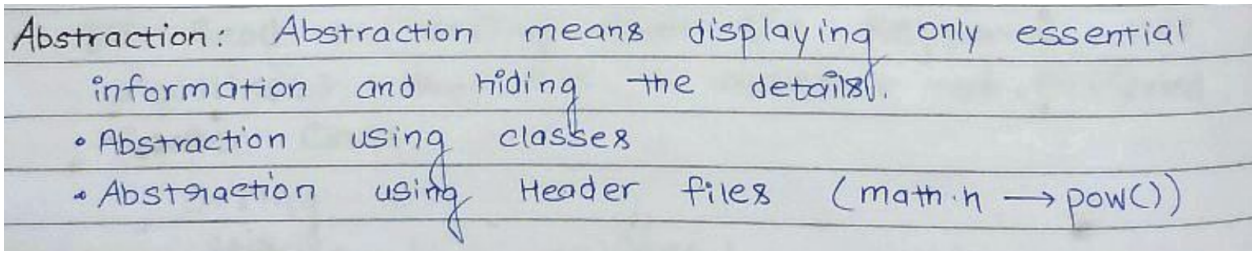
1. **Public Mode**: If we derive a subclass from a public base class. Then the public member of the base class will become public in the derived class and protected members of the base class will become protected in the derived class.
2. **Protected Mode**: If we derive a subclass from a Protected base class. Then both public members and protected members of the base class will become protected in the derived class.
3. **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.

Private members of a class never gets inherited to any subclass. They are accessible only within their class.

Encapsulation:

Encapsulation is the process of combining data and functions into a single unit called class. In Encapsulation, the data is not accessed directly; it is accessed through the functions present inside the class. In simpler words, attributes of the class are kept private and public getter and setter methods are provided to manipulate these attributes. Thus, encapsulation makes the concept of data hiding possible. (**Data hiding:** a language feature to restrict access to members of an object, reducing the negative effect due to dependencies. e.g. "protected", "private" feature in C++).

Abstraction



No need to know about details only we have to use the upper things like in example of header we use it we don't ever think how pow was implemented inside and we don;t need to

Abstraction is achieved by Interface and Abstract class. 100% achieved by interface as abstract class contains non abstract function in it.

Abstract Class:

A class which is declared with the abstract keyword is known as an abstract class in Java. It can have abstract and non-abstract methods (method with the body).

```
abstract class Shape {
    String color;
    abstract double area();
    public abstract String toString();

    // abstract class can have the constructor
    public Shape(String color)
    {
        System.out.println("Shape constructor called");
        this.color = color;
    }

    // this is a concrete method
    public String getColor() {
        return color;
    }
}
```

abstract double area(); - It is an abstract method similar to the abstract methods that we declare in the interface but to make abstract methods here we have to write abstract keywords before any function.

Public string get color(){} - It is a non abstract function.(Concrete methods)

(WE CANNOT CREATE AN OBJECT OF ABSTRACT CLASS AND INTERFACE DIRECTLY BUT CAN ACCESS THE DATA MEMBERS AND METHODS BY OBJECTS OF SUBCLASS)

Points to remember:

- An abstract class must be declared with an abstract keyword.
- It can have abstract and non-abstract methods.
- It cannot be instantiated.
- It can have constructors and static methods also.
- It can have final methods which will force the subclass not to change the body of the method.

In java (Example-2)
Abstraction using abstract class

```
abstract class Shape{
abstract void draw();
}
//In real scenario, implementation is provided by others i.e. unknown by end user
class Rectangle extends Shape{
void draw(){System.out.println("drawing rectangle");}
}
class Circle1 extends Shape{
void draw(){System.out.println("drawing circle");}
}
//In real scenario, method is called by programmer or user
class TestAbstraction1{
public static void main(String args[]){
Shape s=new Circle1();//In a real scenario, object is provided through method, e.g., getShape() method
s.draw();
}
}
```

```
drawing circle
```

In cpp
Abstraction is achieved by

- **Abstract Classes**: In C++ class is made abstract by declaring at least one of its functions as a **pure virtual function**. A pure virtual function is specified by placing "= 0" in its declaration. **Its implementation must be provided by derived classes**.

  Example :

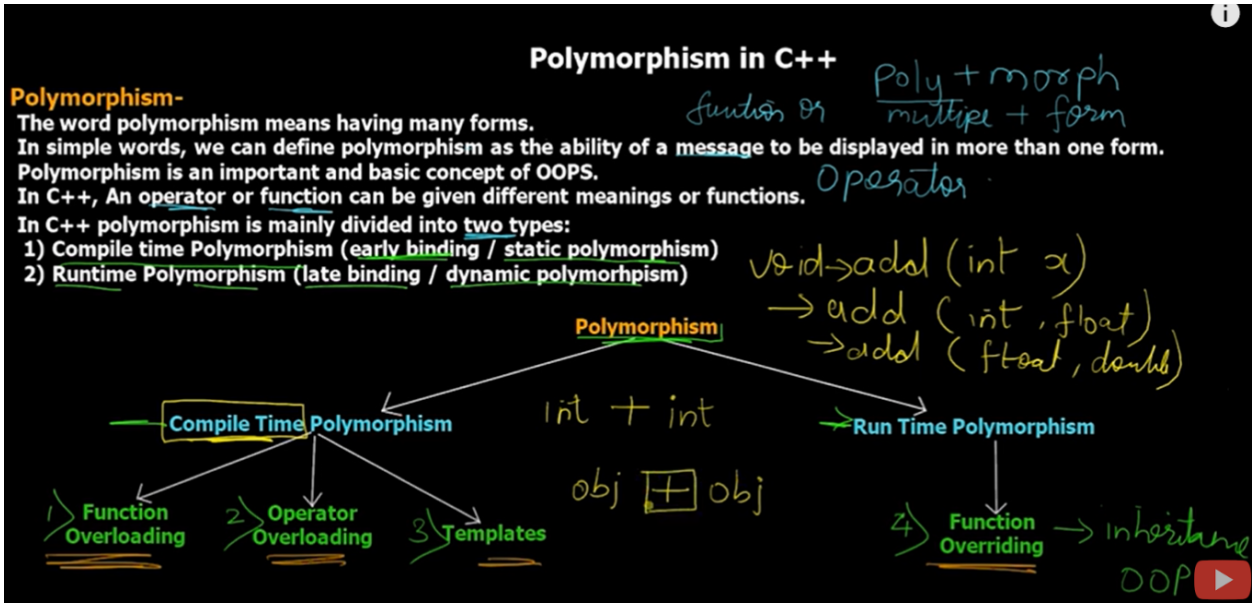
```
#include<bits/stdc++.h>
using namespace std;

// abstract class
class Shape{
    public:
        virtual void draw()=0;
};
class Rectangle : Shape{
    public:
        void draw(){
            cout << "Rectangle" << endl;
        }
};
class Square : Shape{
    public:
        void draw(){
            cout << "Square" << endl;
        }
};

int main(){
    Rectangle rec;
    Square sq;
```



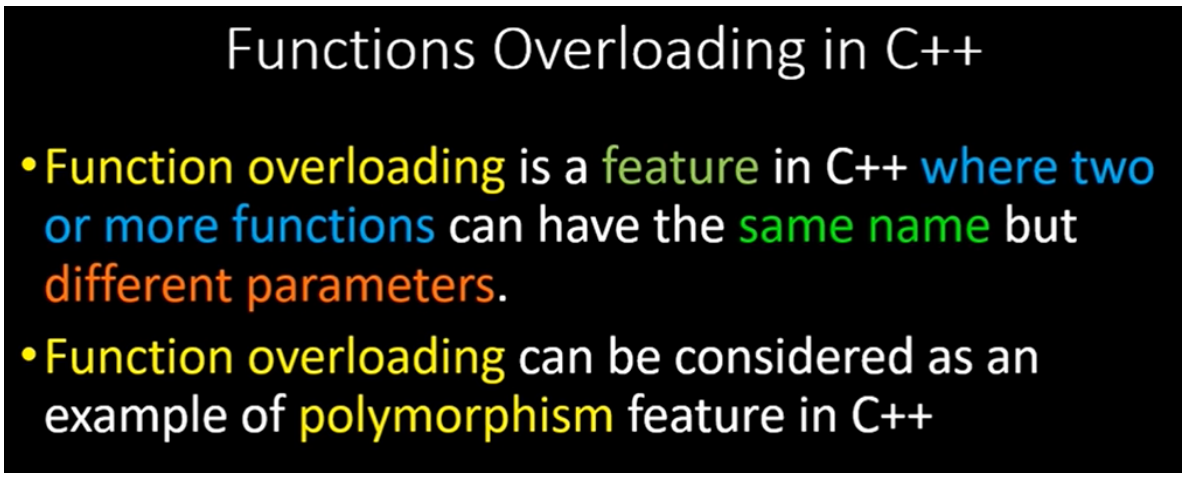
```
    rec.draw();
    sq.draw();
    return 0;
}

/*
Output :
Rectangle
Square
*/
```

Difference between Encapsulation And Abstraction :

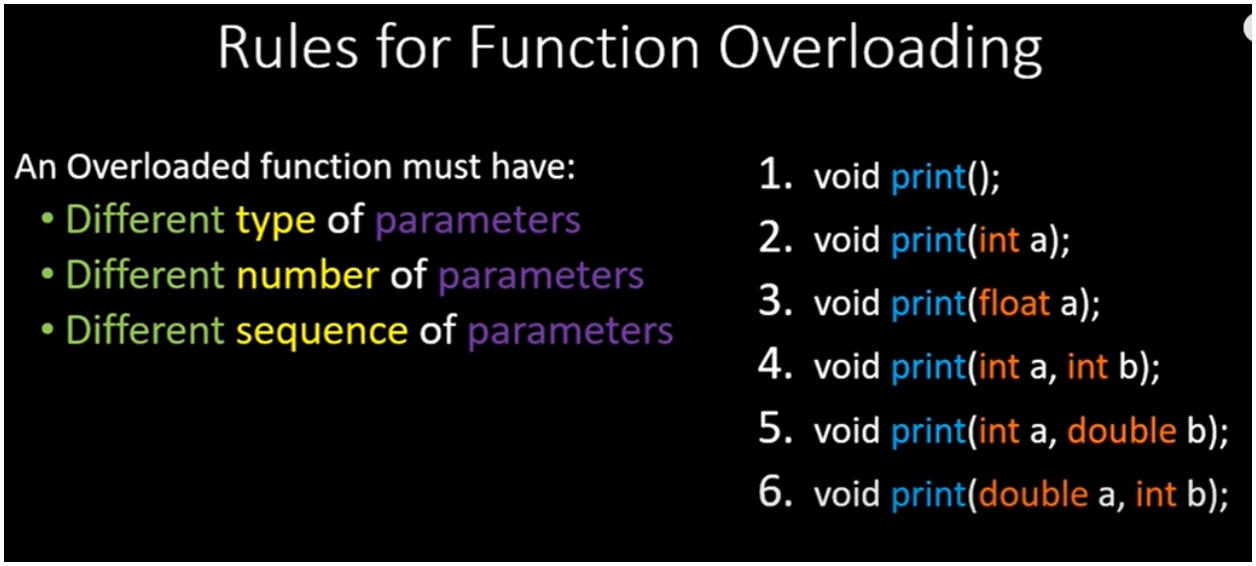
Polymorphism:



Function overloading:



Function overloading occurs in same class only.



Function overloading cannot be done on the basis of only return type. Ex-
Void print(int a, int b){}
String print(int a, int b){}

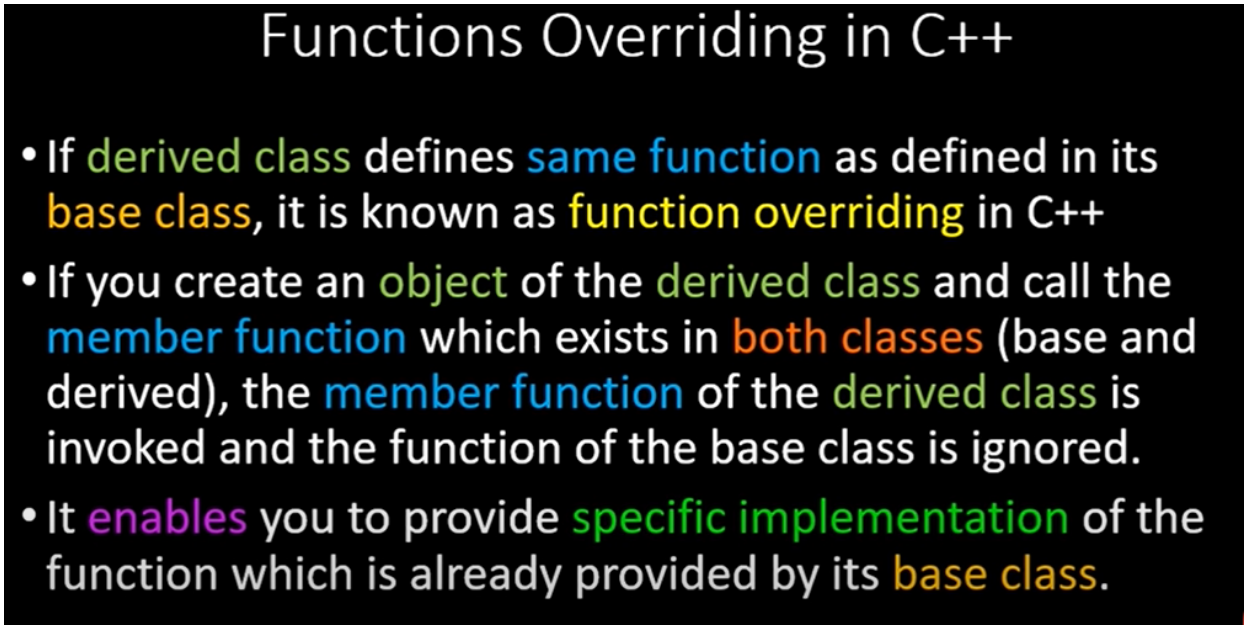
This will give error as the return type are different. But if we also change the passed parameters then its okay!

Code:

```
#include <bits/stdc++.h>
using namespace std;

// Function overloading
class Animal{
    public:
    void add(int x,int y,int z){
        cout<<x+y+z<<"\n";
    }
    void add(int x,int y){
        cout<<x+y<<"\n";
    }

};

int main(){
    Animal a;
    a.add(1,2,3);
    a.add(1,2);
}
```

Output:

Function Overriding:



code:

```cpp
//Function overriding
class Animal{
  public:
  void sum(){
      cout<<"added"<<"\n";
  }
};

class dog: public Animal{
  public:
  void sum(){
      cout<<"not added"<<"\n";
  }
};

int main(){
    //derieved class object
    dog d;
    d.sum();

    //base class object.
    Animal a;
    a.sum();
}
```

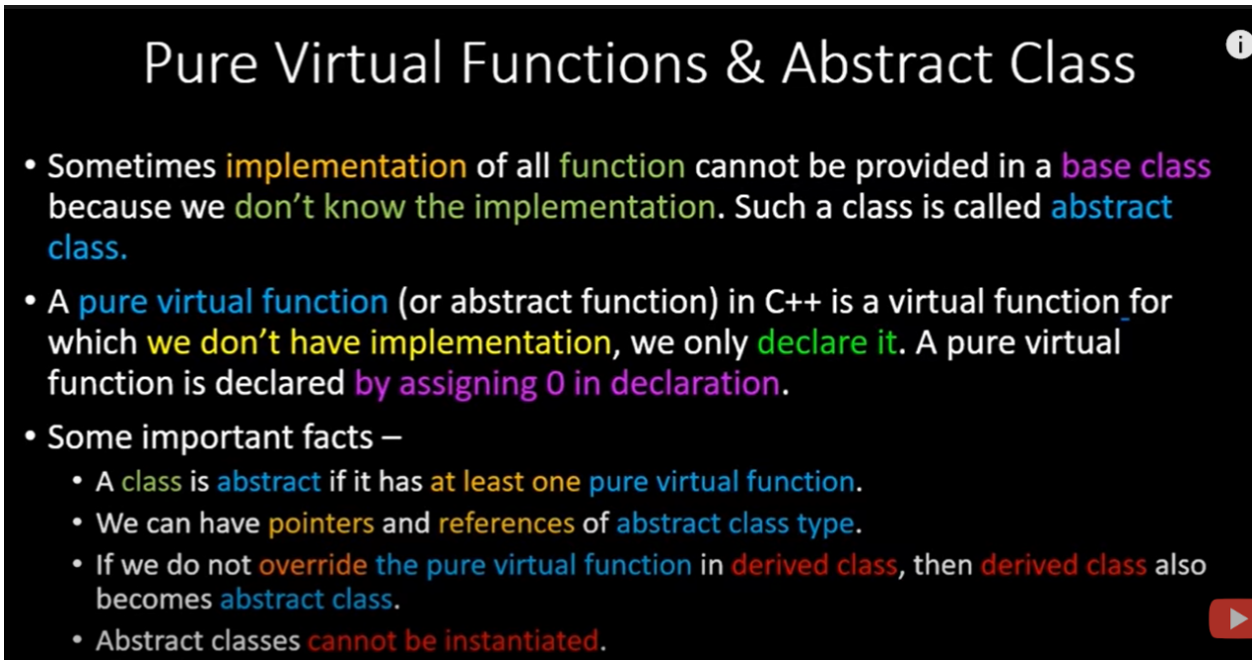
```
not added
added
```

Virtual Function:

- A virtual function is a member function which is declared within base class and is re-defined (Overridden) by derived class.
- When you refer to a derived class object using a pointer or a reference to the base class, you can call a virtual function for that object and execute the derived class's version of the function.
  - Virtual functions ensure that the correct function is called for an object, regardless of the type of reference (or pointer) used for function call.
  - They are mainly used to achieve Runtime polymorphism
  - Functions are declared with a virtual keyword in base class.
  - The resolving of function call is done at Run-time.

By declaring a function virtual, it tells the compiler to perform late binding(run time polymorphism) so that the reference pointers call the right function as needed.

Code: Virrual_Fun_Ex1.cpp (Explain)



Code: Pure_Virtual_Fun_Ex.cpp

To clarify more:
See the code :--- final_vf.cpp

Binding - Connecting the function call to the function body is called binding.

When it is done before the program is run, it is called early binding/static binding/compile time binding.
When it is done after the program gets run, it is called late binding/Dynamic binding/Run time binding.

—----------

Early Binding/Static Binding and Late Binding/ Dynamic Binding:

```
// if function calling is known at compile time then ===> static binding
// if function calling is known at run time then     ===> dynamic binding
```

#1 Function call binding using class objects

```
class Base{
public:
  void show(){ cout << "Base class" << endl; }
};
class Derived : public Base{
public:
  void show(){ cout << "Derived class" << endl; }
}

int main(){
Base b;
Derived d;
b.show();
d.show();
}
```

Output- Base class
Derived class

**(In the above example, we are calling the overridden function using Base class
and Derived class object. Base class object will call base version of the function and derived class's object will call the derived version of the function)**

#2 Function call binding using base class pointers

```
class Base{
public:
  void show(){ cout << "Base class" << endl; }
};
class Derived : public Base{
public:
  void show(){ cout << "Derived class" << endl; }
}

int main(){
Base *ptr;   //Base class pointer
Derived d;   //Derived class object
ptr = &d;    //Base class pointer pointing to derived class object
ptr->show(); //Early binding occurs
}
```

Output- Base class

In the above example, although, the object is of Derived class, still Base class's method is called.
This happens due to Early Binding.
**Compiler on seeing Base class's pointer, set call to Base class's show() function, without knowing
the actual object type.**

Note-

When we use Base class's pointer to hold Derived class's object, base class pointer or reference will always call the base version of the function

To resolve this issue-

Virtual functions in c++

Virtual function is a member function of the base class which is redefined in the derived class and which tells the compiler to perform late binding on this function.

In Late Binding function call is resolved at runtime. Hence, now compiler determines the type of object at runtime, and then binds the function call.

```
class Base{
public:
  virtual void show(){ cout << "Base class" << endl; }
};
class Derived : public Base{
public:
  void show(){ cout << "Derived class" << endl; }
}

int main(){
Base *ptr;   //Base class pointer
Derived d;   //Derived class object
ptr = &d;    //Base class pointer pointing to derived class object
ptr->show(); //Late binding occurs
}
```

Output- Derived class

On using Virtual keyword with Base class's function, Late Binding takes place and the derived version of
function will be called, because base class pointer pointes to Derived class object.

#3 Using Virtual Keyword and Accessing Private Method of Derived class

We can call private function of derived class from the base class pointer with the help of virtual keyword.
Compiler checks for access specifier only at compile time. So at run time when late binding occurs it does
not check whether we are calling the private function or public function.

```
class A{
 public:
   virtual void show(){ cout << "Base class\n"; }
};
class B: public A {
 private:
   virtual void show(){ cout << "Derived class\n"; }
};
int main(){
   A *a;
   B b;
   a = &b;
   a->show();
}
```

Output- Derived class

If we extend class B with private A then we will get error. Private inheritance means that outside the derived class, the inheritance information is hidden. That means you can't cast the derived class to the base class

Public inheritance means that everyone knows that Derived is derived from Base.

Protected inheritance means that only Derived, friends of Derived, and classes derived from Derived know that Derived is derived from Base.*

Private inheritance means that only Derived and friends of Derived know that Derived is derived from Base.

Since you have used private inheritance, your main() function has no clue about the derivation from base, hence can't assign the pointer.

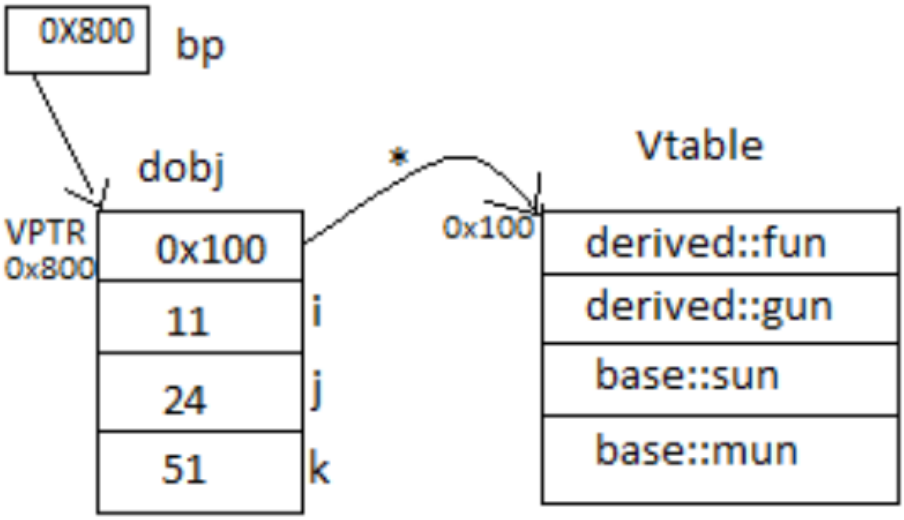
Private inheritance is usually used to fulfill the "is-implemented-in-terms-of" relationship. One example might be that Base exposes a virtual function that you need to override -- and thus must be inherited from -- but you don't want clients to know that you have that inheritance relationship.

Note-

1) If a function is declared as virtual in the base class, it will be virtual in all its derived classes.

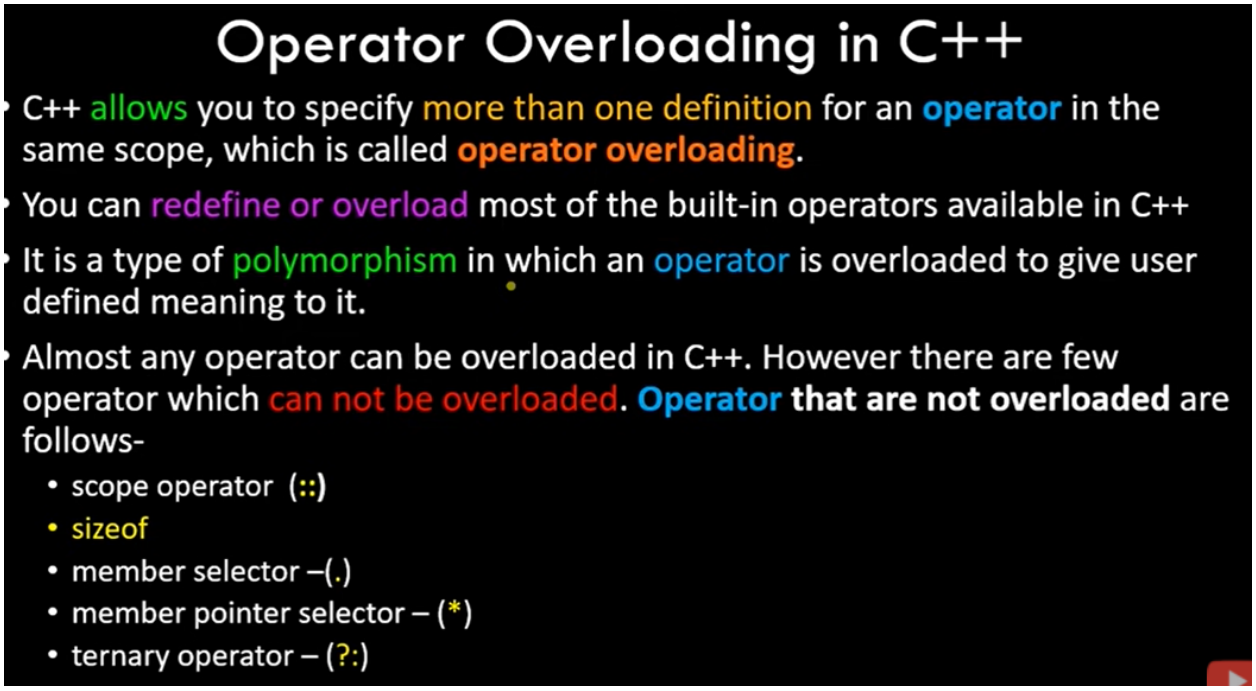
2) To accomplish late binding, Compiler creates VTABLEs, for each class with virtual function.
The address of virtual functions is inserted into these tables. Whenever an object of such
class is created the compiler secretly inserts a pointer called vpointer, pointing to VTABLE
for that object. Hence when function is called, compiler is able to resovle the call by binding
the correct function using the vpointer.

3) The address of the virtual Function is placed in the VTABLE and the compiler uses VPTR(vpointer) to point to the Virtual Function.



VPTR and Vtable internal structure

Operator Overloading :



Operator overloading is done only for user defined objects.
Below is the example.
Java Doesn't support Operator Overloading. Creators ko jyada chull nai tha kyuki Java unko already kaafi complicated laga!

Code:

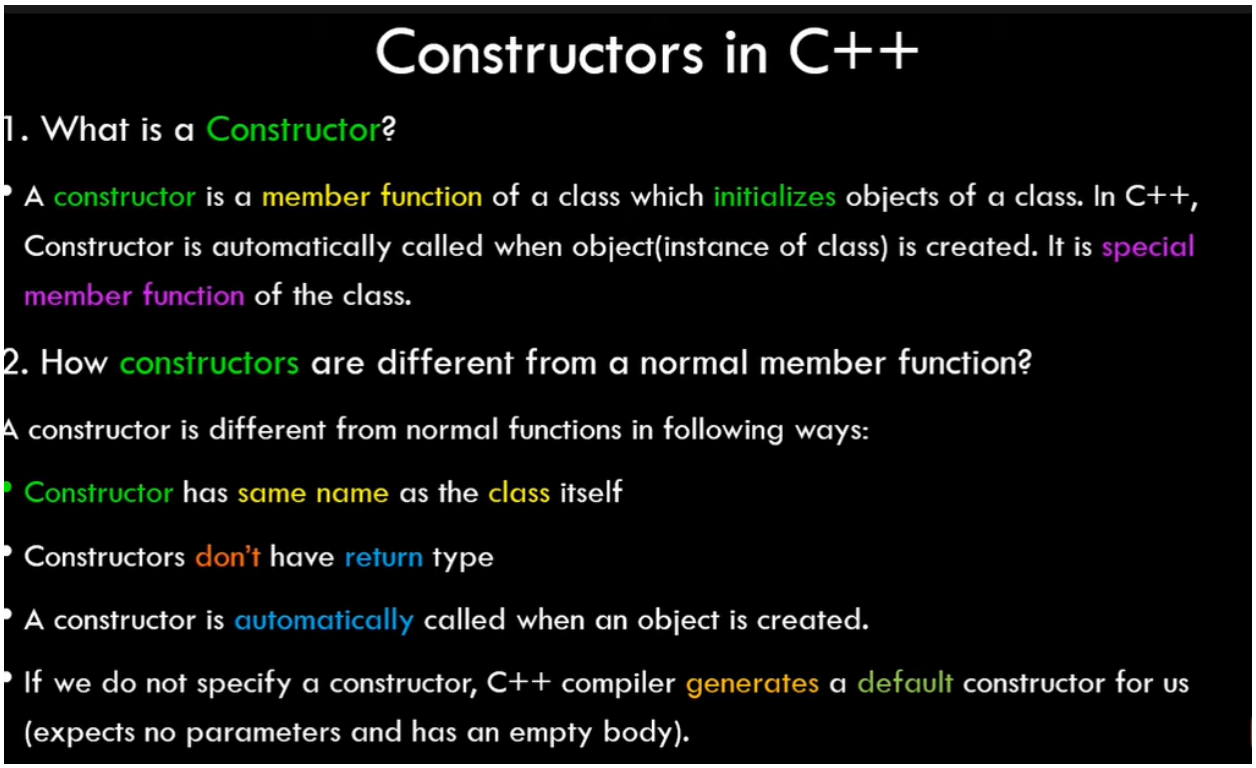
```
#include <bits/stdc++.h>
using namespace std;

//operator overloading

class complexs{
  private:
  int real;
  int img;
```

```
  public:
  complexs(){
     real=0;
     img=0;
  }
  complexs(int real,int img){
     this->real=real;
     this->img=img;
  }
  complexs operator + (complexs c){
     complexs temp;
     temp.real=real+c.real;
     temp.img= img+ c.img;
     return temp;
  }

  void show(){
     cout<<real<<" + "<<img<<"i"<<"\n";
  }
};


int main(){
    complexs c1(2,3);
    complexs c2(5,6);
    complexs c3;
    c3=c1 + c2; // c1.add(c2)
    c3.show();
    // c4= c1+ c2 + c3 means- c1.add(c2.add(c3))
```

}

Difference between “this” and “super” keywords??

Constructors and destructors:



Constructors cannot be private

```
class A {
private:
    A() {}
};
int main()
{
    A a;
}
```

This will give error as constructor in A is private.

Constructors in C++

Types of Constructors?

1. Default Constructors: Default constructor is the constructor which doesn't take any argument. It has no parameters.
2. Parameterized Constructors: It is possible to pass arguments to constructors. Typically, these arguments help initialize an object when it is created. To create a parameterized constructor, simply add parameters to it the way you would to any other function. When you define the constructor's body, use the parameters to initialize the object.
3. Copy Constructor: A copy constructor is a member function which initializes an object using another object of the same class.

Code:copy_cons.cpp

When is a user-defined copy constructor needed?

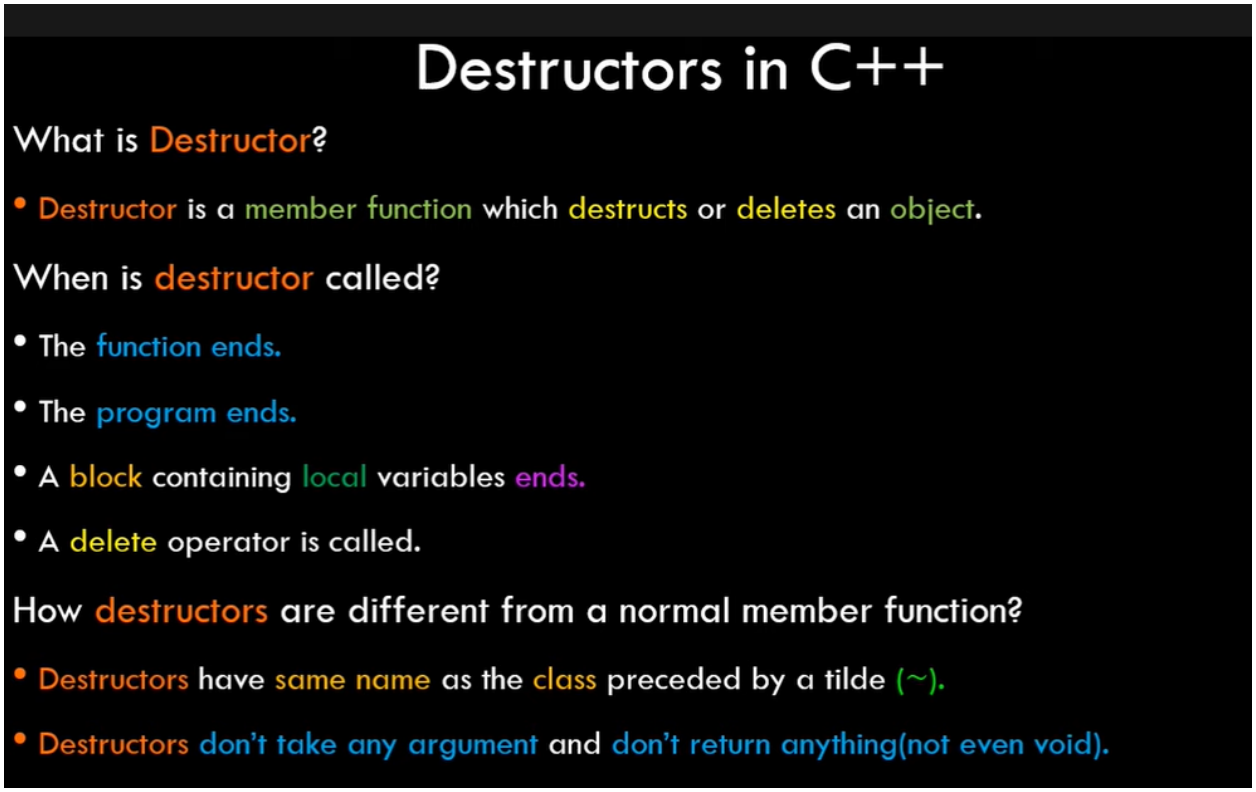
If we don't define our own copy constructor, the C++ compiler creates a default copy constructor for each class which does a member-wise copy between objects. The compiler-created copy constructor works fine in general. We need to define our own copy constructor only if an object has pointers or any runtime allocation of the resource like *file handle*, a network connection, etc.

Code: Deep_copy.cpp

Copy constructor vs Assignment Operator

The main difference between Copy Constructor and Assignment Operator is that the Copy constructor makes a new memory storage every time it is called while the assignment operator does not make new memory storage.

Which of the following two statements calls the copy constructor and which one calls the assignment operator?



A destructor is a member function that is invoked automatically when the object goes out of scope or is explicitly destroyed by a call to delete . A destructor has the same name as the class, preceded by a tilde (~).

Order of Calling of constructors and destructors in inheritance:
I. First all the constructors will call and after that all the destructors will be called in reverse order.

Example:-
Code:- Constructor_ordering.cpp

Incase of Multiple inheritance the class who is written first its constructor will call first then the second class will be called.

```cpp
// First Engineer class constructor will be called then Student class constructor
// If we change the order of inheritance i.e to 'class Nidhi : public Student, Engineer' then
// first Student constructor will be called then Engineer constructor

class Nidhi : public Engineer, Student {
public:
    Nidhi(){ cout << "Nidhi" << endl; }
    ~Nidhi(){ cout << "Nidhi DD" << endl; }
};
```

Virtual Destructor and pure virtual destructor

Destructors can be virtual but constructors can never be virtual. Destructors of the base class must be made virtual for proper destruction of the object when the program exits.

Upcasting without virtual destructor

```cpp
class Base{
  public:
    Base() { cout << "Base constructor\n; }

   ~Base() { cout << "Base Destructor\n"; }
};
class Derived : public Base{
  public:
    Derived() { cout << "Derived constructor\n" }

   ~Derived() { cout << "Derived Destructor\n"; }
};
int main(){
    Base* b = new Derived;     // Upcasting
    delete b;
}
```

```
O/P - Base constructor
      Derived constructor
      Base destructor
```

In the above example, delete b will only call the Base class destructor, which is undesirable because, then the object of Derived class remains undestructed, because its destructor is never called. Which results in memory leak.

Upcasting with virtual destructor

```cpp
class Base{

  public:
               Base() { cout << "Base constructor\n; }
      virtual ~Base() { cout << "Base destructor" << endl; }
};
class Derived : public Base{

  public:
       Derived() { cout << "Derived constructor\n" }
      ~Derived() { cout << "Derived destructor" << endl; }
};
int main(){
  Base* b = new Derived;
  delete b;
}
```

```
O/P-  Base constructor
      Derived constructor
      Derived destructor
      Base destructor
```

When we have a virtual destructor inside the base class, then derived class destructor will be called first then base class destructor will be called. And this is the desired behaviour.

Pure virtual destructors in C++

Pure Virtual Destructors are legal in C++. Also, pure virtual Destructors must be defined,
which is against the pure virtual behaviour.

The only difference between Virtual and Pure Virtual Destructor is, that pure virtual destructor will make its Base class Abstract, hence you cannot create object of that class.
There is no requirement of implementing pure virtual destructors in the derived classes.

```cpp
// C++ Program to demonstrate a pure virtual destructor
#include <iostream>
using namespace std;

// Initilization of base class
class Base {
public:
    virtual ~Base() = 0;
    // Pure virtual destructor
};

// Initilization of derived class
class Derived : public Base {
public:
    ~Derived() { cout << "~Derived() is executed"; }
};

// Driver Code
int main()
{
    // base class pointer which is
    // allocating fresh storage
    // for Derived function object's
    Base* b = new Derived();
    delete b;
    return 0;
}
```

The above code will give error :

```
test.cpp:(.text$_ZN7DerivedD1Ev[__ZN7DerivedD1Ev]+0x4c):
undefined reference to `Base::~Base()'  error: ld returned 1 exit status
```

So we need to always reinitialise the pure virtual explicitly so that it can run

```cpp
#include <iostream>

// Initilization of base class
class Base {
public:
    virtual ~Base() = 0; // Pure virtual destructor
};
Base::~Base() // Explicit destructor call
{
    std::cout << "Pure virtual destructor is called";
}

// Initilzation of derived class
class Derived : public Base {
public:
    ~Derived() { std::cout << "~Derived() is executed\n"; }
};

int main()
{
    // Calling of derived member function
    Base* b = new Derived();
    delete b;
    return 0;
}
```

```
~Derived() is executed
Pure virtual destructor is called
```

It is *important to note* that a class becomes an abstract class(at least a function that has no definition) when it contains a pure virtual destructor.

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

class Test {
public:
    virtual ~Test() = 0;
    // Test now becomes abstract class
};
Test::~Test() {
  cout<<"hii";
}

// Driver Code
int main()
{
    Test p;
    Test* t1 = new Test;
    return 0;
}
```

```
prog.cpp: In function ‘int main()’:
prog.cpp:20:7: error: cannot declare variable ‘p’ to be of abstract type
  Test p;
       ^
prog.cpp:8:7: note:   because the following virtual functions are pure wi
 class Test {
       ^
prog.cpp:13:1: note:    virtual Test::~Test()
 Test::~Test() {
 ^
prog.cpp:21:17: error: invalid new-expression of abstract class type ‘Tes
  Test* t1 = new Test;
                 ^
```

This Pointer:

- **‘this’ Pointer :** this is a keyword that refers to the current instance of the class. There can be 3 main uses of ‘this’ keyword:
    1. It can be used **to pass the current object as a parameter to another method**
    2. It can be used **to refer to the current class instance variable.**
    3. It can be used **to declare indexers.**

Code: this_ex.cpp

In java,

this vs super keyword

The this keyword points to a reference of the current class, while the super keyword points to a reference of the parent class. this can be used to access variables and methods of the current class, and super can be used to access variables and methods of the parent class from the subclass.

super keyword

1. **super** is a reserved keyword in java i.e, we can't use it as an identifier.
2. **super** is used to refer **super-class's instance as well as static members**.
3. **super** is also used to invoke **super-class's method or constructor**.

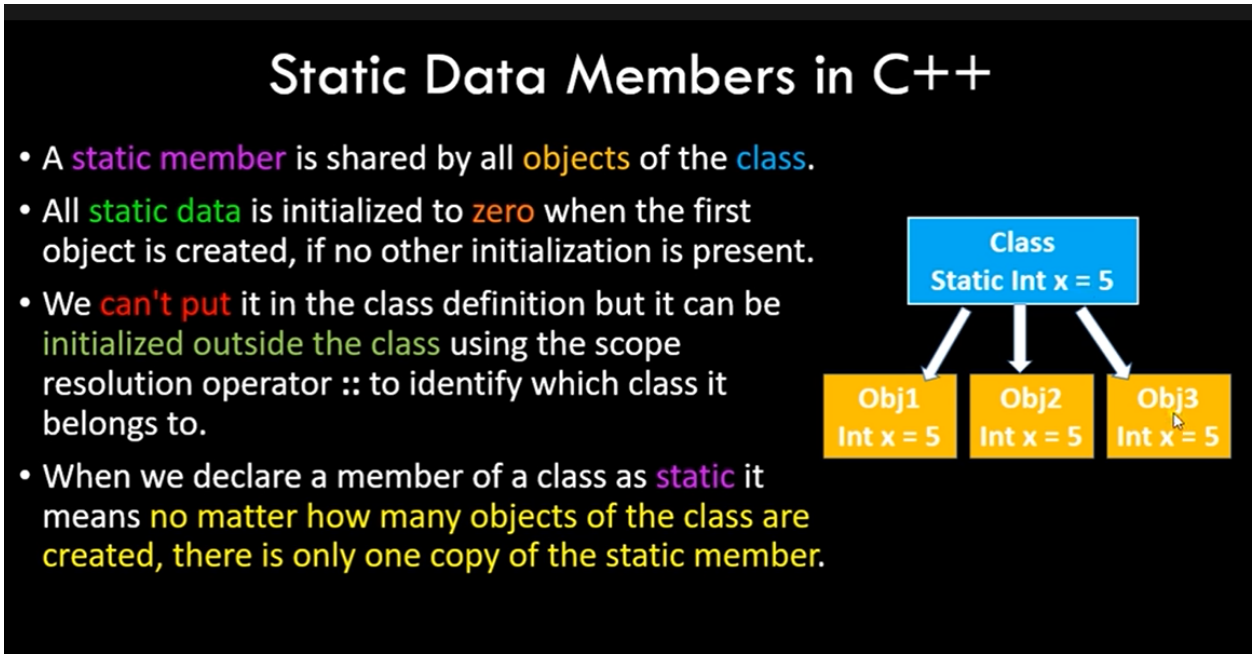
1. **Call to super() must be first statement in Derived(Student) Class constructor.**
2. **If a constructor does not explicitly invoke a superclass constructor, the Java compiler automatically inserts a call to the no-argument constructor of the superclass. If the superclass does not have a no-argument constructor, you will get a compile-time error. Object does have such a constructor, so if Object is the only superclass, there is no problem.**
3. **If a subclass constructor invokes a constructor of its superclass, either explicitly or implicitly, you might think that a whole chain of constructors called, all the way back to the constructor of Object. This, in fact, is the case. It is called constructor chaining..**

Code:

```
// Java Program to illustrate using super
// many number of times

class Parent {
        // instance variable
        int a = 36;

        // static variable
```

```
        static float x = 12.2f;
}

class Base extends Parent {
        void GFG()
        {
                // referring super class(i.e, class Parent)
                // instance variable(i.e, a)
                super.a = 1;
                System.out.println(a);

                // referring super class(i.e, class Parent)
                // static variable(i.e, x)
                super.x = 60.3f;

                System.out.println(x);
        }
        public static void main(String[] args)
        {
                new Base().GFG();
        }
}
```

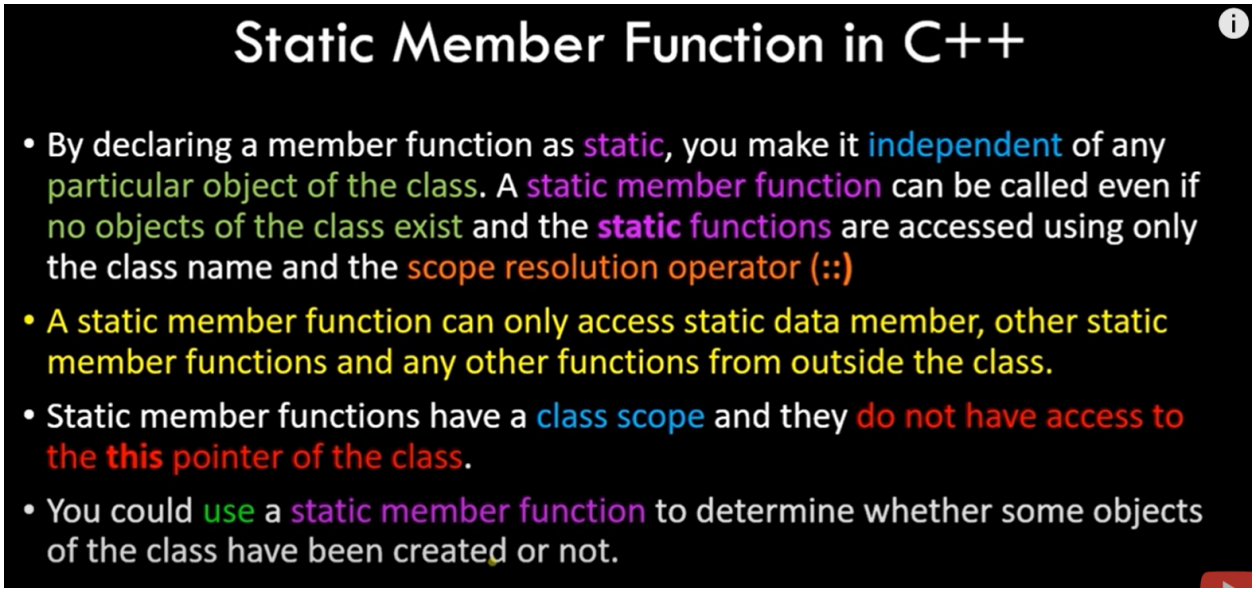
Static Members:



One main use can be:- Count how many objects are created in one class below is the example.

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

class GfG
{
  public:
    static int i;
    GfG(){
        i++;
    }

};

/ / initialization is done outside the class.
```

```
int GfG::i = 0;

int main()
{


  cout<<GfG::i<<"\n";    // it will give 0  as till no object is created
                         //  how it is not giving error as
                         //because static data member works at
                        // class level not  at  object level
  GfG obj1;
  GfG obj2;
//   obj1.i++;
//   obj2.i++;

  // prints value of i
//   cout << obj1.i<<" "<<obj2.i;


// Acces the  static variables using scope resolution operator
    cout<<GfG::i;
}
```




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

class GfG
{
  public:
    int x;
    static int i;
    GfG(){
        GfG::i++;
    }

    //static function cannot acces the non static data members
   //  static int walk(){
   //      return x;
   //  }

   //static member function
    static int walk(){
        return i;
```

```
    }
};

int GfG::i = 0;

int main()
{
  GfG obj1;
  GfG obj2;

    cout<<GfG::walk()<<"\n";
    cout<<GfG::i;
}
```

Friend Class And Friend Function

C++ Friend function

← Prev Next →

If a function is defined as a friend function in C++, then the protected and private data of a class can be accessed using the function.

By using the keyword friend compiler knows the given function is a friend function.

For accessing the data, the declaration of a friend function should be done inside the body of a class starting with the keyword friend.

Declaration of friend function in C++

```
class class_name
{
   friend data_type function_name(argument/s);          // syntax of friend function.
};
```

In the above declaration, the friend function is preceded by the keyword friend. The function can be defined anywhere in the program like a normal C++ function. The function definition does not use either the keyword **friend or scope resolution operator**.

Code:

```
#include <iostream>
using namespace std;
class Box
{
   private:
      int length;
   public:
     Box(){
         length=10;
      }

      //the class in which we want to access through friend
      //function should in the circular braces.
      friend int printLength(Box); //friend function
};

//Friend function is defined outside the class
//All the datatmembers of the Box class is accessed by '.' operator
int printLength(Box b)
{
  b.length += 10;
   return b.length;
}


int main()
{
   Box b;
   //Friend function is called as like a normal function is calledd but with
object as a parameter
   cout<<"Length of box: "<< printLength(b)<<endl;
   return 0;
}
```

Code: Friend_fun.cpp (For more clarity)

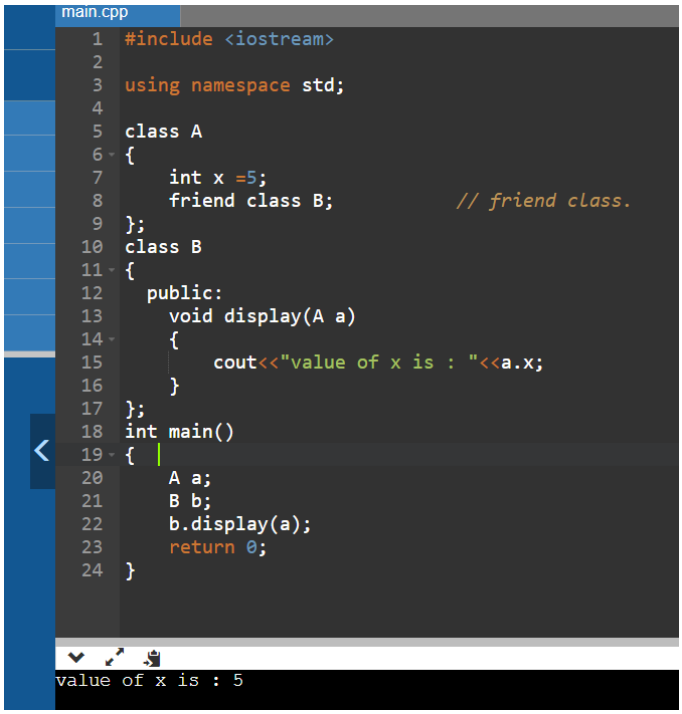
Characteristics of a Friend function:

- The function is not in the scope of the class to which it has been declared as a friend.
- It cannot be called using the object as it is not in the scope of that class.
- It can be invoked like a normal function without using the object.
- It cannot access the member names directly and has to use an object name and dot membership operator with the member name.
- It can be declared either in the private or the public part.

Friend Class:

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 members of other class. For example, a LinkedList class may be allowed to access private members of Node.

Code:



```
-----------------------------------------------------------------------
There are 2 types of relation-ships
1) Has-a   ==> Car has tires   (Aggregation)
2) Is-a   (Inheritance)  ==> Car is a vehicle
(We only derive a class when we can satisfy Is-a relation with base class)
-----------------------------------------------------------------------
```

Aggregation:

It is a process in which one class defines another class as any entity reference. It is another way to reuse the class. It is a form of association that represents the HAS-A relationship.

Code:

```
#include <iostream>
using namespace std;
class Address {
   public:
  string addressLine, city, state;
    Address(string addressLine, string city, string state)
   {
      this->addressLine = addressLine;
      this->city = city;
      this->state = state;
   }
};
class Employee
   {
      private:
      Address* address;  //Employee HAS-A Address
      public:
      int id;
      string name;
```

```
    Employee(int id, string name, Address* address)
    {
        this->id = id;
        this->name = name;
        this->address = address;
    }
   void display()
    {
        cout<<id <<" "<<name<< " "<<
         address->addressLine<< " "<< address->city<< "
"<<address->state<<endl;
    }
  };
int main(void) {
   Address a1= Address("C-146, Sec-15","Noida","UP");
   Employee e1 = Employee(101,"Nakul",&a1);
        e1.display();
  return 0;
}
```

Namespace:

- **Namespaces in C++ :**
  1. The namespace is a logical division of the code which is designed to stop the naming conflict.
  2. The namespace defines the scope where the identifiers such as variables, class, functions are declared.
  3. **The main purpose of using namespace in C++ is to remove the ambiguity.** Ambiguity occurs when a different task occurs with the same name.
  4. For example: if there are two functions with the same name such as add(). In order to prevent this ambiguity, the namespace is used. Functions are declared in different namespaces.
  5. C++ consists of a standard namespace, i.e., std which contains inbuilt classes and functions. So, by using the statement "using namespace std;" includes the namespace "std" in our program.

Problem:

```
// A program to demonstrate need of namespace
int main()
{
    int value;
    value = 0;
    double value; // Error here
    value = 0.0;
}
```

Output :

```
Compiler Error:
'value' has a previous declaration as 'int value'
```

Solution:

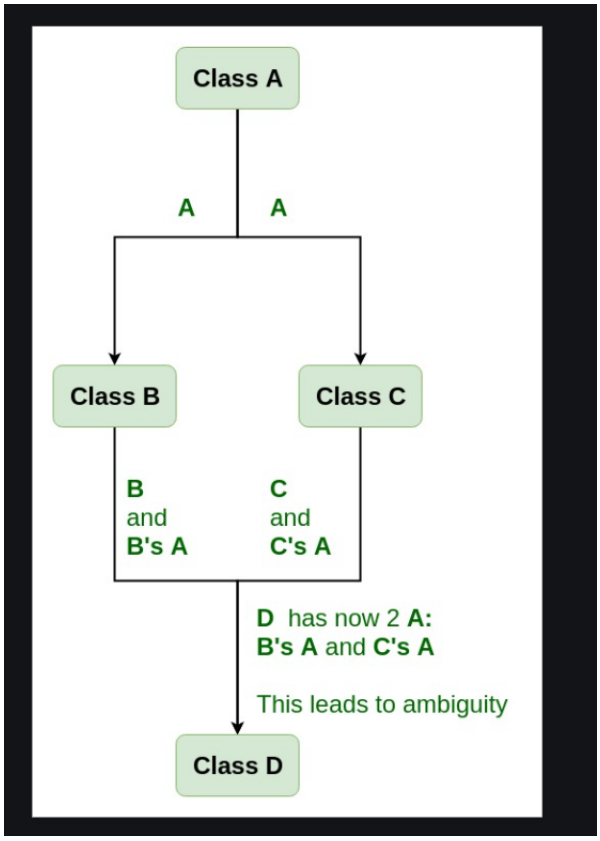
```
// Here we can see that more than one variables
// are being used without reporting any error.
// That is because they are declared in the
// different namespaces and scopes.

#include <iostream>
using namespace std;

// Variable created inside namespace
namespace first
{
   int val = 500;
}

// Global variable
int val = 100;

int main()
```

```
{
    // Local variable
    int val = 200;

    // These variables can be accessed from
    // outside the namespace using the scope
    // operator ::
    cout << first::val << '\n';
    // ::val – global variable

    return 0;
}
```

EXTRA POINTS

- **Delete** is used to release a unit of memory, **delete[]** is used to release an array.
- **Virtual inheritance** facilitates you to create only one copy of each object even if the object appears more than one in the hierarchy.

Virtual inheritance is used when there occurs a diamond problem during multiple inheritance.



As we can see that at class D we are getting two copies of A which creates ambiguity for the compiler to choose which copy .

To handle this we make virtual inheritance .

```cpp
#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
```

```cpp
#include <bits/stdc++.h>
using namespace std;

class A{
  public:
  int a;
  A(){
     a=10;
  }
};

class B: public virtual A{
   public:
   B(){
```

```
        a=5;
    }
};

class C: public virtual A{
     public:
    C(){
        a=50;
    }
};

class D: public C,public B{
    // public:
    // D(){
    //     a=15;
    // }
};

int main(){
    D d;
    cout<<d.a;
}
```

In case of function overriding

```
#include <bits/stdc++.h>
using namespace std;

class A{
  public:
  int a;
  A(){
     a=10;
  }
  void fun(){
     cout<<"A"<<"\n";
  }
```

```
};

class B: public A{
   public:
   B(){
      a=5;
   }

};

class C: public A{
    public:
   C(){
      a=50;
   }
};

class D: public C,public B{
   public:
   // D(){
   //     a=15;
   // }
    void fun(){
      cout<<"D"<<"\n";
  }
};

int main(){
   D d;
   d.fun();
}
```

Remember ambiguity occurs fro both data members and member function of the class but in case of member function if we overrides the member function in class D there will not be any ambiguity and it will work fine.

****
4) We can make a class non-inheritable by using "final" keyword.
If we declare a class as final, then we cannot inherit that class

Eg-

```
class Vehicle final{
public:
  void driveVehicle() {  cout << "Driving the vehicle" << endl;  }
};

class Car : public Vehicle{
public:
  void getTires() {  cout << "A car has 4 tires" << endl;  }
};

int main(){
Car c;
c.driveVehicle();
c.getTires();
}
```

Output- Error, cannot derive Vehicle base class (declared as final class)

##
##
Video- (https://youtu.be/WObyOa2FXwI)
How to make a class non-inheritable without using final keyword???????

Requirements-
1) We need 1 class which will make our class as final class,let's call that class "Final" class

Solution-
1) Make default constructor of final class as private

2) Inherit final class as virtual in our class which we want to make non-inheritable
3) Make our class as friend inside final class
(So that only our class an call the constructor of final class, not the derived class)

Example-

```
class Final{
private:
  Final(){}
  friend class Base;  //If not declared as friend, A base class object will not be able to call Final class constructor (Error)
};

class Base : virtual public Final{
public:
   Base(){}
};

class Derived : public Base {
public:
   Derived(){}
};

int main(){
Derived d;
}
```

O/P- We will get an error -> Because derived class object d will not be able to call the constructor of the final class
Because it is declared as private.
If we make default constructor as public, then code will work fine, but our goal is not achieved in this case.
##

THANK YOU !