C++ Inheritance

⇒ Inheritance is one of the most powerful features of object oriented programming.

⇒ it Allows one class to inherit attributes and methods from another Existing class (base class).

⇒ The new class created ~~is~~ possies (having) all the attributes of base class in addition to its additional features.

⇒ The class which inherits the members of another class is called derived class and the class whose members are inherited is called base class.

⇒ Note: in c++, inheritance is a proces in which one obj acquires all the properties and behaviours of its parent object automatically.

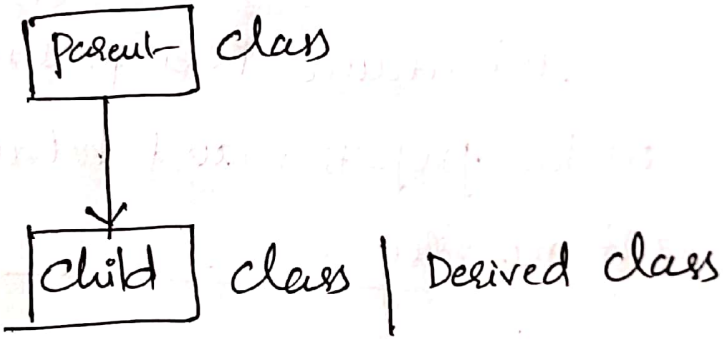
Advantage of C++ Inheritance:

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

Note: When creating a class, instead of writing completely new data members and member-functions, the programmer can designate that the new class should inherit the members of an existing class.

⇒ To inherit from a class, we use : symbol.

⇒ Inheritance is one of the features of OOPs (Object Oriented programming Systems, it allows the child class to acquire the properties (the data members) and functionality (the member f^n's) of parent class.



Advantages: When child class inherits the properties & functionality of parent class, we need not to write the same code again in child class. This makes it easier to reuse the code, makes us write the less code and the code becomes much more readable.

⇒ in simple Code reusability & Readability.

```
# include <iostream>
# define PI 3.14
Class Circle
 {
    protected:
     float radius;
    public:
      void setradius(float r);
      float getArea();
 };
void Circle :: setradius(float r)
 {
   radius = r;
 }
float Circle :: getArea()
 {
  float area = PI * radius * radius;
  return area;
 }
                          // public inheritance
Class Cylinder : Public Circle
 {
    Private:
      float height;
    Public:
       void setHeight(float h);
       float getvolume();
       void showinfo();
 };
void cylinder :: setHeight(float h)
 {
   height = h;
 }
float cylinder :: getvolume()
 {
   float area = getArea();
   float vol = area * height;
   return vol;
 }
void cylinder :: showinfo()   // here we can access radius because it is protected.
 {
   cout << "radius of base circle" << radius;
   cout << "Height of cylinde" << height;
   cout << "volume of " << getvolume();
 }
int main()
 {
   cylinder c;
   c.setradius(4);
   c.setHeight(6);
   c.showinfo();
 }
```

Example 2

```
#include<iostream>
Class Student
{
 protected:
        string name;
        int rollno;
        string course
 Public:
  void setStudentdata(string nm, int r, string c);
   void showStudentdata();
};
void Student::setStudentdata(string nm, int r, chr c)
   {
      name=nm;
      rollno=r;
      course=c;
   }
Void Student::showStudentData()
  {
  Cout<<"Name"<<name;
  Cout<<"rollno"<<rollno;
  Cout<<"Course"<<course;
}

Class Hostel : Public Student
 {
   Private:  string hostel;
             int roomno;
   Public:
      void setHostelData(string hn, int r);
      void showHostelData();
 };
Void Hostlee::setHostelData(string hn,
                            int r);
   {
      hostel=hn;
      roomno=r;
   }
Void Hostlee::showHostelData()
   {
    Cout<<"Name"<<name;
    Cout<<"rollno"<<rollno;
    Cout<<"Course"<<course
    Cout<<"Hostel Name"<<hostel;
    Cout<<"room no"<<roomno
   }
int main()
 {
    Hostlee H;
H.setStudentData("Kaushik", 525, "B.TECH");
H.setHostelData("LEPalesh", 101);
H.showHostlelData();
}
```

Example pgm

```
#include<iostream>
#include
#define
```

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

class xyz
{
    protected: int x;
               int y;

    public:
        void show();
        void display();
};

void xyz::show()
{
    cout<<"Hello";
}
void xyz::display()
{
    cout<<"mgit";
}

class pqr : public xyz
{
    private: int p;
             int q;

    public:
        void display();
        void showdata();
};

void pqr::display()
{
    cout<<"Hello mgit";
}

void pqr::showdata()
{
    cout<<"Hello Hyd";
}

int main()
{
    pqr ob1;
    ob1.show();
    ob1.display();
    ob1.showdata();
}
```

Syntax TO declare a derived class from a base class.

```
Class derived-class : Access-specifier base-class
```

↓

may be a private
public
protected.

Note: if Access-specifier is not specified then by default it is private.

Modes of Inheritance

Note: when deriving a base class, the base class may be inherited through public, protected & private modes.

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 derived class.

② **protected mode:** both public member & protected members of the base class will become protected in derived class.

③ **private mode:** Both public & protected members of the base class will become private in derived class.

Note:- A base class's private members are never accessible directly from a derived class, but can be accessed through calls to the 'public & protected members of the base class.

Ex:-

```
#include<iostream>
Class parent
 {
    private:
        int n1;
    protected:
        int n2;
        int n3;
    public:
        void show()
         {
          cout<<"The Value of N1"<<n1<<endl;
          cout<<"The Value of N2"<<n2<<endl;
         }
 };
Class child : public parent
 {
    Public:
        void input()
         {
          cout<<"enter first value";
          cin>>n1;
          cout<<"enter 2nd value";
         } cin>>n2;
 };
int main()
 {
   child ob1;
   ob1.input(); // raises error you can Access
                   private member n1
   ob1.show();  ∴ Chang n1 as protected
 }
```

```
class parent
{
    private:
    protected:  int n1;
                int n2;
    public:     void show()
                {
                cout<<"n1 value is"<<n1;
                cout<<"n2 value is"<<n2;
                }
};
class child : protected parent
{
    public:
            void input()
            {
            cout<<"enter n1 value"<<endl;
            cin>>n1;
            cout<<"enter n2 value"<<endl;
            cin>>n2;
            } ***** <- show();
};
int main()
{
child ob1;
ob1.input();
ob1.show() x   show() inAccessible
}
```

then write.

```
int main
{
child ob1;
ob1.input();
}
```

```
class parent
{
    private:
    protected:
        int n1;
        int n2;
    public : void show()
        {
            cout<<"n1 value is"<<n1;
            cout<<"n2 value is"<<n2;
        }
};

class child : private parent
{
    public:
        void input()
        {
            cout<<"enter n1 value"
            cin>>n1;
            cout<<"enter n2 value";
            cin>>n2;
            show();
        }
};

int main()
{
    child ob1;
    ob1.input();
    ob1.show(); // inAccesibl.
}
```

add

```
int main()
{
    child ob1;
    ob1.input();
}
```

⇒ We can summarize the different Accen types ⑥ according to - who can Accen them in the following way:

Access	public	protected	private
Same class	Yes	yes	Yes
Derived class	Yes	Yes	NO
Outside class	Yes	NO	NO

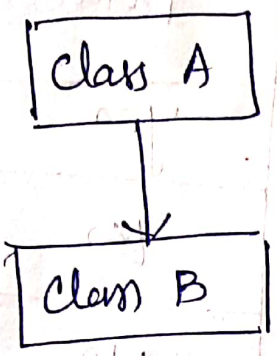
Types of Inheritance

C++ supports the following 5 types of inheritance.

① single inheritance
② multiple "
③ multi-level "
④ Hierachical "
⑤ Hybrid (Virtual) Inheritance.

① Single Inheritance:

→ In single inheritance, a class is allowed to inherit from only one class ie. one sub class is inherited by one base class only.

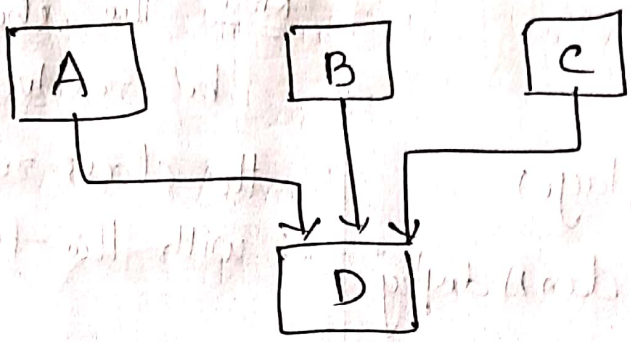


Ex:

```
Class Shape
{
    protected:  int width;
                int height;
    public:  void setdata (int w, int h){
             {
                width= w;
             } height= h;
};
Class Rectangle : public Shape
{
    public: int getArea()
            {
              return (width*height);
            }
};
int main()
{
    Rectangle obj;
    Obj. setdata ( 5,7);
    cout<<" Area of a rectangle is"<<obj.getArea();
}
```

(2) Multiple Inheritance:

→ In multiple inheritance, a class can inherit more than one class. This means that in this type of inheritance a single child class can have multiple parent classes.



Syntax:-

Class derived-class : Access-spec base A, Access-base B ...

```
class shape      // Base 1
{
  protected:
    int width;
    int height;
  public:
    void setdata(int w, int h)
    {
      width = w;
      height = h;
    }
};

class paintcost  // Base 2
{
  public:
    int getcost(int a)
    {
      return (a * 50);
    }
};

class Rectangle : public shape,
                  public paintcost
{
  public:
    int getArea()
    {
      return (width * height);
    }
};

int main()
{
  Rectangle Rect;
  int area;
  Rect.setdata(5, 7);
  area = Rect.getArea();
  cout << "total Area" << area << endl;
  cout << "total paintcost" << Rect.getcost(area);
}
```

~~Abiu~~ Ambiguity in multiple Inheritance

→ Ambiguity can be occured in multiple inheritance when a f^n with the same name occurs in more than one base class.

Ex:

```
Class A
{
    public:
        void display()
        {
            cout<<" class A display"
        }
};
Class B
{
    public:
        void display()
        {
            cout<<"class B display";
        }
};
Class C : public A, public B
{
    public : void show()
        {
            cout<<"Hello";
        }
};
int main()
{
    C c;
    c.display()
}
```

Note:

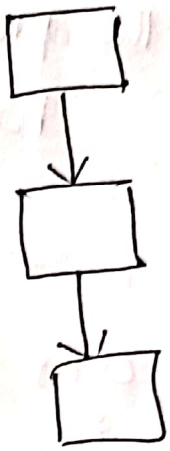
Note: The above issue can be resolved by using the class resolution operator :: with the function.

```
int main()
{
    C c;
    c.A::display();   // calls A's display f^n
    c.B::display()    // invokes B's display f^n
    c.show();
}
```

Multilevel Inheritance:

→ When one class inherits another class which is further inherited by another class is known as multilevel.

Note:- last derived class acquires All the members of all its base classes.



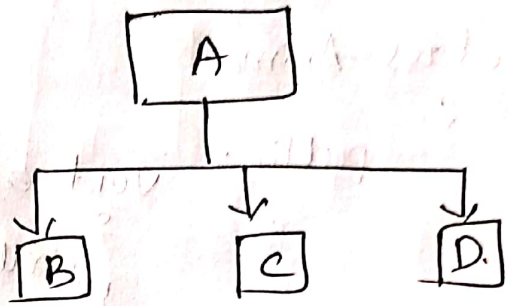
Ex:-
```
class Animal
{
    public: void eat()
            {
                cout<<"Eating"<<endl;
            }
};
class Dog : public Animal
{
    public: void bark()
            {
                cout<<"Barking"<<endl;
            }
};
class BabyDog : public Dog
{
    public: void weep()
            {
                cout<<"weeping";
            }
};
int main()
{
    Babydog d1;
    d1.eat();
    d1.bark();
    d1.weep();
} return 0;
```

Hierarchical Inheritance:

→ In this type of Inheritance, one parent class has more than one child class

(or)

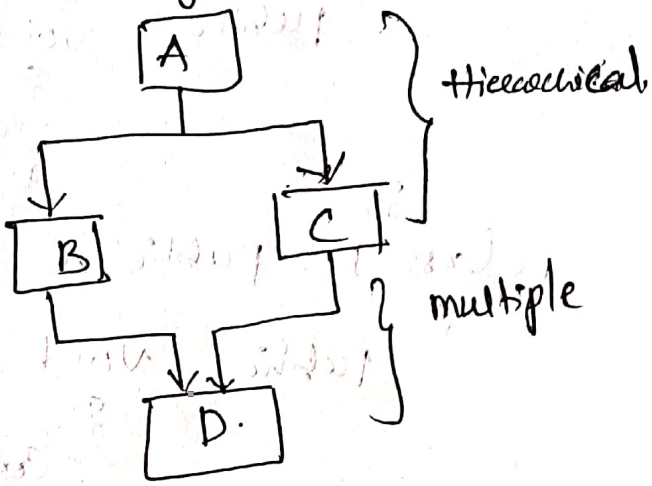
⇒ The process of deriving more than one class from a single base class.



Ex:-
```
class shape
{
    public: int width;
            int height;
    void getdata(int w, int h)
    {
        width = w;
        height = h;
    }
};

class Rectangle : public shape
{
  public:
    int rect_area()
    {
        return (width * height);
    }
};

class Triangle : public shape
{
  public:
    int triangle_area()
    {
        return (0.5 * width * height);
    }
};

int main()
{
  Rectangle r;
  Triangle t;
  r.getdata(5,7);
  cout << "Area of rect is" << r.rect_area();
  t.getdata(8,10);
  int length, breadth, base, height;
  cout << "enter length & breadth";
  cin >> length >> breadth;
  r.getdata(length, breadth);
  cout << r.rect_area();
  cout << "enter base & length";
  cin >> base >> length;
  t.getdata(base, length);
  cout << t.Triangle_area();
}
```

Hybrid Inheritance

→ Combination of more than one type of Inheritance.



Ex:

```
class A
{
    protected:
        int a;
    public:
        void getac)
        {
            cout<<"enter value for a";
        }cin>>a;
};
class B: public A
{
    protected: int b;
    public:
        void getbc)
        {
            cout<<"enter value for b";
        }cin>>b;
};
class C
{
    protected:
        int c;
    public:
        void getcc)
        {
            cout<<"enter c";
        }cin>>c;
};
class D : public B, public C
{
    protected: int d;
    public:
        void mulc)
        {
            getac);
            getbc);
            getcc);
            cout<<"mul of a,b,c is"<<a*b*c;
        }
};
int main()
{
    D d;
    d.mulc);
    return 0;
}
```

Conflict Resolution in Accessing class members

Ex:-
```
class A
{
    public: void display()
            {
                cout<<" A's display";
            }
};
class B: public A
{
    public: void display()
            {
                cout<<"B s display";
            }
};
int main()
{
    B ob;
    ob1.display(); // B's display.
    ob1.A::display() // calls A's display.
}
```

Note:- By default, any reference to a fn from the derived class will invoke the fn in the derived class.

Note:- we can use scope resolution operator (::) to invoke the fn of our choice.

Constructors in Inheritance | Order of Constructor Call with Inheritance:

⇒ When we create an obj of a class, the default constructor of that class is invoked automatically to initialize the members of the class.

⇓

Note: if we inherit a class from another class and create an obj of the derived class, it is clear that the default constructor of the derived class will be invoked but before that the default constructor of the base classes will be invoked.

i.e. The order of invokation is that the base class default constructor will be invoked first and then the derived class default constructor will be invoked.

Ex:-
```
class parent
{
    public:
        parent()
        {
            cout<<"Inside parent class";
        }
};
class child : public parent
{
    public: child()
            {
                cout<<'Inside the child';
            }
};
int main()
{
    child obj;
}
```

O/P: inside parent class
inside child.

Note: Whether derived class's default or parameterized constructor is called, base class's default constructor is always called inside them.

Ex:-
```
class Base
{
    int x;
    public:  Base()
             { cout<<"Base default constructor"; }
};
class Derived : public Base
{
    int y;
    public: Derived()
            { cout<<"Derived default constructor"; }
            Derived(int i)
            { y=i;
              cout<<"Derived parameterized constructor"; }
};
int main()
{
    Base b;
    Derived d1;
    Derived d2(10);
}
```

o/p

Base default constructor
Base " "
Derived " "
Base default constructor
Derived parameterized "

Note: When a class is inherited from other – The data members and member functions of base class comes automatically in derived class based on the Access specifiers. but the definition of these members Exist in base class only. So

⇓

when we create obj of derived class, all of the members of ~~derived~~ base class must be initialized.

Note:- TO call base class's parameterized constructor inside derived classes parameterized constructor, we must mention it explicitly while declaring derived class's parameterized constructor.

Ex:-
```
class Base
  {
    int x;
    public:
        Base (int i)
          {
            x=i;
            cout<<"Base parametrized";
          }
  };
class Derived : public Base
  {
    int y;
    public: Derived (int j): Base(j)
          {
            y=j;
            cout<<" Derived parameterized";
          }
  };
int main()
  {
    Derived d(10);
  }
```

or

```
Derived (int j): Base(100)
  {
  }
```

O/P: Base parameterized constructor
Derived " "

Note: The parameterized constructor of base class can not be called in default constructor of sub class, it should be called in the parameterized constructor of sub class.

Order of Constructor Call in multiple Inheritance

Note:- The base class's constructors are called in the order of inheritance and then the derived class's constructor is called.

```
class parent1
{
public:
        parent1()
        {cout<<"Inside parent1";
        }
};
class parent2
{
public:
        parent2()
        {cout<<"Inside parent2"
        }
};
class child: public parent1, public parent2
{
public:
        child()
        {cout<<"Inside child";
        }
};
int main()
{
child ob;
}
```

o/p: inside parent1
inside parent2
inside child.

```
class parent1
{
public:
        parent1()
        {cout<<"Inside parent1";
        }
};
class parent2
{
public:
        parent2()
        {cout<<"Inside parent2"
        }
};
class child: public parent2, public parent1
{
public:
        child()
        {cout<<"Inside child";
        }
};
int main()
{
child ob1;
}
```

o/p:
inside parent2
inside parent1
inside child.

Destructors in Inheritance:

→ Destructors are called in the opposite order of that of Constructors.

Note:- When the object of a derived class expires first the derived class destructor is invoked followed by the base class destructor.

Ex:-
```
class Base
{
    public:
        Base()
        { cout<<" Base constructor";
        }
        ~Base()
        {
          cout<<" Base destructor";
        }
};
class derived1 : public Base
{
    public:
        derived1()
        {
          cout<<" constructor of derived1";
        }
        ~derived1()
        {
          cout<<" Destructor of derived1";
        }
};
class derived2 : public derived1
{
    public:
        derived2()
        {
          cout<<" Constructor of derived2"
        }
        ~derived2()
        {
          cout<<"destructor of derived2";
        }
};
int main()
{
  derived2 obj;
}
```

Op:
Base constructor
Constructor of derived1
Constructor of derived2
Destructor of derived2
Destructor of derived1
Destructor of Base

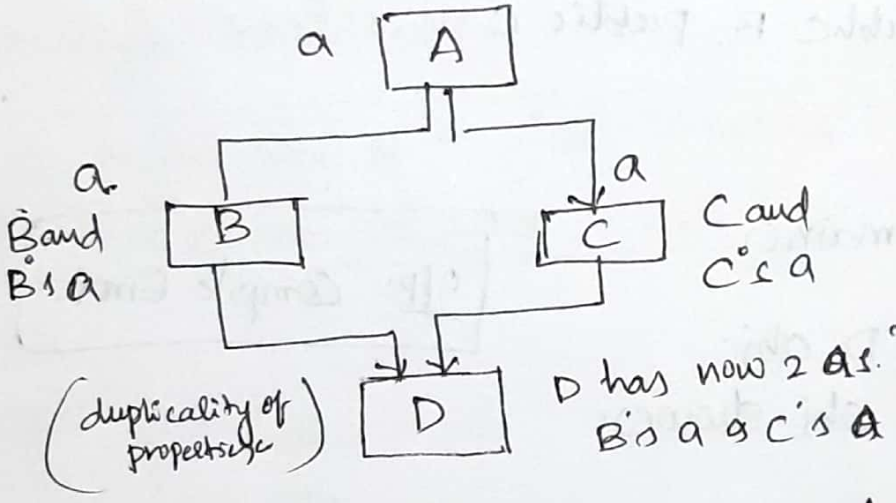
Virtual Base classes in C++

⇒ Hybrid (virtual) inheritance is a combination of hierachical & multilevel inheritance.

(or)

Hybrid inheritance is implemented by combining more than one type of inheritance.

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



⇒ As we can see from the figure that data members/function of class A are inherited twice to class D. (ie one through B second through class C).

⇓

when any data/function member of class A is accessed by an obj of class D, ambiguity arises as to which data/function member would be called?

⇒ This confuses compiler and it displays error and it displays error.

Ex:-
```
class A
{
    public: void show()
    {
        cout<<" hello I am from A";
    }
};

class B : public A
{
};

class C : public A
{
};

class D : public B, public C
{
};

int main()
{
    D obj;
    obj.show();
}
```

O/P: Compile Error

⇒ duplication of inherited members / member f^n's due to multiple paths can be avoided by making the common baseclass as virtual baseclass.

Syntax for virtual Base classes

```
class B : virtual public A
{
};

class C : public virtual A
{
};
```

(or)

```
class B : public virtual A
{
};
```

Class D: public B, public C
{
≡
}

⇑ Now only one copy of A will be inherited

⇒ When A class is made as virtual Base class, C++ compiler takes necessary action to see that only one copy of the class is inherited regardless of how many Inheritance paths Exist.

Note: virtual Base classes offer a way to save space and avoid ambiguities in class hierarchies that use multiple inheritances.

Note: A single copy of its data members is shared by all the base classes that use virtual base.

```
class A
{
  public: void show()
          {
            cout<<"I am from A";
          }
};
class B : public virtual A
{
};
class C: public virtual A
{
};
class D: public B, public C
{
};
    int main()
    {
      D obj;
      obj.show();
    }
```

O/P: I am from A

Polymorphism in C++

⇒ The term "polymorphism" is the combination of "poly" + "morphs" which means many forms.

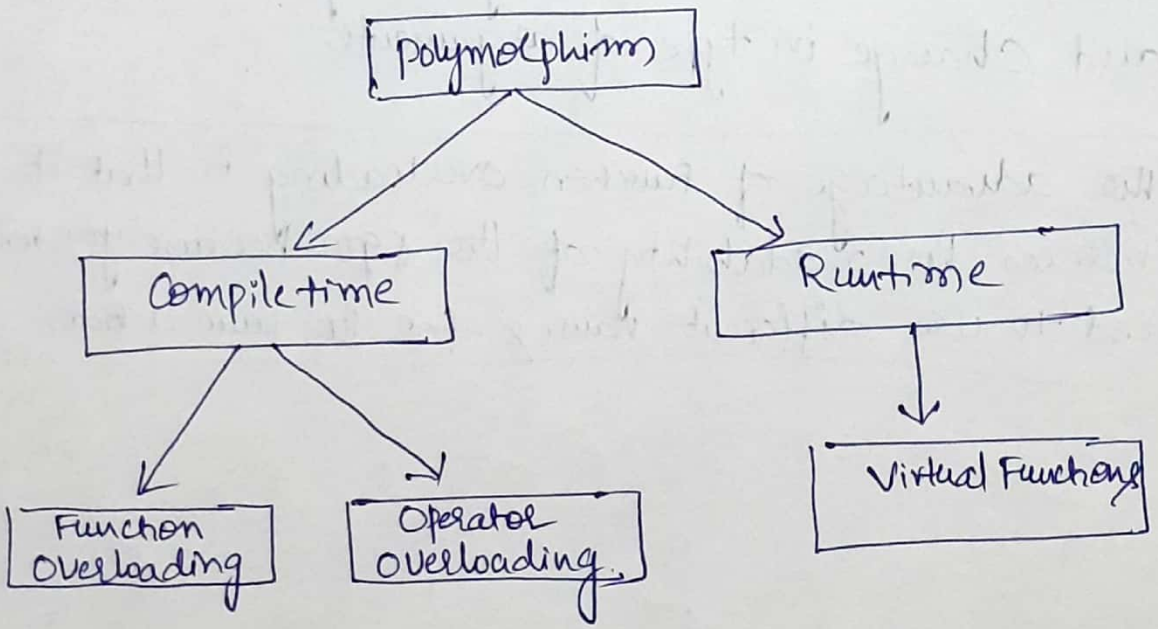
⇒ it is a greek word.

⇒ in object-oriented programming it is one of the important feature/concept.

Example: A lady behaves like a teacher in a classroom, mother at home and customer in a super market. here, a single person is behaving differently according to the situations.

⇒ In C++ polymorphism is mainly divided into two types:

1. Compile time polymorphism
2. Runtime polymorphism



⇒ Compiletime polymorphism is achieved by function overloading & operator overloading.

⇒ Runtime polymorphism is achieved by Function Overriding.

① Compiletime polymorphism:

→ it is achieved by function overloading and operator overloading.

→ The overloaded functions are invoked by matching the type and no. of arguments.

→ compiler selects the appropriate f^n at the compile time.

→ which is also known as static binding or early binding

Function overloading: when there are multiple f^ns with same name but different parameters then these f^ns are said to be overloaded.

⇒ Functions can be overloaded by change in no. of arguments and/or change in type of arguments.

Note:- The advantage of Function overloading is that it increses the readability of the pgm because you don't need to use different names for the same action.

Ex:-

```
class foreload
{
  public: void add(int a, int b)
          {
            cout<<"sum is"<<a+b<<endl;
          }
          void add(int a, int b, int c)
          {
            cout<<"sum is"<<a+b+c<<endl;
          }
};
int main()
{
  foreload ob1;
  ob1.add(10,20);        // 30.
  ob1.add(10,20,30);     // 60
  ob1.add(10.5,20.345);  // 30
  ob1.add(23,34.67);     // 57
  ob1.add(10.345,23);    // 33
}
```

Example:

```
class xyz
{
public:
  void fun(int i)
  {
    cout<<i<<endl;
  }
  void fun(float j)
  {
    cout<<j<<endl;
  }
};
int main()
{
  xyz ob1;
  ob1.fun(12);
  ob1.fun(12.5);
  return 0;
}
```

Note: the pgm shows an error "call of overloaded fun(double) is ambiguous"

⇒ ob1.fun(12) → calls the first function

ob1.fun(12.5) → not having corresponding fun.

Note:- in C++, all the floating point constants are treated as double not as a float.

if we replace ~~float~~ float to double in 2nd function, the pgm works.

```
void fun(double j)
{
  cout<<j<<endl;
}
```

(so replace float → double; 12.5 treated as double)

Note:- Function overloading Ambiguity:

⇒ When the compiler is unable to decide which f^n is to be invoked among the overloaded f^n's, known as f^n overloading ambiguity.

Causes of Function overloading ambiguity:

1. Type Conversion
2. Function with default arguments.
3. Function with pass by reference.

① Type Conversion:

```
Class foverload
{
    public: void add(int a, int b)
            {
              cout<<" sum is "<<a+b;
            }
            void add (int a, int b, int c)
            {
              cout<<" sum is "<<a+b+c;
            }
            void add (double a, double b)
            {
              cout<<" sum is "<<a+b<<endl;
            }
};
int main()
{
    foverload ob1;
    ob1.add(10,20);
    ob1.add(10,20,30);
    ob1.add(10.5, 20.345)
    ob1.add(10, 34.89); //X ambiguous
    ob1.add(23.67, 34); //X ambiguous,
}
```

Example for Function with default arguments

```
class xyz
{
    public:
        void fun(int i)
        {
            cout<<" i value is "<<i<<endl;
        }
        void fun(int a, int b=20)
        {
            cout<<" value of a is "<<a<<endl;
            cout<<" value of b is "<<b<<endl;
        }
};
int main()
{
    xyz ob1;
    ob1.fun(12); // ambiguity
    return 0;
}
```

Runtime polymorphism | Dynamic polymorphism

→ it is achieved by method overriding which is also known as dynamic binding & late binding.

⇒ in method overriding where more than one method is having the same name, no. of parameters and the type of the parameters.

⇒ which fn is to be invoked is known at the runtime.

⇒ it is achieved by virtual functions concept & pointers/references in C++.

Method overriding Example:

```
class Base
{
  public:
    void display()
    {
      cout << "base display";
    }
};
class Derived : public Base
{
  public:
    void display()
    {
      cout << "Derived display";
    }
};
```

```
int main()
{
  Derived ob1;
  ob1.display();
}
```

Always display derived display only.

```
int main()
{
  Base ob1;
  ob1.display(); // base display
  Derived ob2;
  ob2.display(); // derived "
}
```

display() invoking is decided by object invocation.

How to call overridden f^n from the child class:

→ This can be done by Assigning the object of child class to the reference of parent class.

```
class Base
  {
    public:
      void display()
        {
          cout<<"I am in base display"
        }
  };
class Derived : public Base
  {
    public: void display()
              {
                cout<<" Derived display";
              }
  };
int main()
  {
    Base *ptr;
    Derived obj;
    obj.display(); // Derived display
    ptr = &obj;
    ptr -> display(); // Base's display()
    return 0;
  }
```

Note: But when base class pointer contains the Address of the derived class obj, always Executes the base class f^n.

Virtual Functions in C++

⇒ There is a nessity to use the single pointer to refer to all the objects of the different classes. So, we create the pointer to the base class that refers to all the derived objects.

⇓

But, when base pointer contains the address of the derived class obj, always executes the base class function only.

⇒ This issue can only be resolved by using the 'virtual' function.

⇒ A virtual function is a member fⁿ in the base class that should be overridden in the derived class.

⇒ we use 'virtual' keyword to make a member fⁿ of the base class virtual.

⇒ it is used to tell the compiler to perform dynamic linkage or late binding on the function.

⇒ when the function is made virtual, C++ determines which fⁿ is to be invoked at the runtime based on the type of the object pointed by the base class pointer.

Late binding or Dynamic Linkage:

In late binding fⁿ call is resolved during runtime. Therefore compiler determines the type of obj at runtime, and then binds the fⁿ call.

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

class Derived1 : public Base
{
  public:
    void show()
    { cout<<" Derived1 class show"; };
};

class Derived2 : public Base
{
  public:
    void show()
    {
      cout<<"Derived2 class show";
    }
};

int main()
{
  Base *bptr;
  Derived1 d1;
  Derived2 d2;
  bptr = &d1;
  bptr->show(); // o/p: Derived1 class show
  bptr = &d2;
  bptr->show(); // Derived2 class show
}
```

Rules of virtual Functions

1. virtual functions must be members of some class
2. virtual " can not be static members.
3. They are accessed through object pointers.
4. They can be a friend of another class.
5. A virtual f^n must be defined in the base class, even though it is not used.
6. we cannot have a virtual Constructor, but we can have a virtual destructor.

Virtual function call mechanism

or

Working of virtual functions: (VTABLE & VPTR)

```
class Base
{
  public:
    void fun1() { cout << "base-fun1"; }

    virtual void fun2()
    {
      cout << "base-fun2";
    }
    virtual void fun3()
    {
      cout << "base-fun3";
    }
    virtual void fun4()
    {
      cout << "base-fun4";
    }
};
class Derived : public Base
{
  public:
    void fun1()
    {
      cout << "derived-fun1";
    }
    void fun2()
    {
      cout << "derived-fun2";
    }
    void fun4(int a)
    {
      cout << "Derived-fun4";
    }
};
```

```
int main()
{
  Base *ptr;
  Derived obj;
  ptr = &obj;
  ptr->fun1(); // base-fun1 because fun1() is non virtual in base
  ptr->fun2(); // derived-fun2
  ptr->fun3(); // base-fun3
  ptr->fun4(); // base-fun4
```

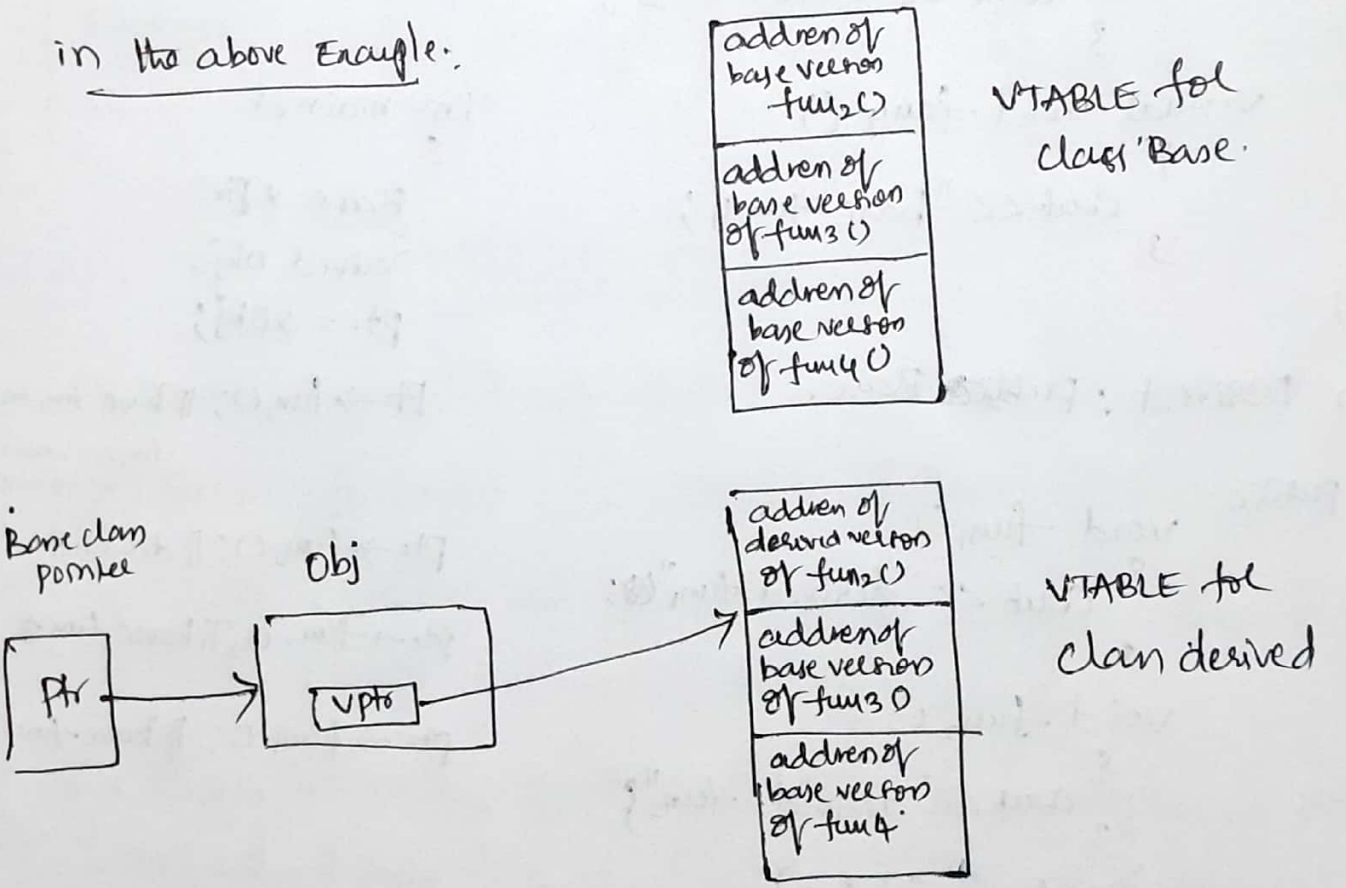
Compiler itself does two things:

* irrespective of object is created or not, a static Array of function pointer called VTABLE where each cell contains the address of each virtual function contained in that class.

* if object of that class is created then a virtual pointer (VPTR) is inserted as a data member of the class to point to VTABLE of that class.

⇓ Note: for each new obj created, a new virtual pointer is inserted as a data member of that class.

in the above Example:



Pure Virtual Functions in C++

⇒ Pure Virtual Functions are virtual-f^n's with no definition.

⇒ When the function has no definition, such-f^n is known as "do-nothing" function.

⇒ do-nothing function is known as pure virtual function.

⇒ A pure virtual function is a f^n declared in the base class that has no definition relative to the base class.

⇒ We declare a pure virtual function by assigning 0 in declaration.

Syntax: virtual returntype f^n name () = 0;

ABSTRACT class

⇒ A class with at least one pure virtual function & abstract function is called abstract class.

⇒ We can't create an object for an abstract class because it has partial implementation of methods.

⇒ The class inheriting the Abstract class must implement abstract function | pure virtual f^n's of the Abstract class.

⇒ We can have pointers | references of Abstract class type.

⇒ if we do not override the pure virtual f^n in derived class, then derived class also becomes abstract class.

⇒ Abstract classes can not be Instantiated.

Ex:-
```
class Shape    // Abstract class
{
  public:
    virtual void getArea() = 0; // pure virtual functions
};
class Circle : public Shape
{
  public: void getArea()
          { int r;
            cout << "enter the circle radius";
            cin >> r;
            cout << "Area of a circle is:" << (3.14 * r * r);
          }
};
class Rectangle : public Shape
{
  public:
    void getArea()
      { int l, b;
        cout << "enter length & breadth of a Rectagle";
        cin >> l >> b;
        cout << "Area of rectagle is" << (l * b);
      }
};
int main()
{
  Shape s1; // you can't create object for shape.
  Circle c1;
  c1.getArea();
  Rectagle r1;
  r1.getArea();
}
```

Virtual Destructors in C++

* Lets first Examine the Example what happens when we do not have a Virtual Base class destructor.

```
Class Base
{
    public:
        Base()
        { cout<<" Constructing base class\n"; }
        ~Base()
        { cout<<" Destructing base class"; }
};
Class Derived: public Base
{
    public:
        Derived()
        { cout<<" Constring derived class"; }
        ~Derived()
        { cout<<" Destructing Derived class"; }
};
int main()
{
    Derived *d = new Derived();
    Base *b = d;
    delete b;
    return 0;
}
```

O/p: Constructing base class
constructing derived class
Destring base class.

Note: 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 since its destructor is never called.

⇒ TO correct this behaviour, the base class should be defined with a virtual destructor.

correct one is as follows:

```
class Base
{
    public:
        Base()
        {
            cout<<"constructing base";
        }
        virtual ~Base()
        {
            cout<<"Destructing base";
        }
};
class Derived : public Base
{
    public:
        Derived()
        {
            cout<<"constructing derived";
        }
        ~Derived()
        {
            cout<<"Destructing derived";
        }
};
int main()
{
    Derived *d = new Derived();
    Base *b=d;
    delete b;
    return 0;
}
```

O/p: Constructing base
constructing derived
Destructing derived
Destructing Base.

Note:- when we made virtual inside the base class, then first derived class's destructor is called and then Base class's destructor is called — which is a desired behaviour.