Types Of Inheritance
C++ supports five types of inheritance:

Single inheritance
Multiple inheritance
Hierarchical inheritance
Multilevel inheritance
Hybrid inheritance

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

```
#include <iostream>
using namespace std;
 class Account {
   protected:
   float bonus = 2000;
 };
   class Programmer: public Account {
   public:
   float monthly_salary = 5000;  //protected:float bonus
   void CalculateSalary()
   {
       cout<<"Total Salary: "<<monthly_salary+bonus<<endl;
   }
   };
int main() {
     Programmer p1;
     p1.CalculateSalary();

    return 0;
}
```

==========

NOTE 1
While creating an object of the derived class, the base class constructor is called first and then the derived class constructor is called. The base class constructor is responsible for initializing the inherited data members and the derived class constructor is responsible for initializing data members of derived class.

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

Derived object has two partss: a base part and a derived part.
The base part of derived object is constructed first. then the derived part is constructed.
therefore the constructor calls are in the order of base->derived.

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

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

CASE 1:

```
/*#include<iostream>
using namespace std;
class A
{
        int a;
public:
        A();
   void display();
 };
A::A()
{
        cout<<"in default of A\n";
        a=10;
}
void A::display()
{
        cout<<a<<endl;
}
class B:public A
{
        int b;
public:
        B();
        void display();
};
```

```
B::B()
{
        cout<<"in default of B\n";
        b=20;
}
void B::display()
{
        A::display();
        cout<<b<<endl;
}
int main()
{
        B bobj;
        bobj.display();
}
```

O/P====>in default of A
in default of B
10 20

CASE 2:

```
#include<iostream>
using namespace std;
class A
{
        int a;
public:
        A();
   void display();
 };
A::A()
{
        cout<<"in default of A\n";
        a=0;
}
void A::display()
{
        cout<<a<<endl;
}
class B:public A
{
        int b;
public:
        B(int,int);
```

```
        void display();
};
B::B(int p,int q)
{
        cout<<"in para of B\n";
        b=q;
}
void B::display()
{
        A::display();
        cout<<b<<endl;
}
int main()
{
        B bobj(10,20);
        bobj.display();
}
```

```
in default of A
in para of B
0
20
```

CASE 3:

```
#include<iostream>
using namespace std;
class A
{
        int a;
public:
        A(int);
   void display();
 };
A::A(int p)
{
        cout<<"in para of A\n";
        a=p;
}
void A::display()
{
        cout<<a<<endl;
}
```

```
class B:public A
{
        int b;
public:
        B(int,int);
        void display();
};
B::B(int p,int q):A(p)//base class initilization list
{
        cout<<"in para of B\n";
        b=q;
}
void B::display()
{
        A::display();
        cout<<b<<endl;
}
int main()
{
        B bobj(10,20);
        bobj.display();
}
in para of A
in para of B
10
20
```

Important Points:

Whenever the derived class's default constructor is called, the base class's default constructor is called automatically.
To call the parameterized constructor of base class inside the parameterized constructor of sub class, we have to mention it explicitly.
The parameterized constructor of base class cannot be called in default constructor of sub class, it should be called in the parameterized constructor of sub class EXCEPT PASSING CONSTANT VALUE FROM SUBCLASS DEFAULT CONSTRUCTOR TO SUPER CLASS PARAMETRIZED CONSTRUCTOR

```
#include<iostream>
using namespace std;
class A
{
```

```
	int a;
public:
	A(int);
 };
A::A(int p)
{
	cout<<"in para of A\n";
	cout<<p<<endl;
}
class B:public A
{
	int b;
public:

	B();
};
B::B() :A(10)
{
	cout<<"in default of B";
}

int main()
{
	B bobj;
}


==================================================
#include<iostream>
using namespace std;
class A
{
	int a;
public:
	A(int);
 };
A::A(int p)
{
	cout<<"in para of A\n";
	cout<<p<<endl;
}
class B:public A
{
```

```
        int b;
public:

        B();
};
B::B() :A(p)//error
{
        cout<<"in default of B";
}


int main()
{
        B bobj;

}
```

==========
==========
==========

Multilevel Inheritance

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

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

```
class A // base class
{
   .....
};
class B : access_specifier A // derived class
{
   .....
} ;
class C : access_specifier B // derived from derived class B
{
   .....
} ;
```

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

```
#include<iostream>
using namespace std;
  class A
{
    public:
        A()
        {
            cout << "Base class A constructor \n";
        }
        ~A()
        {
            cout << "Base class A destructor \n";
        }
};

// Derived class B
class B: public A
{
    public:
        B()
        {
            cout << "Class B constructor \n";
        }
                    ~B()
        {
            cout << " class B DEstructor \n";
        }
};

// Derived class C
class C: public B
{
    public:
        C()
        {
            cout << "Class C constructor \n";

        }
        ~C()
        {
            cout << "Class C DEstructor \n";

        }
```

```
};

// Driver code
int main()
{
    C obj;
    return 0;
}



==========================================================================
class employee
{
        int id;
public:
        employee();
        employee(int);
     void display();
          int findsalary()
        {
                return 0;
        }
};
employee::employee()
{
        cout<<"in default of emp\n";
        id=0;
}
employee::employee(int i)
{
        cout<<"in para of emp\n";
        id=i;
}
void employee::display()
{

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

class wageemployee:public employee
{
        int hrs,rate;
```

```
public:
        wageemployee();
        wageemployee(int,int,int);
         void display();
int findsalary();
};
wageemployee::wageemployee()
{
        cout<<"in default of wage\n";
        hrs=0;
        rate=0;
}
wageemployee::wageemployee(int i,int h,int r)        :employee(i)
{
        cout<<"in para of wage\n";
        hrs=h;
        rate=r;
}
int wageemployee::findsalary()
{
        return hrs * rate;
}
void wageemployee::display()
{
        employee::display();
        cout<<hrs<<endl;
        cout<<rate<<endl;
}
class salesmanager:public wageemployee
{
        int sales,comm;
public:
        salesmanager();
        salesmanager(int,int,int,int,int);
        void display();
        int findsalary();
        void show();
};
salesmanager::salesmanager()
{
        cout<<"in default of sales\n";
        sales=comm=0;
}
```

```
salesmanager::salesmanager(int i,int h,int r,int s,
        int c):wageemployee(i,h,r)
{
        cout<<"in para of sales\n";
        sales=s;
        comm=c;
}
void salesmanager::display()
{
        wageemployee::display();
        cout<<"sales of an emp is "<<sales<<endl;
        cout<<"comm of an emp is  "<<comm<<endl;
}




int salesmanager::findsalary()
{
        return wageemployee::findsalary()  + sales * comm;
}
void salesmanager::show()
{
        cout<<"in show fun\n";
}
int main()
{
        salesmanager sm1(101,10,500,234567,1);
        cout<<"salary is "<<ptr->findsalary();
        ptr->display();

}
```

==========

Destructors in C++ are called in the opposite order of that of Constructors.

Polymorphism

polymorphism is made up of 2 words poly means many and morphism means forms, so polymorphism means many forms.

it means identically " one name " named methods(member functions) that have different behaiour means one methodss have multiple forms.

Same message is pass to inherited classes and this classes will respond to the same message in different ways is called as polymorphism

Binding is the process of associating a function call to an object.
Different types of polymorphism are as follows
1)compile time polymorphism:
When binding occurs at compile time,it is known as compile time binding. All the methods are called on object at compile time. In compile-time polymorphism, the compiler determines which function or operation to call based on the number, types, and order of arguments.
This kind of polymorphism is implemented using function overloading and operator overloading.such polymorphism is called as early binding or static binding because an object is bound to its function call at compile time.
==>At compile time,compiler will check the type of pointer rather than the type of data.

2)Runtime polymorphism:
When the binding process occurs at runtime ,it is called as runtime binding.
In order to invoke the appropriate function of the derived class,the compiler needs to bind that function call to the correct function defination.The decision of which function to be invoked is taken at a later stage(at runtime). Hence this feature is referred to as runtime binding.The generic pointer is capable of pointing to any object at runtime. hence it can invoke any function dynamically depending on the type of the object that it is pointing to . In run-time polymorphism, the decision of which function to call is determined at runtime based on the actual object type rather than the reference or pointer type. It is also known as Dynamic Polymorphism
Run Time Polymorphism can be exhibited by:

Virtual Function
Virtual Function is a member function that is declared as virtual in the base class and it can be overridden in the derived classes that inherit the base class.

Difference Between Compile Time And Run Time Polymorphism
Compile-Time Polymorphism and Run-Time Polymorphism

1)It is also called Static Polymorphism.
It is also known as Dynamic Polymorphism.
2)In compile-time polymorphism, the compiler determines which function or operation to call based on the number, types, and order of arguments.
In run-time polymorphism, the decision of which function to call is determined at runtime based on the actual object type rather than the pointer type.
3)The function to be invoked is known at the compile time.
The function to be invoked is known at the run time.
4)Compile-time Polymorphism can be exhibited by: Function Overloading and Operator Overloading
Run-time Polymorphism can be exhibited by Function Overriding.
5)Faster execution rate.
Comparatively slower execution rate.
6)Inheritance in not involved.

Involves inheritance.
==========
Virtual Function

==>To implement late binding, the function is declared with the keyword virtual in the base class. usually, when there is a possibility that a function in the base class may be overridden in the derived class this function is marked as virtual.
The function in the derived class is overridden with same signature and return type.
==>Polymorphism relies on runtime binding.To achieve runtime binding virtual keyword should be used.virtual keyword will tell the compiler to do binding at runtime and not at compile time.
==>It tells the compiler to perform late binding where the compiler matches the object with the right called function and executes it during the runtime. This technique falls under Runtime Polymorphism.

1)virtual function is a member function that can be redefined for the derived classes.
2)The function in the base class,which are overridden in the subsequent derived classes, are also considered virtual by the compiler.virtual function is redefined in all the derived classes even if the virtual keyword is not written.
3)constructors can not be made Virtual Function
4)destructor can be made Virtual Function
5)virtual keyword should not be written in definition

**At runtime binding the compiler will check the type of data rather than the type of pointer.According to the type of data the compiler will invoke the function of that class.
**The idea is that virtual functions are called according to the type of the object instance pointed to or referenced, not according to the type of the pointer or reference.
In other words, virtual functions are resolved late, at runtime.

==========
==========
==========
==========
//case 1: object creation on stack with out virtual keyword

```
#include<iostream>
using namespace std;
class employee
{
        int id;
public:
        employee();
        employee(int);
    void display();
         int findsalary()
```

```
	{
		return 0;
	}
};
employee::employee()
{
	cout<<"in default of emp\n";
	id=0;
}
employee::employee(int i)
{
	cout<<"in para of emp\n";
	id=i;
}
void employee::display()
{

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

class wageemployee:public employee
{
	int hrs,rate;
public:
	wageemployee();
	wageemployee(int,int,int);
	 void display();
int findsalary();
};
wageemployee::wageemployee()
{
	cout<<"in default of wage\n";
	hrs=0;
	rate=0;
}
wageemployee::wageemployee(int i,int h,int r)		:employee(i)
{
	cout<<"in para of wage\n";
	hrs=h;
	rate=r;
}
int wageemployee::findsalary()
{
```

```
	return hrs * rate;
}
void wageemployee::display()
{
	employee::display();
	cout<<hrs<<endl;
	cout<<rate<<endl;
}

int main()
{
	employee * ptr;//the type of pointer is called as static type
	wageemployee we1(101,5,500);//type of object is called as dynamic type
	ptr = &we1;
	cout<<"salary is "<<ptr->findsalary();//without virtual keyword binding takes
	//at compile time and at compile time type of pointer is checked rather than type
	//of object.depending on type of object,function of that class will be
	//executed
	ptr->display();
}
================================================================================
=
//case 2: object creation on heap with new operator without virtual keyword
#include<iostream>
using namespace std;
class employee
{
	int id;
public:
	employee();
	employee(int);
    void display();
	 int findsalary()
	{
		return 0;
	}
};
employee::employee()
{
	cout<<"in default of emp\n";
	id=0;
}
employee::employee(int i)
```

```
{
	cout<<"in para of emp\n";
	id=i;
}
void employee::display()
{

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

class wageemployee:public employee
{
	int hrs,rate;
public:
	wageemployee();
	wageemployee(int,int,int);
	 void display();
int findsalary();
};
wageemployee::wageemployee()
{
	cout<<"in default of wage\n";
	hrs=0;
	rate=0;
}
wageemployee::wageemployee(int i,int h,int r)		:employee(i)
{
	cout<<"in para of wage\n";
	hrs=h;
	rate=r;
}
int wageemployee::findsalary()
{
	return hrs * rate;
}
void wageemployee::display()
{
	employee::display();
	cout<<hrs<<endl;
	cout<<rate<<endl;
}

int main()
```

```
{
	employee * ptr=new wageemployee(101,5,500);
	cout<<"salary is "<<ptr->findsalary();//without virtual keyword binding takes
	//at compile time and at compile time type of pointer is checked rather than type
	//of object.depending on type of object,function of that class will be
	//executed
	ptr->display();
}
=====================================================================
//case 3: object creation on stack  with virtual keyword
#include<iostream>
using namespace std;
class employee
{
	int id;
public:
	employee();
	employee(int);
   virtual void display();
	virtual int findsalary()
	{
		return 0;
	}
};
employee::employee()
{
	cout<<"in default of emp\n";
	id=0;
}
employee::employee(int i)
{
	cout<<"in para of emp\n";
	id=i;
}
void employee::display()
{

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

class wageemployee:public employee
{
	int hrs,rate;
```

```
public:
        wageemployee();
        wageemployee(int,int,int);
         void display();
int findsalary();
};
wageemployee::wageemployee()
{
        cout<<"in default of wage\n";
        hrs=0;
        rate=0;
}
wageemployee::wageemployee(int i,int h,int r)        :employee(i)
{
        cout<<"in para of wage\n";
        hrs=h;
        rate=r;
}
int wageemployee::findsalary()
{
        return hrs * rate;
}
void wageemployee::display()
{
        employee::display();
        cout<<hrs<<endl;
        cout<<rate<<endl;
}

int main()
{
        employee * ptr;
         wageemployee we1(101,5,500);
         ptr=&we1;
        cout<<"salary is "<<ptr->findsalary();//with virtual keyword binding takes
        //at runtime and at run time type of object is checked rather than type
        //of pointer.depending on type of object,function of that class will be
        //executed
        ptr->display();
}
================================================================================
=====
//case 4: object creation on heap  with virtual keyword
```

```
#include<iostream>
using namespace std;
class employee
{
        int id;
public:
        employee();
        employee(int);
   virtual void display();
        virtual int findsalary()
        {
                return 0;
        }
};
employee::employee()
{
        cout<<"in default of emp\n";
        id=0;
}
employee::employee(int i)
{
        cout<<"in para of emp\n";
        id=i;
}
void employee::display()
{

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

class wageemployee:public employee
{
        int hrs,rate;
public:
        wageemployee();
        wageemployee(int,int,int);
         void display();
int findsalary();
};
wageemployee::wageemployee()
{
        cout<<"in default of wage\n";
        hrs=0;
```

```
	rate=0;
}
wageemployee::wageemployee(int i,int h,int r)	:employee(i)
{
	cout<<"in para of wage\n";
	hrs=h;
	rate=r;
}
int wageemployee::findsalary()
{
	return hrs * rate;
}
void wageemployee::display()
{
	employee::display();
	cout<<hrs<<endl;
	cout<<rate<<endl;
}

int main()
{
	employee * ptr=new wageemployee (101,5,500);
	cout<<"salary is "<<ptr->findsalary();//with virtual keyword binding takes
	//at runtime and at run time type of object is checked rather than type
	//of pointer.depending on type of object,function of that class will be
	//executed
	ptr->display();
}
```

==========
=

1)Virtual functions cannot be friend function.Friend function are not members of the base class and hence cannot be inherited or overrided.
2)Functions that are overriden in the derived class are declared as virtual in the base class. since the base class pointer is used to invoke the appropriate function at runtime.
3)Constructors cannot be made virtual .When an object of a derived class is created, base class constructor should be called first.But if th base class constructor is made virtual, the derived class constructor wil be directly called and the base class members will not be initialized. Therefore forseeing the problem,compiler doesnot allow virtaul constructors.
4)If a function is declared as virtual in the base class then,it will be treated as virtual in derived class even if the keyword virtual is not used.
==========

```
//case 4: object creation on heap  with virtual keyword
#include<iostream>
using namespace std;
class employee
{
        int id;
public:
        employee();
        employee(int);
   virtual void display();
        virtual int findsalary()=0;


};
employee::employee()
{
        cout<<"in default of emp\n";
        id=0;
}
employee::employee(int i)
{
        cout<<"in para of emp\n";
        id=i;
}
void employee::display()
{

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

class wageemployee:public employee
{
        int hrs,rate;
public:
        wageemployee();
        wageemployee(int,int,int);
         void display();
int findsalary();
void show();
};
wageemployee::wageemployee()
{
        cout<<"in default of wage\n";
```

```
	hrs=0;
	rate=0;
}
wageemployee::wageemployee(int i,int h,int r)		:employee(i)
{
	cout<<"in para of wage\n";
	hrs=h;
	rate=r;
}
int wageemployee::findsalary()
{
	return hrs * rate;
}
void wageemployee::display()
{
	employee::display();
	cout<<hrs<<endl;
	cout<<rate<<endl;
}
void wageemployee::show()
{
	cout<<"in show() of wageemployee\n";
}

int main()
{
	employee * ptr=new wageemployee (101,5,500);
	//cout<<"salary is "<<ptr->findsalary();//with virtual keyword binding takes
	//ptr->display();
	ptr->show();
	//with the help of baseclass pointer,we can only invoke overrided function plus
	//that function implementation should also be present in the baseclass pointer
	// type.
}
```

==========
==========
Pure virtual function
1)A function without executable code is called as Pure virtual function.declared by using a specifier(=0) in the declaration of a virtual member function in the class declaration
2) Pure virtual function is represented by(=0) in the function declaration and not in the definition
ex:virtual float computesalary()=0;
3)A class contains atleast 1 pure virtual function than that class is called as abstact class
4)if any class contain all the pure virtual function then that class is calles as pure abstract class
5)we can not create object of an abstract class
6)however we can create pointer of an abstract class.
7)if any base class contains pure virtual function then the pure virtual function has to be overrided in the derived class and give the implementation according to the requirements.if not given,then derived class automatically becomes abstract class
8)Abstract class supports runtime polymorphism.
9)if Pure virtual function is not overrided in the derived class than the derived class automatically becomes abstract class
10)If a class is too generic to define its object then the class is made abstract.so the sole purpose of abstract classes is to provide a baseclass for other classes.
11)An abstract class can have constructors.

Sometimes implementation of all functions cannot be provided in a base class because we don't know the implementation. Such a class is called an abstract class.For example, let Shape be a base class. We cannot provide the implementation of function draw() in Shape, but we know every derived class must have an implementation of draw().other methods can be pure virtual function or virtual function or concrete funtion.
ex: shape is a base class which is abstract.several other classes like circle,rectangle can be derived from it. The common attributes and behaviour of all shapes are bundled together in shape class. draw() and area() are the two common behaviours of all the above mentioned shapes.
==>A shape hierarchy may define abstract class that requires programmers using it to provide their own implementation of the class by creating a derived class. This hierarchy is easily extensible.

ex:
In class Employee,computesalary() was added to implement runtime polymorphism.

```
float computesalary()
{
        return 0.0f;
```

}
Actually this implementation is unnecessary and does not ake any sense. This function is never called ,but its existence in the class Employee is necessary to enable polymorphism.
==>If a virtual function doesnot contain any meaningful code or if it has been added merly to ahieve dynamic binding,then it is made pure virtual funtion.
==========
Why Destructor can be made virtual?

```
class A
{
        public :
        virtual ~A()
        {
                cout<<"In A's Destructor";
        }
};
class B :public A
{
        public :
        ~B()
        {
                cout<<"In B's Destructor";
        }
};
int main()
{
A * aptr=new B();
delete aptr;
}
```

==If a base class pointer points to a derived object created on heap and the memory is freed by delete with base pointer the base class destructor is called.The deried class destructor is not invoked at all. To resole this problem,the base class destructor is made virtual.
ex:
Class A is a base class and B is derived from A. The destructor is made virtual in class A and an object of class B is created on heap using new and the address is stored in the base class A pointer i.e aptr.For the statement delete aptr, the destructor for the derived class is invoked first.
==If the destructor is not made virtual, the base class destructor is called directly without the derived class destructor being called first. This leads to memory related problems.
==========

==========

Similarities between virtual function and pure virtual function
1)These are the concepts of Run-time polymorphism.
2)These functions can't be static.
3)constructor cannot be pure/virtual fun
4)destructor can be pure/virtual
5)can be override

Difference between virtual function and pure virtual function in C++

1)A virtual function is a member function of base class which can be redefined by derived class.
A pure virtual function is a member function of base class whose only declaration is provided in base class and should be defined in derived class otherwise derived class also becomes abstract.

2)Classes having virtual functions are not abstract.
Base class containing pure virtual function becomes abstract.

3)Syntax:

```
virtual<func_type><func_name>()
{
   // code
}
```
Syntax:

```
virtual<func_type><func_name>()
   = 0;
```

4)Definition is given in base class.
No definition is given in base class.

5)Base class having virtual function can be instantiated i.e. its object can be made.
Base class having pure virtual function becomes abstract i.e. it cannot be instantiated.

6)If derived class do not redefine virtual function of base class, then it does not affect compilation.
If derived class do not redefine virtual function of base class, then no compilation error but derived class also becomes abstract just like the base class.

7)All derived class may or may not redefine virtual function of base class.

All derived class must redefine pure virtual function of base class otherwise derived class also becomes abstract just like base class.

==========

==========

Types of Classes:

Concrete class:
A class in which all the functionalities of an object are defined is called as concrete class.

Abstract class:
A class which contains generic/common features that multiple derived class can share.

pure abstract class:
A class in which all the member functions are pure virtual functions is called as pure abstract class. It is just an interface and cannot be instantiated.

Polymorphic class:
A class that contains at least one virtual function is called as polymorphic class.

==========

How virtual function works?

```
class A
{
int x;
        //without virtual function
};
class B
{
int x;
        //with virtual function
};
int main()
{
        A a;
        B b;
        cout<<"size of object a is "<<sizeof(a);
        cout<<"size of object a is "<<sizeof(b);
}
```

class B contains a virtual function whereas class A doesnot contain any virtual function. If the

size of the objects of both the classes A and B is compared, then size of object b is 8bytes more than the size of object a.The reason is, for every class which contains at least one virtual function, the compiler implicitly adds a virtual pointer.

==========
If a class contains a virtual function then the compiler itself does two things.

1)If an object of that class is created then a virtual pointer (VPTR) is inserted as a data member of the class to point to the VTABLE of that class. For each new object created, a new virtual pointer is inserted as a data member of that class.
2)Irrespective of whether the object is created or not, the class contains as a member a static array of function pointers called VTABLE. Cells of this table store the address of each virtual function contained in that class.

Virtual Functions in C++

The most common use of runtime binding is with base class pointers or references calling derived class functions. This is called runtime polymorphism and is implemented in C++ using virtual functions.
A virtual function is a member function that is declared within a base class with the virtual keyword and is re-defined (Overridden) by a derived class. When a class contains a virtual function, it can be overridden in its derived class and tells the compiler to perform dynamic linkage (or late binding) on the function. Now, the virtual functions are implemented in C++ by using vTable and vPtr.

What is VTable ?
==>VTable is a class specific. It is a static member of the class since all the objects of a class refer to the same virtual table. It contains the addresses of the respective virtual functions of that class. VTable is a constant array of function pointers which contains pointers to all virtual functions of the class. All the object of a class refer to its own virtual table.
==>If in the derived class,a function that is declared as virtual in the base class is not overridden, then the compiler uses the address of the base class version in the derived class virtual table.

What is Vptr (Virtual Pointer)?

==>Every object of a polymorphic class has a hidden pointer called virtual pointer(vptr). it is a pointer to VTable and is automatically initialized to the starting address of the VTable in the constructor.
==>The virtual pointer or _vptr is a hidden pointer that is added by the compiler as a member of

the class to point to the VTable of that class. Every object of a class containing virtual functions, a vptr is added to point to the vTable of that class. It's important to note that vptr is created only if a class has or inherits a virtual function.
==>The setting up of the VTable for each polymorphic class, initializing the Vptr and inserting the codefor the function call happens automatically, the programmer doesnot have to worry about it.

consider Employee and Salesperson are polymorphic class.

==>when a class containing a virtual function is created or a class is derived from a class that contains virtual functions,compiler creates a unique VTable for that class.In VTable ,the compiler stores the addresses of all virtual functions.Each object contains vptr that points to VTable of that class.
==>The compiler places Vptr as a data member in the class created per object.Vptr is initialized to the starting address of the appropriate VTable.when salesperson object is created,it contains VPtr as one ofits data members which points to the appropriate VTable.

```
int main()
{
Employee * ptr;
ptr=new Salesperson();
ptr->display();
}
```

The compiler picks up the Vptr of the object pointed by ptr.Thus instead of calling Employee::display() the compiler generates the code that says the function at Vptr should be called.The fectching of Vptr and the determination of the actual function address occur at runtime,thus late binding is achieved.

1)A class contains atleast one virtual function,that class is calles as polymorphic class
2)compiler implicitely add a virtual pointer for every polymorphic class i.e vptr
3)vptr is created per object
4)vtable is called as static array of function pointer.
5)vtable is created per class
6)vtable will store the address of all the virtual functions
7)vptr will point to the starting address of the table.