Inheritance in C++

Inheritance is one of the feature of Object Oriented Programming System(OOPs), it allows the child class to acquire the properties (the data members) and functionality (the member functions) of parent class.

child class:
A class that inherits another class is known as child class, it is also known as derived class or subclass.

parent class:
The class that is being inherited by other class is known as parent class, super class or base class.

Syntax of Inheritance

```
class parent_class
{
   //Body of parent class
};
class child_class : access_modifier parent_class
{
  //Body of child class
};
```

Ex:

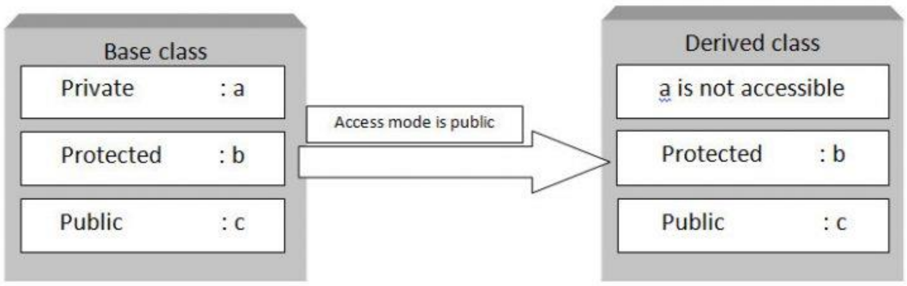
```
#include <iostream>
using namespace std;

//Base class
class Parent
{
   public:
     int id_p;
};

// Sub class inheriting from Base Class(Parent)
class Child : public Parent
{
   public:
     int id_c;
};

//main function
int main()
  {

       Child obj1;

       // An object of class child has all data members
       // and member functions of class parent
       obj1.id_c = 7;
       obj1.id_p = 91;
       cout << "Child id is " <<  obj1.id_c << endl;
       cout << "Parent id is " <<  obj1.id_p << endl;

       return 0;
  }
```

output:

Child id is 7
Parent id is 91

Modes of Inheritance

1. **Public mode**: If we derive a sub class 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.***

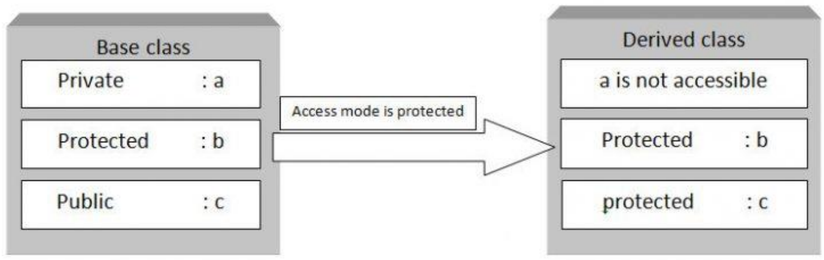


Ex:

```
// public access specifier
#include <iostream>
using namespace std;

class base
{
       private:
       int x;

       protected:
          int y;

       public:
          int z;

       base() //constructor to initialize data members
       {
         x = 1;
         y = 2;
         z = 3;
       }
};

class derive: public base
{
       //y becomes protected and z becomes public members of class derive
       public:
          void showdata()
          {
            cout << "x is not accessible" << endl;
            cout << "value of y is " << y << endl;
            cout << "value of z is " << z << endl;
          }
};
int main()
{
     derive a; //object of derived class
     a.showdata();
     a.z = 30;   //valid
a.showdata();
return 0;
}       //end of program
```

output:
x is not accessible
value of y is 2
value of z is 3
x is not accessible
value of y is 2
value of z is 30

2. **Protected mode**: If we derive a sub class from a ***Protected base class. Then both public member and protected members of the base class will become protected in derived class.***



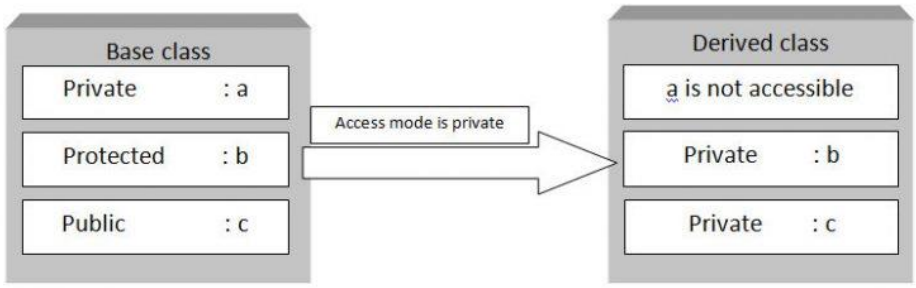
Ex:

```
// protected access specifier
#include <iostream>
using namespace std;

class base
{
      private:
      int x;

      protected:
         int y;

      public:
         int z;

      base() //constructor to initialize data members
      {
         x = 1;
         y = 2;
         z = 3;
      }
};

class derive: protected base
{
      //y and z becomes protected members of class derive
      public:
         void showdata()
         {
           cout << "x is not accessible" << endl;
           cout << "value of y is " << y << endl;
           cout << "value of z is " << z << endl;
         }
};
int main()
{
    derive a; //object of derived class
    a.showdata();
   return 0;
}
```

output:

x is not accessible
value of y is 2
value of z is 3

3. **Private mode**: If we derive a sub class from a ***Private base class. Then both public member and protected members of the base class will become Private in derived class.***



Ex:

```
// private access specifier
#include <iostream>
using namespace std;

class base
{
        private:
        int x;

        protected:
            int y;

        public:
            int z;

        base() //constructor to initialize data members
        {
           x = 1;
           y = 2;
           z = 3;
        }
};

class derive: private base
{
        //y and z becomes private members of class derive and x remains private
        public:
            void showdata()
            {
               cout << "x is not accessible" << endl;
               cout << "value of y is " << y << endl;
               cout << "value of z is " << z << endl;
            }
};
int main()
{
     derive a; //object of derived class
     a.showdata();
     return 0;
}       //end of program
```

output:
x is not accessible
value of y is 2
value of z is 3

Note : The private members in the base class cannot be directly accessed in the derived class, while protected members can be directly accessed

Ex:

```
// C++ Implementation to show that a derived class
// doesn't inherit access to private data members.
// However, it does inherit a full parent object
class A
{
public:
  int x;
protected:
```

```
    int y;
private:
    int z;
};

class B : public A
{
    // x is public
    // y is protected
    // z is not accessible from B
};

class C : protected A
{
    // x is protected
    // y is protected
    // z is not accessible from C
};

class D : private A    // 'private' is default for classes
{
    // x is private
    // y is private
    // z is not accessible from D
};
```

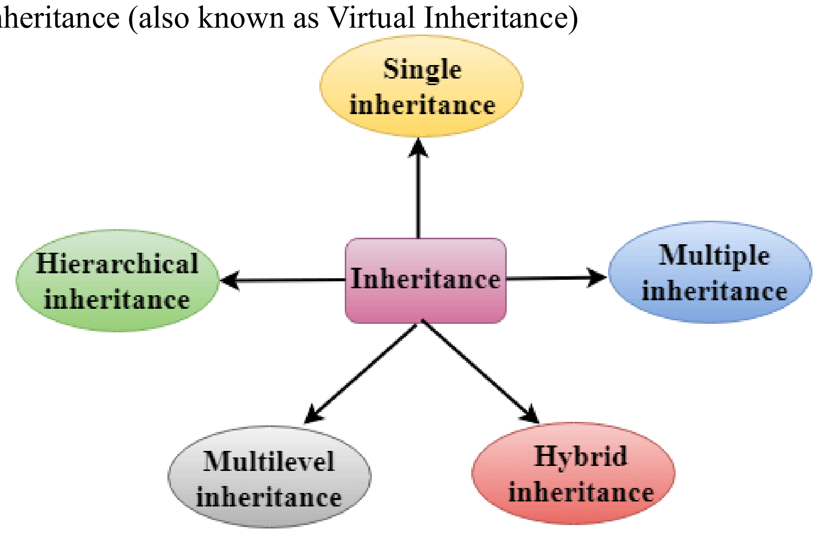
The below table summarizes the above three modes and shows the access specifier of the members of base class in the sub class when derived in public, protected and private modes:

Base class member access specifier	Type of Inheritence		
	Public	Protected	Private
Public	Public	Protected	Private
Protected	Protected	Protected	Private
Private	Not accessible (Hidden)	Not accessible (Hidden)	Not accessible (Hidden)

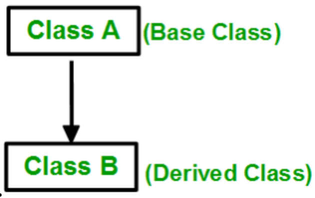
Types of Inheritance

C++ offers five types of Inheritance. They are:

- Single Inheritance
- Multiple Inheritance
- Hierarchical Inheritance
- Multilevel Inheritance
- Hybrid Inheritance (also known as Virtual Inheritance)



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



Syntax:

```
class subclass_name : access_mode base_class
{
  //body of subclass
};
```

Ex:

```
#include <iostream>
using namespace std;
class A {
public:
  A(){
     cout<<"Constructor of A class"<<endl;
  }
};
class B: public A {
public:
  B(){
     cout<<"Constructor of B class";
  }
};
int main() {
   //Creating object of class B
   B obj;
   return 0;
}
```

output:

Constructor of A class
Constructor of B class

Ex:

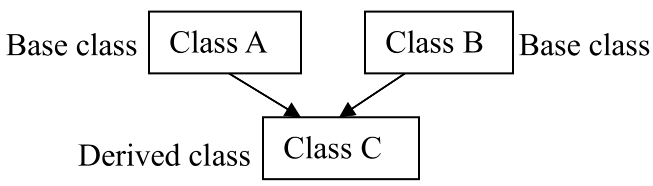
```
// inheritance.cpp
#include <iostream>
using namespace std;
class base    //single base class
{
   public:
     int x;
   void getdata()
   {
     cout << "Enter the value of x = "; cin >> x;
   }
 };
class derive : public base    //single derived class
{
   private:
    int y;
   public:
   void readdata()
   {
     cout << "Enter the value of y = "; cin >> y;
   }
   void product()
   {
     cout << "Product = " << x * y;
   }
```

```
};
int main()
 {
   derive a;     //object of derived class
   a.getdata();
   a.readdata();
   a.product();
   return 0;
 }        //end of program
```

output:

Enter the value of x = 5
Enter the value of y = 4
Product = 20

Multiple Inheritance: Multiple Inheritance is a feature of C++ where a class can inherit from more than one classes. i.e one **sub class** is inherited from more than one **base classes**.



Syntax:

```
class A
 {
  ..........
 };
 class B
 {
  ...........
  } ;
 class C : acess_specifier A,access_specifier A // derived class from A and B
 {
  ...........
 } ;
```

Ex:

```
#include <iostream>
using namespace std;
class A {
public:
  A(){
    cout<<"Constructor of A class"<<endl;
  }
};
class B {
public:
  B(){
    cout<<"Constructor of B class"<<endl;
  }
};
class C: public A, public B {
public:
  C(){
    cout<<"Constructor of C class"<<endl;
  }
};
int main() {
   //Creating object of class C
   C obj;
   return 0;
}
```

output:

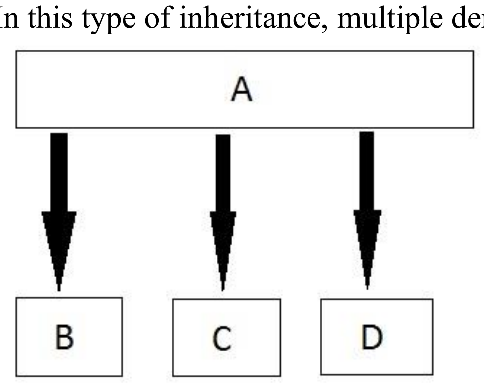
Constructor of A class
Constructor of B class
Constructor of C class

Ex:

```
/ multiple inheritance.cpp
#include <iostream>
using namespace std;
class A
{
       public:
       int x;
       void getx()
   {
          cout << "enter value of x: "; cin >> x;
   }
};
class B
{
       public:
       int y;
       void gety()
       {
          cout << "enter value of y: "; cin >> y;
       }
};
class C : public A, public B   //C is derived from class A and class B
{
       public:
       void sum()
       {
          cout << "Sum = " << x + y;
       }
};

int main()
{
        C obj1; //object of derived class C
        obj1.getx();
        obj1.gety();
        obj1.sum();
             return 0;
}
```

output:
enter value of x: 2
enter value of y: 3
Sum = 5

Hierarchical Inheritance in C++
In this type of inheritance, multiple derived classes inherits from a single base class.



Syntax:

```
class A // base class
{
```

```
    .............
};
class B : access_specifier A // derived class from A
{
    ..........
 } ;
 class C : access_specifier A // derived class from A
 {
    ..........
 } ;
 class D : access_specifier A // derived class from A
 {
     ..........
 } ;
```
Ex:
```
#include <iostream>
using namespace std;
class A {
public:
  A(){
     cout<<"Constructor of A class"<<endl;
  }
};
class B: public A {
public:
  B(){
     cout<<"Constructor of B class"<<endl;
  }
};
class C: public A{
public:
  C(){
     cout<<"Constructor of C class"<<endl;
  }
};
int main() {
   //Creating object of class C
   C obj;
   return 0;
}
```
Output:
Constructor of A class
Constructor of C class

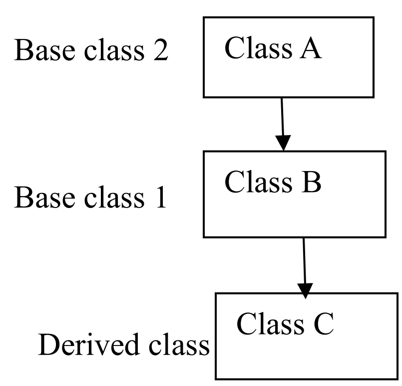
Ex:
```
// hierarchial inheritance.cpp
#include <iostream>
using namespace std;

class A //single base class
{
    public:
        int x, y;
        void getdata()
        {
            cout << "\nEnter value of x and y:\n"; cin >> x >> y;
        }
};
class B : public A //B is derived from class base
{
    public:
        void product()
        {
```

```
            cout << "\nProduct= " << x * y;
        }
};
class C : public A //C is also derived from class base
{
    public:
        void sum()
        {
        cout << "\nSum= " << x + y;
        }
};
int main()
{
    B obj1;          //object of derived class B
    C obj2;          //object of derived class C
    obj1.getdata();
    obj1.product();
    obj2.getdata();
    obj2.sum();
    return 0;
}  //end of program
```

output:

```
Enter value of x and y:
3
4
Product= 12
Enter value of x and y:
4
5
Sum= 9
```

Multilevel Inheritance: In this type of inheritance, a derived class is created from another derived class.



Syntax:

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

Ex:

```
#include <iostream>
using namespace std;
class A {
public:
  A(){
    cout<<"Constructor of A class"<<endl;
  }
};
class B: public A {
public:
  B(){
    cout<<"Constructor of B class"<<endl;
  }
};
class C: public B {
```

```
public:
  C(){
     cout<<"Constructor of C class"<<endl;
  }
};
int main() {
  //Creating object of class C
  C obj;
  return 0;
}
```

Output:
Constructor of A class
Constructor of B class
Constructor of C class

Ex:

```
// Multilevel  inheritance.cpp
#include <iostream>
using namespace std;
class base //single base class
{
        public:
        int x;
        void getdata()
        {
        cout << "Enter value of x= "; cin >> x;
        }
};
class derive1 : public base // derived class from base class
{
        public:
        int y;
        void readdata()
        {
            cout << "\nEnter value of y= "; cin >> y;
        }
};
class derive2 : public derive1   // derived from class derive1
{
        private:
        int z;
        public:
        void indata()
        {
        cout << "\nEnter value of z= "; cin >> z;
        }
        void product()
        {
            cout << "\nProduct= " << x * y * z;
        }
};
int main()
{
     derive2 a;      //object of derived class
     a.getdata();
     a.readdata();
     a.indata();
     a.product();
     return 0;
}              //end of program
```

output:
Enter value of x= 3

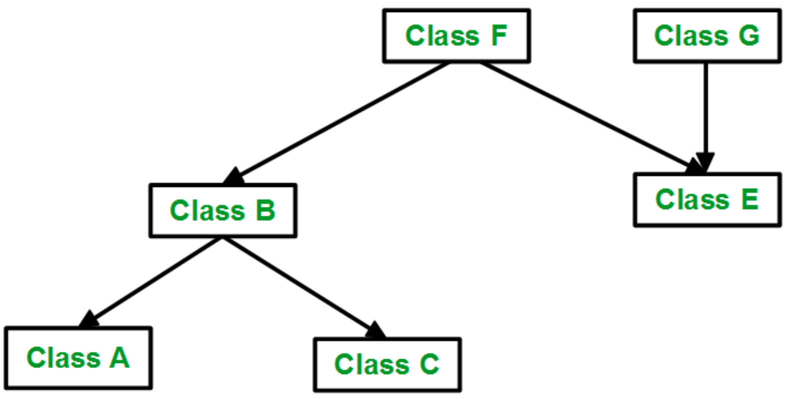
Enter value of y= 2

Enter value of z= 4

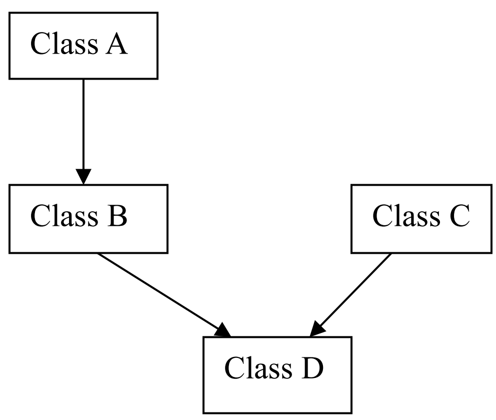
Product= 24

Hybrid Inheritance

Hybrid (Virtual) Inheritance: Hybrid Inheritance is implemented by combining more than one type of inheritance. For example: Combining Hierarchical inheritance and Multiple Inheritance.



Syntax




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

Ex:

```
// hybrid inheritance.cpp
#include <iostream>
using namespace std;

class A
{
        public:
        int x;
};
class B : public A
{
        public:
        B()      //constructor to initialize x in base class A
        {
          x = 10;
        }
};
class C
 {
        public:
        int y;
```

```
        C()   //constructor to initialize y
        {
           y = 4;
        }
};
class D : public B, public C   //D is derived from class B and class C
{
        public:
        void sum()
        {
           cout << "Sum= " << x + y;
        }
};
int main()
{
         D obj1;          //object of derived class D
        obj1.sum();
        return 0;
}                    //end of program
```

output:
Sum= 14

Constructors and Destructors In inheritance
When we are using the constructors and destructors in the inheritance, **parent class constructors and destructors are accessible to the child class** hence when we create an object for the child class, constructors and destructors of both parent and child class get executed

- **Order of execution of constructors:** The order of execution in the the case of constructors are working from top to bottom i.e base class constructor is executed first followed by the derived class constructor
- **Order of execution of destructors**: The order of execution the case of destructors is bottom-up i.e first child class destructors are executed followed by parent class destructors

Ex:
```
#include<iostream>
using namespace std;
class parent //parent class
{

public:
parent()//constructor
{
cout<<"Parent class constructor\n";
}

~parent()//destructor
{
cout<<"Parent class Destructor\n";
}

};

class child : public parent//child class
{

public:
child() //constructor
{
cout<<" child class constructor\n";
}

~ child() //destructor
{
cout<<" child class destructor\n";
```

```
}

};
main()
{

child c;//automatically executes both child and parent class //constructors and destructors because of inheritance

}
```

output:

Parent class constructor
 child class constructor
 child class destructor
Parent class Desctructor

Inheritance in parameterized constructor/ destructor

In the case of the default constructor, it is implicitly accessible from parent to the child class but parametrized constructors are not accessible to the derived class automatically, for this reason, explicit call has to be made in the child class constructor for accessing the parameterized constructor of the parent class to the child class using the following syntax

<class_name>:: constructor(arguments)

Ex:

```
#include <iostream>

using namespace std;

#include <iostream>
using namespace std;
class base
{
 protected:
   int i;
 public:
   base(int x)
   {
     i=x;
     cout << "Constructing base.\n";
   }
   ~base(void) {cout << "Destructing base.\n";}
 };

class derived : public base
{
   int j;
 public:
   derived(int x, int y): base(y){

     j=x;
     cout << "Constructing derived.\n";
   }
   ~derived(void) {cout << "Destructing derived.\n";}
   void show(void) {cout << i << ", " << j << endl;}
 };

int main(void)
{
   derived object(3,4);

   object.show();
}
```

Output:

```
Constructing base.
Constructing derived.
4, 3
Destructing derived.
Destructing base.
```

Ex:

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

int x;
public:
// parameterized constructor
parent(int i)
{
x = i;
cout << "parent class Parameterized Constructor\n"<<"\n x="<<x;
}

};

class child: public parent
{

int y;
public:
// parameterized constructor
child(int j):parent(j)//Explicitly calling
{
y = j;
cout << "\nchild class Parameterized Constructor\n"<<"\n y=  "<<y;
}
};
int main()
{
child c(10);
}
```

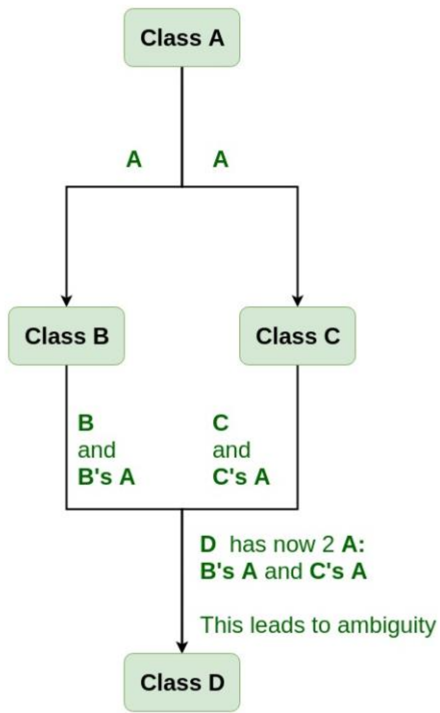
output:

parent class Parameterized Constructor
x= 10
child class Parameterized Constructor
y= 10

Virtual base class in C++

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.

Need for Virtual Base Classes:
Consider the situation where we have one class **A** .This class is **A** is inherited by two other classes **B** and **C**. Both these class are inherited into another in a new class **D** as shown in figure below.



As we can see from the figure that data members/function of class **A** are inherited twice to class **D**. One through class **B** and second through class **C**. When any data / function member of class **A** is accessed by an object of class **D**, ambiguity arises as to which data/function member would be called? One inherited through **B** or the other inherited through **C**. This confuses compiler and it displays error.

Ex:

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

class A {
public:
    void show()
    {
        cout << "Hello form A \n";
    }
};

class B : public A {
};

class C : public A {
};

class D : public B, public C {
};

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

output:

```
vbc.cpp: In function 'int main()':
vbc.cpp:25:12: error: request for member 'show' is ambiguous
     object.show();
            ^
vbc.cpp:7:10: note: candidates are: void A::show()
     void show()
          ^
vbc.cpp:7:10: note:                 void A::show()
```

To resolve this ambiguity when class A is inherited in both class B and class C, it is declared as virtual base class by placing a keyword virtual as :

Syntax for Virtual Base Classes:
Syntax 1:

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

Syntax 2:

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

Note: virtual can be written before or after the **public**. Now only one copy of data/function member will be copied to class **C** and class **B** and class **A** becomes the virtual base class. Virtual base classes offer a way to save space and avoid ambiguities in class hierarchies that use multiple inheritances. When a base class is specified as a virtual base, it can act as an indirect base more than once without duplication of its data members. A single copy of its data members is shared by all the base classes that use virtual base.

Ex:

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

class A {
public:
   void show()
   {
      cout << "Hello form A \n";
   }
};

class B : virtual public A {
};

class C : virtual public A {
};

class D : public B, public C {
};

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

output:

Hello form A

Static Binding:

By default, matching of function call with the correct function definition happens at compile time. This is called static binding or early binding or compile-time binding. Static binding is achieved using function overloading and operator overloading. Even though there are two or more functions with same name, compiler uniquely identifies each function depending on the parameters passed to those functions.

Ex:

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

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

int main(void)
{
    Base *bp;
   Derived d;
   bp=&d;

    // The function call decided at
    // compile time (compiler sees type
    // of pointer and calls base class
    // function.
    bp->show();

    return 0;
}
```

output:
In Base

Ex:2

```
#include<iostream>
using namespace std;
class Test
{
public:
int sum(int x,int y)
{
return x+y;
}
int sum(int x,int y, int z)
{
return x+y+z;
}
};
int main()
{
Test t;
cout<<"sum of 2 nums= "<<t.sum(10,20);
cout<<"sum of 3 nums=  "<<t.sum(10,20,30);
return 0;
}
```

output:
sum of 2 nums= 30
sum of 3 nums= 60

Dynamic Binding :

C++ provides facility to specify that the compiler should match function calls with the correct definition at the run time; this is called dynamic binding or late binding or run-time binding. Dynamic binding is achieved using virtual functions. Base class pointer points to derived class object. And a function is declared virtual in base class, then the matching function is identified at run-time using virtual table entry.

```
// CPP Program to illustrate dynamic or late binding
#include<iostream>
using namespace std;

class Base
{
public:
   virtual void show()
{
cout<<" In Base \n";
}
};

class Derived: public Base
{
public:
   void show()
{
 cout<<"In Derived \n";
 }
};

int main(void)
{
   Base *bp;
   Derived d;
   bp=&d;
   bp->show();  // RUN-TIME POLYMORPHISM
   return 0;
}
```

Output:

```
In Derived
```

Virtual Functions in C++

Virtual Function is a function in base class, which is overrided in the derived class, and which tells the compiler to perform **Late Binding** on this function.

`Virtual` Keyword is used to make a member function of the base class Virtual.

Ex:

the following simple program showing run-time behavior of virtual functions.

```
// CPP program to illustrate
// concept of Virtual Functions
#include<iostream>
using namespace std;

class base
{
public:
    virtual void print ()
    { cout<< "print base class" <<endl; }

    void show ()
    { cout<< "show base class" <<endl; }
};

class derived:public base
{
public:
    void print ()
    { cout<< "print derived class" <<endl; }

    void show ()
    { cout<< "show derived class" <<endl; }
};

int main()
{
    base *bptr;
    derived d;
    bptr = &d;

    //virtual function, binded at runtime
    bptr->print();

    // Non-virtual function, binded at compile time
    bptr->show();
}
```

Output:

```
print derived class
show base class
```

Late Binding in C++

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. Late Binding is also called **Dynamic** Binding or **Runtime** Binding.

Problem without Virtual Keyword

Let's try to understand what is the issue that `virtual` keyword fixes,

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

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

int main()
{
    Base* b;        //Base class pointer
    Derived d;      //Derived class object
    b = &d;
    b->show();      //Early Binding Ocuurs
}
```

Base class

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

---

Using Virtual Keyword in C++

We can make base class's methods virtual by using **virtual** keyword while declaring them. Virtual keyword will lead to Late Binding of that method.

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

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

int main()
{
    Base* b;        //Base class pointer
    Derived d;      //Derived class object
    b = &d;
    b->show();      //Late Binding Ocuurs
```

```
}
```

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 points to Derived class object.

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.

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

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();
}
```

Derived class

Pure Virtual Function

A virtual function will become pure virtual function when you append "=0" at the end of declaration of virtual function. Pure virtual function doesn't have body or implementation. We must implement all pure virtual functions in derived class.
Pure virtual function is also known as abstract function.

A class with at least one pure virtual function or abstract function is called abstract class. We can't create an object of abstract class. Member functions of abstract class will be invoked by derived class object.

Example of pure virtual function

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

class BaseClass       //Abstract class
{

        public:
         virtual void Display1()=0;     //Pure virtual function or abstract
function
         virtual void Display2()=0;     //Pure virtual function or abstract
function
```

```
        void Display3()
        {
                cout<<"\n\tThis is Display3() method of Base Class";
        }

};

class DerivedClass : public BaseClass
{

        public:
        void Display1()
        {
                cout<<"\n\tThis is Display1() method of Derived Class";
        }

        void Display2()
        {
                cout<<"\n\tThis is Display2() method of Derived Class";
        }

};

int main()
{

        DerivedClass D;

        D.Display1();           // This will invoke Display1() method of Derived Class
        D.Display2();           // This will invoke Display2() method of Derived Class
        D.Display3();           // This will invoke Display3() method of Base Class

}
```

```
Output :

        This is Display1() method of Derived Class
        This is Display2() method of Derived Class
        This is Display3() method of Base Class
```

Abstract Class

Abstract class is used in situation, when we have partial set of implementation of methods in a class. For example, consider a class have four methods. Out of four methods, we have an implementation of two methods and we need derived class to implement other two methods. In these kind of situations, we should use abstract class.

A virtual function will become pure virtual function when you append "=0" at the end of declaration of virtual function.

A class with at least one **pure virtual function** or **abstract function** is called abstract class.

Pure virtual function is also known as abstract function.

- We can't create an object of abstract class b'coz it has partial implementation of methods.
- Abstract function doesn't have body
- We must implement all abstract functions in derived class.

Example of C++ Abstract class

```
#include<iostream.h>
using namespace std;
class BaseClass         //Abstract class
{

        public:
        virtual void Display1()=0;      //Pure virtual function or abstract function
```

```
        virtual void Display2()=0;      //Pure virtual function or abstract
function

        void Display3()
        {
                cout<<"\n\tThis is Display3() method of Base Class";
        }

    };

    class DerivedClass : public BaseClass
    {

        public:
         void Display1()
         {
                cout<<"\n\tThis is Display1() method of Derived Class";
         }

         void Display2()
         {
                cout<<"\n\tThis is Display2() method of Derived Class";
         }

    };

    int main()
    {

        DerivedClass D;

        D.Display1();           // This will invoke Display1() method of
Derived Class
        D.Display2();           // This will invoke Display2() method of
Derived Class
        D.Display3();           // This will invoke Display3() method of
Base Class

    }

  Output :

        This is Display1() method of Derived Class
        This is Display2() method of Derived Class
        This is Display3() method of Base Class
```

Virtual Destructors

Lets first see what happens when we do not have a virtual Base class destructor.

Deleting a derived class object using a pointer to a base class that has a non-virtual destructor results in undefined behavior. To correct this situation, the base class should be defined with a virtual destructor. For example, following program results in undefined behavior.

Ex:

```
#include <iostream>

using namespace std;

class base
{
 public:
  base()
  { cout<<"Constructing base \n"; }
  ~base()
  { cout<<"Destructing base \n"; }
};

class derived: public base {
 public:
  derived()
  { cout<<"Constructing derived \n"; }
  ~derived()
```

```
    { cout<<"Destructing derived \n"; }
};

int main(void)
{
  derived *d = new derived();
  base *b = d;
  delete b;
  return 0;
}
```

Output:

```
 Constructing base
 Constructing derived
 Destructing base
```

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.

Destructors in the Base class can be Virtual. Whenever Upcasting is done, Destructors of the Base class must be made virtual for proper destrucstion of the object when the program exits.

```
#include <iostream>
using namespace std;
class base
{
  public:
    base()
    { cout<<"Constructing base \n"; }
    virtual ~base()
    { cout<<"Destructing base \n"; }
};

class derived: public base {
  public:
    derived()
    { cout<<"Constructing derived \n"; }
    ~derived()
    { cout<<"Destructing derived \n"; }
};

int main(void)
{
  derived *d = new derived();
  base *b = d;
  delete b;
  return 0;
}
```

Output:

```
 Constructing base
 Constructing derived
 Destructing derived
 Destructing base
```

When we have Virtual destructor inside the base class, then first Derived class's destructor is called and then Base class's destructor is called, which is the desired behaviour.

NOTE: Constructors are never Virtual, only Destructors can be Virtual

Pure Virtual Destructors

Yes, it is possible to have pure virtual destructor. Pure virtual destructors are legal in standard C++ and one of the most important things to remember is **that if a class contains a pure virtual destructor, it must provide a function body for the pure virtual destructor.**

Why a pure virtual function requires a function body.?

The reason is because destructors (unlike other functions) are not actually 'overridden', rather they are always called in the reverse order of the class derivation. This means that a derived class' destructor will be invoked first, then base class destructor will be called. If the definition of the pure virtual destructor is not provided, then what function body will be called during object destruction? Therefore the compiler and linker enforce the existence of a function body for pure virtual destructors.

```
#include <iostream>
using namespace std;
class Base
{
public:
   virtual ~Base()=0; // Pure virtual destructor
};
Base::~Base()
{
   cout << "Pure virtual destructor is called";
}

class Derived : public Base
{
public:
   ~Derived()
   {
      cout << "~Derived() destructor  is executed\n";
   }
};

int main()
{
   Base *b = new Derived();
   delete b;
   return 0;
}
```

Output:

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