Accessing Derived Class Objects Through Base Pointers

A base class pointer can be used to point to a derived class object.

If the base class does not contain any virtual functions then the members referred by using such a pointer are those belonging to the base class.

Virtual Destructors

```
class A {
public :
   A( ){cout<<"base cons\n" ;}
   ~A( ){cout<<"base destr\n" ;}
} ;

class B : public A {
public :
   B( ){cout<<"der cons\n" ;}
   ~B( ){cout<<"der destr\n" ;}
} ;

main( ) {
   A     * ptr = new B ;
   delete(ptr) ;
}
```

Output :

base cons
der cons
base destr

- The object is not destroyed fully as the derived class destructor is never invoked.

- If we can force the invocation of the derived destructor then the base destructor is anyway automatically called.

- This can be done by declaring the base destructor as virtual.

Virtual destructors are possible because :

- During destruction the type of the object is already known and the VPTR already initialized.

- This is not the case with the constructor.

Virtual Destructors

```
class A {
public :
   A( ){cout<<"base cons\n" ;}
   ~A( ){cout<<"base destr\n" ;}
} ;

class B : public A {
public :
   B( ){cout<<"der cons\n" ;}
   ~B( ){cout<<"der destr\n" ;}
} ;

main( ) {
   A     * ptr = new B ;
   delete(ptr) ;
}
```

Output :
base cons
der cons
base destr

Virtual Destructors

```
class A {
public :
   A( ){cout<<"base cons\n" ;}
   ~A( ){cout<<"base destr\n" ;}
} ;

class B : public A {
public :
   B( ){cout<<"der cons\n" ;}
   ~B( ){cout<<"der destr\n" ;}
} ;

main( ) {
   A     * ptr = new B ;
   delete(ptr) ;
}
```

Output :
base cons
der cons
base destr

If the type of the object pointed by the base pointer is known during code generation then the following can be used :

*((B *) base_ptr) -> g( ) ; //base address casted to derived address*

Casting the derived address to base address is always safe.

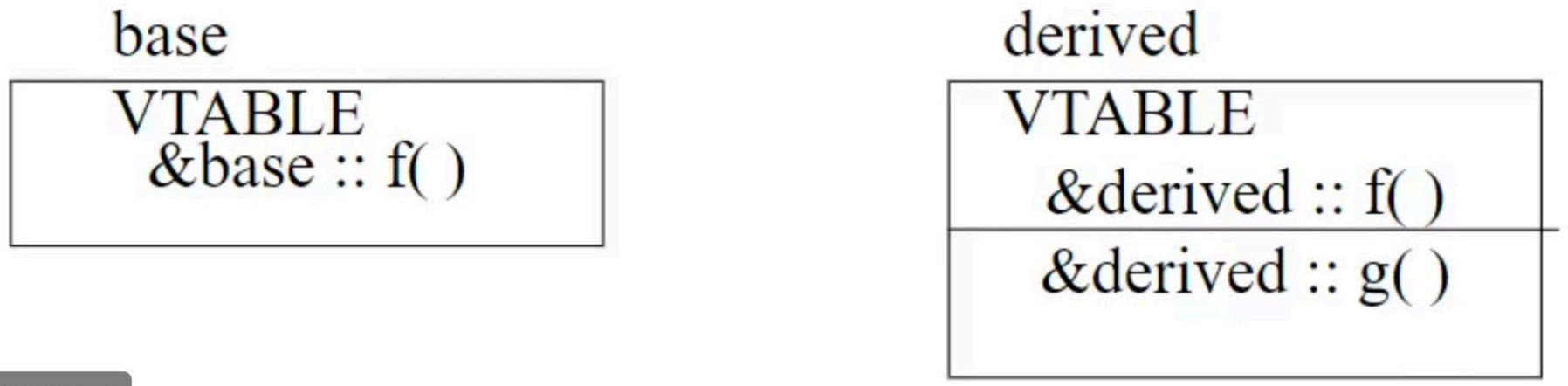
The reverse is not true.

This code generates a compilation error because *g( )* is not a member of class *A*.

Though the VTABLE contains both the functions *f( )* and *g( )*, but the compiler does not know what object's address is available in the *base_ptr*.

So the compiler decision whether the member invoked is correct or not is based on the type of *base_ptr*.

Once the code compiles successfully then only the VTABLES come into picture for the virtual functions.



```
class A {
    public :
    virtual void f( ) { cout <<"f :: AAAA\n" ; }
} ;

class B : public A {
    public :
    void f( ) { cout <<"f :: BBBB\n" ; }
    virtual void g( ) { cout <<"g :: BBBB\n" ; }
} ;

int main( )
{
    B   derived_obj ;
    A   * base_ptr  = & derived_obj ;

    base_ptr -> g( ) ;      //compilation error
}
```

- Whenever a virtual function is invoked using the base class pointer, the compiler inserts the code to fetch the VPTR and to look up the function address in the VTABLE.

- This causes late binding to take place to invoke the correct virtual function because the VPTR gets initialized at run-time.

When a virtual function is invoked on a base class pointer containing the address of a derived class object then :

- The compiler reads the contents of the base class pointer.
- Through this the compiler reads the derived object's VPTR.
- Through this VPTR the derived class VTABLE is accessed.
- From this table address of function being invoked is fetched.
- Using this address, function of derived class is accessed.

With virtual functions in a class the following is true :

- The compiler creates a VTABLE for each class that contains virtual functions and for the classes derived from it.

- This VTABLE of a class contains addresses of the virtual functions for that particular class.

- If a derived class does not redefine a virtual function of the base class then the compiler places the address of the base class version itself in the VTABLE of the derived class.

- The compiler adds a void pointer to an object of such base and derived classes containing virtual functions.

- This pointer, called the *vpointer* (VPTR) points to the class's VTABLE.

- The size of class B is 8 : size of 2*int data type.
- The size of class A is 16 : (2*size of int) + (size of a void pointer)
- The sizes of class C and D are 1 : so that objects of these classes have non-zero sizes. (There can not be an array of zero sized objects)

```
class A {
    int i, j ;
    public : virtual void f() {cout<<"AAA\n" ;}
} ;
class B {
    int i, j ;
    public : void f() {cout<<"BBBB\n" ;}
} ;
class C {
    public : void f() {cout<<"CCCC\n" ;}
} ;
class D { } ;

int main() {
    cout<<sizeof(A)<<endl<<sizeof(B)<<endl<<sizeof(C)<<endl<
<sizeof(D)<<endl ;
}
```

VTABLES

- A base class pointer can point to a derived class object but the reverse is not true.

- When a base pointer points to a derived class, incrementing or decrementing it does not make it point to the next object of the derived class. It is incremented or decremented only relative to its base type.

- It is not necessary to redefine a virtual function in the derived classes. If not redefined in the derived classes the base version is invoked.

Rules for Virtual Functions

- The virtual functions must be members of some class.
- They cannot be static members
- A virtual function can be a friend of another class.
- A virtual function in a base class must be defined, even though it may not be used.
- The prototypes of the base class version of a virtual function and all the derived class versions must be identical.
- Virtual constructors are not possible but virtual destructors are possible.

How use of virtual functions results in dynamic polymorphism?

- In case the base class pointer is pointing to an object of a derived class and a virtual function of the base class is invoked through such a pointer then :
  - the compiler selects the function to be called based on the contents of the pointer and not on the type of the pointer
  - but the compiler does not know the contents of ptr because ptr gets initialized to the derived address only at run time
  - so it is not known at compile time the class to which the function belongs
  - so the decision of which version of the function is to be called, is deferred till the program is running.

Example :

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

class der_b : public base {
public : void f() {cout<<"der1\n" ;} } ;

class der_d : public der_b {
public : void f() {cout<<"der2\n" ;} } ;

void main() {
    base *bptr ; der_b    dObj ;  der_d   ddObj ;
    bptr = &dObj ;     bptr->f() ;  //der 1
    bptr = &ddObj ;    bptr->f() ; //der 2

    der_b   *dptr = &ddObj ;
    dptr->f() ;                  //der 2
}
```

Example :

```
class A {
    public :    virtual void f( ) { cout<<"ff in AAAA\n" ; }
                void p( ) { cout<<"pp in AAAA\n" ;}
} ;
class B : public A {
    public :    void f( ) { cout<<"ff in BBBB\n" ; }
                void p( ) { cout<<"pp in BBBB\n" ;}
} ;
main( )
{   B   derived_obj ;  A    base_obj ;
    A   * base_ptr = & base_obj ;
    base_ptr->f( ) ; base_ptr->p( ) ; // base versions of f( ) & p( ) called

    base_ptr = & derived_obj ;
    base_ptr->f( ) ; // derived versions of f( ) called
    base_ptr->p( ) ; // base versions of p( ) called
}
```

Example :

```
class A {
    public :    virtual void f( ) { cout<<"ff in AAAA\n" ; }
                void p( ) { cout<<"pp in AAAA\n" ;}
} ;
class B : public A {
    public :    void f( ) { cout<<"ff in BBBB\n" ; }
                void p( ) { cout<<"pp in BBBB\n" ;}
} ;
main( )
{   B   derived_obj ;  A    base_obj ;
    A   * base_ptr = & base_obj ;
    base_ptr->f( ) ; base_ptr->p( ) ; // base versions of f( ) & p( ) called

    base_ptr = & derived_obj ;
    base_ptr->f( ) ; // derived versions of f( ) called
    base_ptr->p( ) ; // base versions of p( ) called
}
```

Dynamic Polymorphism

Dynamic polymorphism is achieved by the use of **Virtual Functions**.

How is dynamic polymorphism achieved ?

- Pointer to base class is used to refer to a derived class object.
- The function with the same name (say *test*) in the base and the derived class is preceded by the keyword *virtual* in the base class.
- When a base class pointer is referring to a derived class object then if the function *test* is invoked using such a pointer then the derived class version is executed.
- If in the base class this function is not preceded by the keyword *virtual* then it is the base class version of the function is invoked.

Dynamic Polymorphism And VTABLE

Accessing Derived Class Objects Through Base Pointers

A base class pointer can be used to point to a derived class object.

If the base class does not contain any virtual functions then the members referred by using such a pointer are those belonging to the base class.

```
class base {
   public :    int x, y ;
   void f() {cout<<"base\n" ;}
   void f(int i) {cout<<"base int\n" ;}
   void g() {cout<<"base g\n" ;}
   void h() {cout<<"base h\n" ;} } ;
class der:public base{
   public :     int x ;
   der() {x=9 ; base::x=90 ;}
   void f() {cout<<"der\n" ;}
   void f(int i, int j) {cout<<"der int int\n" ;}
   void g(int b) {} } ;
main() {
   der dobj ;
   cout<<sizeof(der)<<" "<<dobj.base::x<<" "<<dobj.x<<" "<<dobj.y<<endl ; ;
   dobj.base::f() ; //base
   dobj.f() ; //der
   dobj.base::f(9) ;
   //dobj.f(8) ; //err
   dobj.f(2,2) ;
   dobj.y = 99 ;
   //dobj.g() ;err
   dobj.h() ;
}
```

Overloading, Overriding, Hiding

```
class base {
        public :
        void f(int x) {cout<<"base f -> 1arg\n" ;} // f is overloaded
        void f(int x, int y) {cout<<"base f -> 2args\n" ;}
        void g(int x) {cout<<"base\n" ;}
        void h(int x) {cout<<"base\n" ;}
} ;

class der : public base {
        public :
        void g(int x) {cout<<"der\n" ;} //g of base is overridden
        void h(int x, int y) {cout<<"der\n" ;} // h of base is hidden
} ;
```

Another example :

```
#include<iostream>
using namespace std ;
class base {
   public :
      int x;
      base() {cout<<"base constr\n";x=10;}
      ~base() {cout<<"base destr\n";}
};
class der1 : public virtual base{
   public :
      der1() {cout<<"der1 constr\n";}
      ~der1() {cout<<"der1 destr\n";}
};
class der2 : public virtual base{
   public :
      der2() {cout<<"der2 constr\n";}
      ~der2() {cout<<"der2 destr\n";}
};

class sub : public der1, public der2 {};

main() {
   sub subObj;
   cout<<sizeof(sub)<<endl; //size = 4(x) + 8(vptr.der1) + 8(vptr.der2)
}
```

without virtual bases der1 and der2 → constr/destr of base for subObj are called twice

```
class D : public B1, public B2 {
public :
    D(int a, int b, int c):B2(c, a), B1(b, c)
    {
        cout<<"derived\n" ;
    }
} ;

main()
{
    D   obj(1,2,3) ;
}
```

Output :

```
B1 : 2  3
B2 : 3  1
derived
```

```
class gp {
   public :    gp(int a, int b) {cout<<a<<" "<<b<<endl ;} } ;

class p1 : virtual public gp {
   public :    p1(int x) : gp(1, 2) {cout<<"p1:"<<x<<"\n" ;} } ;

class p2 : virtual public gp {
   public :    p2(int x) : gp(12, 22) {cout<<"p2:"<<x<<"\n" ;} } ;

class ch : public p1, public p2 {
   public :    ch() : p1(6), p2(7), gp(3, 4) {cout<<"ch\n" ;} } ;

class gc : public ch {
   public :    gc() : gp(78, 672) {} } ;

main() { ch     chobj ;       gc      gcobj ;}
```

In this case the *ch* class passes arguments to its non immediate parent constructor. This is because, when *ch* object is created, neither *p1* nor *p2* can do the argument passing to *gp*.

Hence :

First, virtual base classes are created before non-virtual base classes, which ensures all bases get created before their derived classes.

Second, note that the *p1* and *p2* constructors still have calls to the *gp* constructor. If we are creating an instance of *ch*, these constructor calls are simply ignored because *ch* is responsible for creating the *gp*, not *p1* or *p2*.

However, if we were to create an instance of *p1* or *p2*, the virtual keyword is ignored, those constructor calls would be used, and normal inheritance rules apply.

Third, if a class inherits one or more classes that have virtual parents, the most derived class is responsible for constructing the virtual base class. In this case, *ch* (or *gc*) inherits *p1* and *p2*, both of which have a *gp* virtual base class. *ch* (or *gc*) , the most derived class, is responsible for creation of *gp*.

Hence :

First, virtual base classes are created before non-virtual base classes, which ensures all bases get created before their derived classes.

Second, note that the *p1* and *p2* constructors still have calls to the *gp* constructor. If we are creating an instance of *ch*, these constructor calls are simply ignored because *ch* is responsible for creating the *gp*, not *p1* or *p2*.

However, if we were to create an instance of *p1* or *p2*, the virtual keyword is ignored, those constructor calls would be used, and normal inheritance rules apply.

Third, if a class inherits one or more classes that have virtual parents, the most derived class is responsible for constructing the virtual base class. In this case, *ch* (or *gc*) inherits *p1* and *p2*, both of which have a *gp* virtual base class. *ch* (or *gc*) , the most derived class, is responsible for creation of *gp*.

```
class gp {
   public :    gp(int a, int b) {cout<<a<<" "<<b<<endl ;} } ;

class p1 : virtual public gp {
   public :    p1(int x) : gp(1, 2) {cout<<"p1:"<<x<<"\n" ;} } ;

class p2 : virtual public gp {
   public :    p2(int x) : gp(12, 22) {cout<<"p2:"<<x<<"\n" ;} } ;

class ch : public p1, public p2 {
   public :    ch() : p1(6), p2(7), gp(3, 4) {cout<<"ch\n" ;} } ;

class gc : public ch {
   public :    gc() : gp(78, 672) {} } ;

main() { ch     chobj ;       gc      gcobj ;}
```

In this case the *ch* class passes arguments to its non immediate parent constructor. This is because, when *ch* object is created, neither *p1* nor *p2* can do the argument passing to *gp*.

```
class gp {
   public :    gp(int a, int b) {cout<<a<<" "<<b<<endl ;} } ;

class p1 : virtual public gp {
   public :    p1(int x) : gp(1, 2) {cout<<"p1:"<<x<<"\n" ;} } ;

class p2 : virtual public gp {
   public :    p2(int x) : gp(12, 22) {cout<<"p2:"<<x<<"\n" ;} } ;

class ch : public p1, public p2 {
   public :    ch() : p1(6), p2(7), gp(3, 4) {cout<<"ch\n" ;} } ;

class gc : public ch {
   public :    gc() : gp(78, 672) {} } ;

main() { ch     chobj ;       gc      gcobj ;}
```

In this case the *ch* class passes arguments to its non immediate parent constructor. This is because, when *ch* object is created, neither *p1* nor *p2* can do the argument passing to *gp*.

```
class gp {
   public :    gp(int a, int b) {cout<<a<<" "<<b<<endl ;} } ;

class p1 : virtual public gp {
   public :    p1(int x) : gp(1, 2) {cout<<"p1:"<<x<<"\n" ;} } ;

class p2 : virtual public gp {
   public :    p2(int x) : gp(12, 22) {cout<<"p2:"<<x<<"\n" ;} } ;

class ch : public p1, public p2 {
   public :    ch() : p1(6), p2(7), gp(3, 4) {cout<<"ch\n" ;} } ;

class gc : public ch {
   public :    gc() : gp(78, 672) {} } ;

main() { ch     chobj ;       gc      gcobj ;}
```

In this case the *ch* class passes arguments to its non immediate parent constructor. This is because, when *ch* object is created, neither *p1* nor *p2* can do the argument passing to *gp*.

Example :

```
class B1 {
    int     i1, j1 ;
public :
    B1(int x, int y) {
        i1 = x, j1 = y ;
        cout<<"B1:"<<i1<<" "<<j1<<"\n" ; }
} ;

class B2 {
    int     i2, j2 ;
public :
    B2(int x, int y) {
        i2 = x, j2 = y ;
        cout<<"B2:"<<i2<<" "<<j2<<"\n" ; }
} ;
```

```
class D : public B1, public B2 {
public :
    D(int a, int b, int c):B2(c, a), B1(b, c)
    {
        cout<<"derived\n" ;
    }
} ;

main()
{
    D   obj(1,2,3) ;
}
```

Output :

```
B1 : 2  3
B2 : 3  1
derived
```

Example :

```
class B1 {
    int     i1, j1 ;
public :
    B1(int x, int y) {
        i1 = x, j1 = y ;
        cout<<"B1:"<<i1<<" "<<j1<<"\n" ; }
} ;

class B2 {
    int     i2, j2 ;
public :
    B2(int x, int y) {
        i2 = x, j2 = y ;
        cout<<"B2:"<<i2<<" "<<j2<<"\n" ; }
} ;
```

- The constructors of virtual base classes are invoked before any non virtual base class constructors.

  Example :
  class D : public A, virtual public B {} ;
  execution order : B, A, D

- The derived class must pass the initial values to the base class constructors.

  It does so for only its immediate parents.

- The derived class constructors receive the entire list of arguments and pass them to the base constructors in the order in which they are declared in the base class.

- The constructors of virtual base classes are invoked before any non virtual base class constructors.

  Example :
  class D : public A, virtual public B {} ;
  execution order : B, A, D

- The derived class must pass the initial values to the base class constructors.

  It does so for only its immediate parents.

- The derived class constructors receive the entire list of arguments and pass them to the base constructors in the order in which they are declared in the base class.

- In multiple inheritance, the base class constructors are executed in the order in which the base classes appear in the declaration of the derived class.

  Example :
  class D : public A1, public A2, public A3 {} ;
  constructor execution order : A1, A2, A3, D

- In case of multilevel inheritance the constructors are executed in the order of inheritance.

  Example :
  class B : public A {} ;
  class D : public B {} ;
  constructor execution order : A, B, D

Constructors in Derived Classes

- As long as no base class constructor takes an argument, the derived class need not have a constructor function.

  If the base class contains at least one constructor with at least one argument then it is compulsory for the derived class to have a constructor and pass arguments to base class constructors.

- When an object of derived class is created the base constructor is executed before the derived constructor.

  Example :
  class D : public A {} ;
  constructor execution order : A, D

Another example :

```
#include<iostream>
using namespace std ;
class base {
   public :
      int x;
      base() {cout<<"base constr\n";x=10;}
      ~base() {cout<<"base destr\n";}
};
class der1 : public virtual base{
   public :
      der1() {cout<<"der1 constr\n";}
      ~der1() {cout<<"der1 destr\n";}
};
class der2 : public virtual base{
   public :
      der2() {cout<<"der2 constr\n";}
      ~der2() {cout<<"der2 destr\n";}
};

class sub : public der1, public der2 {};

main() {
   sub subObj;
   cout<<sizeof(sub)<<endl; //size = 4(x) + 8(vptr.der1) + 8(vptr.der2)
}
```

without virtual bases der1 and der2 → constr/destr of base for subObj are called twice

Another example :

```
#include<iostream>
using namespace std ;
class base {
   public :
      int x;
      base() {cout<<"base constr\n";x=10;}
      ~base() {cout<<"base destr\n";}
};
class der1 : public virtual base{
   public :
      der1() {cout<<"der1 constr\n";}
      ~der1() {cout<<"der1 destr\n";}
};
class der2 : public virtual base{
   public :
      der2() {cout<<"der2 constr\n";}
      ~der2() {cout<<"der2 destr\n";}
};

class sub : public der1, public der2 {};

main() {
   sub subObj;
   cout<<sizeof(sub)<<endl; //size = 4(x) + 8(vptr.der1) + 8(vptr.der2)
}
```

without virtual bases der1 and der2 → constr/destr of base for subObj are called twice

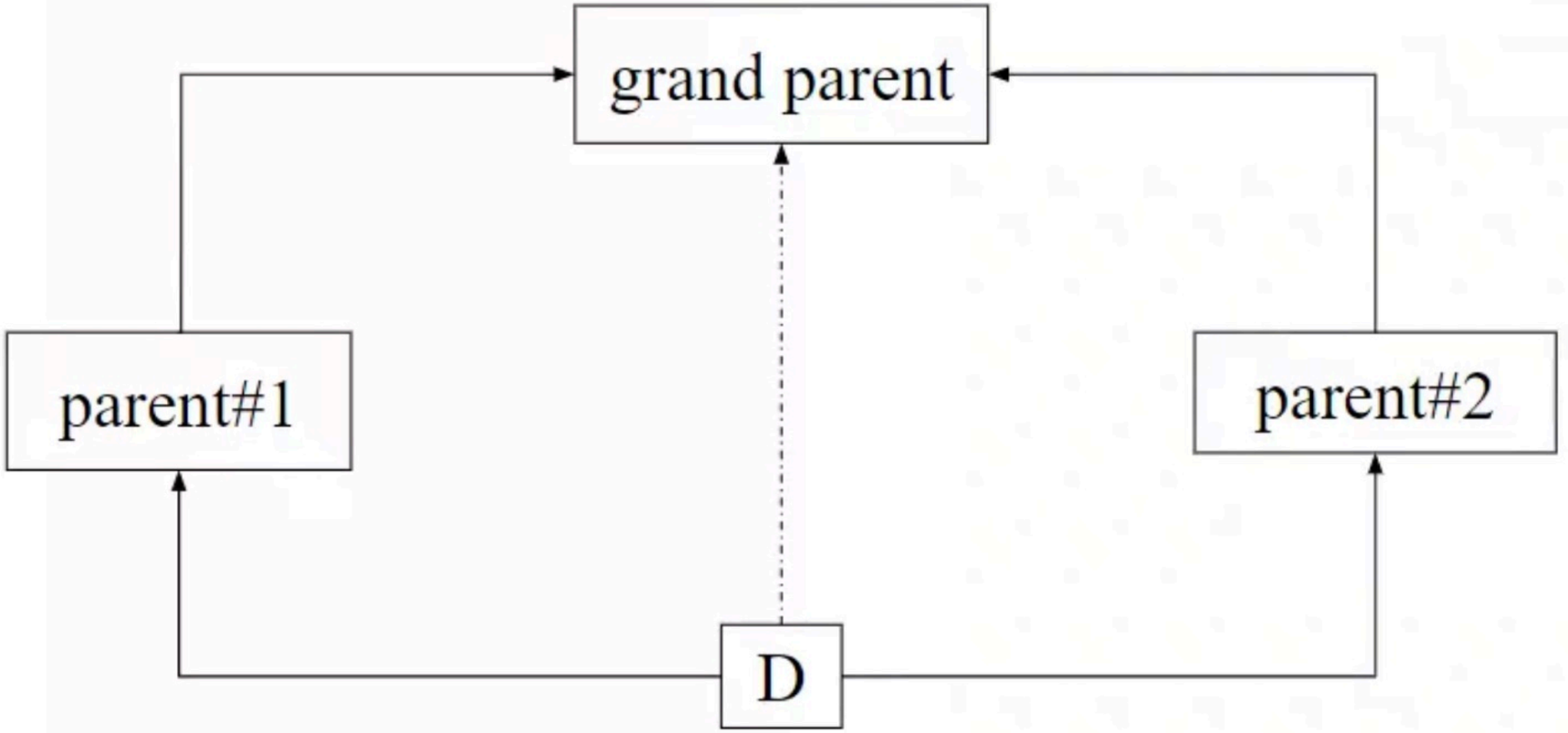
It is essential to declare the common base class as a **virtual base class** as shown below :

```
class G {
// mems of G
} ;

class P1: virtual public  G {
} ;

class P2: public virtual G {
} ;

class D: public P1, public P2 {
} ;
```

This ensures that only one copy of the parent class *G* is available in D.

```
class G {
public :
      int  i ;
} ;
class P1:public  G {
} ;
class P2:public G {
} ;
class D: public P1, public P2 {
} ;

main()
{
      D d1 ;
      cout<<d1.i ; // error due to ambiguity
}
```

In case B1 and B2 have the same parent as shown in the following figure :

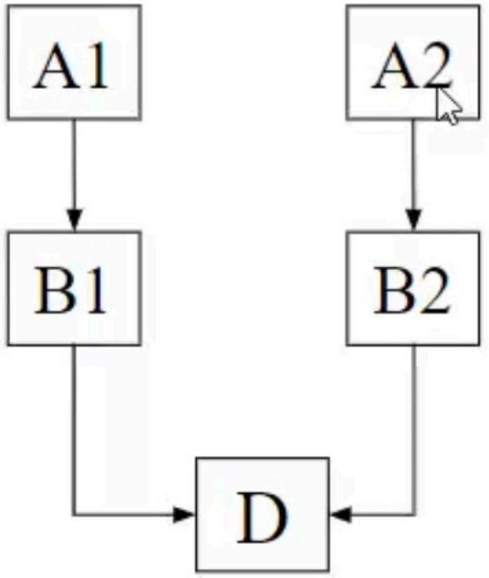


D inherits the properties of class *grandparent* through *parent#1* as well as *parent#2*.

This is an ambiguous situation.

Virtual Base Classes

For the following example :



D inherits the properties from the following classes :

- B1
- B2
- A1 through B1
- A2 through B2

Overloading, Overriding, Hiding

```
class base {
        public :
        void f(int x) {cout<<"base f -> 1arg\n" ;} // f is overloaded
        void f(int x, int y) {cout<<"base f -> 2args\n" ;}
        void g(int x) {cout<<"base\n" ;}
        void h(int x) {cout<<"base\n" ;}
} ;

class der : public base {
        public :
        void g(int x) {cout<<"der\n" ;} //g of base is overridden
        void h(int x, int y) {cout<<"der\n" ;} // h of base is hidden
} ;
```

```
class base {
   public :    int x, y ;
   void f() {cout<<"base\n" ;}
   void f(int i) {cout<<"base int\n" ;}
   void g() {cout<<"base g\n" ;}
   void h() {cout<<"base h\n" ;} } ;
class der:public base{
   public :     int x ;
   der() {x=9 ; base::x=90 ;}
   void f() {cout<<"der\n" ;}
   void f(int i, int j) {cout<<"der int int\n" ;}
   void g(int b) {} } ;
main() {
   der dobj ;
   cout<<sizeof(der)<<" "<<dobj.base::x<<" "<<dobj.x<<" "<<dobj.y<<endl ; ;
   dobj.base::f() ; //base
   dobj.f() ; //der
   dobj.base::f(9) ;
   //dobj.f(8) ; //err
   dobj.f(2,2) ;
   dobj.y = 99 ;
   //dobj.g() ;err
   dobj.h() ;
}
```

Multiple Inheritance

Syntax is as follows :

```
class D : visibility B1, visibility B2, …
{
    // mems of D
} ;
```

The following table shows the visibility of the inherited members depending on the derivation mode.

Base class visibility	Derived class visibility		
	private derivation	protected derivation	public derivation
private	not inherited	not inherited	not inherited
protected	private	protected	protected
public	private	protected	public

Derivation Modes

A derived class can be created by inheriting the base class in the following modes :

- private
- protected
- public

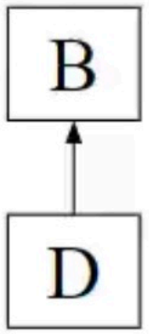
The default mode of derivation is ***private***.

```
class der-class-name : visibility mode base-class-name
{
// mems of derived class
}
```

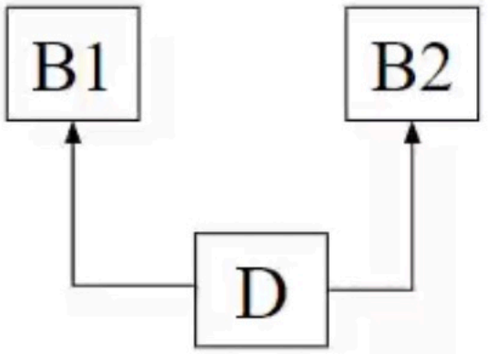
Single Inheritance

Example :

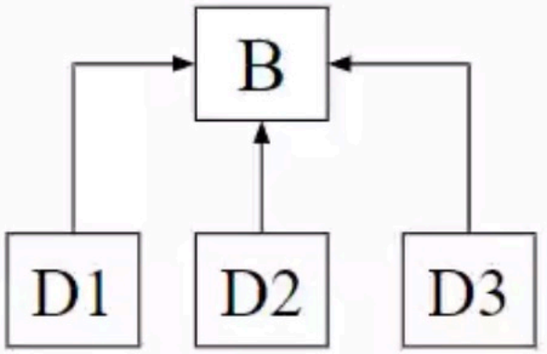
```
class  Base
{
    int  intVar;
    public:
        int i;
        void funcB(unsigned x) { /* code */ }
} ;
class Derived : public Base
{
    public:
        void funcD(unsigned x) { /* code */ }
        // belongs to the derived class
};
```



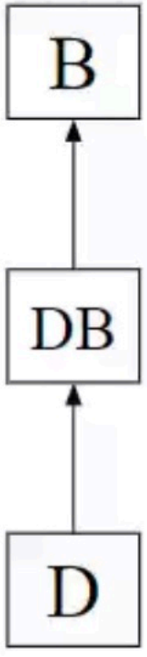
single inheritance



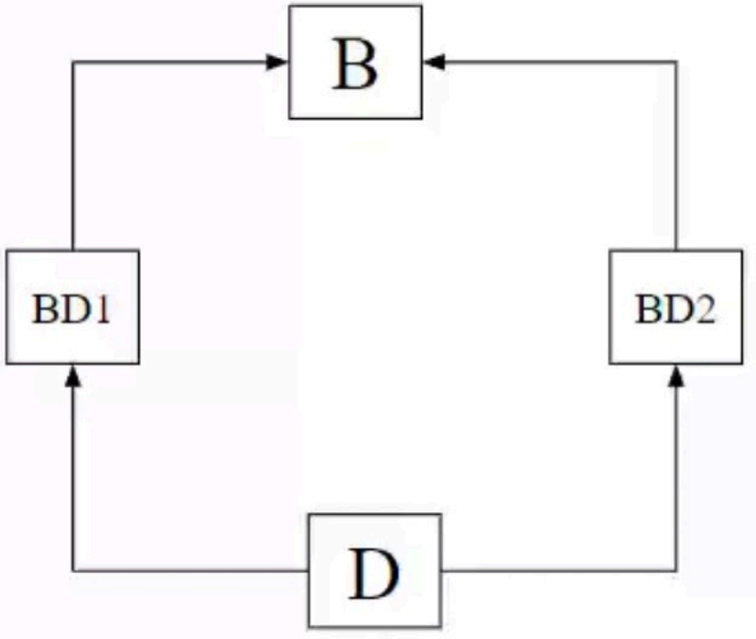
multiple inheritance



hierarchical inheritance



multilevel inheritance



hybrid inheritance

Various Forms of Inheritance

THUS, inheritance

- Provides for the reusability of code
- One class can be derived from another
- The class that inherits from another is the **derived class**
- The class inherited from is the **base class**
- An object of newly derived class is also an object of the base class

- Inheritance offers following benefits:
  - Subclasses provide specialized behaviors from the basis of common elements provided by the superclass
  - Through the use of inheritance, programmers can reuse the (object) code in the superclass many times

Inheritance

- A class inherits state and behavior from its superclass
- Example :
  - mountain bikes, road bikes, and tandems are all kinds of bicycles
  - mountain bikes, road bikes, and tandems are all *subclasses* of the bicycle class
  - the bicycle class is the *superclass* of mountain bikes, road bikes, and tandems.
- Subclass *inherits* state (data members) from the superclass
  - Mountain bikes, road bikes, and tandems share some states: eg speed
- Subclass *inherits* behaviour (methods) from the superclass
  - Mountain bikes, road bikes, and tandems share some behaviors: applyBreaks, changePedalSpeed

*Untitled - Notepad

File Edit Format View Help

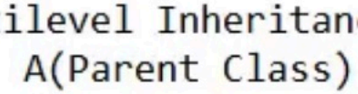
Inheritance:

1. Single Inheritance
2. Multiple Inheritance

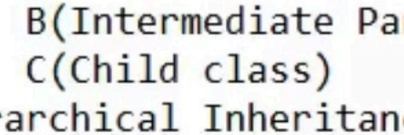
3. Multilevel Inheritance

A(Parent Class)
B(Intermediate Parent / Child class)
C(Child class)

4. Hierarchical Inheritance



5. Hybrid Inheritance



100% Windows (CRLF) UTF-8

9:36 AM

Inheritance

- A class inherits state and behavior from its superclass
- Example :
  - mountain bikes, road bikes, and tandems are all kinds of bicycles
  - mountain bikes, road bikes, and tandems are all *subclasses* of the bicycle class
  - the bicycle class is the *superclass* of mountain bikes, road bikes, and tandems.
- Subclass *inherits* state (data members) from the superclass
  - Mountain bikes, road bikes, and tandems share some states: eg speed
- Subclass *inherits* behaviour (methods) from the superclass
  - Mountain bikes, road bikes, and tandems share some behaviors: applyBreaks, changePedalSpeed