176

* private inheritance means is-implemented-in-terms of. It is usually inferior to composition. But it makes sense when a derived class needs access to protected base class members or needs to redefine inherited virtual fn.

ex.

```
class Person{ ... };
class Student : public Person{};
    void eat(const Person &p)
    void study(const Student &s)

Person p;
Student s

eat(p)
eat(s)
study(s)
study(p) // error
```

Compiler converts a derived class object into a base class object if inheritance is public

ex-1

```
class Person{ ... };
class Student{ ... };
class Student : private Person{ ... };
    void eat(const Person &p)
    void study(const Student &s)
```

177

Person p;
Student s;
eat(p)
eat(s); // error

compiler will not convert a derived class object into a Base class object if inheritance relⁿ is private.

class Student : private Person

It is not semantic mean to specialize. It is saying that student is possibly implemented as person therefore a functionality of person will not be available to student. It is a component functionality not a generalization/specialization.

protected inheritance

- does not mean generalization/specialization.
- private inheritance means something entirely diff and protected inheritance is something whose meaning eludes me to this day.

Visibility across Access and Inheritance

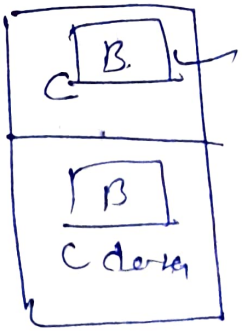
| Visibility | public inheritance | protected inheritance | private inheritance |
|---|---|---|---|
| public | public | protected | private |
| private | private | pro private | private |
| protected | protected | protected | private |

Visibility of members in Base class

Visibility of members in child class.

Example program

```
class B {
  protected:
    B() { cout << "B" << endl; }
    ~B() { cout << "~B"; }
};
class C : public B {
  protected:
    C() { cout << "C"; }
    ~C() { cout << "~C"; }
};
class D : private C {
    C data;
  public:
    D() { cout << "D"; }
    ~D() { cout << "~D"; }
};
int main()
  D d
```


B C B C D
~D ~C ~B ~C
~B

Note: In terms of private inheritance also life time of object is also not get affected.

workout example.

```
class A
{ private: int x;
    protected: int y;
    public int z;
};
class B: public A{
    private: int u
    protected: int v
    public: int w; void f() {x;}
};
class C: protected A{
    private: int u
    protected: int v;
    public: int w; void f(){x;}
};
class D: private A{
    private: int u;
    protected: int v;
    public: int w; void f() {x;}
};
class E: public B{
    public: void f() {x; u;}
};
class U: public D
{   public: void f() {x; y; z; u;}
};
void f(A &a, B &b, C &c, D &d, E &e, F &f,
       U &g)
{ a.z; b.z; b.w; c.w; e.z; e.w;
  f.w; g.w; }
```

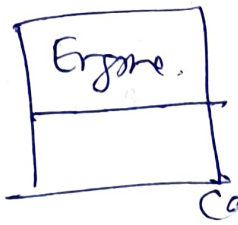
(composition or private inheritance
car has engine.

simple composition

```
class Engine
{ public:
    Engine(int numOfCylinders) { }
    void start() { }
};

class Car
{ public:
    Car() : e_(8) { }
    void start()
    { e_.start(); }
private:
    Engine e_;
};

Car c
c.start()
```

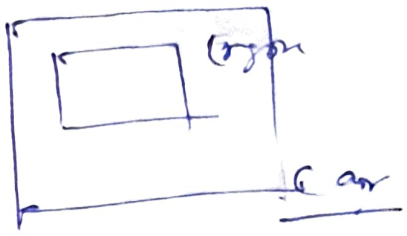



private inheritance

```
class Engine
{ public:
    Engine(int numOfCylinders) { }
    void start() { }
};
```

```
class Car : private Engine
{
public:
    Car() : Engine() {}
};
```

```
Car c
c.start()
```




~~We~~ we should use composition when we can, private inheritance when we have to.

## Casting - Basic Rule

* Casting is performed when a value (variable) of one type is used in place of some other type.

```
int i = 3;
double d = 2.5;
double res = d / i; // i cast to double and used.
```

## Casting can be implicit or explicit

```
int i = 3
double d = 2.5
double xp = (double) i;
d = i // implicit
i = d // implicit we get warning.
```

(183)

```
i = (int) d  // explicit
i = p  // error
i = (int) p  // explicit
```

* Implicit casting between unrelated classes is not performed.

```
class A { int i; };
class B { double d; };
A a
B b
A *p = &a
B *q = &b;
a = b // error
a = (A) b // error
b = a // error
b = (B) a // error
p = q // error
q = p // error
```

* we can forced cast the pointer of unrelated object.

```
p = (A*) &b // Forced okay
q = (B*) &a // Forced okay
```

(18) If we do forced conversion between unrelated classes (causes problems) then the result is unpredictable

C.

~~A a~~

class A { public: int i; };

class B { public: double d; };

A a

B b

a.i = 5

b.d = 7.2

A *p = &a

~~B~~ *q = &b

cout << p->i << endl; — 5

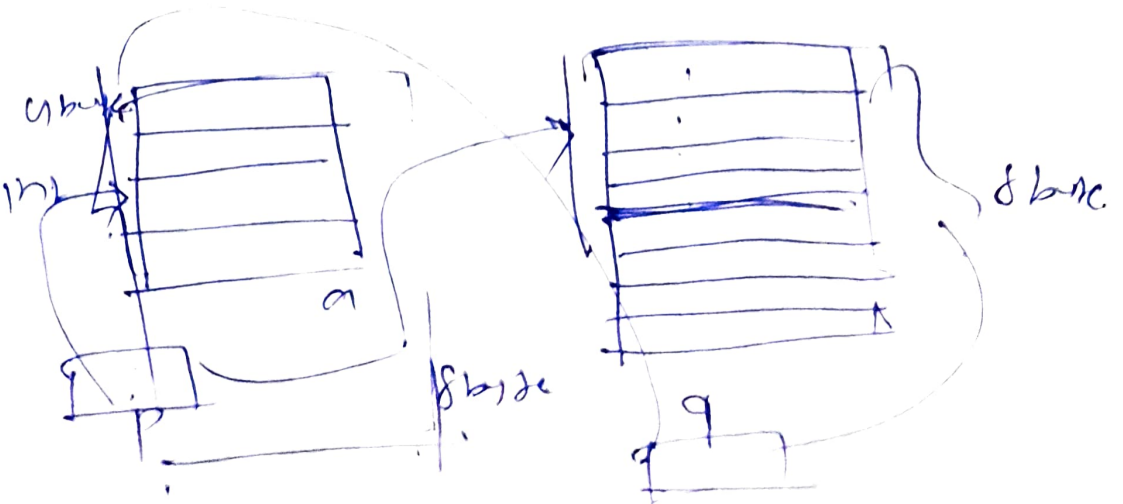
cout << q->d << endl — 7.2

p = (A*) &b ✓

q = (B*) &a

cout << p->i << endl; } — garbage o/p.

cout << q->d << endl



(184) Casting in Hierarchy

Casting in hierarchy is performed in limited ways

class A { };

class B : public A { };

A *pa = 0

B *pb = 0

void *pv = 0

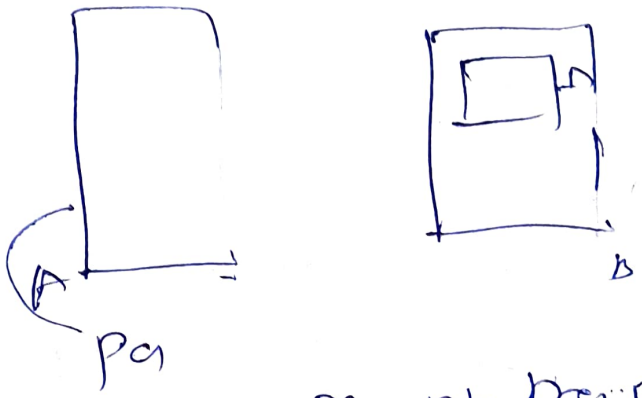
down cast

A

B

move from specialization to generalization

pa = pb // upcast




when we assign pa = pb then pa will have its exact A object part from B object which is valid.

pb = pa // error downcast Not allowed

pv = pa

pv = pb

pv = pc

} have its type

pa = pv

pb = pv

pc = pv

} Not allowed

(IES) * Base class pointer can point to derived class object. (pointer)

* Upcasting is safe

class A { public: int dataA; };

class B : public A { public: int dataB; };

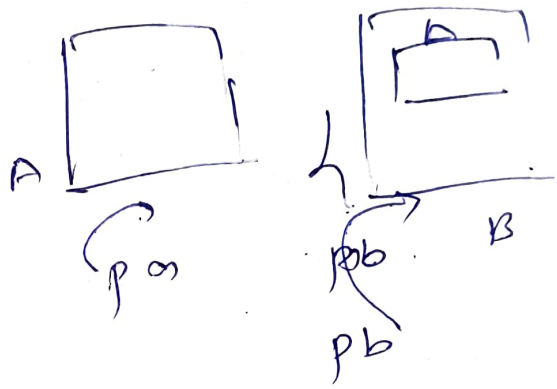
A a;

B b;

a. dataA = 2

b. dataA = 3

b. dataB = 4




A *pa = &a

B *pb = &b

cout << pa->dataA << endl; — 2

cout << pb->dataA << pb->dataB << endl

(2, 3)

pb = pa = &b

cout << pa->dataA << endl; 2

cout << pa->dataB << endl; // error

compilation

# static and dynamic binding

```
class B {
public:
    void f() { cout << "B::f()" << endl; }
    virtual void g()
    { cout << "B::g()" << endl;
};
class D: public B
{ public:
    void f() { cout << "D::f()" << endl;
    virtual void g()
    { cout << "D::g()" << endl;
};
int main()
{ B b;
  D d;
  B *pb = &b;
  B *pd = &d // upcast
  B &rb = b;
  B &rd = d;
```

| | |
|---|---|
| b.f() | ~~pb->f()~~ |
| b.g() | pb->g |
| d.f() | |
| d.g() | |

(187)

| | | |
|---|---|---|
| pb → f(); | // static binding | B::f() |
| pb → g(); | // Dynamic binding | B::g() |
| pd → f() | // static " | B::f() |
| pd → g() | // Dynamic " | D::g() |
| *b.f() | // static " | B::f() |
| *b.g() | // Dynamic " | B::g() |
| *d.f() | // static " | B::f() |
| *d.g() | // Dynamic " | D::g() |

~~If pb is of fy~~ pb and pd are of type B class. But since g() is virtual so it will do dynamic binding based on the content of pb or pd

* Given an exp the return of invocation we decide which fn to get calls
* when the binding is based on pointer type (statically)
* Dynamic binding does not depend on the type of pointer


pd → g()

B::g()

D::g()

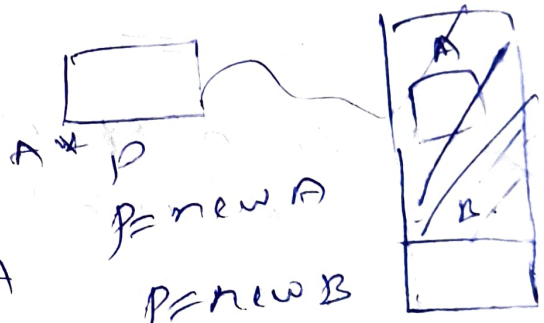
## Type of Objects

* The static type of the object is the type declared for the object while writing the code compiler sees the static type

* The dynamic type of the object is determined by the type of the object to which it currently refers. compiler does not see dynamic type.

Ex

```
class A{};
class B: public A
A *p;
p= new B();
// static type of p=A
dynamic type of p=B
```




Static Binding (early binding) when a fn invocation binds to the function definition based on the static type of object. This is done at compile time.

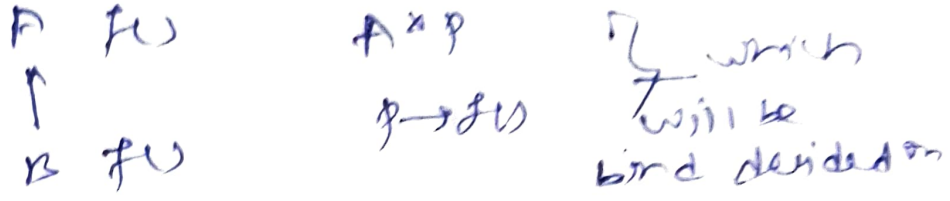
Eg. Normal fn call, Overloaded fn call, Overloaded operators.

```
void f(int)
void f()
```

f(3) f() } resolve at compile time

(IFQ) Dynamic binding (Late binding): When a fn is invoked binds to the fn definition based on the dynamic type of object. This is done at (compile / run time.

eg: fn pointers, virtual fn




actual type of object.

```
typedef void (*fp)();
fp myf;     void f()
            void g()
myf() → decided based on
        myf = f or myf = g.
```

Static binding. ex:

```
class B {public:
        void f1() {}
};
class D : public B {public:
        void g() {}
};
int main()
{
    B b;
    D d;
    b.f1(); d.f1(); d.g();
}
```

B::f1()
B::f1()
D::g()

## Overriden method

```
class B { public:
          void f() { }
};
class D: public B { public:
          void f() { }
};
   B b
   D d                          // If we override
   b.f() // B::f()              then Derived class func
   d.f() // d::f()              Base class fn. i.e base
                                class fn is not overridable.
```

## Override and overload

```
class B {
  public: void f(int);
          void g(int i);
};
class D: public B {
          void f(int)
          void f(string s)
          void h(int i)
};
   B b
   D d
   b.f(1) // calls B::f(int);
```

b.g(2) // calls B::g(int)

d.f(3) // calls D::f(int)

d.g(4) // calls B::g(int)

d.f("red") // calls D::f(string)

d.h(5) // calls D::h(int)

## Using to avoid method hiding

* If we overload any fn of Base class in child class, then the Base class fns are not available in child class. If we try to invoke Base class fn of that version in this situation with child class object then it will be compile time error. If we want to use Base class fn and overloaded child class fn in this situation then we should use using:

using BaseClassName::fnName; in child class.

Eg. class A { public: void f() {} };

class B: public A

{ void f(int) {} };

B b

b.f(3) ✓

b.f() // error

(192) using keyword is used to resolve this

```
class A{ public:
        void f(){}
};
class B: public A{
        using A::f;
        void f(int){}
};
```


```
B b
b.f(3)
b.f() // class A::f()
```

Dynamic Binding

```
class B{public: void f(){}};
class D: public B
{  public:
        void f(){}
};
B b
D d
B *p;
p=&b; p->f() // class B::f()
p=&d; p->f() // class D::B::f()
```

↓
Static binding

Can be resolved during virtual fn

```
class B {public:
        virtual void f() {}
};
class D: public B {public:
        virtual void f() {}
};
```

```
B b;
D d;
B *p;
p = &b; p->f() // calls B::f()
p = &d; p->f() // calls D::f()
```

polymorphic Type: virtual functions

* Dynamic binding is possible only for pointers and reference data types and for members functions that are declared as virtual in the Base class. These are called virtual functions

* If a member fn is declared as virtual, it can be overriden in the derived class

* If a member fn is not virtual and it is re-defined in the derived class then later defn hides the former one.

* Any class containing a virtual memb fn by defn or by inheritance is called polymorphic type (since it

(19) can take dif forms based on run time Objects

A hierarchy may be polymorphic or non-polymorphic. If the base class or some class of hierarchy has virtual fn then hierarchy as a whole is polymorphic.

→ A non-polymorphic has a structural value but it has diff the value from the computation point of view.

NOTE: If a fn is virtual in base class. then wherever the fn is overridden or inherited in derived class it will be virtual in derived class. ~~Even~~ it is not mandatory to write virtual in derived class it will be virtual.

(ii) Any non virtual fn can be made virtual at any stage from that points onwards it will be continues to be virtual.

(iii) Compiler will always take the static type of the pointer and go to the class see whether fn is virtual or non virtual if it is non virtual it will use static binding and if its is virtual it will create code for dynamic binding.

Ex: class A { public:

```
void f() { cout << "A::f()" << endl;
virtual void g()
{ cout << "A::g()" << endl;
}
void h()
{ cout << "A::h()" << endl;
}
};
class B: public A { public:
{ void f()
{ cout << "B::f()" << endl;
}
void void g()
{ cout << "B::g()" << endl;
}
void virtual void h()
{ cout << "B::h()" << endl;
}
};
class C: public B { public:
{ void f() { cout << "C::f()" << endl;
void g() { cout << "C::g()" << endl;
void h() { cout << "C::g()" << endl;
};
```

B *q = new C ;

B *q = new C;

A *p = q;

p->f()

p->g()

p->h()

q->f()

q->g()

q->h()



q

O/p:

A :: f()
C :: g()
A :: h()
B :: f()
C :: g()
C :: h()

## Virtual destructor

```
class B {
    int data;
public:
    B(int d): data(d)
    { cout<<"B()"<<endl;
    }
    ~B() { cout<<"~B()"<<endl;}
    virtual void print()
    {   cout<<data<<endl;
    }
};
```

```
class D: public B
{
    int *ptr;
public:
    D(int d1, int d2): B(d1); ptr(new
    int(d2)
    {
        cout<<"D()"<<endl;
    }
    ~D()
    {
        cout<<"~D()"<<endl;
        delete ptr;
    }
    void print()
    {
        B::print();
        cout<<" "<<*ptr;<<endl;
    }
};
```

* Destructors are never inherited

```
B *p= new B(2);
B *q= new D(2,3);
p->print();
q->print()
delete p;
delete q
```

O/P: of B()
B()
D()
2
2, 3
~B()
~B()

Prob: Here destructor of D is not called.

Soln: make destructor of B as virtual.

(198) making class B destructor as virtual

```
class B {
    int data;
public:
    B(int d) : data(d) { cout << "B()" << endl; }
    virtual ~B()
    { cout << "~B()" << endl; }
    virtual void print()
    { cout << data << endl; }
};
class D : public B
{   int *ptr;
public:
    D(int d1, int d2) : B(d1), ptr(new
      int(d2))
    { cout << "D()" << endl;
    ~D()
    { cout << "~D()" << endl;
      delete ptr;
    }
    void print()
    { B::print();
      cout << *ptr << endl;
    }
};
```

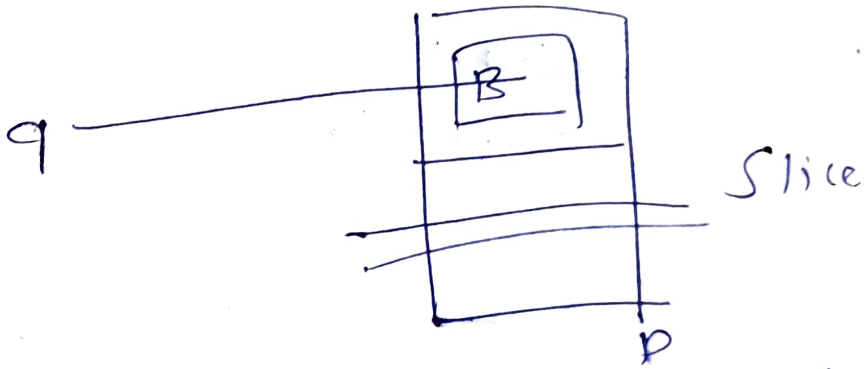
(199) B *p = new B(2);

B *q = new D(2,3);

p->print();
q->print();
delete p;
delete q;

O/p:
B()
B()
D()
2
2,3
~B()
~D()
~B()

Note: If destructor of Base class is virtual then by inheritance destructors of Derived class will also be virtual. But we do not explicitly write in Derived class.




If the destructor is not virtual in a polymorphic hierarchy it leads to slicing

Destructor must be virtual in the base class.

## pure virtual function

* A pure virtual fn has signature but no body

## Abstract Base class

* A class containing atleast one pure virtual function is called an Abstract Base class
* A pure virtual fn may be inherited or defined in the class.
* No instance can be created for an abstract base class.
* Naturally it does not have constructor destructor
* An abstract base class, however may have other virtual (non pure) and non-virtual member fn as well as data members.
* Data members in an abstract base class should be protected. Of course private and public data are also allowed
* member fn in an abstract base class should be public Of course, private and protected methods are also allowed.
* A concrete class must override and implement all pure virtual fn so that it can be instantiated.