

(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)
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)

(77)

Person p:

Student s:

eat(p)

ear(s) / savor

compilers will not convert
a derived class object
into a base class
object if inheritance
reln is private.

class student: private Person

This is not semantic means to specialize. This
says 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 general
ization/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

	public	protected	private
public	public	protected	private
private	private	private	private
protected	private	protected	private
Visibility of members in child class.			
Visibility of members in Base class.			

Q 188

example program

Class B {

protected:

B() { cout << "B" << endl; }

~B() { cout << "h.B"; }

}

Class C : public B {

protected:

C() { cout << "C" << endl; }

~C() { cout << "h.C"; }

}

Class D : private C {

C data;

public:

D() { cout << "D" << endl; }

~D() { cout << "h.D"; }

}

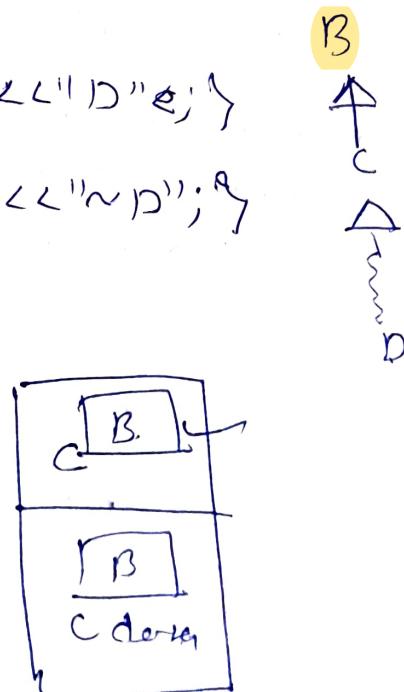
int main()

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.

(179)

Class A

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 B: public D {

public: void f() { x; y; z; u; }

{}

void f(A x, B y, C z, D w, E v, F p,

(78g)

{ x; y; z; w; v; p; }

{ u; }

Composition or private inheritance

(180)

Car has engine

Simple composition

Class Engine

↳ public:

Engine (int numofcylinders) { }

void starts () { }

}

Class Car

↳ public:

Car() : e_(8) { }

void starts()

↳ e_.starts(); }

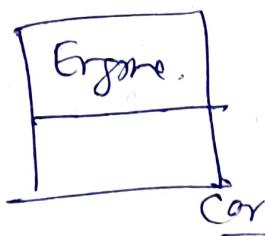
private:

Engine e_;

}

Car e

C.starts()



private interface

Class Engine

↳ public:

Engine (int numofcylinders) { }

void starts () { }

}

Cancer: private type

(181)

↳ public:

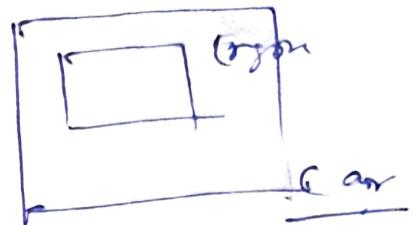
Car(): Engine (F) { }

word engine:: start;

?;

Car C

constant



* 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 = 2*d;

d = i // implicit

i = d // implicit we get warning

(182)

$i = (\text{int}) d$ // explicit

$i = p$ // error

$i = (\text{int}) p$ // error

~~Implicit casting~~ between unrelated classes is not performed.

Class A { int a; };

Class B { double b; };

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.

$p = (\text{A}^*) \&b$ // forced okay

$q = (\text{B}^*) \&a$ // forced okay

(P3) If we do forced casting from unregistered classes
(using pointers) then the result is unpredictable

C.

A a

Class A L public: int i;

Class B L public: double d;

A a

B b

$$a.i = 5$$

$$b.d = \underline{7.2}$$

$$A * p = \&a$$

~~$$B * q = \&b$$~~

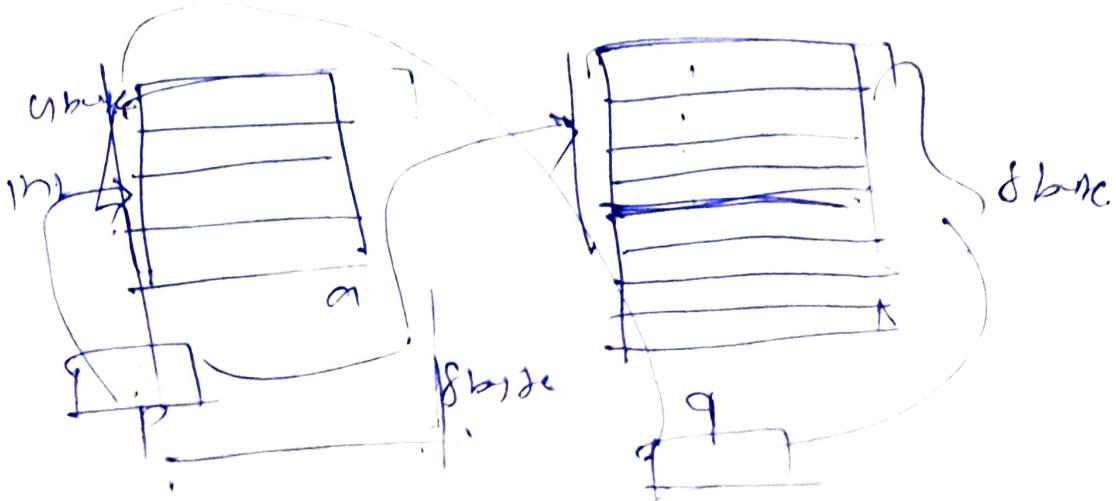
control p \rightarrow i << endl; — 5

control q \rightarrow d << endl; — 7.2

$$p = (A *) \&b \checkmark$$

$$q = (B *) \&a$$

control p \rightarrow i << endl; }
control q \rightarrow d << endl; } } garbage
O/P.



(184)

Converting a hierarchy

casting on hierarchy is performed in limited sense

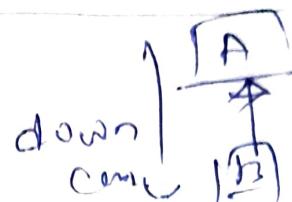
Class A {;

Class B: public A {;

$$A * pa = 0$$

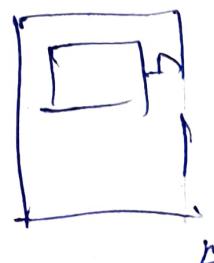
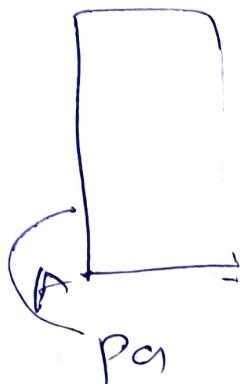
$$B * pb = 0$$

$$void *pv = 0$$



↑ move from
specialization
to generalization

$pa = pb \text{ / upcast}$



Given we assign $pa = pb$ then pa will have
an exact A object part from B object
which is valid.

$pb = pa \text{ / downcast Not}$
allowed

$$pv = pa .$$

$$pv = pb .$$

$$pv = pc .$$

more is type

$$pa = pv$$

$$pb = pv$$

$$pc = pv$$

Not allowed.

(15) ~~* Base cast class pointer can't or reference
for take child class object (pointer)~~

Hypothetical code

class A { public: int dataA; };

class B { public:

 A dataA; int dataB; };

A a;

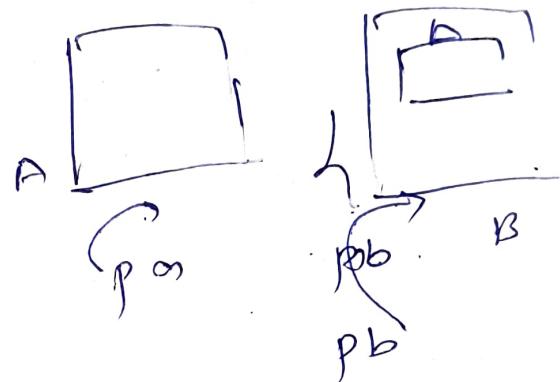
B b;

C dataA=2

or dataA=2

b. dataA=3

b. dataB=4



$$\underline{A} \times pa = \&a$$

$$\underline{B} \times pb = \&b$$

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

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

$$pb = pa = \&b$$

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

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

(186)

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 &r = b;

 B &r = d;

b.f()

b.g()

d.f()

d.g()

pb->f()

pb->g()

187

$\text{pb} \rightarrow \text{f1();}$ // static binding B::f1()

$\checkmark \text{ph} \rightarrow \text{g1();}$ // Dynamic binding B::g1()

$\text{pd} \rightarrow \text{f1();}$ // static "

B::f1()

$\checkmark \text{pd} \rightarrow \text{g1();}$ // Dynamic "

D::g1()

pb.f1(); // static "

B::f1()

pb.g1(); // dynamic "

B::g1()

rd.f1(); // static "

B::f1()

$\checkmark \text{rd.g1();}$ // dynamic "

D::g1()

* If ~~pb is of type~~ pb and pd are of type B class. But since g1() is virtual so it will do dynamic binding based on the content of pb or pd

* Given one exp. interpretation of invocation we decide which fn to get call

* When no binding is based on pointer type (statically)

* Dynamic binding does not depend on the type of pointer



$\text{pd} \rightarrow \text{g1();}$

B::g1()

$\Rightarrow \text{B::g1();}$

(18)

Type of Objects

- * The static type of the object is the type declared for the object, while writing the code (compile time).
- * 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.

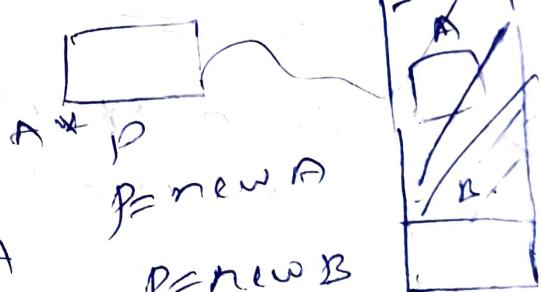
Ex Class A {

Class B : public A

A * p;

p = new B();

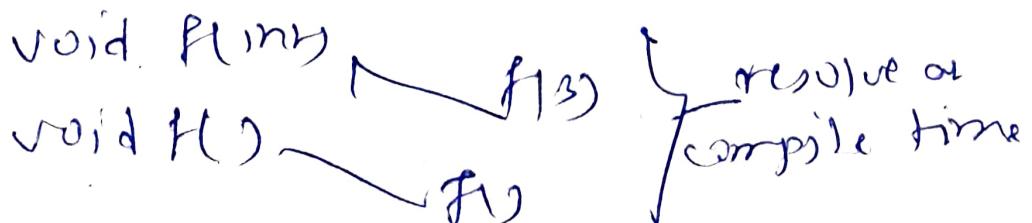
// static type of p=A



dynamic type of p=B

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

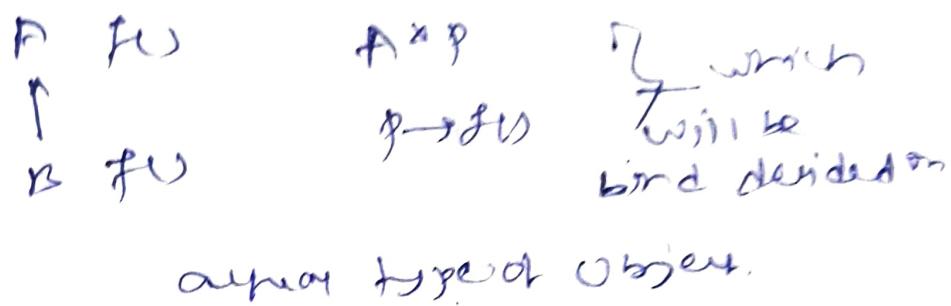
e.g. Normal funcy, Overloaded funcy,
overloaded operators.



(189) Dynamic Binding (late binding): When a function
variable binds to the function definition based on the
dynamic type of objects. This is done at
runtime.

sur hme

e.g. for purposes, virtual fn



fynched wi^d (x^tp) ()

fp myt: void HS
void gy

myf(t) \rightarrow decided to use t_0
• myf = f or myf = g.

Stonic binding.

class B < public:
void func;

3;

class **B**: public **A** {
public:
 void g() {}}

3

mt $B \xrightarrow{b} B \cdot H$
 $D \xrightarrow{d} D \cdot H$
 $B \cdot H : d \cdot H ; \underline{d \cdot g} \leq D \cdot g$

Overridden method

(190)

Class B { public:
 void f() {}
};

Class D: public B { public:
 void f() {}
};

B b
D d

b.f() // D:: f() Then derived class must
d.f() // d:: f() Base class fn. i.e base
class fn is unavailable.

Override & overload

Class B {
 public: void f(int);
 void g(int);
};

Class D: public B {
 void f(int)
 void f(string)
 void h(int)
};

B b
D d

b.f() // calls B:: f();

- (191)
- b. g(2) // (ans B:: g(m))
 - d. f(3) // (ans D:: f(m))
 - d. g(4) // (ans B:: f(m))
 - d. f(red')) // (ans D:: f(m))
 - d. h(5) // (ans D:: h(m))

Using to avoid method hiding

→ If we overload ~~only~~ b fn of Base class in child class, then the Base class fn is not available in child class. if we try to invoke Base Class fn of that version is 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

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

class B : public A

h. void f(m) {}

y;

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() { }

}

B b

b.f()

b.f() // calls A::f()

Dynamic Binding

class B public: void f() { }

class D: public B

 public:

 void f() { }

}

B b

D d

~~B~~ B *p;

p = &b; p->f() // calls B::f()

p = &d; p->f() // calls D::f()

Dynamic Binding

(193)

Can be resolved during Virtual fn

class B public:

virtual void f() {}

y:

Class D: public B & public:

virtual void f() {}

y:

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 overridden 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

- (194) can take diff forms based on run time Obj.
- A hierarchy may be polymorphic or non-polymorphic. If one 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 little relevance for computation point of view.

~~Note~~ If a fn is virtual in base class, then wherever this is overidden or inherited in derived class it will be virtual in derived class. ~~Even if~~ if it is not overridden by derived class it will be virtual.

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

(11) Compiler will always ~~not~~ take the static type of the pointer and goto the class see whether the fn is virtual or non virtual if it is non virtual it will use static binding and if it is virtual it will create code for dynamic binding.

Ex class A : public : (195)
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 A() { cout << "C:: A()" << endl;

void g() { cout << "C:: g()" << endl;

void h() { cout << "C:: h()" << endl;

?
.

~~B * q = new C ; (19)~~ ↗

~~A D p~~ ↗
B * q = new;

p → f()

↙ p → g()

p → h()

q → f()

↙ q → g()

q → h()

B * q = new C;

A * p = q;



P → C

q

Op: A :: fu

C :: gu

A :: hu

B :: fu

C :: gu

C :: hu

Virtual destructor

Class B {

int d;

public:

B (int d) : ~~d (d)~~ (d)

~B () cout << "B ()" << endl;

}

~B () cout << " ~B ()" << endl; }

Virtual void print()

~ cout << "descend!"

Y;

class D: public B

(197)

↳ 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;

↳

* Destruktoren

wegen

new

};

B *p = new B(2);

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

p->print();

obj of B()

B()

D()

2

2, 3

~B()

~B()

~~q->print()~~

~~q->print()~~

~~q->print()~~

~~q->print()~~

~~q->print()~~

✓ prob: here destructor of B is not called.

Soln: make destructor of B virtual.

(98)

making class B destroyer as virtual.

Class B
int d1
public:
B(int d) : d1(d) {} cout << "B(" << endl;
Virtual ~B()
{ cout << "~B()" << endl; }
Virtual void print()
{ cout << "class B" << endl; }
};
Class D: public B
int d2
public:
D(int d) : m1(d) {} cout << "D(" << endl;
m1(d2)
cout << "D()" << endl;
~D()
{ cout << "~D()" << endl; }
Delete ptr
};
void print()
{ B::print();
cout << *ptr << endl; }
};

(199) $B \times P = \text{new } B(2)$

$B \times Q = \text{new } D(2, 3)$

$P \rightarrow \text{ptr}(p); \quad \underline{D(p)}: \quad B()$

$Q \rightarrow \text{ptr}(q); \quad B()$

$\text{delete } p; \quad D()$

$\text{delete } q; \quad 2$

$2, 3$

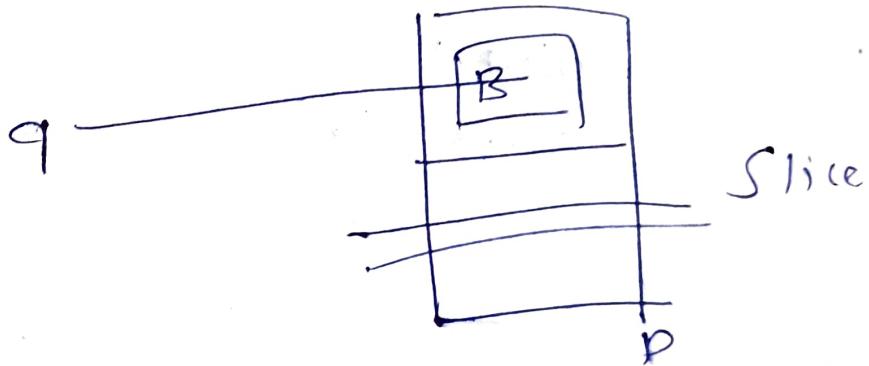
$\sim D()$

$\sim D()$

~~Note: If destructor of Base class is ~B()~~

~~virtual member inheritance destruction~~

~~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 (20)

* A pure virtual fn has signature but no body

Abstract Base Class

* A class containing at least 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;