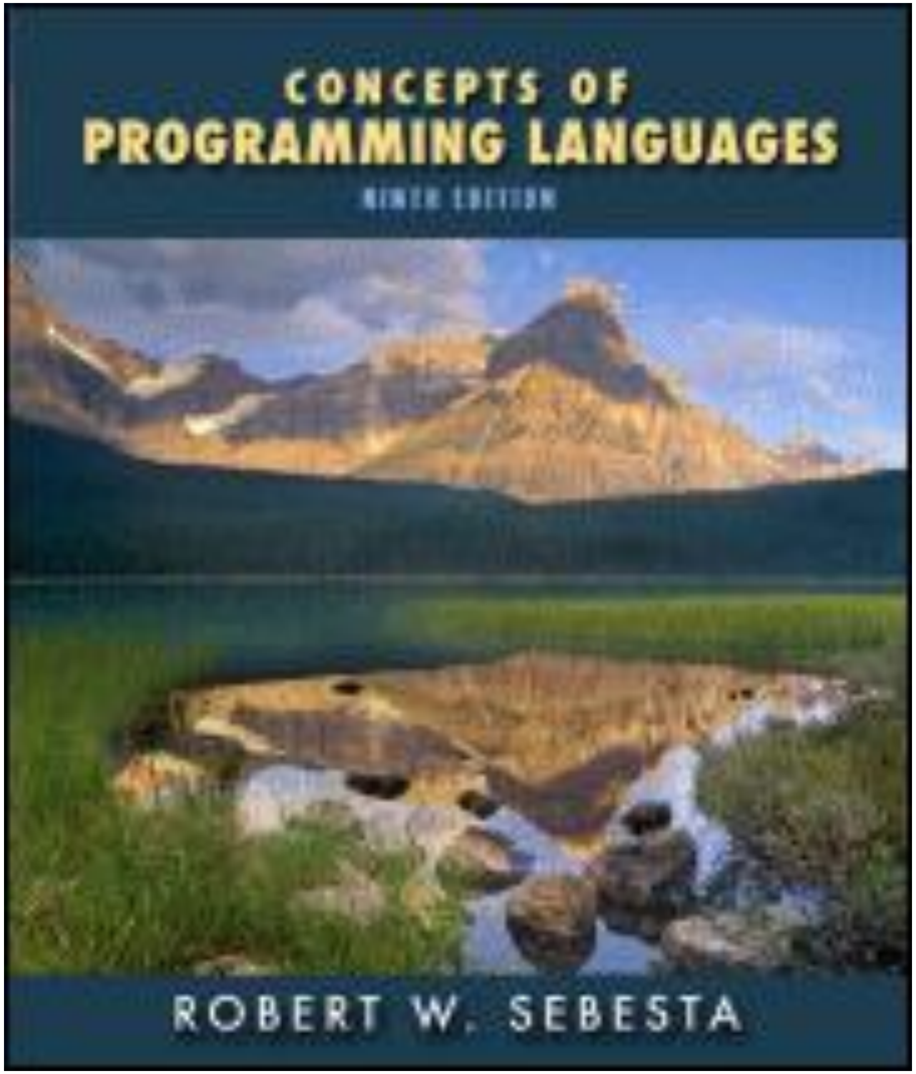
Chapter 12

Support for Object-oriented Programming



Ch12 – Support for Object-oriented Programming

12.1~12.9 Skipped

12.10 Implementation of Object-Oriented Constructs

Implicit object parameter

- Implicit object parameter
  - class A {

    public: compiled to

    A() : x(0) {} ⇒ A(A* this) : this->x(0) {}

    void f() { x++; } ⇒ void f(A* this) { this->x++; }

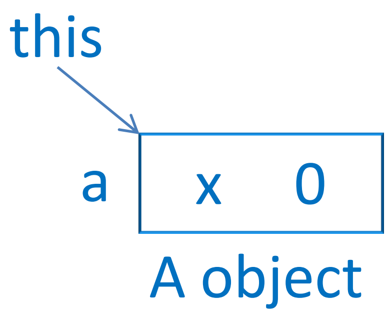
    private:

    int x;

    };

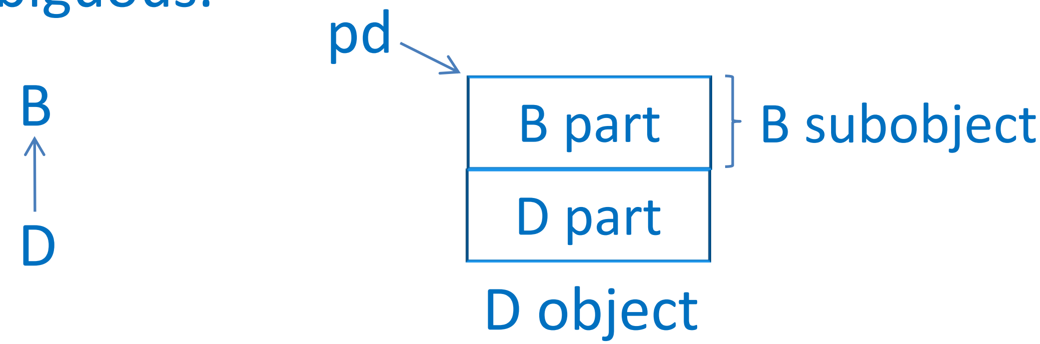
    A a; ⇒ A(&a);

    a.f(); ⇒ f(&a);



Standard pointer conversion

- Standard pointer conversion
  - D* can be (implicitly) upcast to B* or
    D can be (implicitly) upcast to B&,
    if B is accessible (i.e. a public member of B is accessible)
    and unambiguous.



- Upcast with single inheritance
  - Easy, the pointer remains the same, only its type changes.
  - No ambiguity

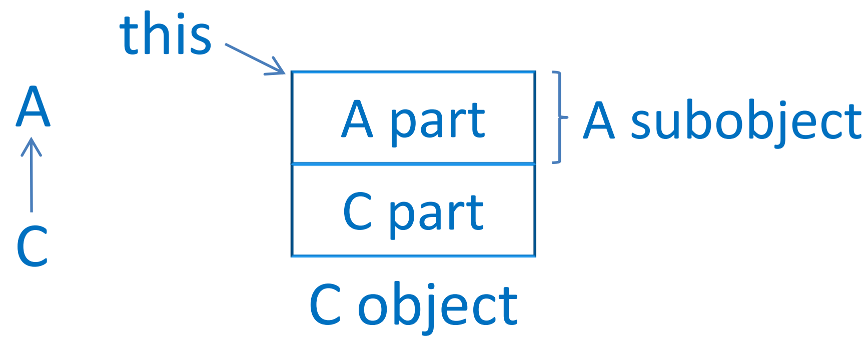
Single inheritance

- Upcast with single inheritance
  - class A { public: void f() {} }   ⇒ void f(A* this) {}

    class C : private A {   upcast C* → A*

    public: void g() { this->f(); }   ⇒ void g(C* this) { f(this); }

    };

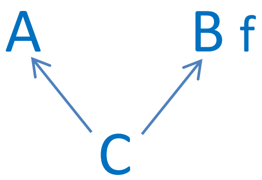


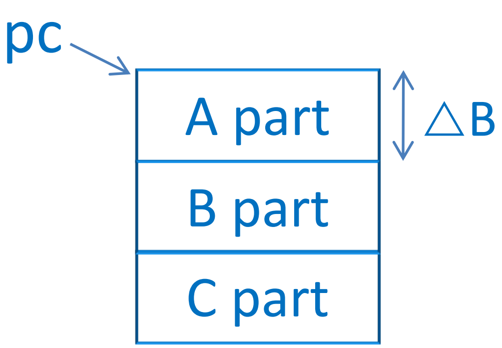

```
void X()
{
    A& ra = *new C;      // no, A is inaccessible
    ra.f();              // otherwise, one can access C's
}                        // private member f() through ra
```

Multiple inheritance

- Upcast with multiple inheritance
  - May need to adjust the pointer with some offset
  - May be ambiguous
  - class A {};
class B { public: void f(); };
class C: public A, public B {};

    C* pc = new C;
pc->f()
⇒ ((B*) pc) ->f()
⇒ ((B*) ((char*) pc+△B))->f()



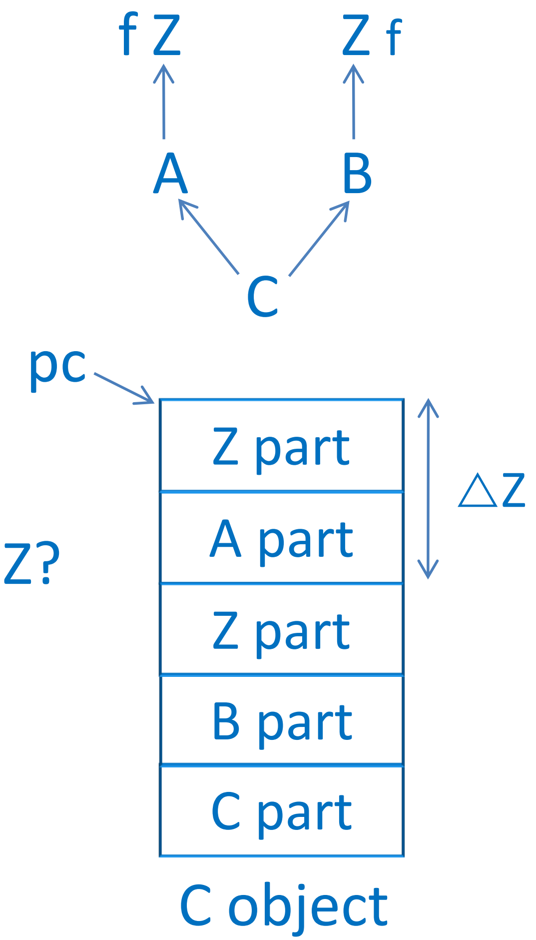


C object

Multiple inheritance

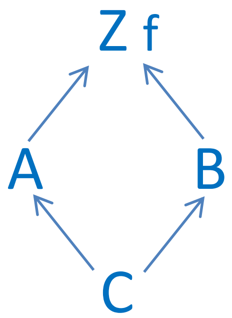
- Upcast with multiple inheritance
  - ```
    class Z { public: void f(); };
    class A: public Z {};
    class B: public Z {};
    class C: public A, public B {};

    C* pc = new C;
    pc->f()              // ambiguous, which Z?
    pc->B::f()           // ok, C::B::Z
    ((B*)pc) ->f()       // the same
    ((Z*)(B*)pc)->f()    // the same
    ```

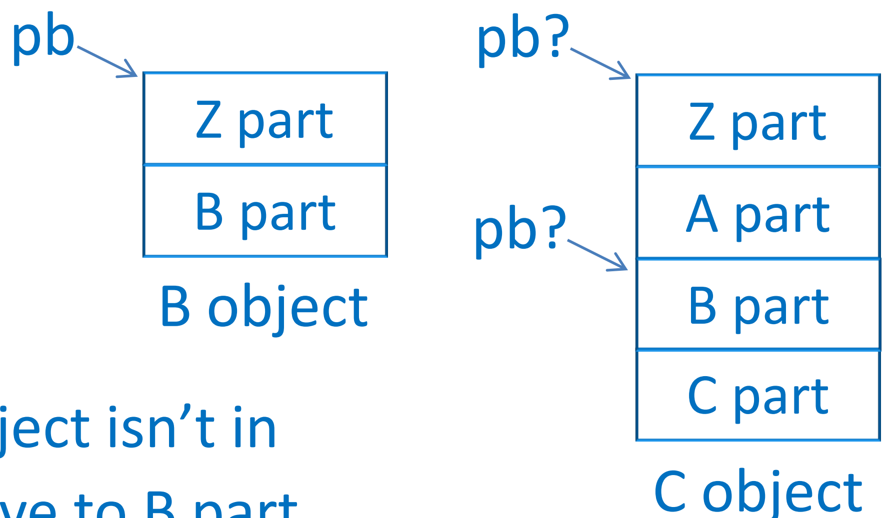

    ```

Virtual base

- Virtual base
  - class Z { public: void f(); };
    class A: public virtual Z {};
    class B: public virtual Z {};
    class C: public A, public B {};



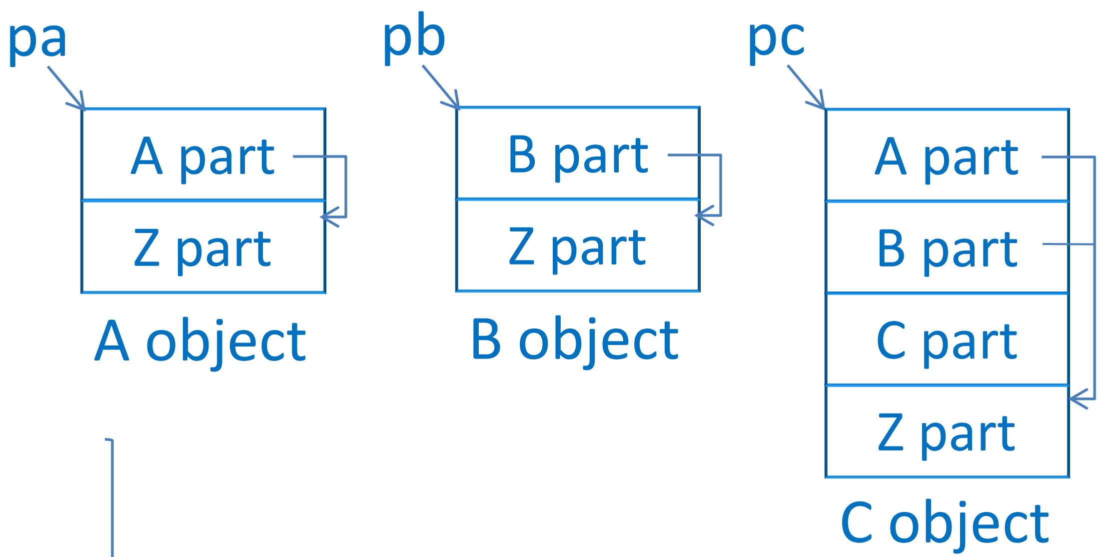
B* pb = new B;
B* pb = new C;



Bad layout, the Z subobject isn't in the same position relative to B part.

Virtual base

- Virtual base
  - The solution is to store a pointer to the Z subobject in all objects of classes that have Z as a virtual base.
  - A* pa = new A;
    B* pb = new B;
    C* pc = new C;

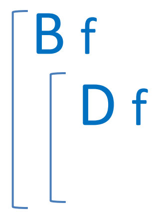


pa->f()
pb->f()
pc->f(), ((A*)pc)->f()
((B*)pc)->f()

Use the stored pointer to upcast

Member name lookup

- Member name lookup
  - Class hierarchy considered as block structure



D::f hides B::f

  - In

exp.f   // f is looked up in the static type of exp

exp.A::f   // f is looked up in the qualified type A

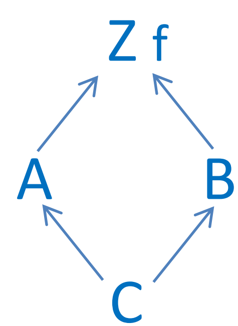
where A must be a base type of the static type of exp.

e.g. C* pc = new C

pc->f(), i.e. (*pc).f()

pc->A::f()

((A*)pc)->f()

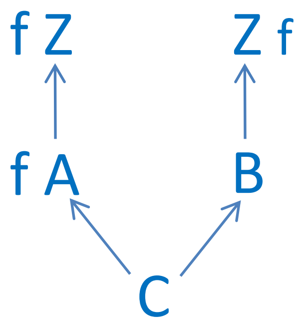


Member name lookup

- Member name lookup
  - It is ambiguous, if the resulting set of declarations are not all from subobjects of the same type.

    e.g. C* pc = new C;

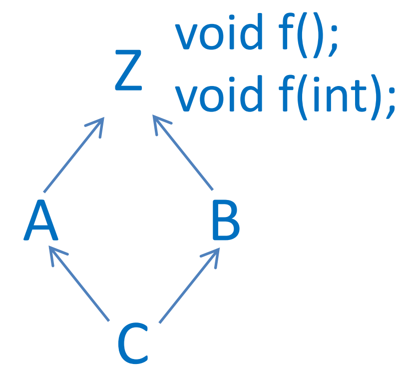
    ```
    pc->f()           // ambiguous
    pc->A::f()        // A::f
    pc->B::f()        // B::Z::f
    pc->A::Z::f()     // A::Z::f
    ```



  - Otherwise, apply overloading resolution to the resulting set of declarations

    e.g. C* pc = new C;

    ```
    pc->f()
    ```



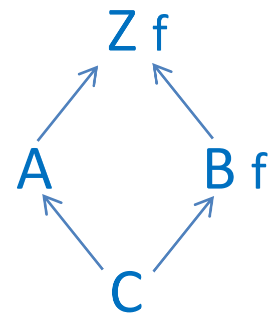
Member name lookup

- Member name lookup
  - Any hidden declaration of f is eliminated from consideration

    e.g. C* pc = new C;

    pc->f()   // B::f

    pc->A::f()   // Z::f

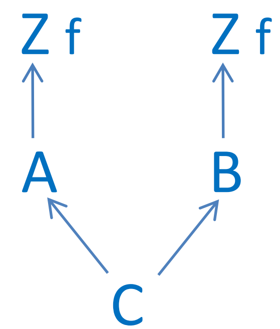


  - It is ambiguous, if the resulting set of declarations has a nonstatic member and includes members from distinct subobjects of the same type.

    e.g. C* pc = new C;

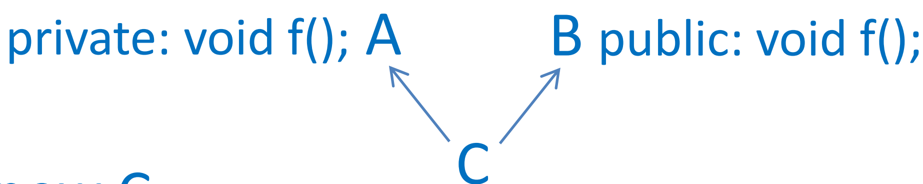
    pc->f()   // ambiguous, unless f is static

    pc->A::f()   // A::Z::f



Member name lookup

- Member name lookup
  - Name lookup → overload resolution → access control
  - Name lookup → access control



C* pc=new C;

pc->f()   // ambiguous, even if A::f is private

Why?

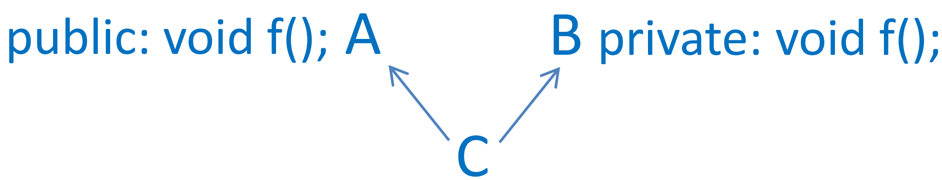
Access control is related to data encapsulation, and has nothing to do with the meaning of a program.

Thus, changing the accessibility of a class member should not change the meaning of a program.

Member name lookup

- Member name lookup
  - (Cont'd)

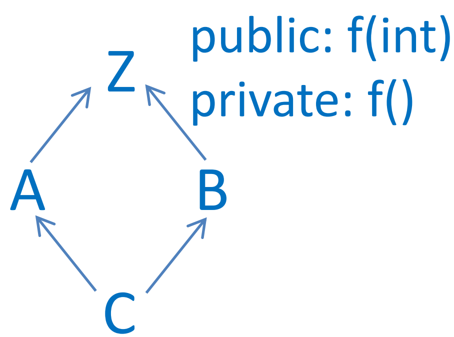
    Were access control taken place before name lookup, B::f would be selected above, but A::f would be selected below



  - Overload resolution → access control

    C* pc=new C;

    pc->f()   // error, call private Z::f()



    Were access control taken place before overloading resolution, the error would be "no viable function".

Virtual function

- Virtual function
  - A virtual function call depends on the object’s dynamic type
  - class A {

```
public: virtual void f();
};
class C: public A {
public: virtual void f();   // overwrite A::f
};
A* pa=new A;                // dynamic type A; static type A
pa->f();                    // call A::f
A* pa=new C;                // dynamic type C; static type A
pa->f();                    // call C::f
```

OOP = Virtual function
+ Inheritance

Virtual function lookup

- Virtual function lookup
  - Step 1

    Use member name lookup to resolve the function as usual.

    (The virtual specifier is ignored in this step.)

    (It is erroneous, if ambiguity arises.)
  - Step 2

    Find the *unique* final overrider of the virtual function along the path(s) from the object's dynamic type to the class containing the resolved function.

    (The class hierarchy is ill-formed, if there is no unique final overrider.)

Virtual function lookup

- Virtual function lookup
  - Example

A f
↑
C f

A* pa=new C;		
pa->f()	step 1: A::f	step 2: C:f
C* pc=new C;		
pc->f()	step 1: C::f	step 2: C:f
A* pa=new A;		
pa->f()	step 1: A::f	step 2: A:f

Convention: A virtual function overrides itself.

Virtual function lookup

- Virtual function lookup
  - Difference between step 1 and step 2

Step 1

Where to start	the static or qualified type
Where to search	all base types of the static or qualified type
Condition	hiding (parameters aren't considered)

Step 2

Where to start	the dynamic type
Where to search	only the path(s) from the dynamic type to the class containing the resolved function
Condition	overriding (parameters are considered)

Virtual function lookup

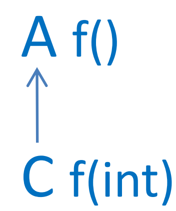
- Virtual function lookup
  - C* pc=new C;

    pc->f()   step 1 C::f(int) not viable

    A* pa=new C;

    pa->f()   step 1 A::f()

    step 2 A::f()

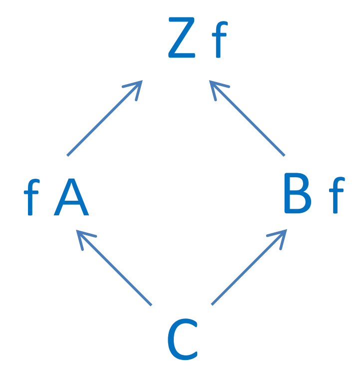


It doesn’t matter if C::f is virtual or not

  - Z* pz=new C;

    pz->f()   step 1 Z::f

    step 2 error



This class hierarchy is illegal, since Z::f has two final overriders: A::f along the path Z-A-C , and B::f along Z-B-C

Virtual function lookup

- Virtual function lookup
  - Z* pz=new C;

    pz->f() step 1 Z::f

    step 2 A::f (downcast)

    B* pb=new C;

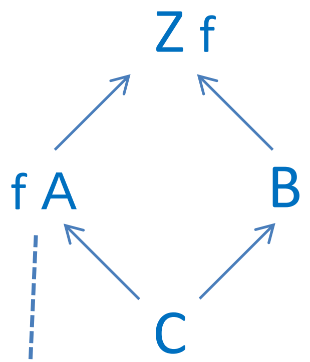
    pb->f() step 1 Z::f

    step 2 A::f (crosscast)

    C* pc=new C;

    pc->f() step 1 A::f

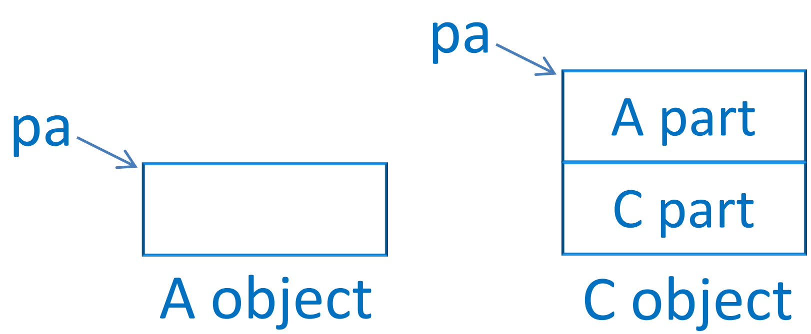
    step 2 A::f (upcast)



Inheritance via dominance since A::f dominates.

Implementation of virtual functions

- How to tell the dynamic type?
  - 

    A* pa=new A;

    A* pa=new C;

    As such, there is no way to tell the dynamic type of *pa.

    i.e. if pa points to a stand-alone or an embedded A object?

Implementation of virtual functions

- Virtual table (vtble)
  - There is one vtble per polymorphic class (i.e. class that declares or inherits virtual functions) .
  - The vtble contains information about virtual functions; nonvirtual functions are not in the vtble.
- Virtual pointer (vptr)
  - Each object contains a vptr pointing to the vtble of its class.
- Single inheritance
  - $$\begin{tikzcd} A \text{ ~A,f,g} \\ C \text{ ~C,f} \arrow[u] \end{tikzcd}$$

    Assume that all members are public virtual and all inheritances are public.

    Virtual dtors guarantee correct deletion, e.g.

    A* pa=new C; delete pa;

Implementation of virtual functions

- Single inheritance
  - Virtual table

A ~A,f,g

↑

C ~C,f

By analyzing this inheritance lattice, the compiler constructs a virtual table for each class.

code pointer

offset

A::~A	0
A::f	0
A::g	0

A's vtble

Dynamic type A

stand-alone A objects

C::~C	0
C::f	0
A::g	0

C/A's vtble

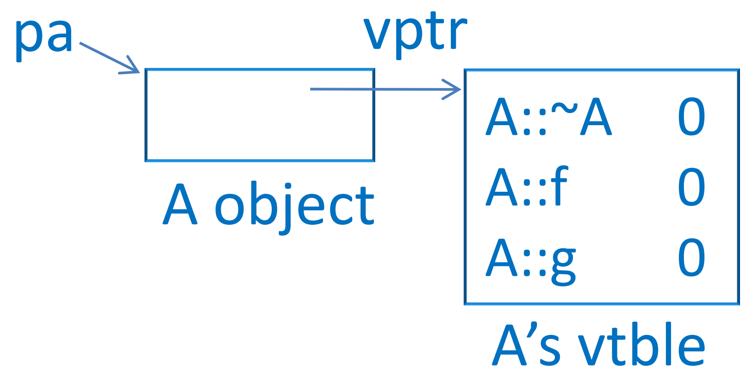
Dynamic type C

C objects or embedded A objects

Implementation of virtual functions

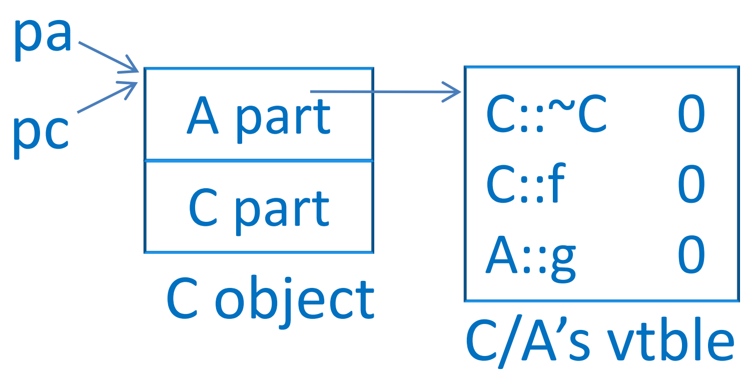
- Single inheritance
  - Virtual pointers

A* pa=new A;



A* pa=new C;

C* pc=new C;



The call pa->f() can tell the dynamic type of *pa, but the call pa->g() can't. The point is: It doesn't matter.

Implementation of virtual functions

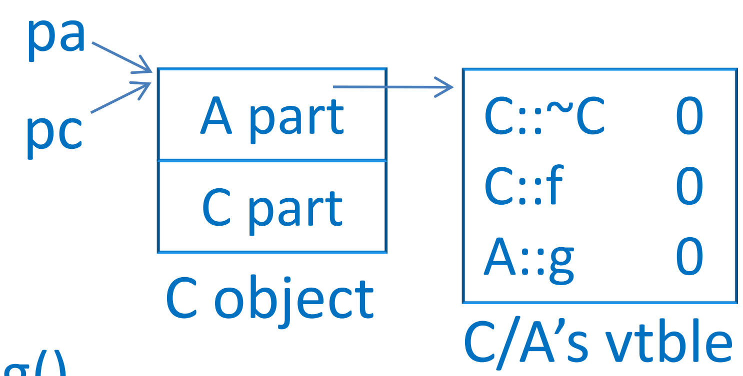
- Single inheritance
  - Observation 1 – Offsets help upcast, downcast, and crosscast

A ~A,f,g

C ~C,f

A* pa=new C;

C* pc=new C;



pc->g() upcasts to ((A*)pc)->g()

pa->f() downcasts to ((C*)pa)->f()

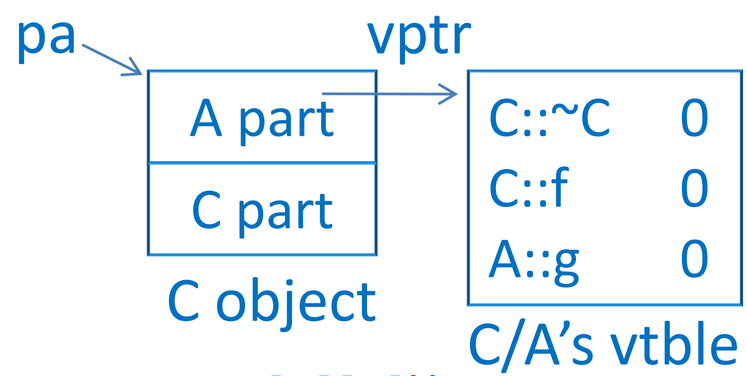
Due to single inheritance, the pointers remain the same in both cases. Thus, the offsets are all 0's.

N.B. If a language allows only single inheritance, the offsets needn't be stored in virtual tables.

Implementation of virtual functions

- Single inheritance
  - Observation 2

    The set of virtual functions of a class is fixed. So, there is no need of searching the vtble – the compiler knows where each function is.

    For example,

    pa->f()

    is compiled to

    pa->vptr[1][0]((C*)((char*)pa+pa->vptr[1][1]))

    N.B. The small piece of code used to implement virtual function call is called a *thunk*. (The vtble contains code pointers + offsets – similar to closures.)

Implementation of virtual functions

- Single inheritance
  - Observation 3

    Wherever it appears, the vtble of a class must have the same layout.

    For example, stand-alone A's vtble and embeded A's vtble must have the same layout so that

    pa->f()

    can always be compiled to

    pa->vptr[1][0]((C*)((char*)pa+pa->vptr[1][1]))

A::~A	0
A::f	0
A::g	0

A's vtble

C::~C	0
C::f	0
A::g	0

C/A's vtble

Implementation of virtual functions

- Multiple inheritance
  - ~A,f,g A B ~B,f,h

    C ~C,f
  - A* pa=new A; B* pb=new B;

pa

A object

A::~A	0
A::f	0
A::g	0

A's vtble

pb

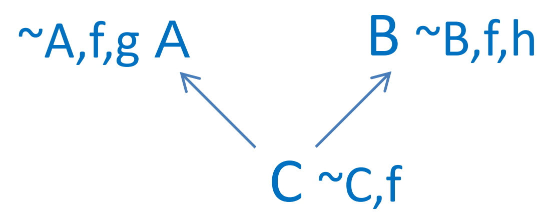
B object

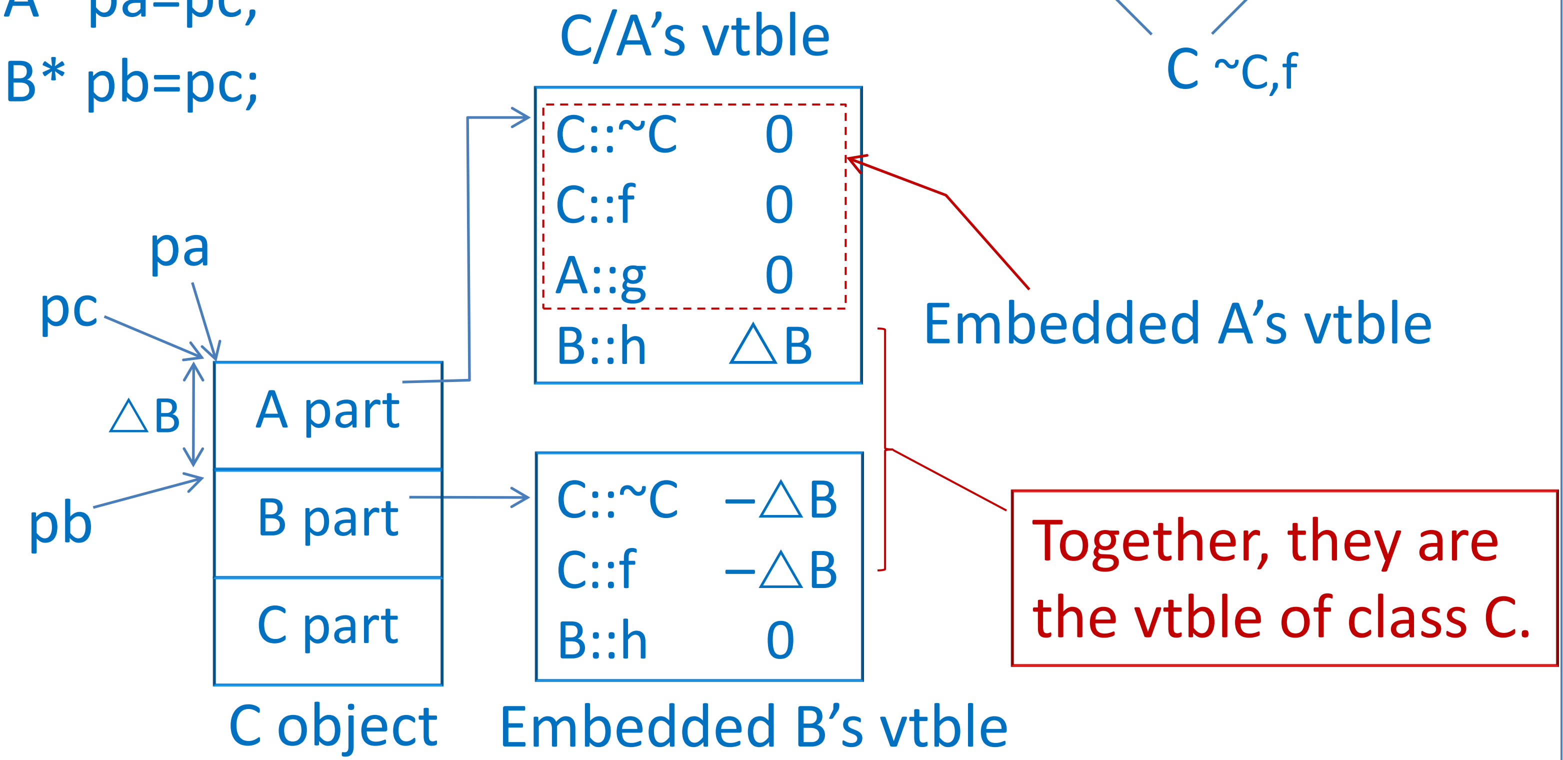
B::~B	0
B::f	0
B::h	0

B's vtble

Implementation of virtual functions

- Multiple inheritance
  - C* pc=new C;
    A* pa=pc;
    B* pb=pc;

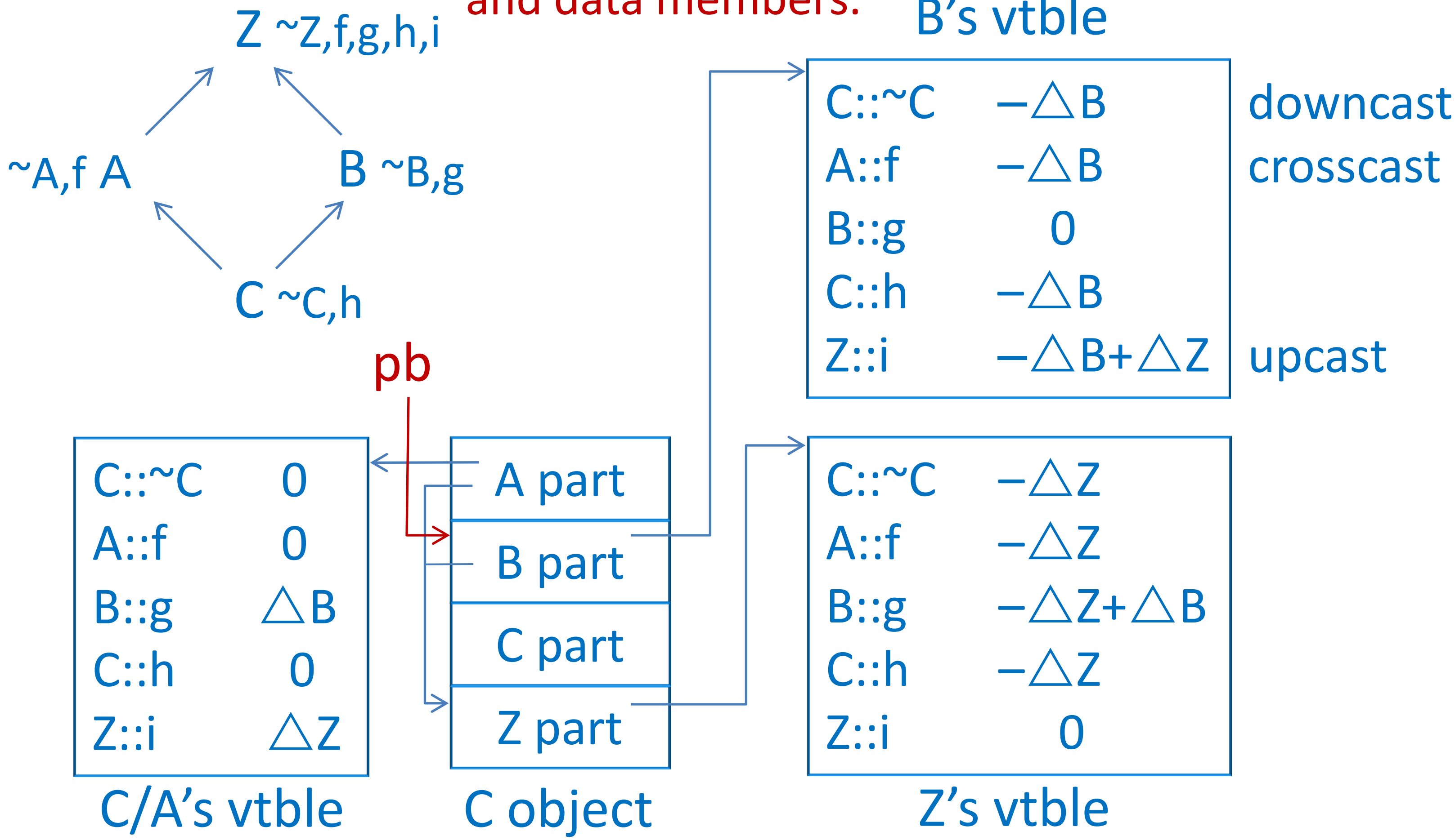




Implementation of virtual functions

- Virtual base

Virtual base pointers are for nonvirtual functions and data members.

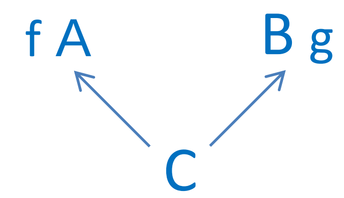


Run-time type information (RTTI)

- RTTI has 3 components
  - type_info class
    record type information of a class
  - typeid operator
    obtain a type_info object
  - dynamic_cast operator
    browse the class hierarchy (upcast, downcast, crosscast)

Run-time type information (RTTI)

- RTTI
  - class A { public: virtual void f() {} };
    class B { public: virtual void g() {} };
    class C: public A, public B {};

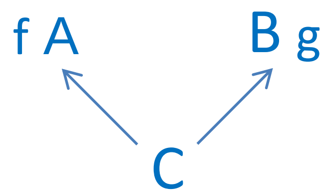


  - Upcast

```
C* pc=new C;
cout << typeid(*pc).name();   // class C (up to compiler)
typeid(*pc)==typeid(C)        // true
dynamic_cast<B*>(pc)->g();    // upcast, compile-time check
static_cast<B*>(pc)->g();     // upcast, compile-time check
pc->g();                      // upcast, compile-time check
```

Run-time type information (RTTI)

- RTTI
  - Downcast




```
A* pa=new C;
cout << typeid(*pa).name();      // class C
dynamic_cast<C*>(pa)->g();       // runtime check, ok
static_cast<C*>(pa)->g();        // unchecked, but ok
A* pa=new A;
cout << typeid(*pa).name();      // class A
dynamic_cast<C*>(pa)             // fail; yield a null pointer
static_cast<C*>(pa)              // undefined
```

Run-time type information (RTTI)

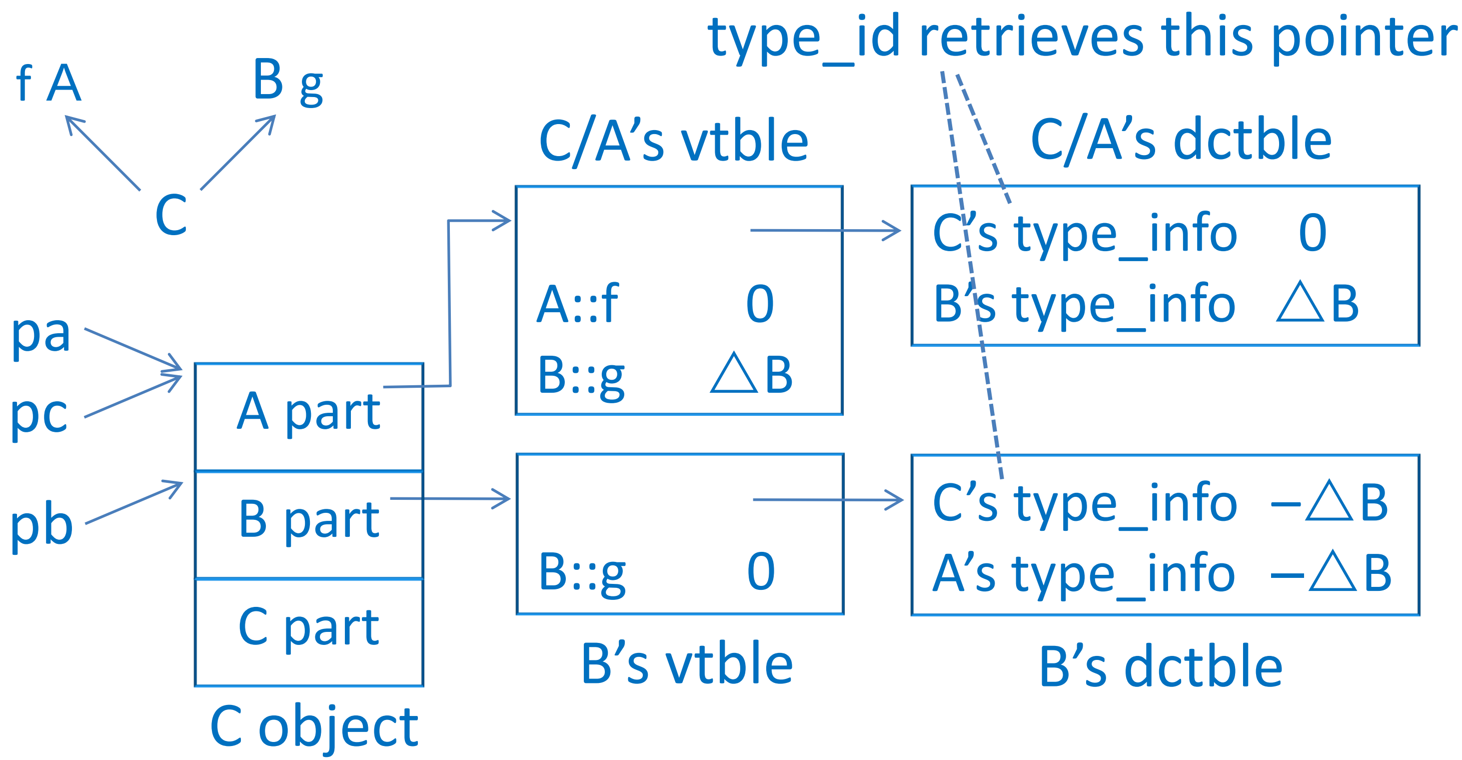
- RTTI
  - Crosscast

```
A* pa=new C;
dynamic_cast<B*>(pa)->g();     // runtime check, ok
static_cast<B*>(pa)->g();      // unchecked, but ok

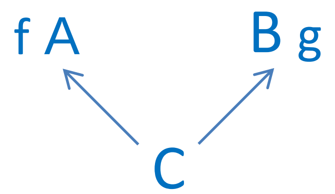
A* pa=new A;
dynamic_cast<B*>(pa)           // fail; yield a null pointer
static_cast<B*>(pa)            // undefined
```

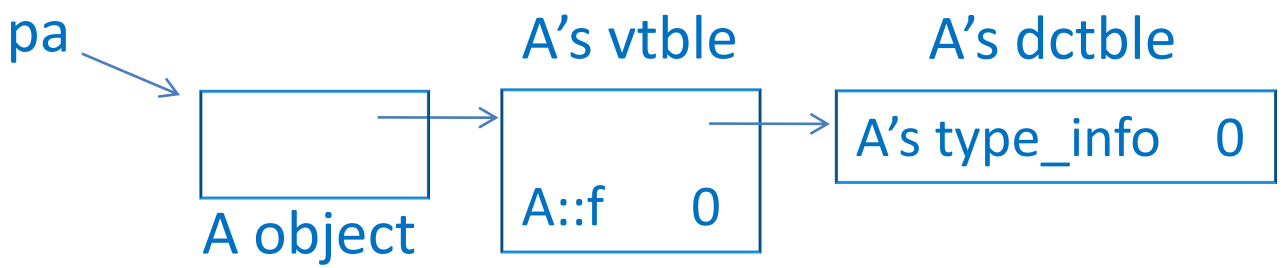
Run-time type information (RTTI)

- Dynamic_cast table (dctble)
  - Dctbles contain information necessary for type_id and dynamic_cast. (No information for upcast is stored.)



Run-time type information (RTTI)

- Dynamic_cast table (dctble)
  - 

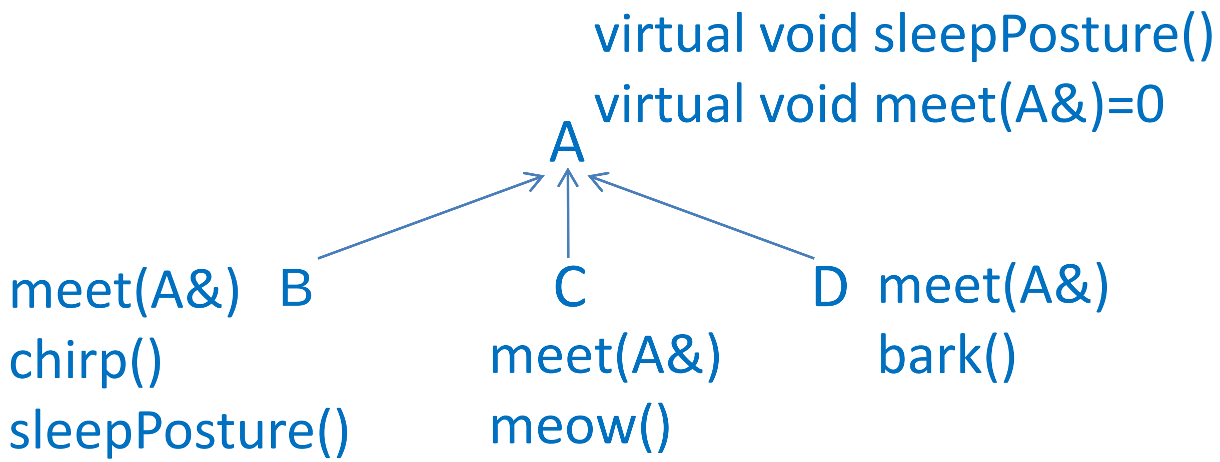


e.g.

dynamic_cast<C*>(pa) and dynamic_cast<B*>(pa) fail.

Double dispatching

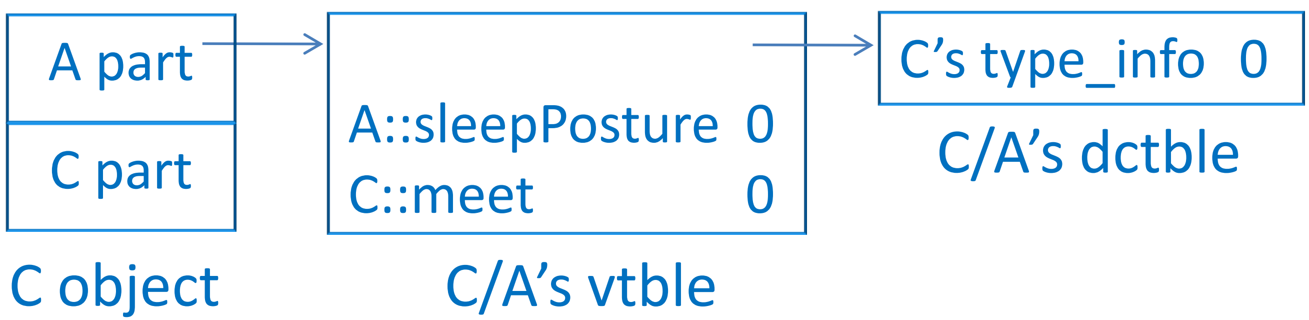
- Abstract and concrete class



  - A virtual function, e.g. A::sleepPosture() , provides an interface as well as a *default* implementation that may be overwritten.
  - A pure virtual function, e.g. A::meet(A&) provides only an interface.

Double dispatching

- Abstract and concrete class
  - A is an abstract class; B, C, and D are concrete classes.
  - A concrete class provides its own implementation of inherited pure virtual functions.
  - No objects of an abstract class can be created except as subobjects of a class derived from it.
    Thus, an abstract class doesn’t have a stand-alone vtble, but has an embedded vtble as a subobject.



Double dispatching

- Single dispatching
  - A virtual function call that depends on the dynamic type of one object is called single dispatching, e.g.
    ```
    A* pa=new B;
    pa->sleepPosture()        // B::sleepPosture()
    ```
  - Virtual function mechanism directly supports single dispatching.
- Double dispatching
  - A virtual function call that depends on the dynamic types of two objects is called double dispatching, e.g.
    ```
    A &a1=*new D, &a2=*new C;
    a1.meet(a2);              // D::meet(A&) , where A is C
    ```

Double dispatching

- Double dispatching
  - Method 1 – Use typeid

```
void D::meet(A& a)
{
    bark();
    if (typeid(a)==typeid(B))
        static_cast<B&>(a).chirp();
    else if (typeid(a)==typeid(C))
        static_cast<C&>(a).meow();
    else if (typeid(a)==typeid(D))
        static_cast<D&>(a).bark();
}
```

static_cast is better than dynamic_cast, as it is efficient and guaranteed safe here

Double dispatching

- Double dispatching
  - Method 2 – Use dynamic_cast, pointer version

```
void D::meet(A& a)
{
    bark();
    if (B* pb=dynamic_cast<B*>(&a)) pb->chirp();
    else if (C* pc=dynamic_cast<C*>(&a)) pc->meow();
    else if (D* pd=dynamic_cast<D*>(&a)) pd->bark();
}
```

A failed dynamic_cast to a pointer type yields a null pointer

A failed dynamic_cast to a reference type throws a bad_cast object.

Double dispatching

- Double dispatching
  - Method 2 – Use dynamic_cast, reference version

```
void D::meet(A& a)
{
    bark();
    try { dynamic_cast<B&>(a).chirp(); }
    catch (bad_cast&) {
        try { dynamic_cast<C&>(a).meow(); }
        catch (bad_cast&) {
            dynamic_cast<D&>(a).bark();
        }
    }
}
```

Double dispatching

- Double dispatching
  - Method 3 – Use two single dispatchings

```
class B; class C; class D;
class A {
public:
    virtual void sleepPosture() { cout << "lying prone\n"; }
    virtual void meet(A&)=0;
    virtual void meet(B&)=0;
    virtual void meet(C&)=0;
    virtual void meet(D&)=0;
};
```

```
A& a1=*new D;
A& a2=*new C;
a1.meet(a2);
```

resolved to A::meet(A&)
dispatch D::meet(A&)

Double dispatching

- Double dispatching
  - class B: public A {

```
class B: public A {
public:
    void chirp() { cout << "chirp\n"; }
    virtual void sleepPosture() { cout << "standing\n"; }
    virtual void meet(A& a) { a.meet(*this); }
    virtual void meet(B& b) { chirp(); b.chirp(); }
    virtual void meet(C& c); // { chirp(); c.meow(); }
    virtual void meet(D& d); // { chirp(); d.bark(); }
};
```

Double dispatching

- Double dispatching

```
class C: public A {
public:
    void meow() { cout << "C::meow\n"; }
    virtual void meet(A& a) { a.meet(*this); }
    virtual void meet(B& b) { meow(); b.chirp(); }
    virtual void meet(C& c) { meow(); c.meow();  }
    virtual void meet(D& d);// { meow(); d.bark();  }
};
```

```
A& a1=*new C;
B& a2=*new B;

a1.meet(a2);
```

resolved to A::meet(B&)
dispatch C::meet(B&)

Double dispatching

- Double dispatching

resolved to A::meet(D&)
dispatch C::meet(D&)

```
class D: public A {
public:
    void bark() { cout << "D::bark\n"; }
    virtual void meet(A& a) { a.meet(*this); }
    virtual void meet(B& b) { bark(); b.chirp(); }
    virtual void meet(C& c) { bark(); c.meow(); }
    virtual void meet(D& d) { bark(); d.bark(); }
};
void B::meet(C& c) { chirp(); c.meow(); }
void B::meet(D& d) { chirp(); d.bark(); }
void C::meet(D& d) { meow(); d.bark(); }
```