Advanced Issues
in Object Oriented Programming

- C++ implementation questions -
– part II -
Multiple Inheritance

Multiple inheritance allows more than one direct superclass for a given class.

```
class Secure { // code, call police
            …    }
class Door   { // open, close
            …    }
class SecureDoor :
      public Secure, public Door
      { // open only if the correct
        // code entered previously
      }
```

Similar examples are

- `WindowWithBorder`, `WindowWithScrollBar` and `WindowWithBorderAndScrollBar`
- `Displayed`, `Task` and `DisplayedTask`
- `FrTaxPayer`, `USTaxPayer` and `FrUSTaxPayer`
- `InputStream`, `OutputStream` and `InputOutputStream`
- `Vehicle`, `Residence` and `MotorHome`

Virtual vs non-virtual base classes

A *virtual base class* is inherited only once.

For example:

```
class A{ int fa; };

class B : public virtual A{
    void g(){ fa++;}
    void init(){ fa=0;} };

class C : public virtual A{
    void h(){ cout <<  fa;}
    void init(){ fa=0;} };

class D : public B, public C {  };

D d; d.B::init(); d.C::init(); d.g();d.h();
```

returns

whereas, on the other hand:

```
class A{ int fa }
class B : public A{
    void g(){ fa++;}
    void init(){ fa=0;} }
class C : public A{
    void h(){ cout <<  fa;}
    void init(){ fa=0;} }
class D : public B, public C {  }

D d; d.B::init(); d.C::init(); d.g();d.h();
```

returns

Question Where would we use virtual/non-virtual multiple inheritance in the following examples?

- `Window`, `WindowWithBorder`, `WindowWithScorllBar` and `WindowWithBorderAndSrollBar`
- `TaxPayer`, `FrTaxPayer`, `USTaxPayer` and `FrUSTaxPayer`

Question: In the following code

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

how many `A`-sub-objects are there within in a `D`-object?

Controversy around multiple inheritance

There were heated discussions around the inclusion of multiple inheritance in C++, *cf*

- T. A. Cargill: The Case Against Multiple Inheritance in C++, Usenix Computing Systems, vol 4, no. 1, 1991
- Jim Waldo: The Case For Multiple Inheritance in C++, Usenix Computing Systems, vol 4, no. 2, 1991

The Case Against Multiple Inheritance

- **too complicated** (to learn, write and read)
- **not needed** - the examples given for multiple inheritance would be better expressed through single inheritance and aggregation, i.e. every published example written using MI could just as well be written without MI
- **implementation is expensive**

The complexity of MI -- static checks

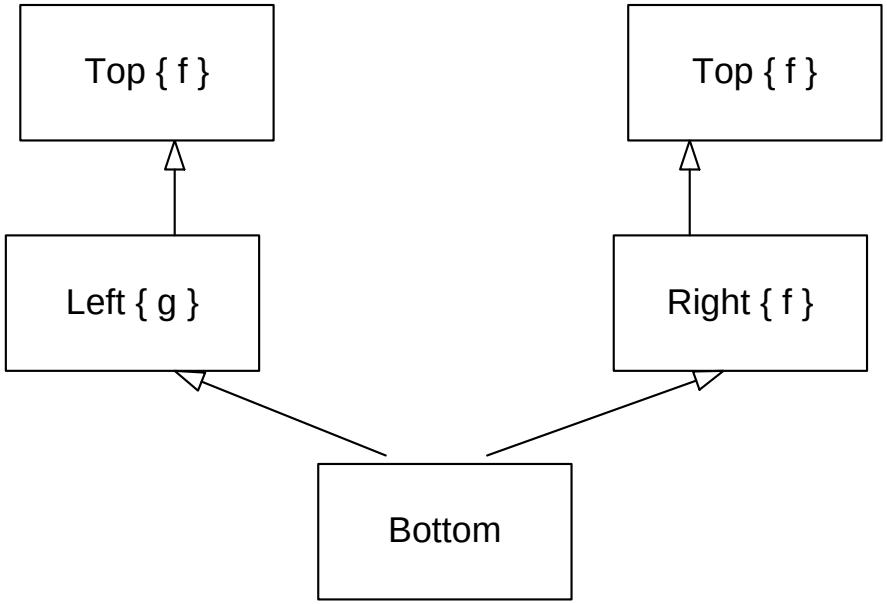
```
class A  { int g(){ 1 } int f(){ 2 } }

class B  { int g(){ 3 } int f(){ 4 } }}

class C : public A, public B
          { int g(){ 5 } }

C c;
c.f()      // Compile time error, ambiguous
c.g()      // returns 5
```

More on complexity of MI (example by Cargill)

… with non-virtual multiple inheritance…..

```
class Top{
   virtual void f(){ cout << "Top::f() ";}
};
class Left : public   Top{
   void g(){ f();}
};
class Right : public   Top{
   virtual void f(){ cout << "Right::f() ";}
};
class Bottom : public Left, public Right{ };
```

or, diagrammatically:



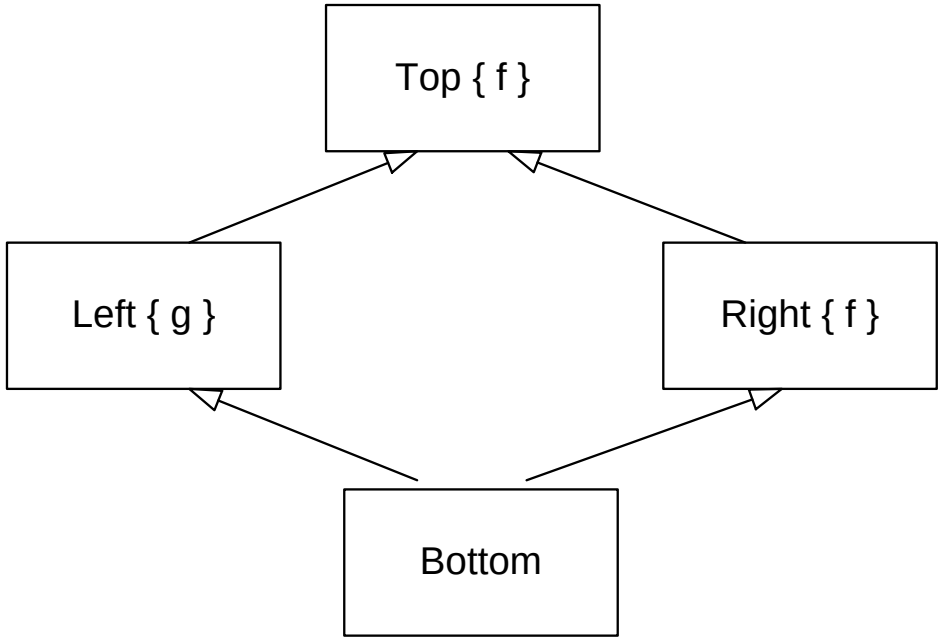
Then

```
Bottom b;
b.f();   is ambiguous
b.g();   prints
```

… with virtual multiple inheritance…..

```
class Top{
      virtual void f(){ cout << "Top::f() ";}
};
class Left : public virtual Top{
      void g(){ f();}
};
class Right : public virtual Top{
      virtual void f(){ cout << "Right::f() ";}
};
class Bottom : public Left, public Right{
};
```

or, diagrammatically:



Then

```
Bottom b;
b.f();    prints
b.g();    prints
```

The Case For Multiple Inheritance

- not that complicated
- it is needed in large software – not toy examples
- it exists in other languages (Eiffel)
- the implementation is not expensive

Jim Waldo's case for MI

MI supports interface inheritance, where

- *interface inheritance* (a subclass provides the same functionality as its superclass possibly through a different implementation, i.e. no code sharing)
- *implementation inheritance* (a subclass shares code with its superclass)

Implementation inheritance allows MI classes to share code among each other. In contrast, interface inheritance allows clients of MI classes to treat them in uniform way.

Jim Waldo's multiple interface inheritance example:

First define some abstract classes

```
class Persistent{ // entities that persist
      void save()=0;
      void restore()=0;
}
class Remote{  // remote entities
      remote_id  delegateTo; void forward …
}
class Employee{ void setSalary(float) =0; … };
```

Then define some concrete classes

```
class EmplPerst: public Employee, public Persist
{
    // bodies for save() restore()
    // setSalary calls save()
}

class EmplRem: public Employee, public Remote
{
    // bodies for forward()
    // setSalary calls forward()
}
```

Now, we can treat all sorts of employees in uniform way, e.g.

```
void review(Employee* p){ … e->setSalary(…)… }
```

Question How does the above compare with Java's design?

The Case For Multiple Inheritance - finally

Grady Booch: MI is like a parachute – not often needed, but when needed, it is indispensable

In most people's minds multiple inheritance was *the* feature of version 2.0. … Multiple inheritance belongs to C++, but is far less important than parameterised types … introduced in version 3.0. There were several reasons for choosing to work on multiple inheritance at the time: … design advanced …. Nobody seemed to doubt that I could implement templates efficiently. Multiple inheritance, on the other had, was widely supposed to be difficult to implement efficiently.

Bjarne Stroustrup *The Design and Evolution of C++*

Non-virtual MI is easier to implement, so we discuss it first:

C. Virtual member functions, multiple inheritance, non-virtual base classes

For example

```
class A{
    int fa;
    virtual void g(){ fa++;}
    virtual void j(){ g(); }
}
class B {
    int fb;
    virtual void f(){ h();}
    virtual void h(){ fb++; }}
```

```
class C : public A, public B {
    int fc;
     virtual void g(){ cout <<  fa;}
     virtual void h(){ fa =fb+fc; }
     virtual void k(){ fa =fb; }
}

C c, *cp; A a, *ap; B b, *bp;
```

Question: Draw object layout diagram for assignments

```
ap = new A(); bp = new B()
```

and for the assignments

```
ap = new C(); bp = new C()
```

and consider the representation of

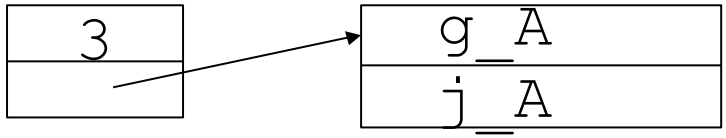
```
ap->fa; bp->fb;
```

C.1. Object Layout Store sub-objects one after the other. Virtual table of first sub-object together with virtual table of subclass. Other virtual tables at the end of the corresponding sub-object.

Layout `A` objects:

fa

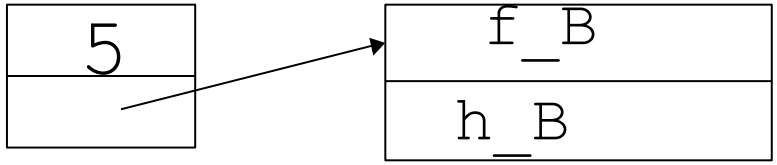
vtab



Layout `B` objects:

fb

vtab



Layout `C` objects

????

???????

Remember

- INVARIANT 1: The layout of objects of any class is a prefix of the layout of objects of any subclass.
- INVARIANT 2: Method look-up table of any class is a prefix of method look-up table of any subclass.

Problem:

- INVARIANT 1 cannot be satisfied
- INVARIANT 2 cannot be satisfied

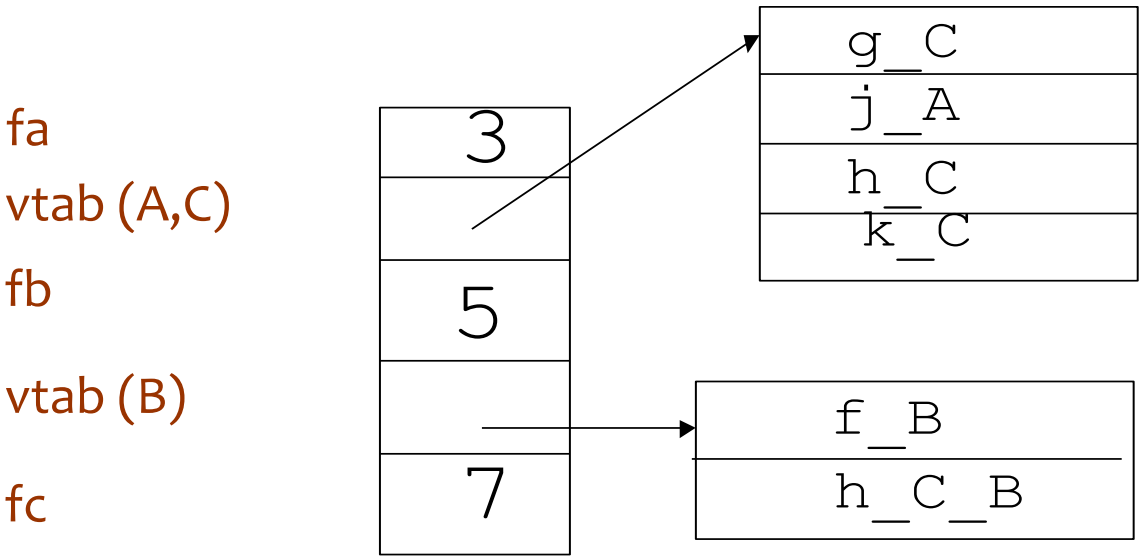
Remember

- INVARIANT 1: The layout of objects of any class is a prefix of the layout of objects of any subclass.
- INVARIANT 2: Method look-up table of any class is a prefix of method look-up table of any subclass.

Stroustrup's idea:

- If `C'` is a subclass of `C`, then an object of class `C'` contains a `C`-sub-object at offset `offset_C_C'`.

Layout `C` objects:



The two virtual tables in `C` are necessary because …

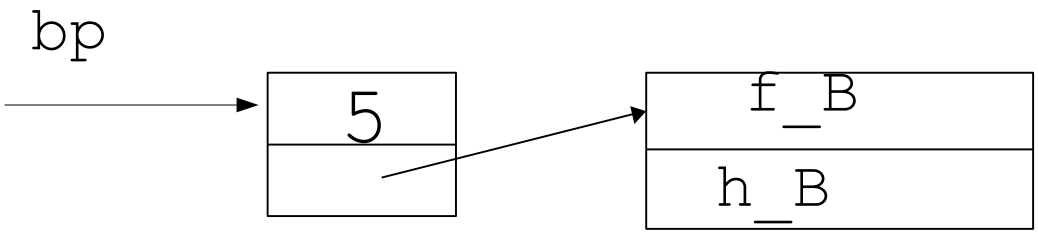
Note, two ways of calling `h` on `C` objects (more later).

`h_C_B` is the body of `h` in class `C` when receiver has static type `B*` (more later).

Therefore, after execution of

```
bp = new B();bp->fb=5;
```

we would have

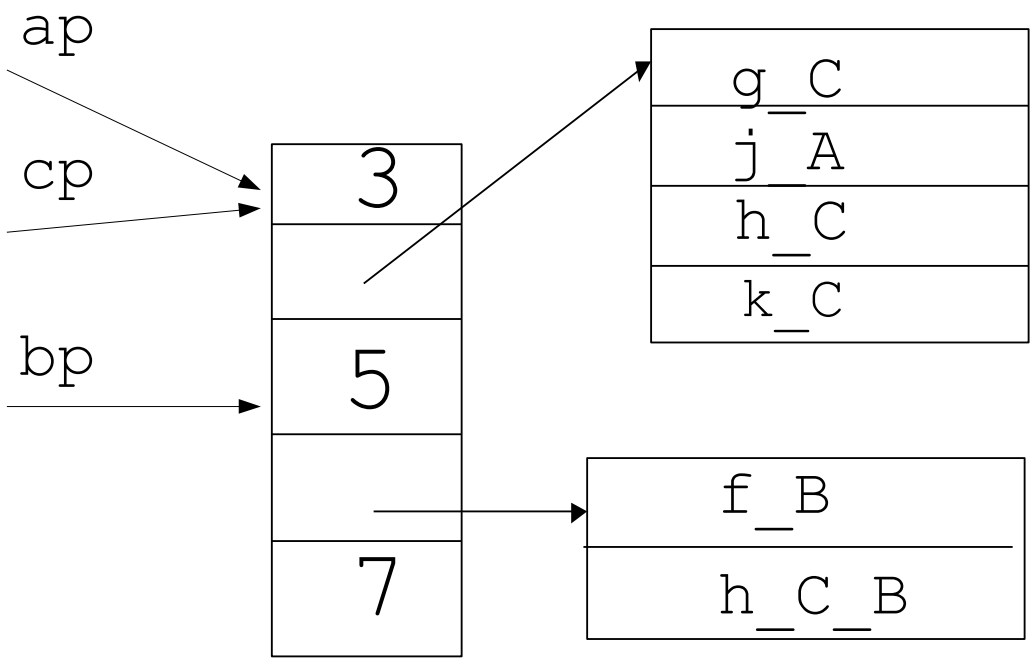


which is not surprising at all.

But, after execution of

```
cp=new C(); cp->fa=3; cp->fb=5; cp->fc=7; ap=cp; bp=cp;
```

we would have



We now have

- INVARIANT 1' For classes `c'`, `c`, where `c'` a subclass of `c`, there exists a constant `offset_c_c'`, such that layout of objects of class `c` is a prefix of layout of objects of class `c'` at offset `offset_c_c'`
- INVARIANT 2' The virtual table of an object of class `c`, a direct subclass of c_1, … c_n, consists of the virtual tables of the c_1-subobject, … c_n -subobject.
- If `c'` is a subclass of `c` through single inheritance, then `offset_c_c'=0`

Aside

In the example earlier on, we decided to first store the `A` sub-object and then the `B` subobject. Could we have done the opposite? If yes, then what would the object layout look like?

End aside

C.2. Data Member access depends on static type of object whose field is accessed. Eg:

$$Lc(\texttt{a.fa}) \equiv \alpha(\texttt{a})$$
$$Lc(\texttt{c.fa}) \equiv \alpha(\texttt{c})$$

$$Lc(\texttt{b.fb}) \equiv \alpha(\texttt{b})$$
$$Lc(\texttt{c.fb}) \equiv \alpha(\texttt{c}) + 2$$

Note, similarity between $Lc(\texttt{a.fa})$ and $Lc(\texttt{c.fa})$, and difference between $Lc(\texttt{b.fb})$ and $Lc(\texttt{c.fb})$

Similarly for pointers. So, we have:

$$Lc(\texttt{ap->fa}) \equiv *(\alpha(\texttt{ap}))$$
$$Lc(\texttt{cp->fa}) \equiv *(\alpha(\texttt{cp}))$$
$$Lc(\texttt{bp->fb}) \equiv *(\alpha(\texttt{bp}))$$
$$Lc(\texttt{cp->fb}) \equiv *(\alpha(\texttt{cp})) + 2$$

C.3. Assignment

C.3.1 Object (value) assignment

… corresponds to copying their fields. Therefore,

assignment	represented as:
`a=c;`	α(a) := *(α(c))
`b=c;`	α(b) := *(α(c) + 2)

C.3.2 Pointer assignment

… corresponds to copying of addresses. Therefore,

assignment	represented as:
`ap=cp;`	α(ap) := *(α(cp))
`bp=cp;`	α(bp) := *(α(cp)) + 2

C.4. Virtual functions and function calls

Some thought is necessary for inherited functions in conjunction with the weaker INVARIANT 1':

Consider

```
B* bp; bp = new C(); bp->h()        // Case_1
C* cp; cp = new C(); cp->h()        // Case_2
```

In both **Case 1** and **Case 2** the function `void C::h` will be executed.

However, the two cases differ in that the position of the contents of the pointer with respect to object is different.

E.g., the offset of field `fa` from the receiver object in **Case_1** in and **Case_2** are different.

Therefore, we need code for *three* functions

```
κ(h_B)    =  pThis { } //B* pThis
// from class B,
// static type of receiver B*
κ(h_C)    =  pThis { } //C* pThis
// from class C,
// static type of receiver C*
κ(h_C_B)  =  pThis { } //B* pThis
// from class C,
// static type of receiver B*
```

In general,

`func_c1_c2`

stands for a function body from class `c1`, for objects with static type `c2*`, where `offset_c2_c2` ≠ `0`.

We use name mangling to distinguish between member functions from different classes *and also to distinguish the static type of the receiver* – where necessary.

So, we have:

```
K(g_A)   =  pThis {                } //A* pThis
K(j_A)   =  pThis {                } //A* pThis
K(f_B)   =  pThis {                } //B* pThis
K(h_B)   =  pThis {                } //B* pThis
K(g_C)   =  pThis {                } //C* pThis
K(h_C)   =  pThis {                } //C* pThis
K(k_C)   =  pThis {                } //C* pThis
K(h_C_B) =  pThis {                } //B* pThis
```

C.4.1 **Function call** requires indexing the function in the appropriate virtual table.

Representation of calls for receivers with static type `C*` is similar to representation of calls with static type `A*`:

```
ap->g();        // function look-up
                *(*( *( α(ap))+1))
                    // pass receiver
                    ( *( α(ap)) )
cp->g();        // function look-up
                *(*( *( α(cp)) +1))
                    // pass receiver
                    ( *( α(cp)) )
```

```
ap->j();        // function look-up
                *(*( *( α(ap)) +1) +1)
                   // pass receiver
                     ( *( α(ap)) )
cp->j();        // function look-up

                 // pass receiver
```

On the other hand, representation of calls for receivers with static type `C*` is different from calls for receivers with static type `B*`, because of:

- the use of `vtab(B)`,
- the adjustment to receivers - in some cases

```
bp->f();        // function look-up
                *(*( *( α(bp)) +1))
                     // pass receiver
                      ( *( α(bp)) )
cp->f();        // function look-up
                *(*( *( α(cp))+3))
                     // pass receiver
                     ( *( α(cp)) +2 )
```

```
bp->h();          // function look-up
                  *( *( *( α(bp)) +1) +1)
                  // pass receiver
                     ( *( α(bp)) )
cp->h();             // function look-up
                     *( *( *( α(cp))+1) +2)
                     // pass receiver
                       ( *( α(cp)))
// also poss. less desirable (more later):
// *( *( *( α(cp))+3) +1)
//  *( α(cp)) +2 )
```

Question: Compare the function call resolution when

```
bp=new B(); bp->h();
```

with the function call resolution when

```
bp=new C(); bp->h();
```

C.4.2 Function bodies need to distinguish between the static type of `this` and the dynamic class of `this` (ie the class containing the function body)

For example, for `this` of class `C`, and static type `C*`:

```
κ(h_C)=    pThis    // C* pThis
// body from class C
          { . . .
          // the address of  this->fa
          *( α(pThis))
          // the address of  this->fb
          *( α(pThis))+2
          // the address of  this->fc
          *( α(pThis))+4
}
```

On the other hand, for `this` of class `C`, and static type `B*`, we *could* have

```
κ(h_C_B_poss) =     pThis   // B* pThis
// body from class C
        {. . .
        // the address of  this->fa
        *( α(pThis))-2
        // the address of  this->fb
        *( α(pThis))
        // the address of  this->fc
        *( α(pThis))+2
    }
```

Instead, the receiver is adjusted at the beginning, and original function called

```
κ(h_C_B) =    pThis  // B* pThis
    // body from class C
    // calculate address of beginning
    // of the C object,
    // while considering possibility of 0
    α(pNewThis) := *( α (pThis)) ?
                   *( α (pThis)) -2 : 0
     // NOTE above is a cond expr.
     // call original function
     h_C (*( α (pNewThis) ) )     }
```

Question: Why was `h_C_B` chosen rather than `h_C_B_poss`?

Question Why does the representation of `C` objects not require a separate `vtab(C)`?

Question In an object of class `C`, is the entry for `h` in `vtab(A,C)` indispensable? Could we rely on the entry for `h` in `vtab(B)` instead?

Question Change the C++ code, so that the entry for `h` in `vtab(A,C)` is indispensable.

D. Virtual member functions, multiple inheritance, virtual base classes

For example

```
class A{
    int fa;
    virtual void f(){ h();}
    virtual void h(){ fa++;}
};

class B : public virtual A{
    int fb1; int fb2;
    virtual void f(){ fa+=fb2; }
    virtual void g(){ fa+=fb2; }
};

class C : public virtual A{
    int fc;
    virtual void h(){ fa+=fc; }
```

```
};
class D : public B, public C{
    int fd;
     virtual void k(){ fc =fb1 + fa; }
     virtual void f(){ fc =fb2 + fd; }
};

D d, *dp; C c, *cp; A a, *ap; B b, *bp;
```

Objects of class B have an A-part.

Objects of class C have an A-part.

Objects of class D have a B-part and C-part,
which *share* their A-part.

Therefore, offset between B-part and A-part is different within B objects than within D objects.

Problem: INVARIANT 1' cannot be satisfied.

INVARIANT 1" Assuming that `c3` is subclass of `c2`, and `c2` derives virtually class `c1`, objects of class `c3` contain a "subobject" of class `c1` at offset `offset_c1_c2_c3`.

Ideas:

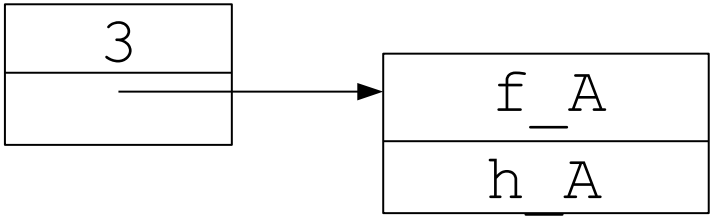
- Store `offset_c1_c2_c3` within `vtab(c2)` for `c3`
- Function `fName_c1_c2_c3` indicates a function defined in class `c1`, within the `c2` subobject of an object of class `c3`.

D.1. Object Layout

Store the part from the virtually inherited superclasses last. Store with the virtual table of a class the offset from its virtual superclasses – store it at the negative indices of the table.

Layout `A` objects

fa
vtab



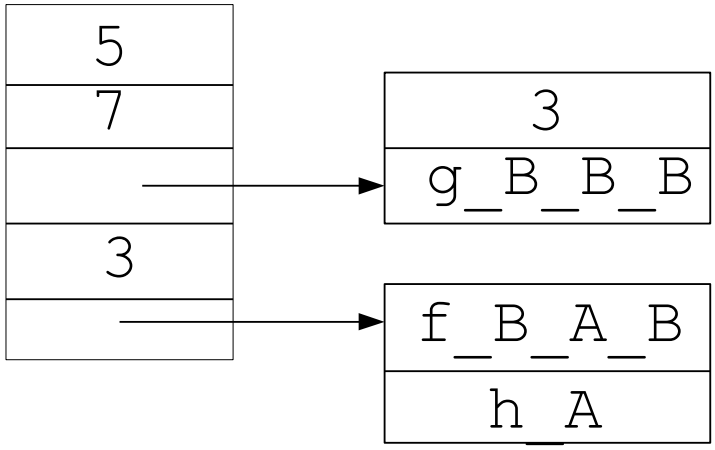
Layout `B` objects

the B – part

fb1
fb2
vtab (B)

the A-part

fa
vtab(A)

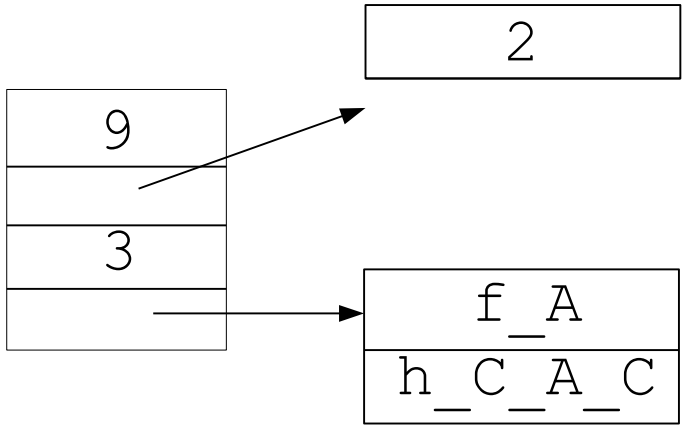


Layout C objects

the C -part

the A-part

fc
vtab (C)
fa
vtab (A)



Layout D objects

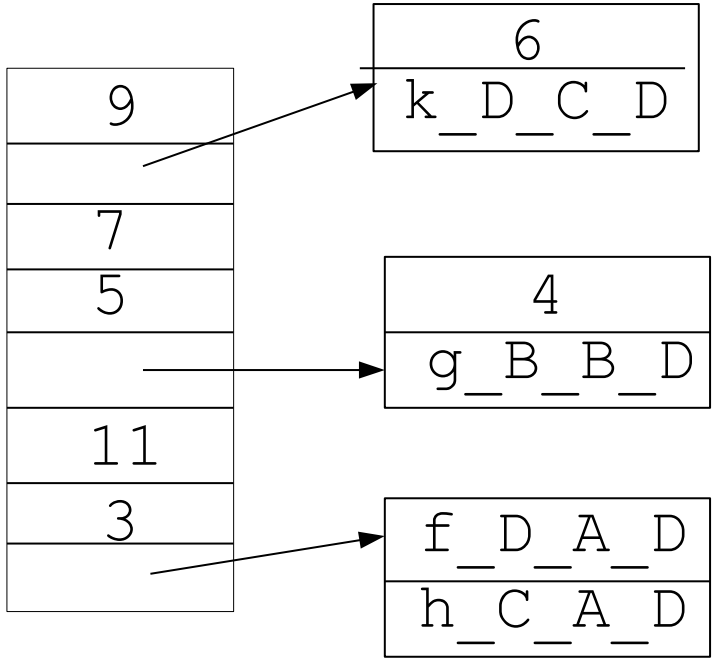
the C-part

the B-part

the D-part

the A-part

fc
vtab (C,D)
fb1
fb2
vtab (B)
fd
fa
vtab(A)



Aside -1

In the example earlier on, we decided to first store the `C` sub-object and then the `B` subobject. Could we have done the opposite? If yes, then what would the object layout look like?

Aside -2

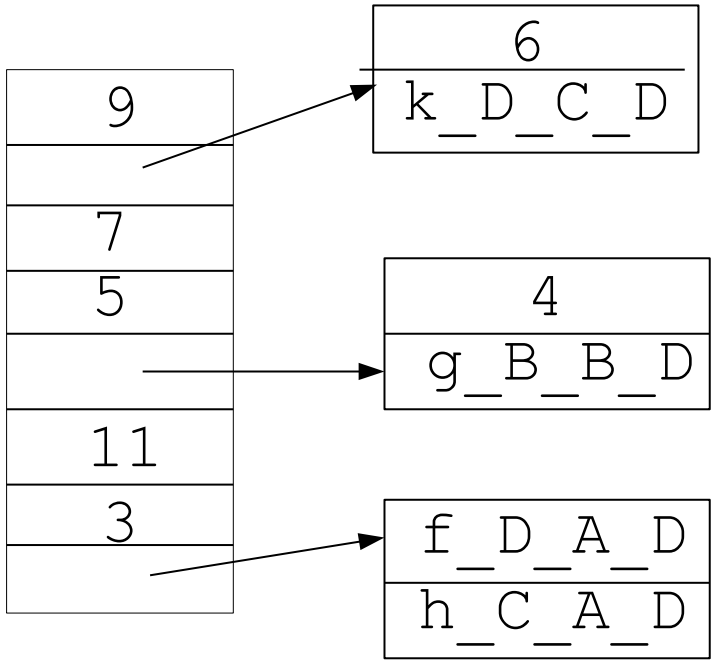
In the example earlier on, we stored the `A` sub-object at positive offset. Could we have stored it at negative offset?

End aside

Assuming

```
dp= new D(); dp->fa=3; dp->fb1=7;
             dp->fb2=5; dp->fc=9; dp->fd=11;
ap = dp; bp = dp; cp = dp;
```

To which part of the `D` object will the pointers `ap`, `bp`, `cp`, point?



D.2. Data Member access

For objects, access to fields is according to static type of object containing the field:

$$Lc(\texttt{a.fa}) \equiv \alpha(\texttt{a})$$

$$Lc(\texttt{b.fa}) \equiv \alpha(\texttt{b}) + 3$$

$$Lc(\texttt{c.fa}) \equiv \alpha(\texttt{c}) + 2$$

$$Lc(\texttt{d.fa}) \equiv \alpha(\texttt{d}) + 6$$

$$Lc(\texttt{b.fb2}) \equiv \alpha(\texttt{b}) + 1$$

$$Lc(\texttt{d.fb2}) \equiv \alpha(\texttt{d}) + 3$$

For pointers, access to fields from non-virtual superclass, eg:

$$Lc(\texttt{ap->fa}) \equiv *(\alpha(\texttt{ap}))$$

$$Lc(\texttt{bp->fb2}) \equiv *(\alpha(\texttt{bp})) + 1$$

$$Lc(\texttt{dp->fb2}) \equiv *(\alpha(\texttt{dp})) + 3$$

is different from access to fields from virtual superclass

$$Lc(\texttt{bp->fa}) \equiv *(\alpha(\texttt{bp})) + *(*(*(\alpha(\texttt{bp}))+2)-1)$$

$$Lc(\texttt{cp->fa}) \equiv *(\alpha(\texttt{cp})) + *(*(*(\alpha(\texttt{cp}))+1)-1)$$

$$Lc(\texttt{dp->fa}) \equiv$$

D.3.1 Object (value) assignment… corresponds to copying their fields. Therefore

```
a = b;          α(a)   := *( α(b)+4)
b = d;          …
                …
                …
```

D.3.2 **Pointer assignment** takes offsets between subobjects into account.

As for D.2, pointer assignment is different if *lhs* is a non virtual superclass of *rhs*,

```
cp=dp;      α(cp)   := *( α(dp))
bp=dp;      α(bp)   := *( α(dp)+2)
```

than if *lhs* is a virtual superclass of *rhs*:

```
ap=bp;      α(ap)  := *(α(bp)) + *(* (*(α(bp) )+2)-1)
ap=cp;      α(ap)  :=
ap=dp;      α(ap)  :=
```

Questions:

- Why is `offset_c1_c2_c3` stored with the `vtab` of `c2` in `c3`, instead of directly with the object?
- Why did we not store `offset_D_A_D`?
- Why no need for separate `vtab(D)` in `D` objects?
- Why do we have a separate `vtab(A)`?

- Is distinction between `h_C_A_C` and `h_C_A_D` necessary?
- Do we need both `g_B_B_B` and `g_B_B_D`?

D.4. Virtual functions and function calls

`fName_C1_C2_C3` stands for the implementation of function `fName` defined in class `C1` in the `C2` sub-object in class `C3`.

Call member function inherited from non-virtual superclass

```
bp->g();              // function look-up
                         *( *( *( α(bp)) +2))
                        // pass receiver
                         ( *( α(bp)) )

dp->g();            // function look-up
                         *( *( *( α(dp)) +4))
                        // pass receiver
                         ( *( α(dp)) +2 )
```

Call member function inherited from virtual superclass

```
ap->h();              // function look-up
                         *(*(*(α(ap))+1)+1)
                       // pass receiver
                         ( *( α(ap)) )

bp->h();              // find start of A-subobject
                   tmp =  *(α(bp))+*(*(*(α(bp))+2)-1);
                   // function look-up
                            *(*( *( α(tmp)) +1) +1)
                          // pass receiver
                          ( *( α(tmp)) )
```

Note readjustment required to find the appropriate sub-object when calling a virtually inherited function (`tmp` above)..

Bodies of member functions

In κ(g_B_B_B)

Lc(this->fa) ≡ *(α(pThis))+3

Lc(this->fb2) ≡ *(α(pThis))+1

In κ(g_B_B_D)

Lc(this->fa) ≡ *(α(pThis))+4

Lc(this->fb2) ≡ *(α(pThis))+1

In κ(h_C_A_C)

Lc(this->fa) ≡ *(α(pThis))

Lc(this->fc) ≡ *(α(pThis)) - 2

In κ(h_C_A_D)

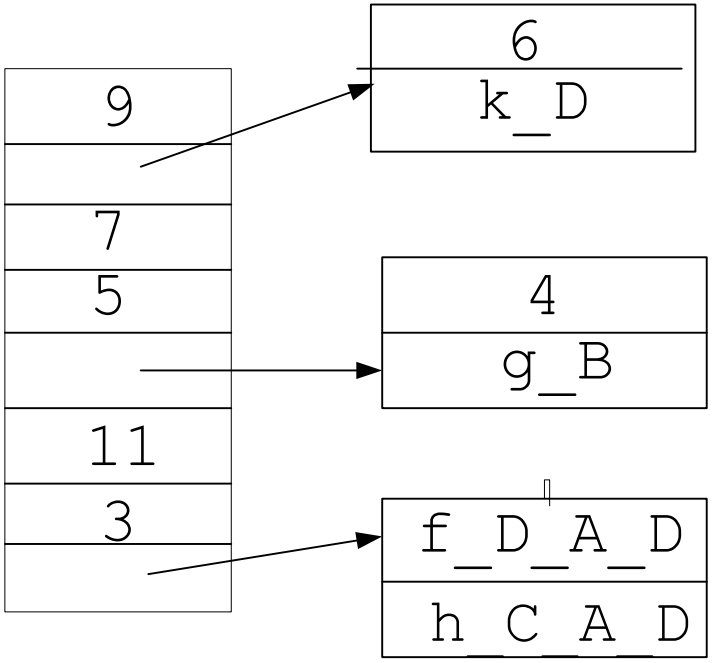
Lc(this->fa) ≡ *(α(pThis))

Lc(this->fc) ≡ *(α(pThis)) - 6

Therefore, we need to adjust inherited methods for *each* subclass.

D.5. 1st Alternative

The creation a different version for each inherited method accessing fields from virtual superclass (e.g. `g_B_B_B` and `g_B_B_D`) can be avoided by using the offset stored in vtab



In `κ(g_B)`

***Lc*(this->fa) ≡**

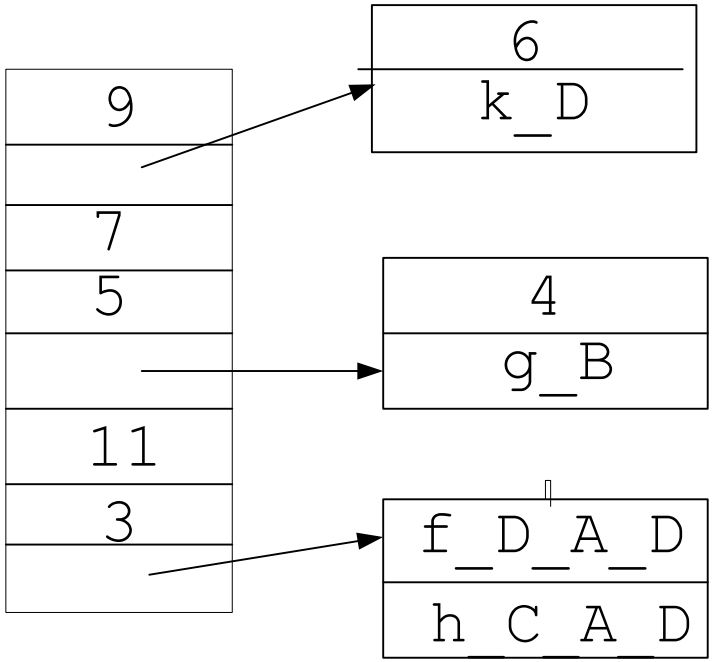
Lc`(this->fb2)` ≡

In `κ(h_C_A_D)`

Lc`(this->fa)` ≡

Lc`(this->fc)` ≡

D.5. 1st Alternative – revisited



In κ(g_B)

Lc(this->fa) ≡

*α(pThis)) + *(*(*α(pThis) +2) -1)

Lc(this->fb2) ≡ *α(pThis) +1

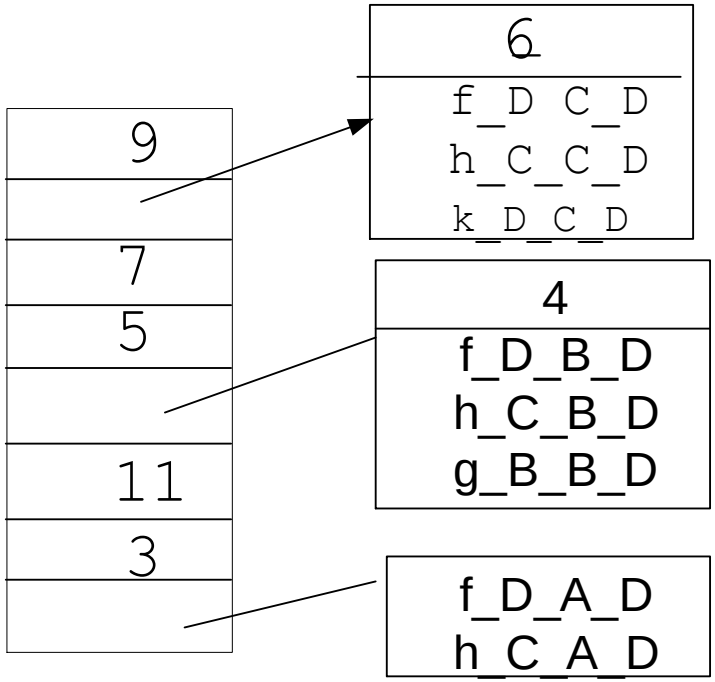
In κ(h_C_A_D)

Lc(this->fa) ≡ *α(pThis))

Lc(this->fc) ≡ *α(pThis))-6

D.6. 2nd Alternative

The adjustment necessary *before* method can be avoided by extending the vtabs in all subobjects




```
bp->f()    *(*(*(α(bp))+2)))
                 (*α(bp))

bp->h()    *(*(*(α(bp))+2))+1)
                 (*α(bp))

bp->g()    *(*(*(α(bp))+2))+2)
                 (*α(bp))
```

Bodies of member function

In κ(f_D_C_D)

Lc(this->fa) ≡ *Lc*(this->fb2) ≡

In κ(h_C_B_D)

Lc(this->fa) ≡ *Lc*(this->fb2) ≡

In κ(b_B_B_D)

Lc(this->fa) ≡ *Lc*(this->fb2) ≡

More alternatives possible.

For exams, any alternative (which works correctly) full marks.

WOW!

- vtabs necessary for lookup of virtual functions
- with single inheritance only one vtab required, and layout of subclass extends layout of superclass (part B).
- with multiple inheritance
    - sub-object at some offset within object of subclass
    - multiple vtabs required
    - vtab for subclass stored together with vtab of one of the superclasses
- offset between subclass and superclass fixed (part C).
- offset between subclass and virtually inherited superclass within another subclass fixed (part D).