Advanced Issues

in Object Oriented Programming

-

C++ implementation questions

Motivation

L1 and L2 reflect the essence of object orientation, but their naïve implementation would be extremely expensive, because

- all objects allocated on the heap, and not on the stack.
- dynamic method look-up ($M(P,c,m)$) is a recursive process unravelling the class hierarchy, and matching names.
- field look-up ($F(P,c,f)$) is a dynamic process matching identifiers.

Smalltalk is an interpreted, dynamically typed language; its dynamic nature exploited for prototype programming; first implementations required very powerful machines…

… enter C++ … using static types to provide (some) type safety, and to enhance performance

> *… a language easy enough to understand to attract users and easy enough to implement to attract developers …*
>
> *… not sufficient to provide a feature, it had to be provided in an affordable form. … "affordable on hardware common among developers" as opposed to "affordable in a couple of years when hardware will be cheaper"*
>
> *… should be implementable without using an algorithm more complicated than linear search*
>
> Bjarne Stroustrup *The Design and Evolution of C++*

We shall discuss some implementation issues in C++.

We shall consider

- object layout
- method call

in the presence of

- inheritance
- virtual and non-virtual methods
- pointers and (value) objects
- multiple inheritance

We shall first give a language background, and then discuss implementation.

Language background:
Value objects, pointers, virtual, non-virtual functions in C++

In C++, superclass ~ base class, subclass ~ derived class, with the following syntax:

```
class Food {
     // forget public, private members
     bool tasty(…){ … }
     …
  }
class Pizza: public Food{
     …
};
```

Subsumption and assignment

Objects of a derived class may appear wherever objects of a base class are expected. Pointers to a derived class may be implicitly converted to pointers to a base class. Assignment of (value) objects requires copying of fields; assignment of pointers is copying of addresses.

For example:

```
class A {
    int f1, f2;
  }

class B: public A{
    int fb;
 }
```

Then:

```
A a1, a2; A* ap; B b1,b2; B* bp;

            // TYPES

a1 = b1;
// b2 = a1;                  TYPE ERROR
// a1.fb;                    TYPE ERROR
ap = bp;
// bp = ap;                  TYPE ERROR
// ap->fb ;                  TYPE ERROR
```

```
            // Subsumption for (value) objects
a1.f1 = 2;    a1.f2 = 4;
b1.f1 = 3;    b1.f2 = 5;    b1.fb = 7;


a1 = b1;
b1.f2 = 33;
a1.f1 + a1.f2 ;            // returns ???

            // Subsumption for pointers
ap = new A();
ap->f1 = 2;    ap->f2 = 4;
bp = new B();
bp->f1 = 3;    bp->f2 = 5;    bp->fb = 7;


ap = bp;
bp->f2 = 33;
ap->f1 + ap->f2 ;            // returns ????
```

```
            // Subsumption for (value) objects
a1.f1 = 2;    a1.f2 = 4;
b1.f1 = 3;    b1.f2 = 5;    b1.fb = 7;


a1 = b1;
b1.f2 = 33;
a1.f1 + a1.f2 ;            // returns 8

            // Subsumption for pointers
ap = new A();
ap->f1 = 2;    ap->f2 = 4;
bp = new B();
bp->f1 = 3;    bp->f2 = 5;    bp->fb = 7;


ap = bp;
bp->f2 = 33;
ap->f1 + ap->f2 ;            // returns 36
```

Subsumption and function call

- *Method binding* is the process by which we determine which method to execute for a given call. In C++ we have static and dynamic method binding.
- Non-virtual functions are bound statically, according to the (static) type of the receiver.
- Virtual functions *may* be bound dynamically. When a virtual function is called for `pointer`, then the function is bound according to the *class* of the object pointed at by `pointer`, and not according to the *type* of `pointer`.
- The notation `Class::memFunction` ensures that the virtual mechanism is not used.

For example, consider classes `A`, `B`, and `C`. We declare `A::f()` as virtual, and `A::g()` as a non-virtual.

```
class A{
public:
    virtual int f( ) {return 100;};
    int g( ) {return 150;}; };

class B: public A{
public:
    virtual int f( ) {return 200;};
    int g( ) {return 250;}; };

class C: public B{
public:
    virtual int f( ) {return 300;};
    int g( ) {return 350;}; };
```

```cpp
void main(){
// static binding for (value) objects

    B  b1;
    A  a1;

   a1.f();                // returns ???
   a1.g();                // returns ???

   b1.f();                // returns ???
   b1.g();                // returns ???

   a1 = b1;
   a1.f();                // returns ???
   a1.g();                // returns ???
```

```
// dynamic binding for pointers

B* bp   = new B();
A* ap1 = new A();
A* ap2 = new C();

ap1->f();              // returns ???
ap1->g();              // returns ???

ap2->f();              // returns ???
ap2->g();              // returns ???

bp->f();               // returns ???
bp->g();               // returns ???

ap1 = bp;
ap1->f();              // returns ???
ap1->g();              // returns ???
```

```
void main(){
// static binding for (value) objects

    B  b1;
    A  a1;

   a1.f();              // returns 100
   a1.g();              // returns 150

   b1.f();              // returns 200
   b1.g();              // returns 250

   a1 = b1;
   a1.f();              // returns 100
   a1.g();              // returns 150
```

```
// dynamic binding for pointers

B* bp    = new B();
  A* ap1 = new A();
  A* ap2 = new C();

  ap1->f();             // returns 100
  ap1->g();             // returns 150

  ap2->f();             // returns 300
  ap2->g();             // returns 150

  bp->f();              // returns 200
  bp->g();              // returns 250

  ap1 = bp;
  ap1->f();             // returns 200
  ap1->g();             // returns 150
```

In other words:

- the class of the object executing the member function determines which function will be executed.
- for (value) objects, the class of the object is known at compile time, therefore static binding even for virtual functions.
- for pointers , the class of object unknown at compile time, therefore dynamic binding, if the function is virtual.

Language Design Philosophy: In other OO languages, e.g. Smalltalk, Java, there is only dynamic binding. Which mode is more important for OO? Why are there two modes of binding in C++? How many modes of binding for C#?

Language Design Philosophy

Static binding results in faster programs. Dynamic binding allows for flexibility at run-time.

Programmers should use dynamic binding only when necessary - but *not* hard-code it!

In C++

- as much static binding as possible (i.e. for non-virtual functions, for non-pointer receivers).
- dynamic binding only when necessary (i.e. only for calls of virtual function if the receiver is a pointer).
- type system is (probably) sound. (work on C#, possible projects) I

Comment 1: dynamic binding continues throughout calls

```
class A{ public:
    virtual int f( ) { return 10;};
    virtual int g( ) { return 20;};
};
class B: public A{ public:
    virtual int f( ) { return g();};
    virtual int g( ) { return 50;};
};
class C: public B{
public:
    virtual int g( ) { return 100;};
};

A* ap;
ap = new A(); ap->f();   // returns
ap = new C(); ap->f();   // returns
```

Comment 2: dynamic binding may lead to accessing statically inaccessible members

```
class A{
public:
    virtual int f( ) { return 10;}; };

class B: public A{
public:
    int i;
    virtual int f( ) { return g() + i;};
    virtual int g( ) { return 100;}; };
A* ap; B* bp;
ap = new A(); ap->f();       // returns 10
bp = new B(); bp->i = 20; ap = bp;
// ap->g(), ap->i compile time errors
ap->f();                     // returns 120
```

Comments end

We have now covered sufficient material for you to do the coursework.

We now discuss how C++ supports efficient implementations.

In particular, we discuss

1. object layout
2. data member access
3. assignment (object value, pointer)
4. functions, and function call

We consider these questions in the following order:

A. Classes with data members, non-virtual member functions only, and single inheritance
B. as above + virtual member functions
C. as above + multiple (not virtual) inheritance
D. as above + multiple + virtual inheritance

These questions are also discussed at

Margaret Ellis, Bjarne Stroustrup
The Annotated C++ Reference Manual (ARM)
Addison Wesley

Bjarne Stroustrup
The Design and Evolution of C++
Addison Wesley

Stanley Lippman
Inside the C++ Object Model
Addison Wesley

Usenix issues between 1989 and 1993

Paul Anderson, Gail Anderson
Navigating C++ and Object-Oriented Programming
Addison Wesley

Before discussing the implementation, we introduce a

High Level Target Language

abstracting from general compilation issues (eg runtime organization), concentrating on implementation of oo features.

- We assume a word addressed machine.
- We assume that integers & addresses take up one word,
- α`(ident)` gives the address which contains `ident`.

```
Instr   ::=   Loc := Loc
              // overwrites lhs with the rhs
        |     Ident( Loc* )
              // call method with identifier, 0 or more args
        |     Loc( Loc* )
              // call method from loc_1, with 0 or more args
        |     Instr; Instr
```

```
Loc        ::=  α(Ident)
                // location of ident
                |    * Loc
                // the contents of loc
                |    Loc + Loc
                // the  loc augmented by offset
                |    intergerConstant
                // eg 200

MethBody   ::=  Ident *  { Instr * }
                //  names of params, instructions of method
```

Examples in High Level Target Language

The following are valid/invalid target language instructions:

$x := 5$

$\alpha(x) := y$

$\alpha(x) := 5$

$\alpha(x) := \alpha(y)$

$\alpha(x) := \alpha(y) + 1$

$\alpha(x) := *(\alpha(y) + 1)$

$*(\alpha(x) + *(\alpha(y))) := *(\alpha(y)) + 1$

$f(14)$

$(\alpha(x) +2)(14)$

$(\alpha(x) +2)(Y)$

$*(\alpha(x) +2)(14)$

Examples in High Level Target Language - revisited

The following are valid/invalid target language instructions:

$x := 5$	✗
$\alpha(x) := y$	✗
$\alpha(x) := 5$	✓
$\alpha(x) := \alpha(y)$	✓
$\alpha(x) := \alpha(y) + 1$	✓
$\alpha(x) := *(\alpha(y) + 1)$	✓
$*(\alpha(x) + *(\alpha(y))) := *(\alpha(y)) + 1$	✓
$f(14)$	✓
$(\alpha(x) + 2)(14)$	✓
$(\alpha(x) + 2)(Y)$	✗
$*(\alpha(x) + 2)(14)$	✓

Memory

A memory, m, maps integers to integers or identifiers:

$$m : int \rightarrow int \cup Ident$$

(Note: a more realistsic view of memory is a map of integers to integers, but our higher level model allows a more succinct presentation of the issues we are dealing with).

…	…
200	348
…	…
400	aStudent
401	344
…	…
500	4
…	…

Execution in High Level Target Language

A "position" function, π, gives location for each identifier, while "code" function, κ, gives method body for an identifier:

π : Ident -> int

κ : Ident -> MethBody

Assuming "global" κ, executing instr has format

instr, π, m $\rightsquigarrow$ m'

and evaluation of location, loc, or value, val, has the format

loc, π, m $\rightsquigarrow$ i $\qquad\qquad$ $i \in$ int

val, π, m $\rightsquigarrow$ i

Locations

$$\frac{}{\alpha(x),\ \pi,\ m \rightsquigarrow \pi(x)} \qquad \frac{}{i,\ \pi,\ m \rightsquigarrow i}$$

$$\frac{loc,\ \pi,\ m \rightsquigarrow i}{{}^*loc,\ \pi,\ m \rightsquigarrow m(i)} \qquad \frac{loc,\ \pi,\ m \rightsquigarrow i \qquad loc',\ \pi,\ m \rightsquigarrow i'}{loc{+}loc',\ \pi,\ m \rightsquigarrow i + i'}$$

Instructions

$$\frac{loc,\ \pi,\ m \rightsquigarrow i \qquad loc',\ \pi,\ m \rightsquigarrow i'}{loc{:=}loc',\ \pi,\ m \rightsquigarrow m[i \mapsto i']} \qquad \frac{instrs,\ \pi,\ m \rightsquigarrow m'' \qquad instr,\ \pi,\ m'' \rightsquigarrow m'}{instrs;\ instr,\ \pi,\ m \rightsquigarrow m'}$$

$$\frac{\begin{array}{l} loc_k,\ \pi,\ m \rightsquigarrow i_k \quad \text{for } k=1..n \\ \kappa(id) = par_1, \ldots\ par_n \ \{\ instrs\ \} \\ \pi' = par_1 \mapsto i'_1 \ \ldots\ par_n \mapsto i'_n \quad \text{where } i'_1, \ldots\ i'_n \text{ fresh in } m \\ m'' = m\,[i'_1 \mapsto i_1 \ \ldots\ i'_n \mapsto i_n] \\ instrs,\ \pi',\ m'' \rightsquigarrow m' \end{array}}{id(loc_1, \ldots\ loc_n),\ \pi,\ m \rightsquigarrow m' \backslash \{\ i'_1, \ldots\ i'_n\ \}}$$

$$\frac{\begin{array}{l} loc,\ \pi,\ m \rightsquigarrow i \\ m(i) = id \\ id(loc_1, \ldots\ loc_n),\ \pi,\ m \rightsquigarrow m' \end{array}}{loc(loc_1, \ldots\ loc_n),\ \pi,\ m \rightsquigarrow m'}$$

Questions

What does π represent? Why is π *not* global?

What is the difference between π and α?

What does κ represent? Why do we assume a “global” κ?

Why do we need to allow the memory to map addresses to identifiers?

Example 1

Assume $\pi(x)=300$, $\pi(y)=400$. Memory as in lhs table, execute:

$\alpha(x)+1 := \alpha(y)\,;\ \ \alpha(x)+2 := *(\alpha(x)+1)\,;\ \ \alpha(x)+3 := *(\alpha(x))+1\,;$

before execution

...	...
300	44
...	...
400	500
...	...
500	300
...	...

after execution

...	...
300	44
...	...
400	500
...	...
500	300
...	...

Example 2

For the same π, execute:

$$*(*(\alpha(\mathtt{y}))) := 10; \quad *(\alpha(\mathtt{y})) := 100; \quad \alpha(\mathtt{y}) := 1000;$$

before execution

. . .	. . .
300	44
. . .	. . .
400	500
. . .	. . .
500	300
. .	. . .

after execution

. .	. . .
300	44
. . .	. . .
400	500
. . .	. . .
500	300
. . .	. . .

Example 3

Assume:

$$\kappa\,(f) = x, y \{ *\alpha(y) := * * \alpha(x) \}$$

What is the effect of executing

$f(2,3)$

in the following memory

before execution

1	11
2	22
3	33

during execution

1	
2	
3	
...	...
...	...
...	....

$\pi' = \ldots.$

after execution

1	
2	
3	

Example 1 - revisited

Assume $\pi(x)=300$, $\pi(y)=400$. Memory as in lhs table, execute:

$$\alpha(x)+1 \;:=\; \alpha(y)\,;\;\; \alpha(x)+2 \;:=\; *(\alpha(x)+1)\,;\;\; \alpha(x)+3 \;:=\; *(\alpha(x))+1\,;$$

before execution

...	...
300	44
...	...
400	500
...	...
500	300
...	...

after execution

...	...
300	44
301	400
302	400
303	45
....	...
400	500

Example 2 - revisited

For the same π, execute:

$$*(*(\alpha(y))) := 10; \quad *(\alpha(y)) := 100; \quad \alpha(y) := 1000;$$

before execution

. . .	. . .
300	44
. . .	. . .
400	500
. . .	. . .
500	300
. . .	. . .

after execution

. .	. . .
300	10
. . .	. . .
400	1000
. . .	. . .
500	100
. . .	. . .

Example 3 -- revisited

Assume:

$$\kappa(f) = x, y \{ *\alpha(y) := **\alpha(x) \}$$

What is the effect of executing

$f(2,3)$

in the following memory

before execution

1	11
2	22
3	33

during execution

1	11
2	22
3	~~33~~ 22
4	2
5	3

$\pi'(x)=4,\ \pi'(y)=5$

after execution

1	11
2	22
3	22

A. Classes with data members, non-virtual member functions, and single inheritance

Consider

```
class A{
public:
    int fa1, fa2;
    void g( ) { fa1++; }
    void h( ) { g(); }
}

class B: public A{
public:
    int fb;
    void g( ) { fb++; }}

A   a, a1, *ap; B b, *bp;
```

A.1. Object Layout

Lay out all members of a class - most probably in the order of their declaration (compilers also take into account alignment requirements).

Pointers to objects are represented through the address of the first cell of the object.

Then, after execution of

```
 a.fa1=3;    a.fa2=5;
 b.fa1=4;    b.fa2=6;    b.fb=8;
ap = new B();
ap->fa1=5;    ap->fa2=7;
```

we will have

… somewhere in memory …	address	contents
fa1	π(a)	3
fa2	π(a) + 1	5
… somewhere else in memory		
fa1	π(b)	4
fa2	π(b) + 1	6
fb	π(b) + 2	8
… somewhere else in memory		
	π(ap)	vvv
… somewhere else in memory		
	vvv	5
		7
		…

Compare the previous with the L1 representation of `a`

    (A, (fa1 ↦ 3, fa2 ↦ 5))

which is equivalent with

    (A, (fa2 ↦ 5, fa1 ↦ 3))

and, also with the L2 representation of `b`:

    (B, (fa2 ↦ 4, fa1 ↦ 6, fb ↦ 8))

Questions:

- Why did we choose a more “liberal” representation in L1?
- Why can we get away with a more succinct representation in C++?

INVARIANT 1: The layout of objects of any class is a prefix of the layout of objects of any subclass.

A.2. Data Member access

Offsets of fields are known at compile time. Thus field access can be implemented through statically known offsets.

Thus, we have

$$\mathit{Lc}(\texttt{a.fa1}) \equiv \alpha(\texttt{a})$$

$$\mathit{Lc}(\texttt{a.fa2}) \equiv \alpha(\texttt{a}) + 1$$

$$\mathit{Lc}(\texttt{b.fa1}) \equiv \alpha(\texttt{b})$$

$$\mathit{Lc}(\texttt{b.fa2}) \equiv \alpha(\texttt{b}) + 1$$

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

(where symbol $\equiv$ signifies “is equivalent with”, *not* assignment, and where Lc maps C++ path expressions to locations expressed in the intermediate target language.)

What about pointers? How do we represent, eg,

`ap->fa2`

Remember, that `ap` may be pointing to an object of class `A`, or to an object of class `B`. The representation of `ap->fa2` should work for either case!

Because of INVARIANT 1, this is automatically achieved.

So, we have:

$$Lc(\texttt{ap->fa1}) \equiv *(\alpha(\texttt{ap}))$$

$$Lc(\texttt{bp->fa1}) \equiv *(\alpha(\texttt{bp}))$$

$$Lc(\texttt{ap->fa2}) \equiv *(\alpha(\texttt{ap})) + 1$$

$$Lc(\texttt{bp->fa2}) \equiv *(\alpha(\texttt{bp})) + 1$$

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

A.3. Assignment

A.3.1 Object (value) assignment

Remember, that assignment of objects corresponds to copying their fields – modulo copy constructor calls, which we shall disregard here.

Therefore,

the assignments	are represented as:
`a=a1;`	$\alpha(a) := *(\alpha(a1))$
	$\alpha(a)+1 := *(\alpha(a1)+1)$
`a=b;`	$\alpha(a) := *(\alpha(b))$
	$\alpha(a)+1 := *(\alpha(b)+1)$

A.3.2 Pointer assignment

Remember, that assignment of pointers corresponds to copying of addresses.

Therefore,

the assignment is represented as:

`ap=bp;` $\alpha(ap) := *(\alpha(bp))$

Example 4

Give the representation for the following:

```
a.fa1=2;
ap=&b;
ap->fa2=b.fb;
*ap = * bp;
```

A.3.2 Pointer assignment

…

Example 4 -- revisited

Give representation for the following:

`a.fa1=2;`	α(a) := 2;
`ap=&b;`	α(ap) := α(b);
`ap->fa2=b.fb;`	*α(ap)+1 := *(α(b)+2);
`*ap = *bp;`	*α(ap) := **α(bp);
	*α(ap)+1 := *(*α(bp)+1);

A.4. Functions and function calls

Stroustrup's aim was to make non-virtual function call as fast as normal (global) function call.

- Member functions represented by "normal" functions, with additional `pThis` parameter, representing `this`.
- Through name mangling distinguish between member functions from different classes (and also between overloaded functions).

In above example, we have following function bodies

```
K(g_A) =  pThis { . . . } // body of A::g()
                          // with A* pThis
K(h_A) =  pThis { . . . } // body of A::h()
                          // with A* pThis
K(g_B) =  pThis { . . . } // body of B::g()
                          // with B* pThis
```

Calls of non-virtual member functions can be resolved (bound) statically.

Therefore,

the calls	will be represented as:
`a.g();`	`g_A ( α(a) )`
`ap->g();`	`g_A ( *( α(ap)) )`
`b.h();`	`h_A ( α(b) )`
`bp->h();`	`h_A ( *( α(bp)) )`

Example 5

Remember that

```
class A{   …
   void g( ) { fa1++; }
   void h( ) { g(); } // ie this->g()
   }
class B : public A{   …
   void g( ) { fb++; }
}
```

fill in the representation of the function bodies:

```
K(g_A) =  pThis {      ...      }
K(h_A) =  pThis {      ...      }
K(g_B) =  pThis {      ...      }
```

Example 5 – revisited

```
class A{   …
   void g( ) { fa1++; }
   // ie this->fa1 = this->fa1 +1
   void h( ) { g(); } // ie this->g()
   }
class B : public A{   …
   void g( ) { fb++; }
     // ie this->fb = this->fb +1
}
```

fill in the representation of the function bodies:

κ(g_A) = pThis
    { *(α(pThis)) := *(*(α(pThis)))+1 }
κ(h_A) = pThis { g_A(*(α(pThis))) }
κ(g_B) = pThis
    { *(α(pThis))+2 := *(*(α(pThis))+2)+1 }

Example 6

Consider

```
class A{
    int fa;   A* next;
    void g(A* p)
        { this->next = p; p->next = this; }
}
```

Fill in the representation of the function body:

```
K(g_A) =  pThis, p {                                }
// with   A* pThis, A* p
```

Example 6 -- revisited

Question Consider

```
class A{
    int fa;  A* next;
    void g(A* p) { this->next = p; p->next = this; }
}
```

Representation of function body:

$$\kappa(g\_A) = pThis,\ p \quad \{ *(\alpha(pThis))+1 := *(\alpha(p)); \quad *(\alpha(p))+1 := *(\alpha(pThis)); \}$$

Example 7 The effect of executing `g_A(1,3)` in the memory

before execution

1	10
2	0
3	20
4	0

after execution

1	
2	
3	
4	

Example 7 - revisited

κ(g_A) = pThis, p
{ *(α(pThis))+1 := *(α(p)); *(α(p))+1 := *(α(pThis)); }

The effect of executing g_A(1,3) in the memory

before execution

1	10
2	0
3	20
4	0

during execution

1	10
2	~~0~~ 3
3	20
4	~~0~~ 1
5	1
6	3

π'(pthis)=5, π‘(p)=6

after execution

1	10
2	3
3	20
4	1

Question Why were we able to replace the L_1 string comparison function $F(P,c,f)$ through statically determined function locations?

Question Do the operations described so far preserve INVARIANT 1?

Question Pointers contain the address of the *first* cell of the object - why not the address of the *last* cell?

… enter virtual functions …

B. Classes with data members, possibly virtual member functions, single inheritance

We now now add virtual member functions, *i.e.* the possibility of dynamic method binding. The dynamic method look-up function M(`P`,`c`,`m`) in L_2 expresses that.

Stroustrup's aim was to make virtual function call as fast as non-virtual function call + a small constant.

Stroustrup's idea is, basically, an application of the treatment of fields, to the treatment of functions.

Stroustrup's idea was:

- Each class has a function look-up table.
- Function selection is represented through an index into the look-up table of the class of the receiver .
- **INVARIANT 2: Method look-up table of any class is a prefix of method look-up table of any subclass.**

In C++ literature, the function look-up table is often called the *virtual table*, and a pointer to it is represented by a field `vtab`.

Consider

```
class C: public B{
public:
     int fc;
    virtual void hh( ) { gg(); }
    virtual void gg( ) { fa1+=fc; }
    }

class D: public C{
public:
     int fd;
     virtual void gg( ) { g(); }
     virtual void kk( ) { fc+=fd; }
     void k ( ) { gg(); }
     }
C c, c1, *cp;   D d, d1, *dp;
```

B.1. Object Layout

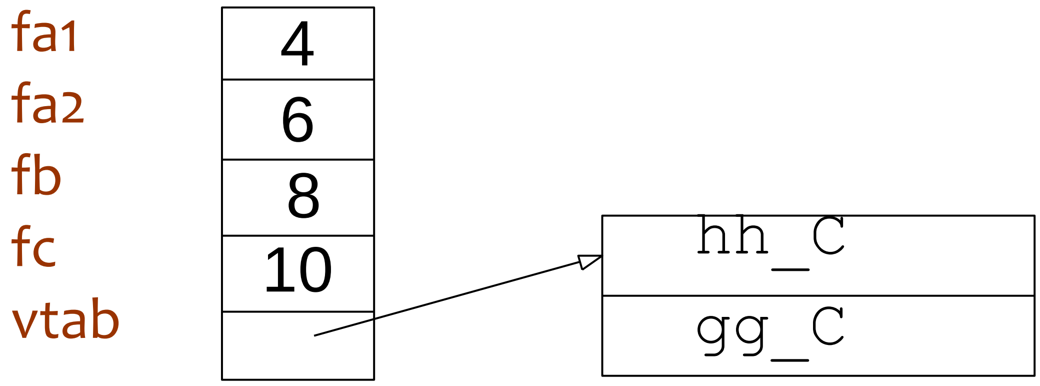
Lay out all members of a class, after the first class with virtual members put a pointer to the virtual table.

Consider execution of

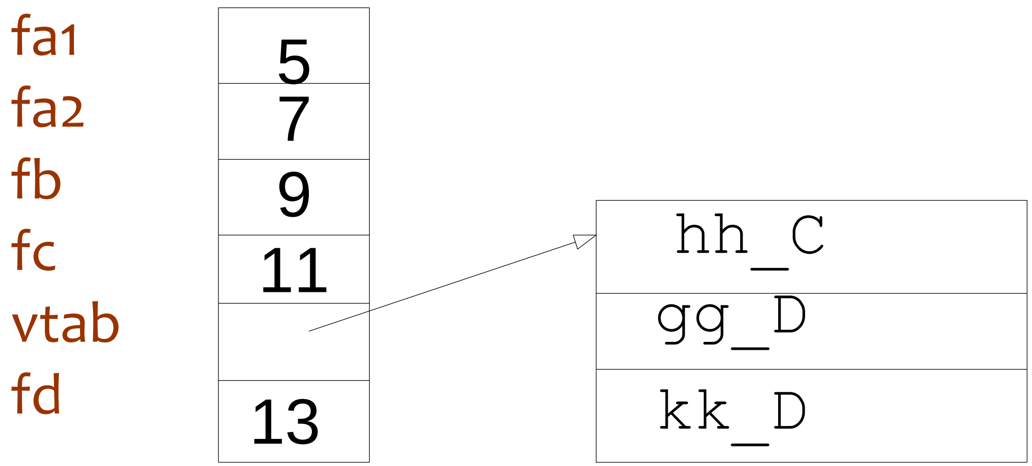
```
c.fa1=4;
c.fa2=6;
c.fb =8;
c.fc=10;

d.fa1=5;
d.fa2=7;
d.fb =9;
d.fc=11;
d.fd =13
```

Then,

at α `(c)` we have



at α `(d)` we have



Question: why are the other functions not in `vtab`?

B.2. Data Member access

INVARIANT 1 holds, and therefore field access can be implemented through a statically known offset

Therefore:

$$\mathit{Lc}(\texttt{d.fa1}) \equiv \alpha(\texttt{d})$$

$$\mathit{Lc}(\texttt{d.fa2}) \equiv \alpha(\texttt{d}) + 1$$

$$\mathit{Lc}(\texttt{d.fb}) \equiv \alpha(\texttt{d}) + 2$$

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

$$\mathit{Lc}(\texttt{d.fd}) \equiv \alpha(\texttt{d}) + 5$$

Similarly for pointers. So, we have:

$$\mathit{Lc}(\texttt{dp->fa1}) \equiv *(\alpha(\texttt{dp}))$$

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

B.3. Assignment

B.3.1 Object (value) assignment

… assignment of objects corresponds to copying their fields

Therefore,

the assignments will be represented as:

```
a=d1;      α(a)    := *( α(d1))
           α(a)+1  := *( α(d1) +1 )
c=d;       α(c)    := *( α(d))
           α(c)+1  := *( α(d)  +1 )
           α(c)+2  := *( α(d)  +2 )
           α(c)+3  := *( α(d)  +3 )
```

Question: why not also

```
           α(c)+4  := *( α(d)  +4 )
```

B.3.2 Pointer assignment

… assignment of pointers corresponds to copying of addresses

Therefore,

the assignment will be represented as:

```
cp=dp;          α(cp)   :=  *( α(dp))
```

Question give the representation for the following:

```
d = d1;
```

B.4. Virtual functions and function calls

Stroustrup's aim was to make virtual function call as fast as normal (global) function call plus small constant.

As for non-virtual functions, functions are represented through "normal" functions, with an additional `pThis` parameter; name mangling distinguishes between member functions from different classes

So, we have:

```
K(hh_C)  =  pThis {          } //C* pThis
K(gg_C)  =  pThis {          } //C* pThis
K(gg_D)  =  pThis {          } //D* pThis
K(kk_D)  =  pThis {          } //D* pThis
K(k_D)   =  pThis {          } //D* pThis
```

For virtual member functions and pointers the function calls can only be resolved (bound) dynamically. Otherwise, function call can be resolved statically.

Because of INVARIANT_2, lookup can be performed through an index into the virtual table.

Therefore,

the calls will be represented as:

```
// static function look-up
c.hh();            hh_C ( α(c) )
dp->g();            g_B ( *( α(dp))   )
dp->k ();           k_D ( *( α(dp))   )
```

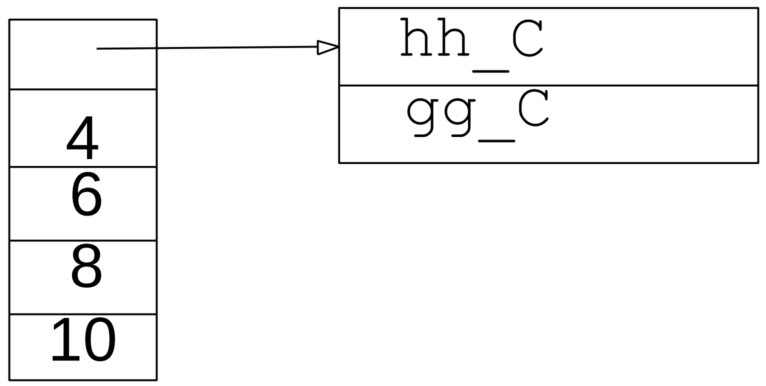
```
// dynamic function look-up
cp->hh();          // function look-up
                   *(*(*( α(cp))+4))
                        // pass receiver
                        ( *( α(cp)) )
cp->gg();          // function look-up
                    *(*(*( α(cp))+4)+1)
                        // pass receiver
                        ( *( α(cp)) )
```

Question fill in the representation of the function bodies:

```
K(hh_C)  =  pThis {            } //C pThis
K(gg_C)  =  pThis {            } //C pThis
K(gg_D)  =  pThis {            } //D pThis
K(kk_D)  =  pThis {            } //D pThis
K(k_D)   =  pThis {            } //D pThis
```

Question: Could one choose the following alternatives?

…. . store pointer to virtual table at the beginning of the object. Then, `c` would be: (and similarly for `d`)



or

… store pointer to virtual table at the end of the object. Then, `c` would be as in previous slides, but `d` would be: