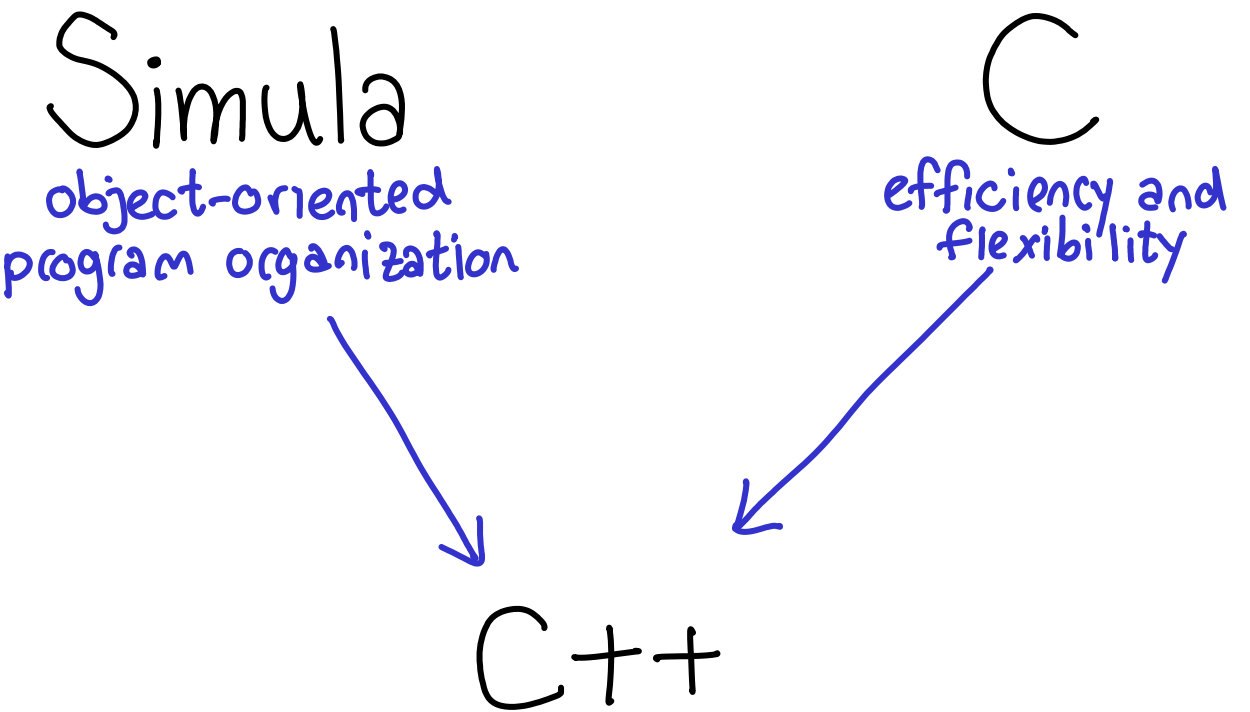
Virtual Tables

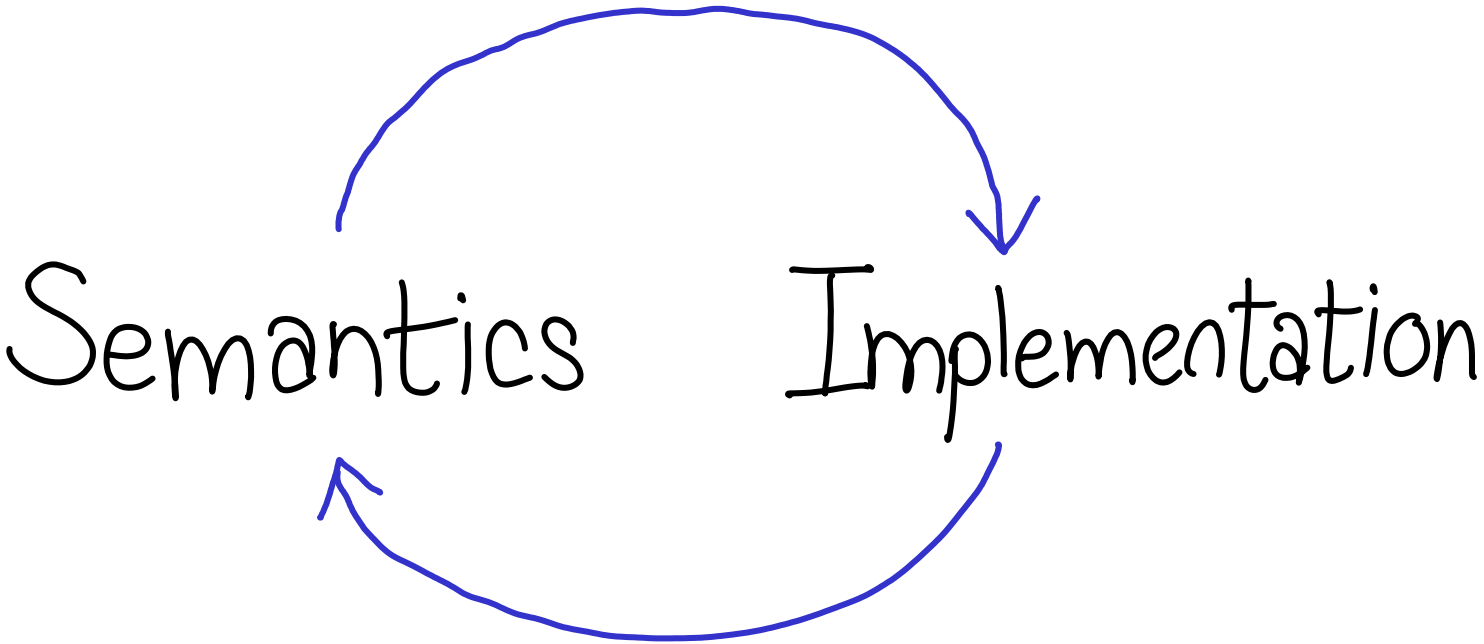
Edward Z. Yang


Simula
object-oriented
program organization
C
efficiency and
flexibility
C++

"What you don't use, you don't pay for."

—Bjarne Stroustrup

(There is a direct mapping of C++ language constructs to hardware.)



C++ features are cheap
but sometimes complicated

The basics

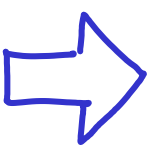
Example

syntax simplification:
visibility modifiers
omitted

```
class A {
    int a;
    void f(int i);
}
```

```
int a;
```

```
A* pa;
pa->f(2);
```



compiles to

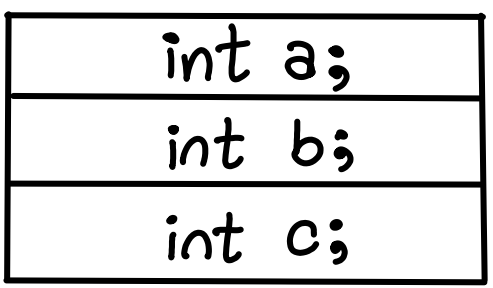
```
__A_f(pa,2);
```

object passed in as argument

ordinary fn call

Inheritance

```
class A { int a; void f(int);}
class B : A  {int b; void g(int);}
class C : B  {int c; void h(int);}
```

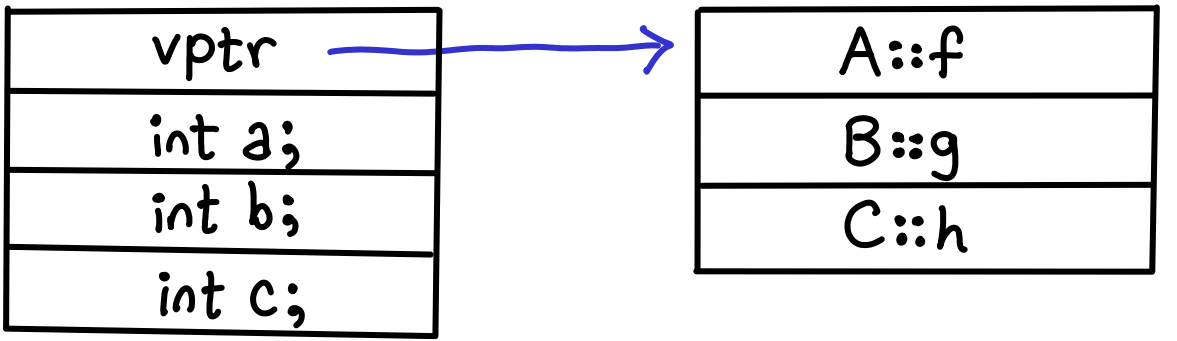


compiler "knows" position of each field

Virtual methods

```
class A {
    int a;
    virtual void f(int);
    virtual void g(int);
    virtual void h(int);
}

class B : A { int b; void g(int); }
class C : B { int c; void h(int); }
```

class C object:

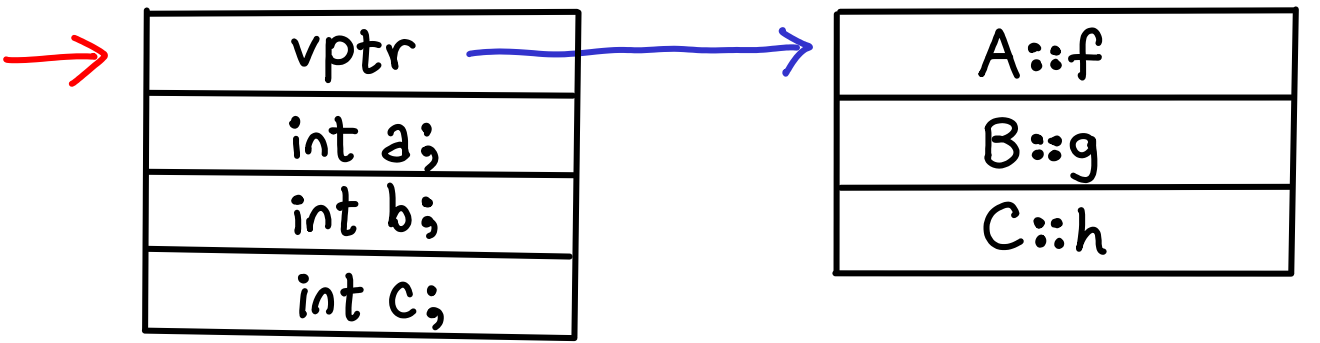



```
class A {
    int a;
    virtual void f(int);
    virtual void g(int);
    virtual void h(int);
}
```

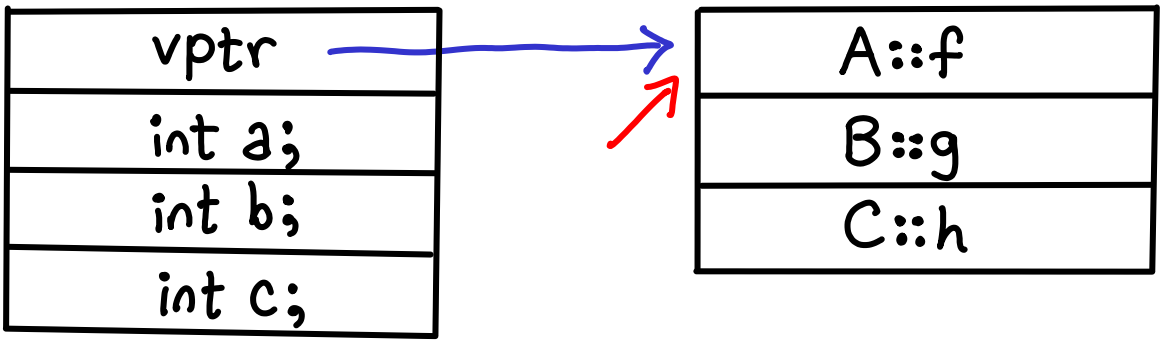
```
class B : A { int b; void g(int); }
class C : B { int c; void h(int); }
```

```
C* pc;
pc->g(2);
```

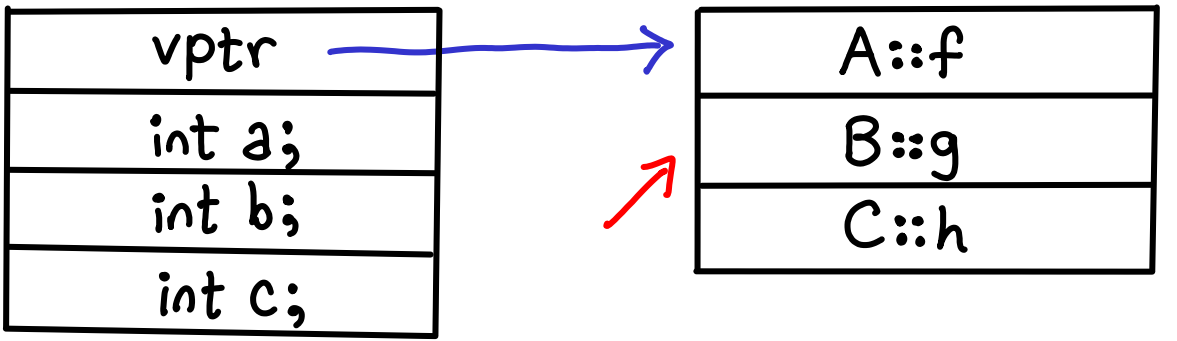
```
(*(pc->vptr[1]))(pc,2)
```



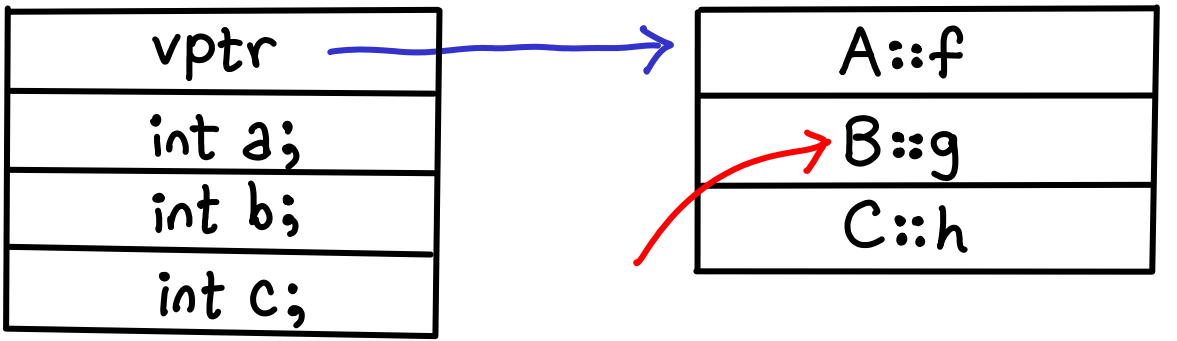
(*(pc->vptr[1]))(pc,2)



(*(pc→vptr[1]))(pc,2)



(*(pc->vptr[1]))(pc,2)

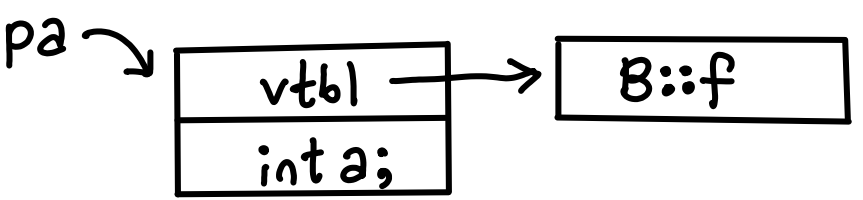


(*(pc->vptr[1]))(pc,2)

Implementation → Semantics

Non-virtual	versus	Virtual
Direct fn call		Indirection through vtable
Cannot be redefined (except via overloading)		Can be redefined

```
class A {
    int a;
    virtual void f() { printf("parent"); }
}
class B: A {
    virtual void f() { printf("child "); }
}
```

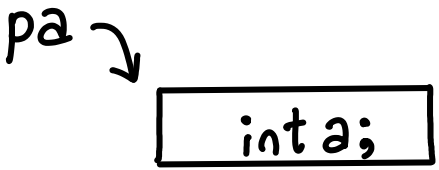



```
A* pa = new B();
pa->f();  // child
```

```
(*(pa->vtbl[0]))();
```

```
class A {
    int a;
    void f() { printf("parent"); }
}
class B: A {
    void f() { printf("child "); }
}
```

don't do this!

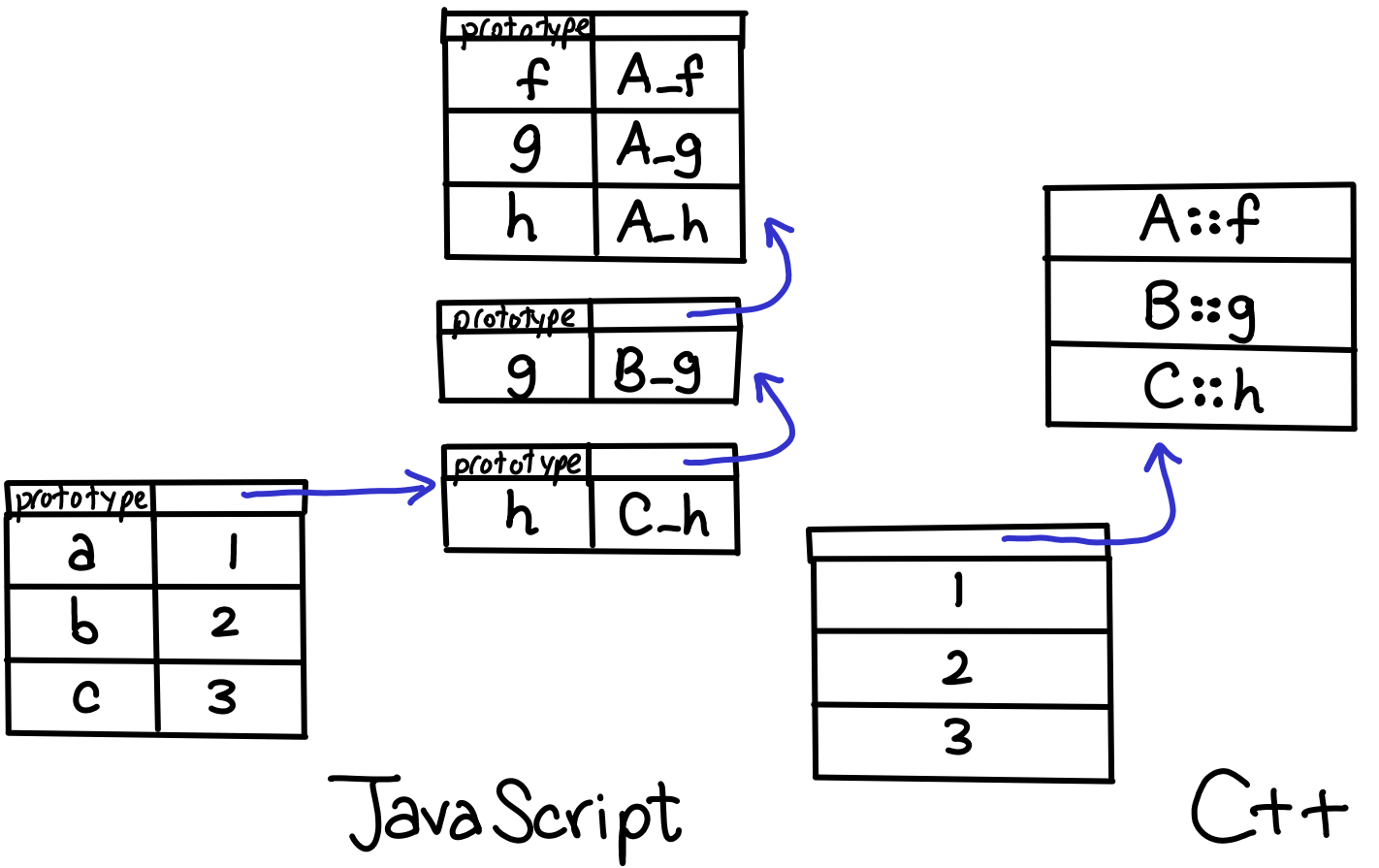



```
A* pa = new B();
pa->f(); // parent
```

--A_f(pa);

type-directed dispatch
(overloading)

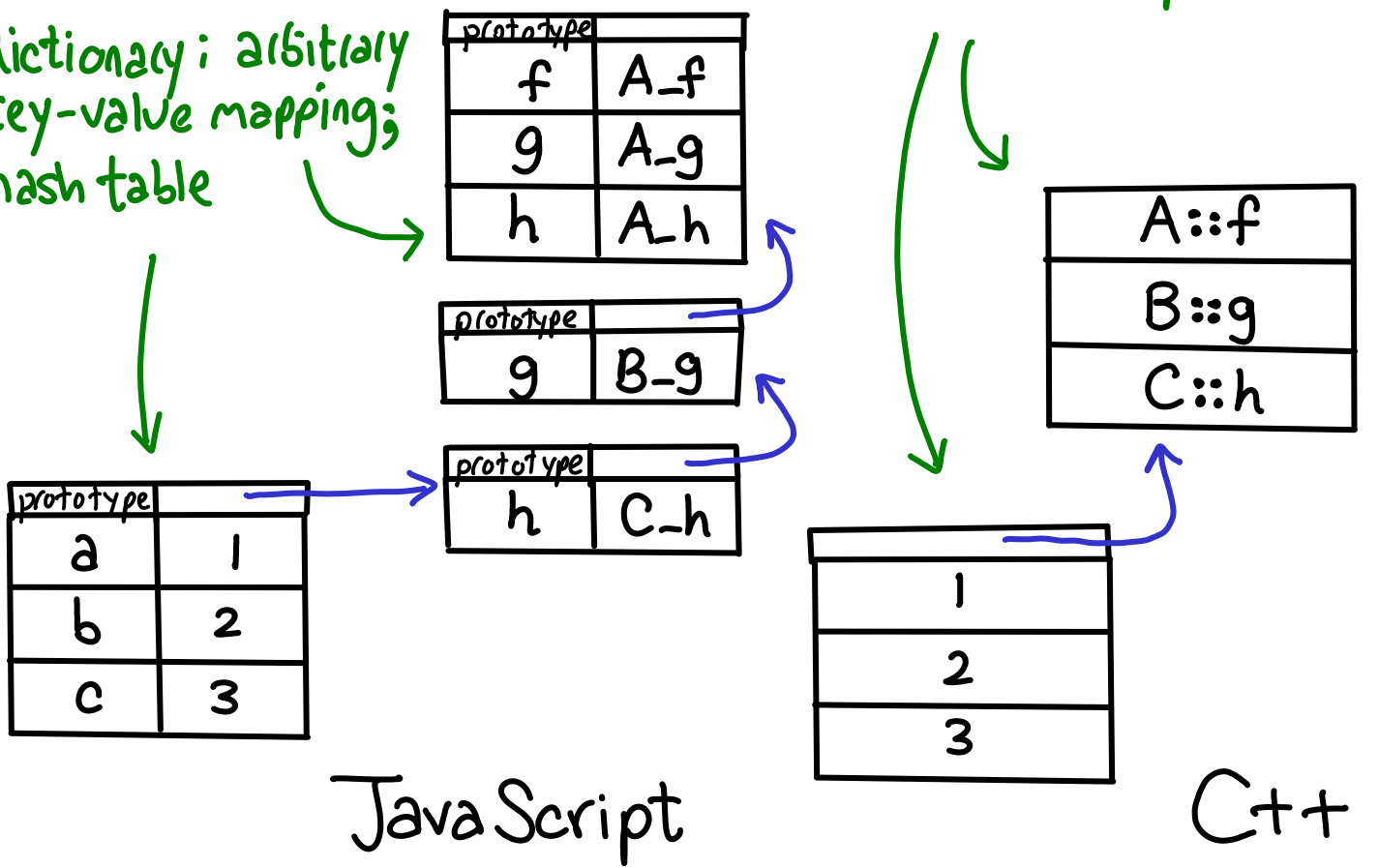
Comparison


prototype
f
A_f
g
A_g
h
A_h
prototype
g
B_g
prototype
h
C_h
prototype
a
1
b
2
c
3
JavaScript
A::f
B::g
C::h
1
2
3
C++

Comparison

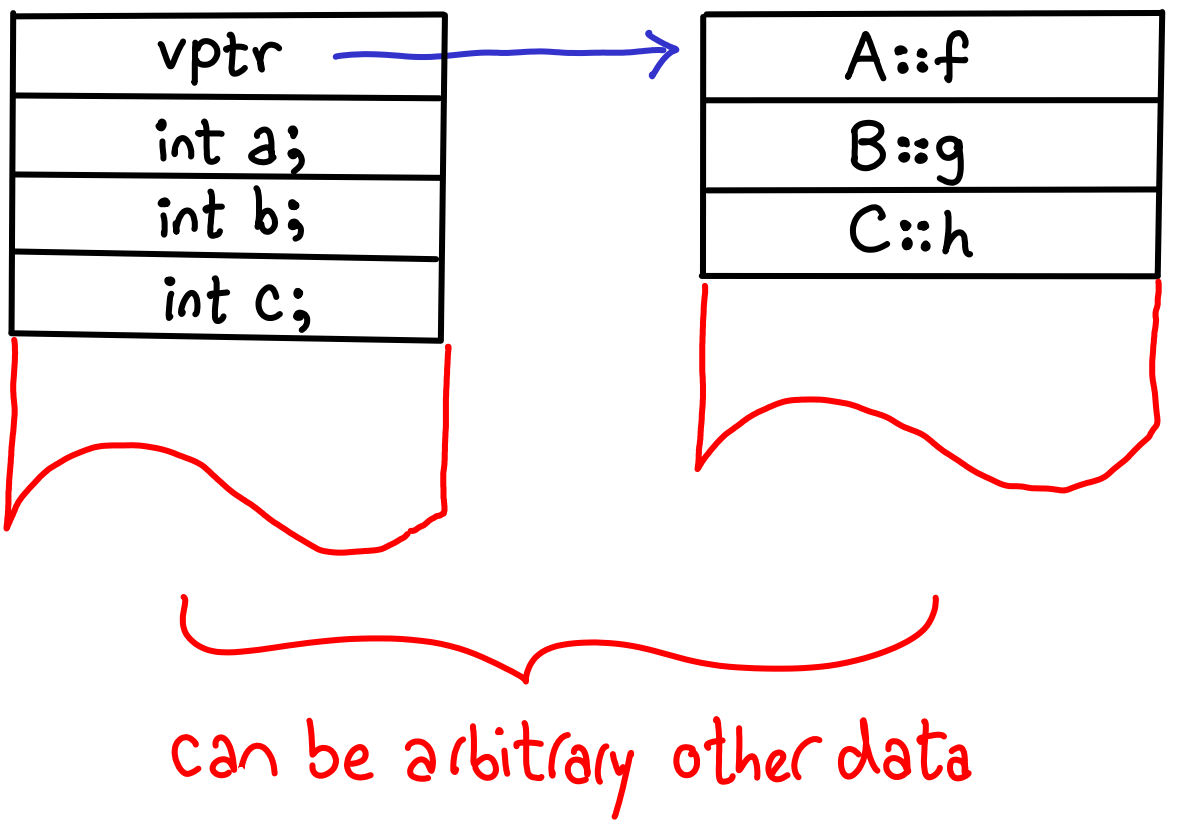
dictionary: arbitrary key-value mapping; hash table

array: offset known to compiler



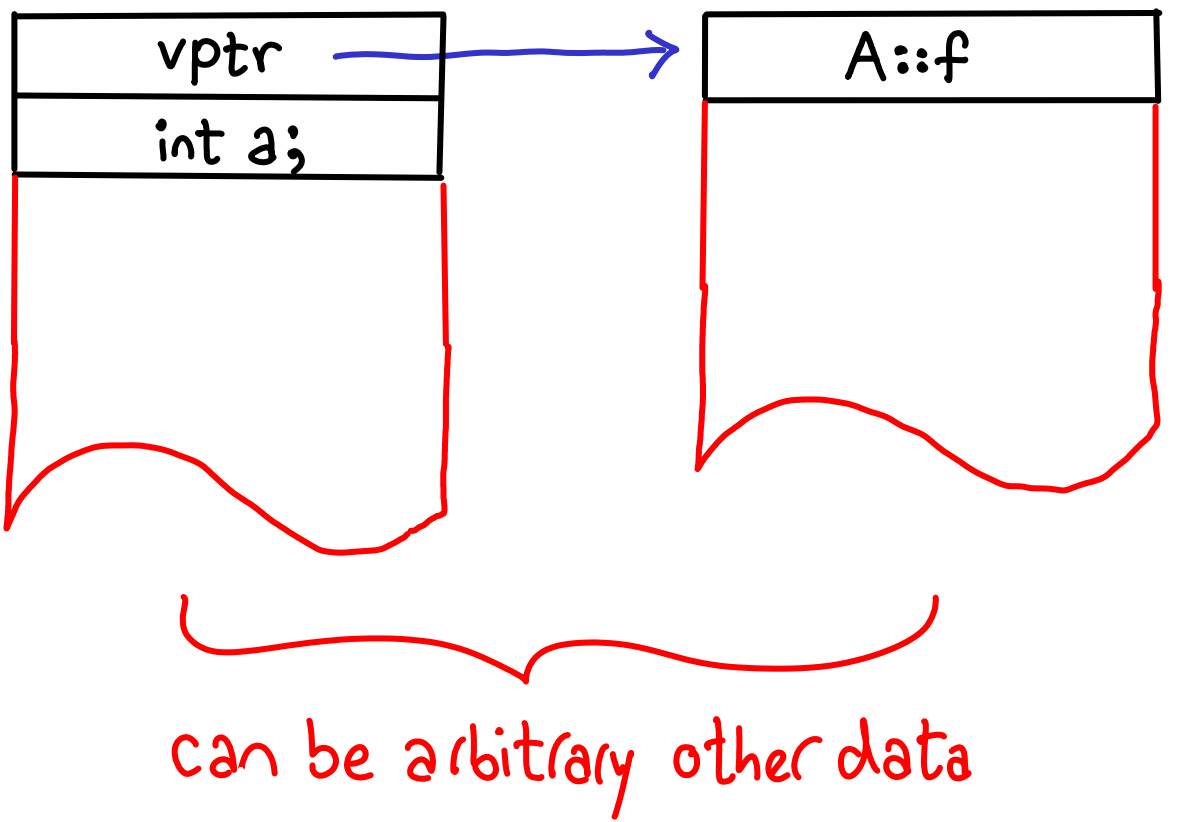
C++

class C object:



C++

class A object:



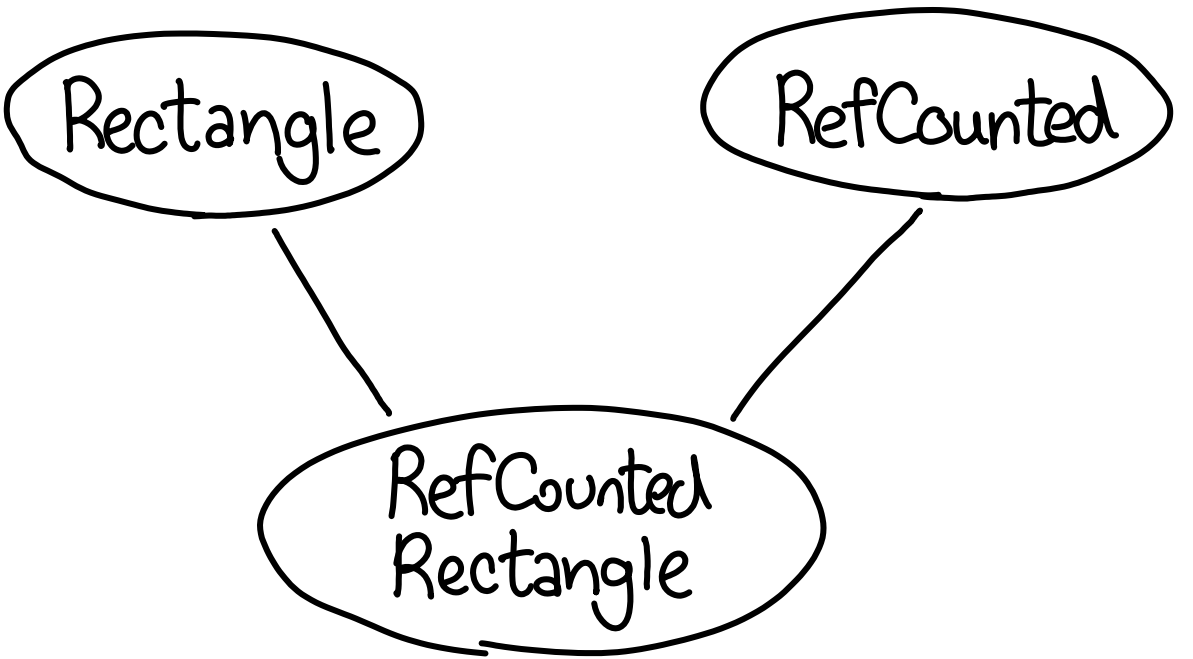
Virtual method call

= Invoke function pointer at fixed offset in vtable

Multiple inheritance

"Multiple inheritance is like a parachute, you don't need it very often, but when you do it is essential."

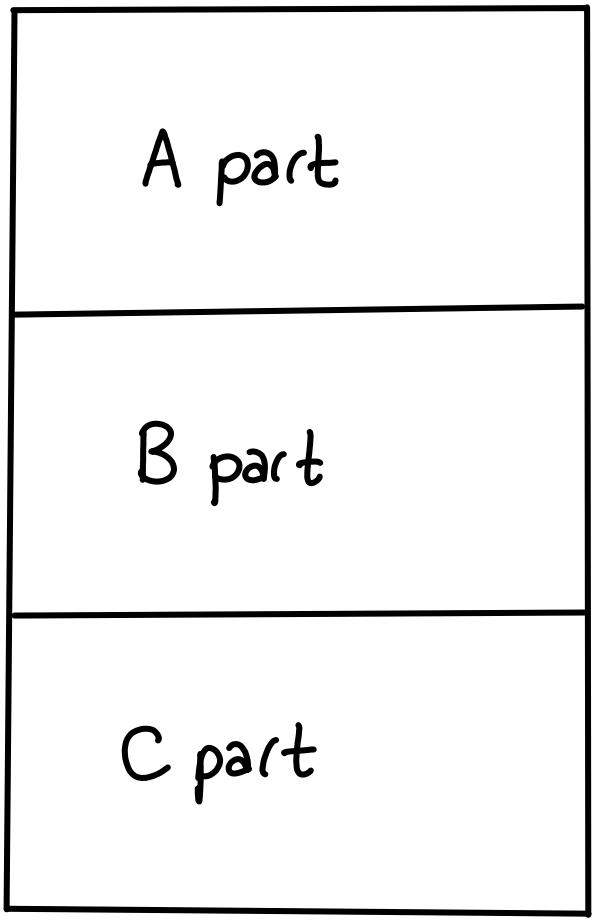
— Grady Booch


Rectangle
RefCounted
RefCounted
Rectangle

Key idea: Maintain ~same implementation strategy

Virtual method call
= Invoke function pointer at fixed offset in vtable

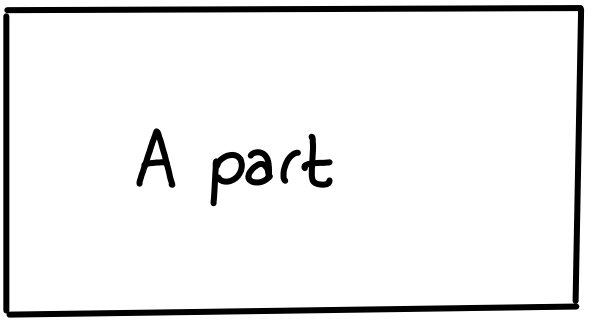
```
class A {...}
class B {...}
class C : A , B {...}
```



```
class A { void af(int);}
```

non-virtual

pa



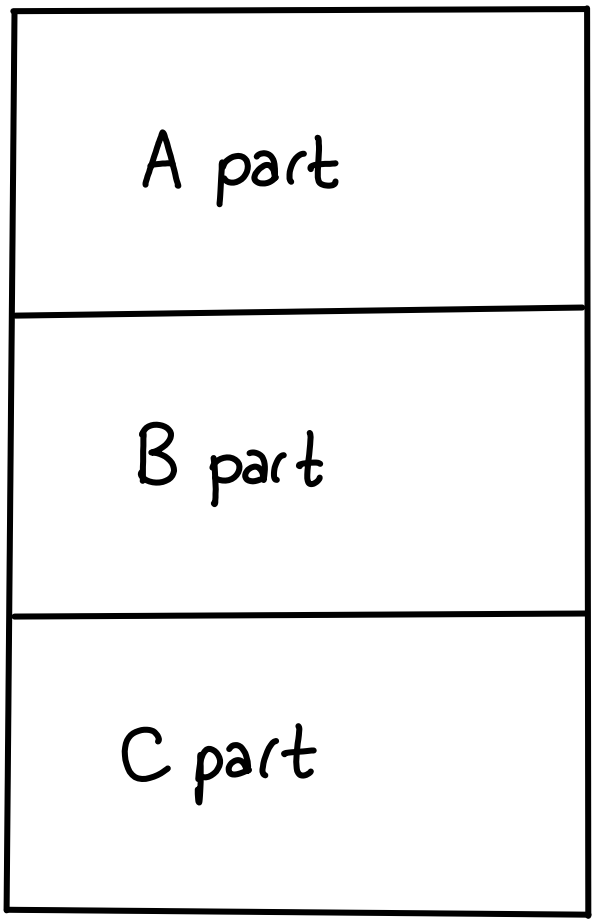

```
A* pa;
pa->af(2);
```

```
__A_af(pa);
```

class A { void af(int); }

non-virtual

pc



C* pc;
pc->af(2);

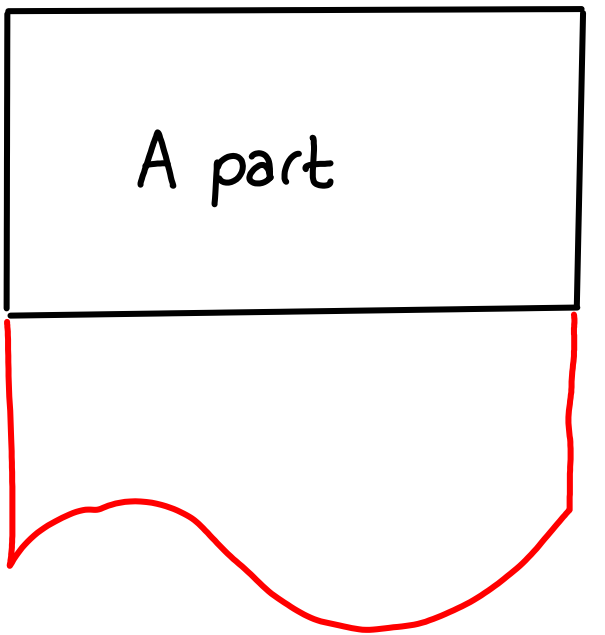
__A_af((A*)pc);

no-op

class A { void af(int); }

non-virtual

(A*) pc



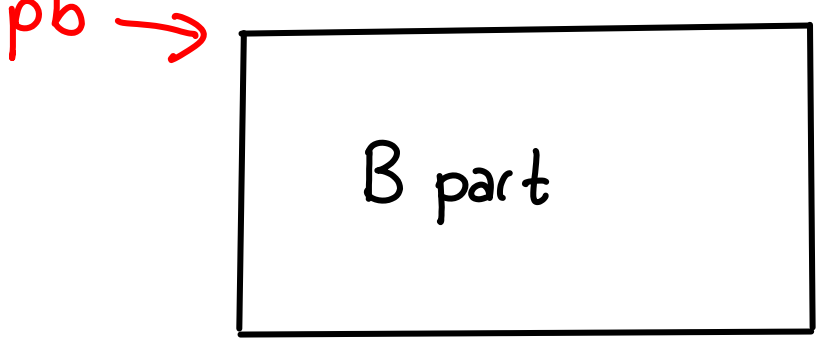
C* pc;
pc->af(2);

__A_af((A*) pc);

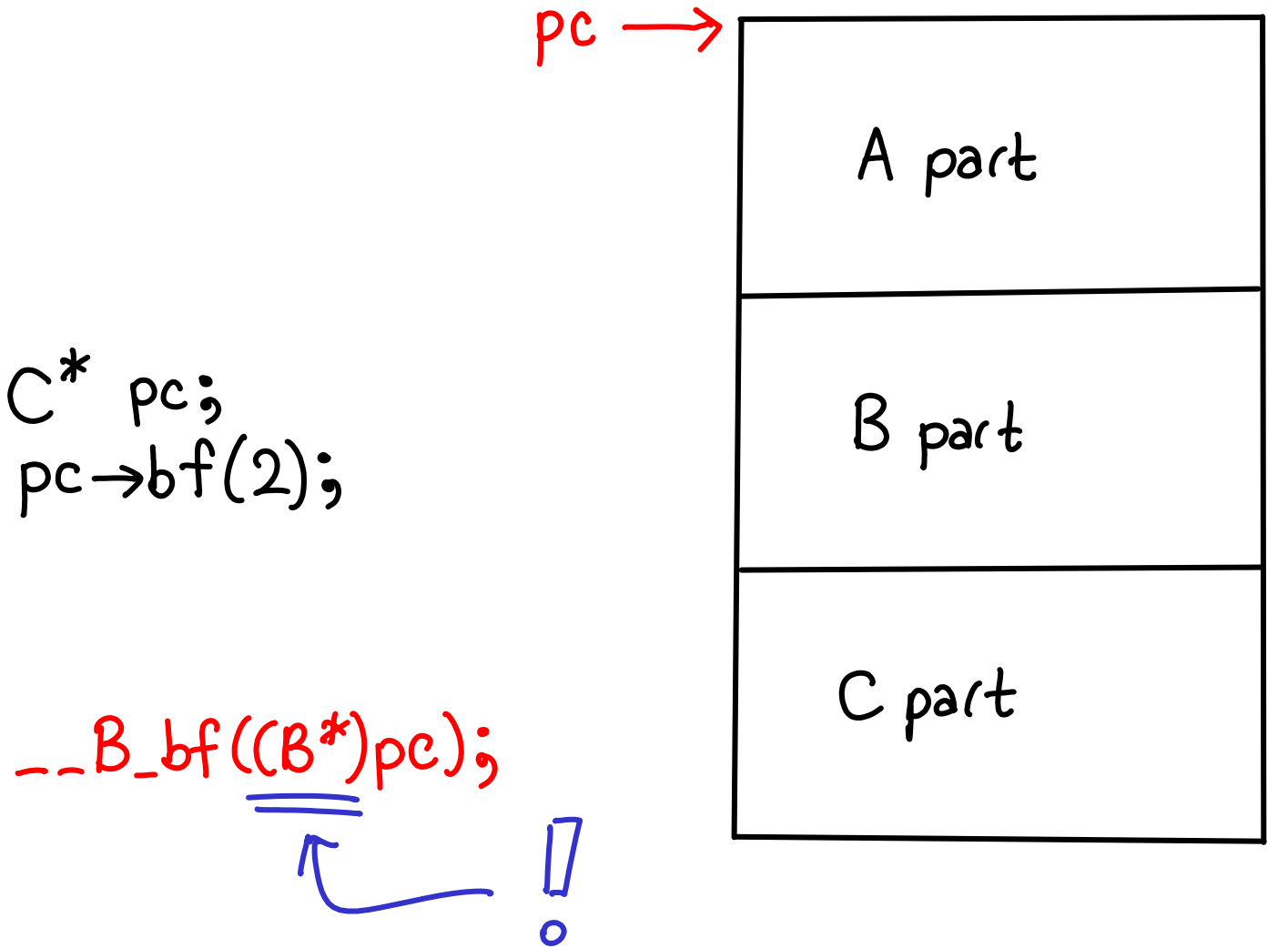
no-op

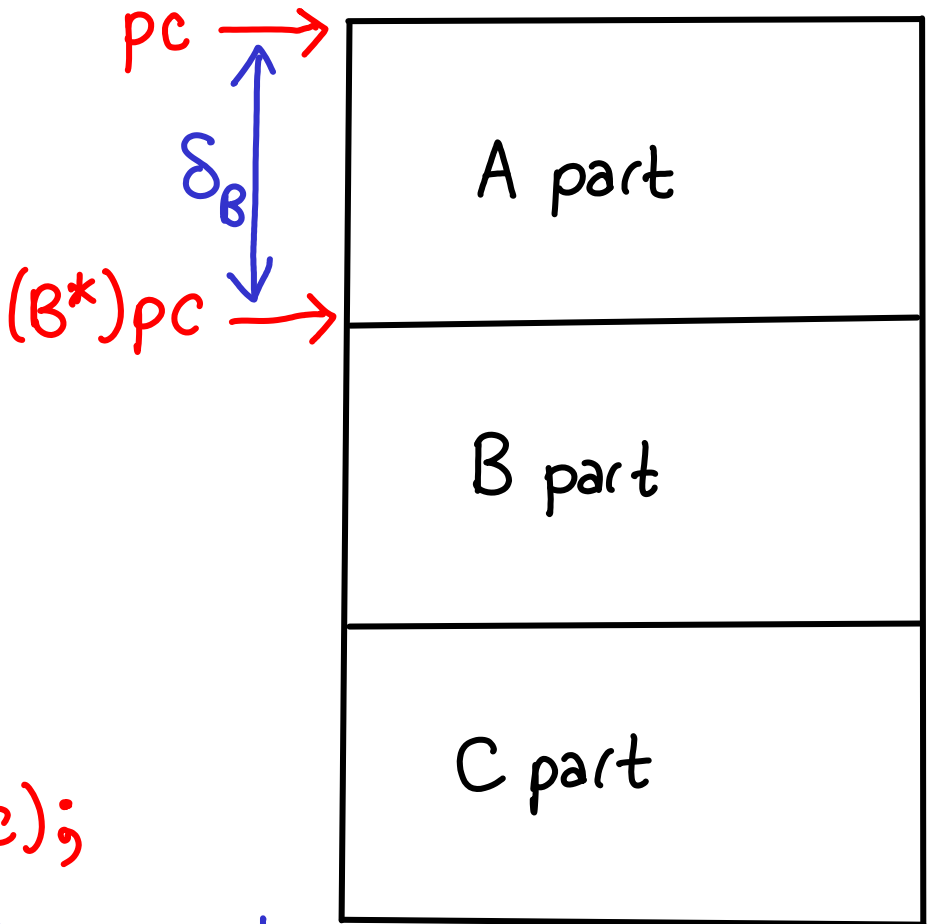
```
class B  { void bf(int);}
```

```
B* pb;
pb->bf(2);
```




```
__B_bf(pb);
```


pc
A part
B part
C part
C* pc;
pc->bf(2);
__B_bf((B*)pc);
!

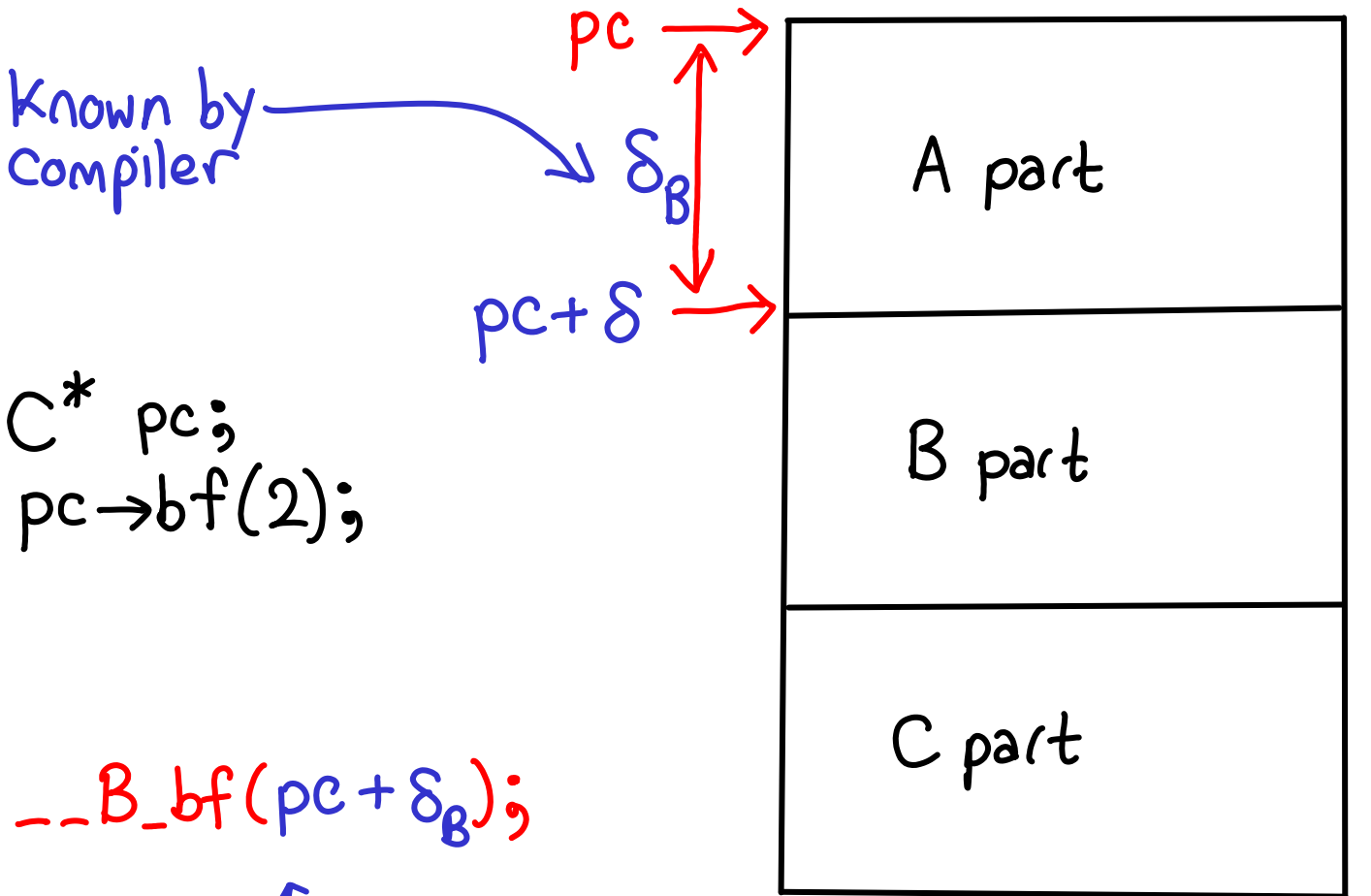



```
C* pc;
pc->bf(2);
```

```
__B_bf((B*)pc);
```

not a no-op!

Remark: (char*)(B*)pc ≠ (char*)pc




```
C* pc;
pc->bf(2);
```

```
__B_bf(pc+δB);
```

↳ add δ_B to pointer (overhead!)

Aside: name clashes

```
class Cowboy      { void draw(); }
class Displayable { void draw(); }

class CowboyWidget : public Cowboy,
                     public Displayable {}

CowboyWidget* pc;
pc->draw();
```

???

Aside: name clashes

```
class Cowboy      { void draw(); }
class Displayable { void draw(); }

class CowboyWidget : public Cowboy,
                     public Displayable {}

CowboyWidget* pc;
pc→Displayable::draw();
```

explicitly say which one

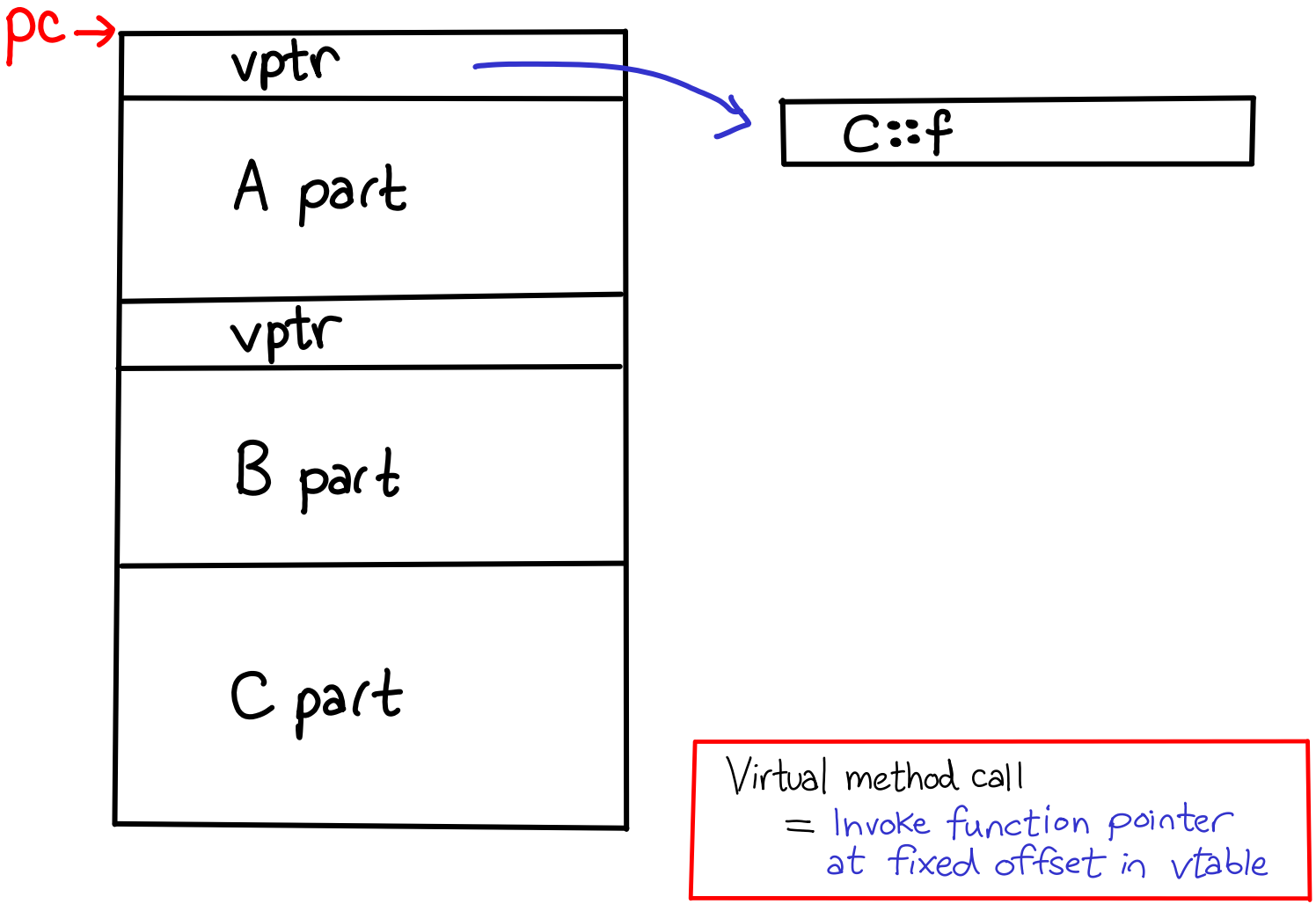
Virtual functions

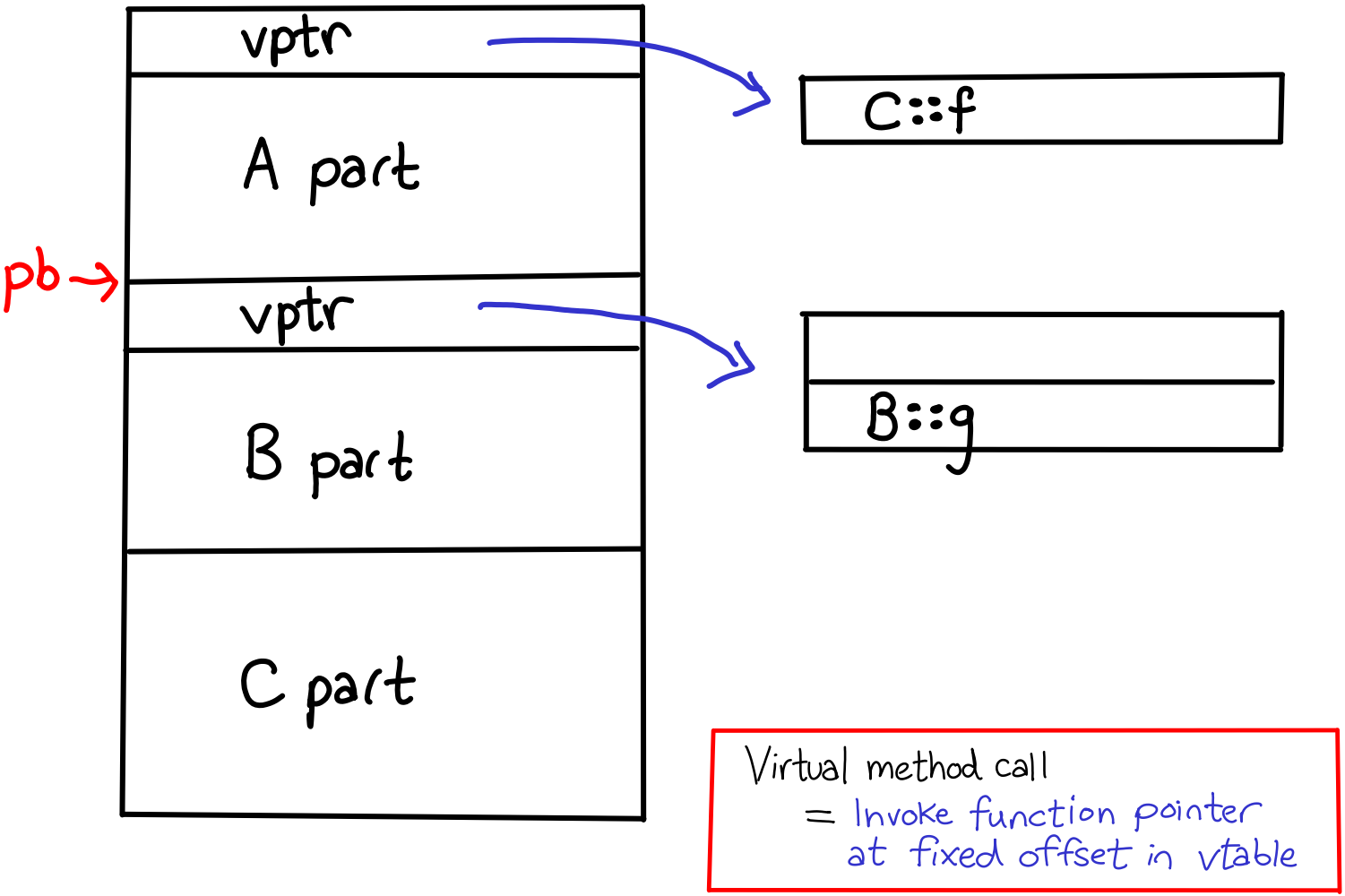
```
class A { virtual void f(); }
class B { virtual void f();
          virtual void g(); }
class C : A, B { void f(); }
```

implicitly virtual

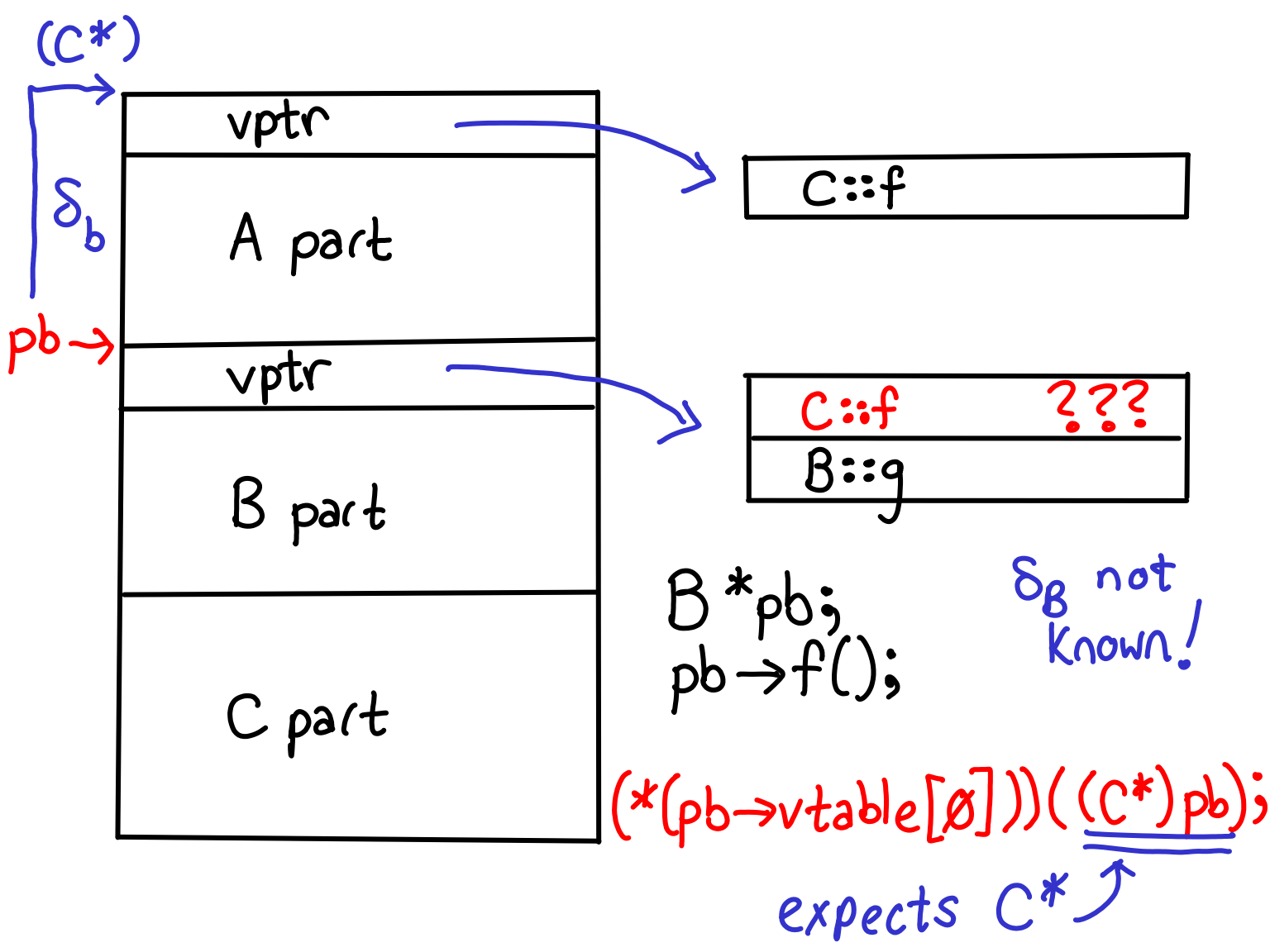
```
A* pa = new C;
B* pb = new C;
C* pc = new C;

pa->f();
pb->f();
pc->f();
```

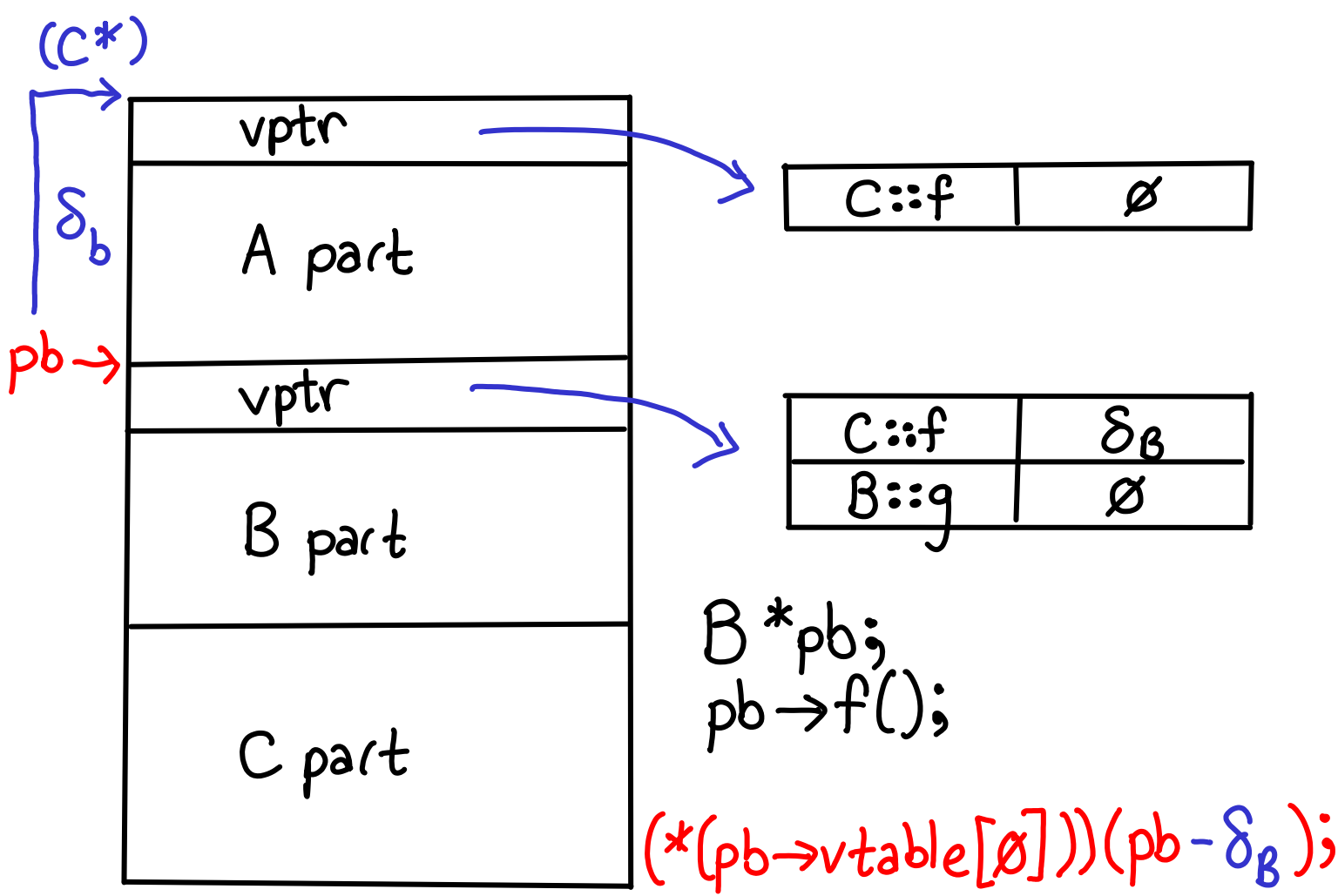
all invoke C::f()


pc
vptr
A part
vptr
B part
C part
C::f
Virtual method call
= Invoke function pointer
at fixed offset in vtable

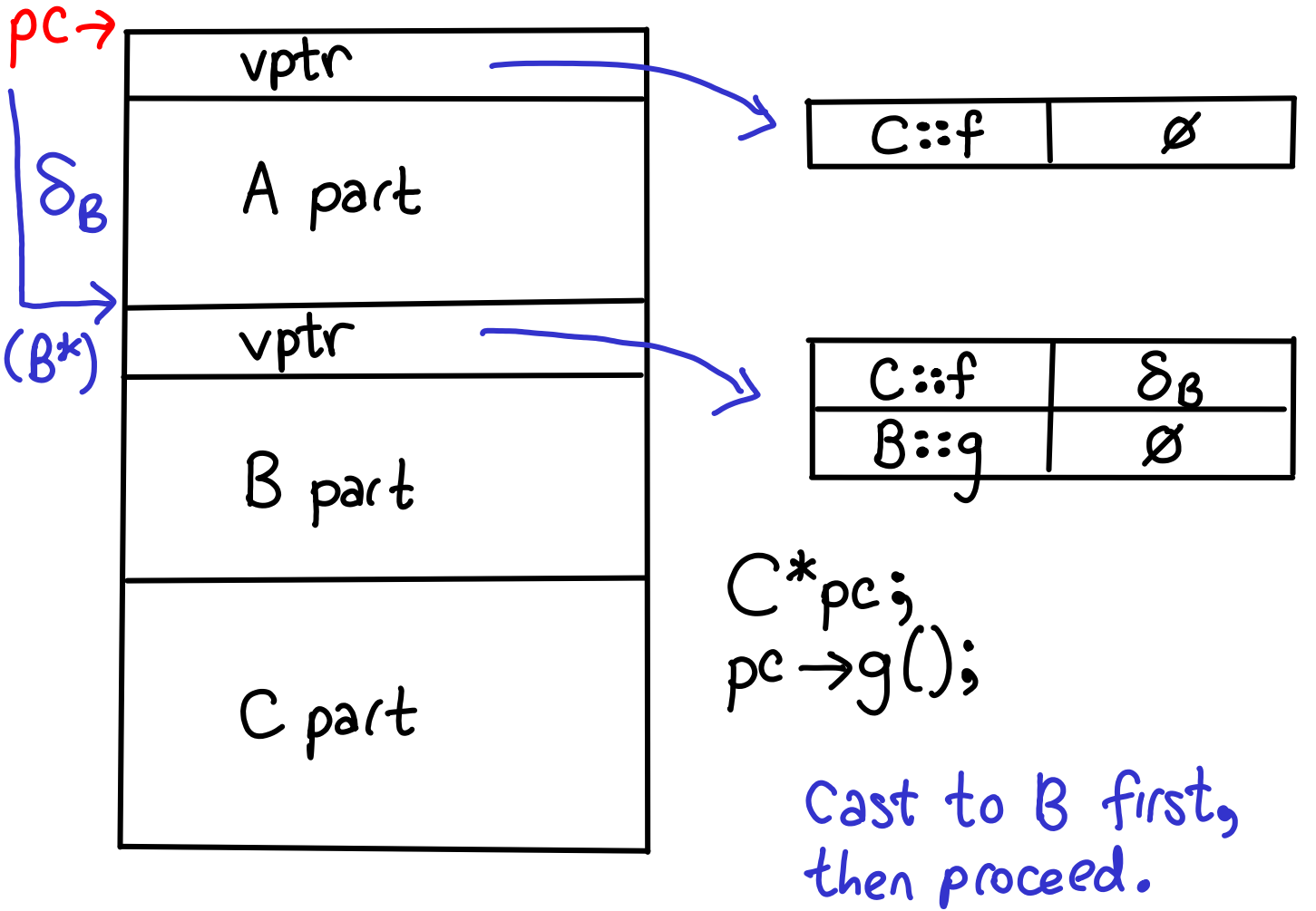


Virtual method call
= Invoke function pointer
at fixed offset in vtable


(C*)
vptr
A part
δb
pb
vptr
B part
C part
C::f
C::f ???
B::g
B *pb;
pb->f();
δB not Known!
(*(pb->vtable[0]))((C*)pb);
expects C*

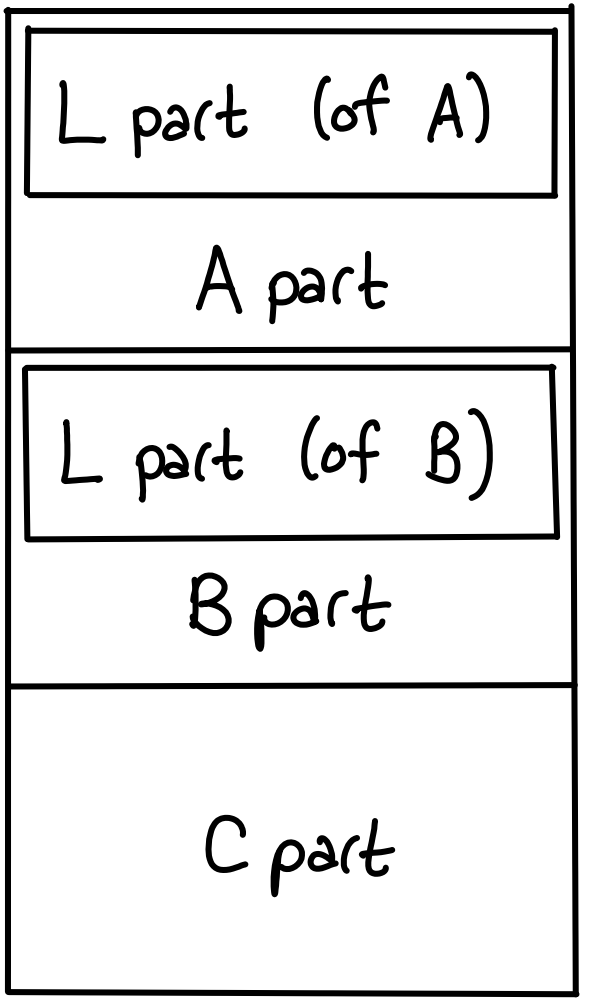


Note: sign often reversed here


pc→
vptr
A part
δB
(B*)
vptr
B part
C part
C::f
ø
C::f
δB
B::g
ø
C*pc;
pc→g();
Cast to B first,
then proceed.

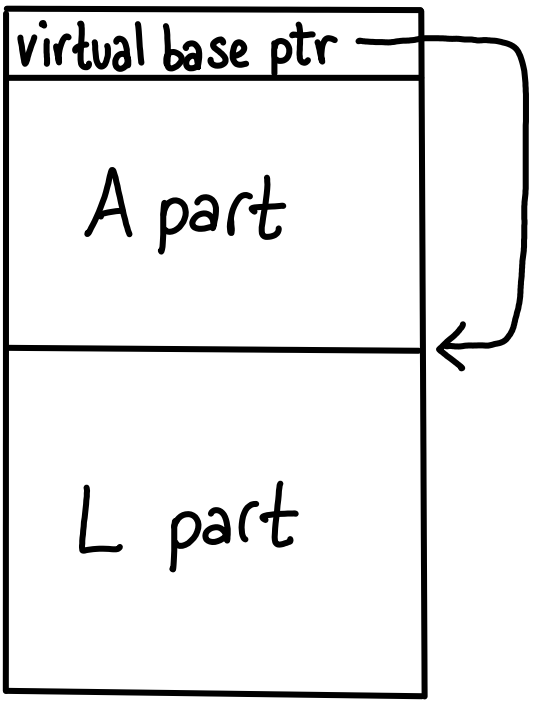
Diamond inheritance

```
class L {...}
class A : L {...}
class B : L {...}
class C : A, B {...}
```



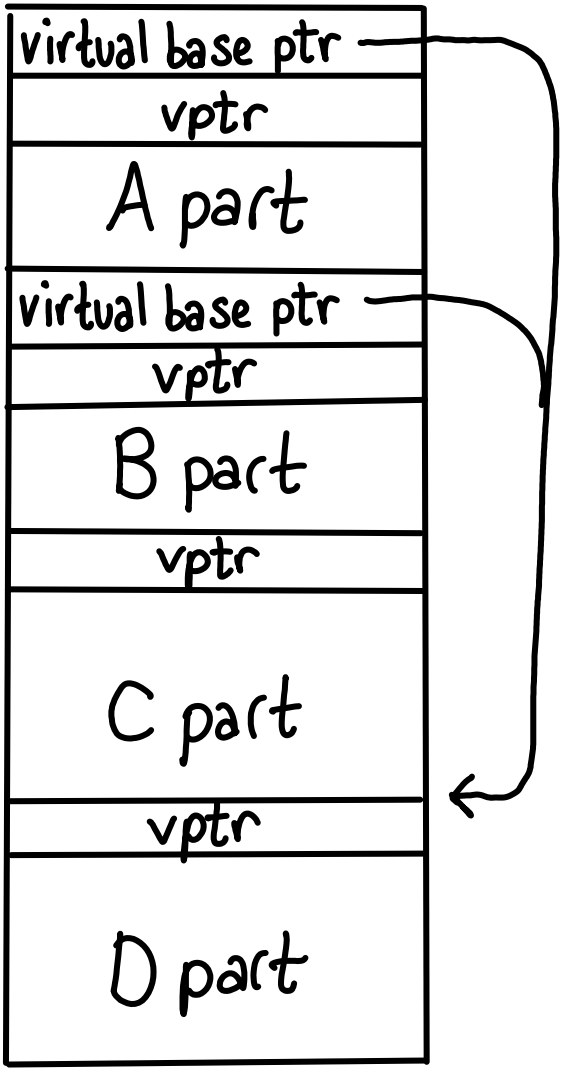
Virtual base classes

```
class L {...}
class A : virtual L {...}
class B : virtual L {...}
class C : A, B {...}
```



Virtual base classes

```
class L {...}
class A : virtual L {...}
class B : virtual L {...}
class C : A, B {...}
```



also need deltas...

Virtual base classes comparison

With virtual:

use a pointer to share L

don't use a pointer; inline contents

```
data L = L {...}
data A = A {alParent :: L, ...}
data B = B {blParent :: L, ...}
data C = C {caParent :: {-# UNPACK #-}!A,
            cbParent :: {-# UNPACK #-}!B, ...}
```

like in Haskell, can't go from L → A

Virtual base classes comparison

Without virtual:

```
data L = L {...}
data A = A {alParent :: {-# UNPACK #-}! L, ...}
data B = B {blParent :: {-# UNPACK #-}! L, ...}
data C = C {caParent :: {-# UNPACK #-}! A,
            cbParent :: {-# UNPACK #-}! B, ...}
```

Multiple inheritance summary

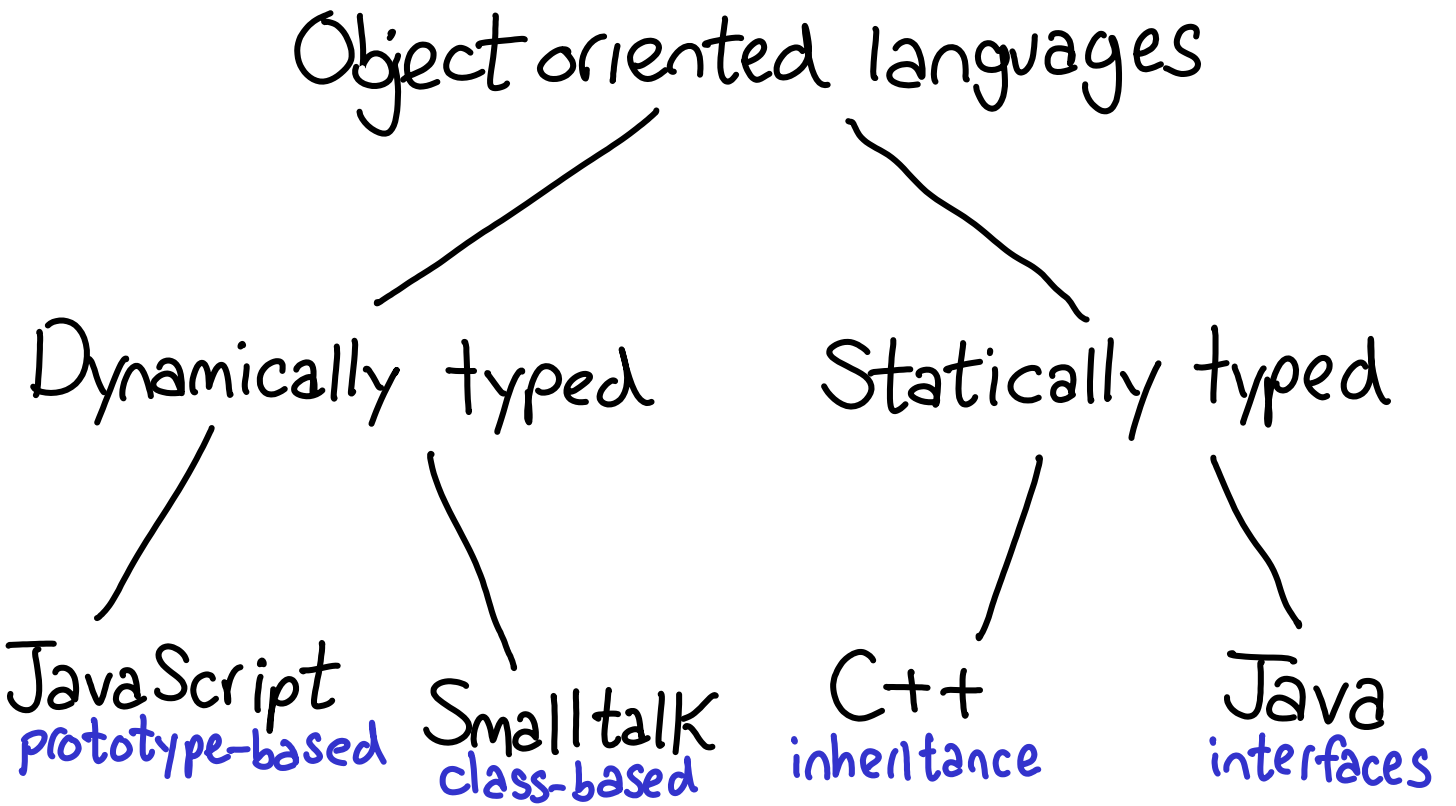
- One vtable per combination of base class and derived class
  - In general, need 1+ parents vtable, but derived vtable can be shared w/ first parent
- Delta required for all vtables multiply inherited class

The costs

Member access * of second or subsequent base class	subtraction (cast)
Virtual table	one word per function (delta)
Virtual call	deref (delta) + subtraction (cast)

* Only paid when using the feature!

enough work...

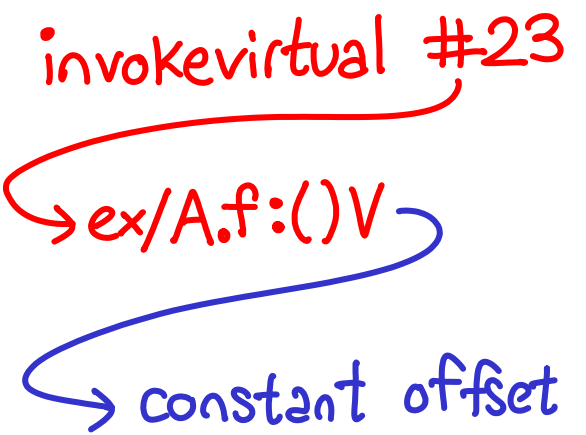

Object oriented languages
Dynamically typed
Statically typed
JavaScript
prototype-based
Smalltalk
class-based
C++
inheritance
Java
interfaces

C++	Java
OO extension of C	Simpler than C++
No GC	GC'd
Close to the machine	Efficiency secondary
Compiles to native	Compiles to bytecode Portability, Safety
No runtime	Runtime w/ JIT compiler

Java methods = C++ virtual methods

```
class A {
    public void f() {...}
}
class B extends A {
    public void f() {...}
}
```

```
A a = new B();
a.f();
```



(no non-virtual methods)

Java interfaces ≠ C++ multiple inheritance

```
interface A {
    void f();
}
interface B {
    void g();
}
class C implements A,B {
    public void f() {...}
    public void g() {...}
}
```

```
A a = new C();
a.f()
```

invokeinterface #24

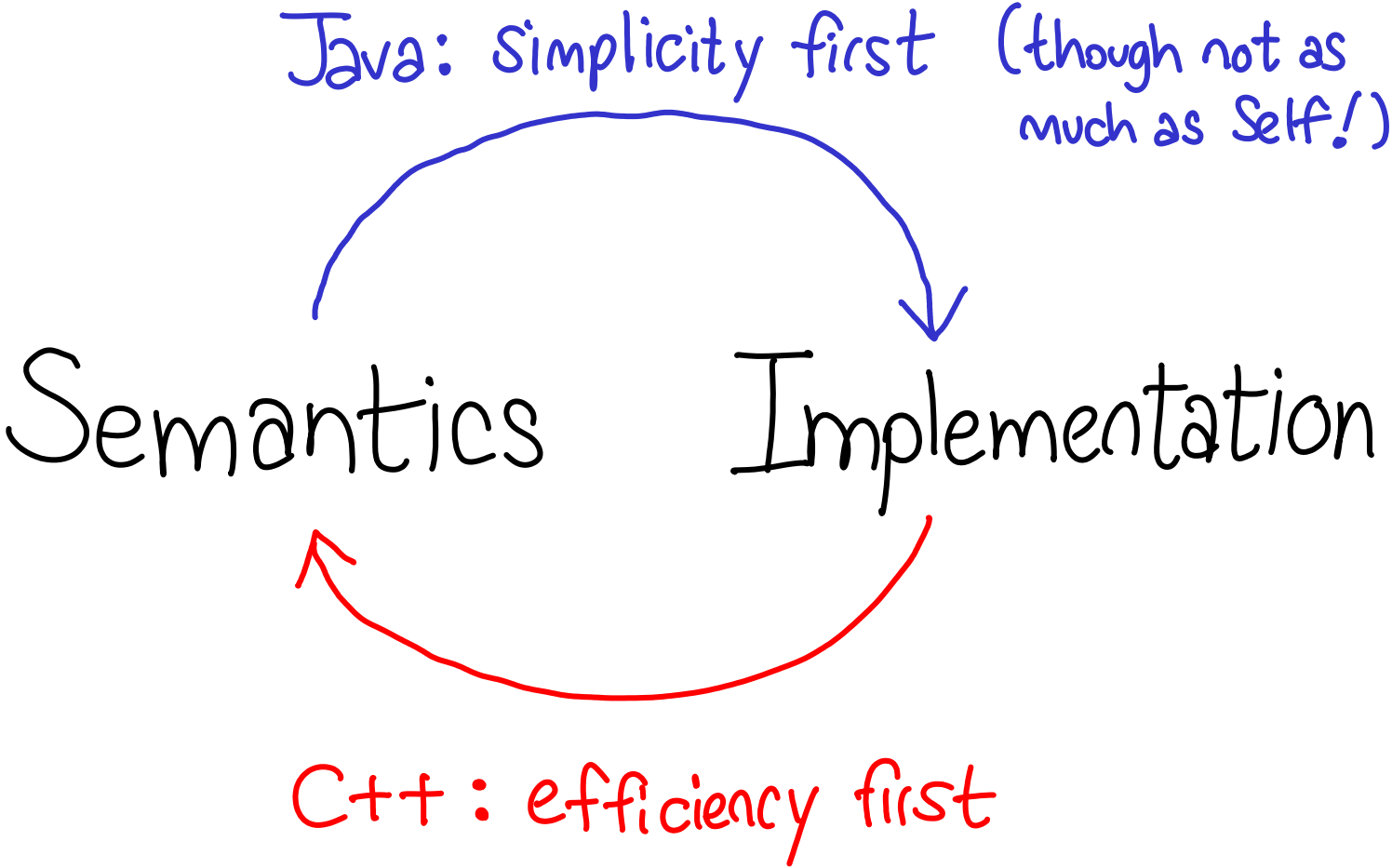
ex/A.f:()V

dynamic lookup!

Java: Inheritance to share code

C++: Inheritance to

share (binary) interface


Java: simplicity first (though not as much as Self!)
Semantics
Implementation
C++: efficiency first

Bonus problem

```
class InterfaceA {
    virtual void f();
}
class A : InterfaceA {
    void f() { ... }
}
class InterfaceB : InterfaceA {
    virtual void g();
}
class B : A, InterfaceB {
    void g() { ... }
}
```

http://stackoverflow.com/questions/1723469/interface-inheritance-in-c

Bonus problem 2

```
class W { virtual void f();
          virtual void g();
          virtual void h();
          virtual void k();
          ... }

class AW : virtual W { void g(); ... }
class BW : virtual W { void f(); ... }
class CW : AW, BW { void h(); ... }

CW* pcw = new CW;
```