Compiling Object-Oriented Languages

Key features:

- inheritance (possibly multiple)
- subtyping & subtype polymorphism
- message passing, dynamic binding

Inheritance & subtype polymorphism

Represent subclass as extension of superclass's representation

```
class A {
  int i;
  char* p;
};
class B : A {
  float m;
  A* a;
};

B* b = new B;
...
A* a = b;       // subtype polymorphism
...
... a->i ...    // works
```

Multiple inheritance

```
class A {
  int i;
};
class B {
  int j;
};
class AB : A, B {
  int ij;
};

AB* ab = new AB;
A* a = ab;
... a->i ...  // works
B* b = ab;
... b->j ...  // breaks
```


A solution

[Stroustrup 87]

When casting to a non-primary superclass,
change pointer to point to the middle of the object

```
B* b = ab;     // adds sizeof(A) to ab
... b->j ...  // works
```

+ can access fields directly w/o overhead
+ supports MI

– some assignments, casts execute code
– complicates runtime system (e.g. GC, method lookup)

– breaks when casting through e.g. `void*`:

```
void* p = ab;
...
B* b = (B*) p;
... b->j ...  // breaks
```

Representing & calling virtual functions

```
class A {
  int i;
  virtual int foo() { return i; }
};
class B : A {
  int j;
  virtual int foo() { return j; }
  virtual void set_foo(int f) { j = f; }
};
...
A* a = ...;
... a->foo() ...    // what foo gets called?
```

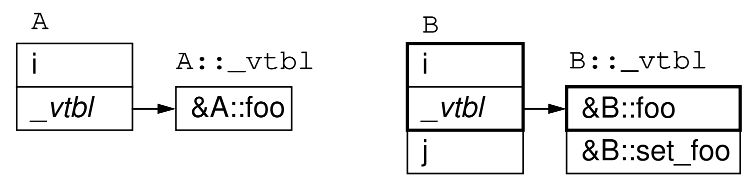
Could store pointer to virtual fn in each object

- \+ simple
- \+ reasonably fast to invoke virtual fn
- \+ flexible: each object could have its own methods
- – space-consuming

Virtual function tables

Observation:
each object in a class has the same virtual fn pointers

Idea:
factor out virtual fns into table shared by all instances of class



Call sequence:

```
... a->foo() ...
  ⇒
r1 := *(a + _vtbl_offset);
r2 := *(r1 + foo_index*4);
call *r2;
```

- \+ only 1 word of space overhead per object
- – extra indirection on call

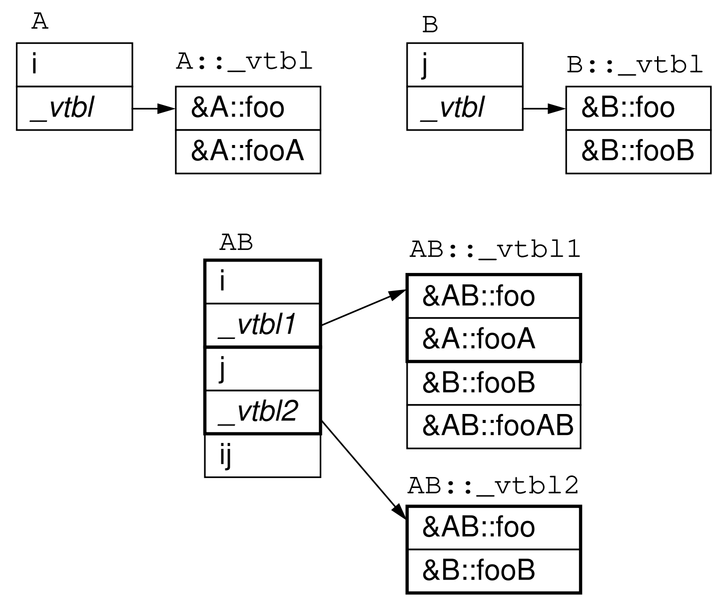
Multiple inheritance & virtual functions

Example source code:

```
class A {
  int i;
  virtual int foo()  { return fooA(); }
  virtual int fooA() { return i; }
};
class B {
  int j;
  virtual int foo()  { return fooB(); }
  virtual int fooB() { return j; }
};
class AB : A, B {
  int ij;
  virtual int foo()   { return fooAB(); }
  virtual int fooAB() { return ij; }
};
```


Object layout of example

Multiple vtbl pointers required in `AB` representation



A problem

```
AB* ab = new AB;
... ab->fooAB() ...// works
... ab->fooA()  ...// works
... ab->fooB()  ...// breaks
```

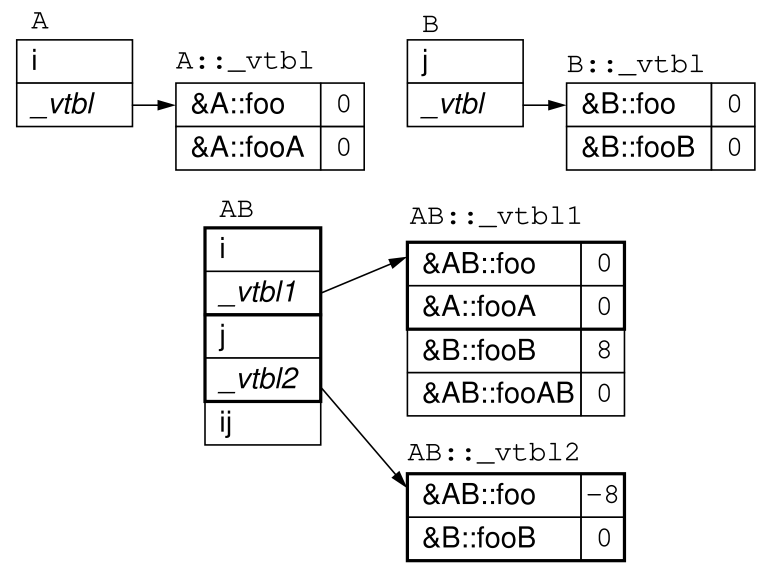
Need to perform implicit cast of receiver argument
as part of virtual function call

- – caller can't do it ⇒ doesn't know callee method
- – callee can't do it ⇒ doesn't know static type of actual
- + can be done in virtual tables
  - vtbl is specific to actual's static type in callee & callee method is known

One solution

Each entry of vtbl is a pair:

- pointer to callee function
- offset to be added to receiver pointer




```
... a->foo() ...
  ⇒
r1 := *(a + _vtbl_offset);
r2 := *(r1 + foo_index*8);
r3 := *(r1 + foo_index*8+4);
r4 := add a, r3;
call *r2;
```

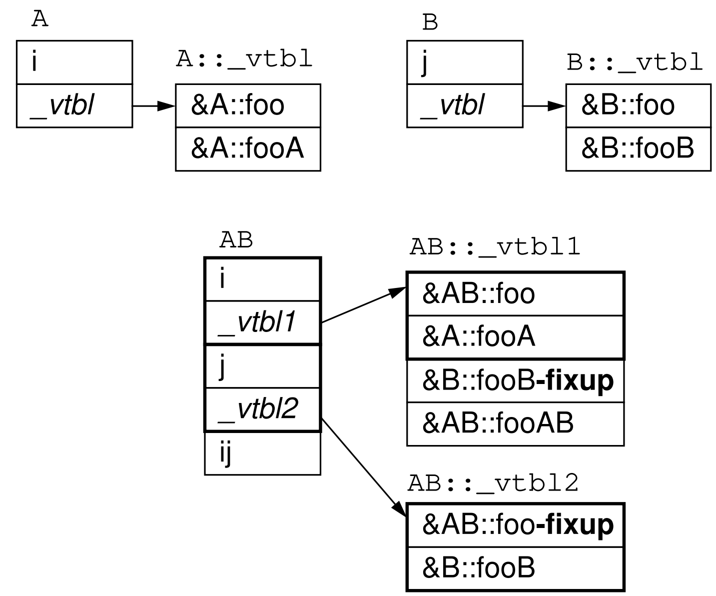
Evaluation

+ virtual functions can coexist with multiple inheritance
– calling sequence has 2 more instructions
– vtbls are twice as big
– does not support implicit cast for other args, result
    ⇒ forces “nonvariance” for function overriding in C++

An alternative

Observation: most receiver offsets are 0

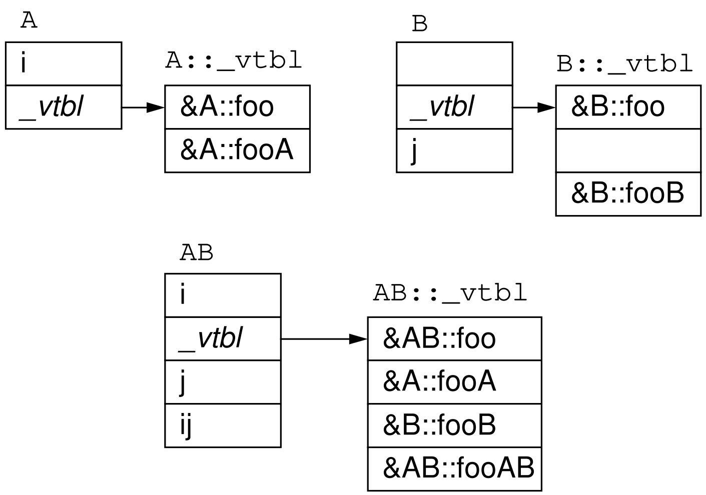
Idea: use single inheritance call sequence,
    insert intermediary fns to implement non-0 offsets



+ fast call sequence, usually
+ pay for MI only when used

Another alternative

Insert padding in assigned field/function offsets to avoid conflicts among parents
[Dixon *et al.* 89], [Pugh & Weddell 90]



+ always fast access, invocation
+ no pointers to middle of objects
+ pay (at run-time) for MI only when used
– space cost
– complex link-time analysis (graph coloring!)

Static analysis of OO programs

[Chambers & Ungar 91]

If knew exact class of receiver at compile-time,
could replace dynamically-dispatched send
with statically-bound call

+ no run-time dispatching overhead
+ can apply inlining, interprocedural analysis

If knew receiver was one of small # of classes at compile-time,
could still do decent interprocedural analysis

Particularly important in dynamically-typed and/or pure OOLs

- message sends frequent
- no static type declarations to help implement them efficiently

In Self:

– w/o static analysis (and other tricks):
10-1000x slower than optimized C
+ w/ static analysis: 50% of optimized C

[caveat: small integer/array benchmarks]

Intraprocedural class analysis

Propagate forwards sets of bindings of variables to sets of classes

e.g. $\{x \rightarrow \{Int\}, y \rightarrow \{Vector,String\}\}$

Base cases:

- x := *const*;
- x := new *class*;

Propagate through assignments

Exploit at sends, type tests

- if x.class = Vector goto ...
- x := send “foo”(y,z)

Weakness: don’t know class of

- formals
- results of non-inlined messages
- contents of instance variables

Specialization

To get better static info, specialize source method w.r.t. inheriting receiver class

+ compiler knows statically the class of the receiver formal

```
class Rectangle {
  virtual int area() {
    return length() * width(); }
  virtual int length() = 0;
  virtual int width() = 0;
};

class Square : Rectangle {
  int size;
  virtual int length() { return size; }
  virtual int width() { return size; }
};
```

If specialize `Rectangle::area` as `Square::area`, can inline-expand `length()` & `width()` virtual fn calls

What to specialize?

In Sather, Trellis: specialize for all inheriting receiver classes

- in Trellis, reuse superclass's code if no change

In Self: specialize at run-time

- Self compiles everything at run-time, incrementally as needed
- will only specialize for (classes $\times$ messages) actually used at run-time

In Cecil: use profile-derived weighted call graph to guide specialization
[Dean *et al.* 95a]

- only specialize if high frequency & provides benefit
- can specialize on args, too

Class hierarchy analysis

[Dean *et al.* 95b]

Even if don't know exact receiver class, examining class hierarchy often enables static binding

Example:

```
class Shape {
  virtual int area() = 0;
};
class Rectangle : Shape {
  virtual int area() {
    return length() * width(); }
  virtual int length() = 0;
  virtual int width() = 0;
};
class Square : Rectangle {
  int size;
  virtual int length() { return size; }
  virtual int width() { return size; }
};


Rectangle *r = ...;
... r->area() ...
```

Interprocedural static class analysis

Want to do interprocedural class analysis

- more effective optimization
- relieve pressure to over-inline

Problem: circularity

- need to compute call graph in parallel with receiver classes

Current research area

[Palsberg & Schwartzbach 91,92,93]

[Plevyak & Chien 94]

[Agesen 94]

[Grove *et al.* 9x]

Issues:

- handling polymorphic data structures & functions effectively
- balancing cost of analysis and precision of results

Profile-guided receiver class prediction

Can exploit dynamic profile information if static info lacking

Monitor receiver class distributions for each send

- traditional off-line profiling [Grove *et al.* 94]
- on-line profiling in a dynamic-compilation-based system [Hölzle & Ungar 94]

Recompile program, inserting run-time class tests for common receiver classes

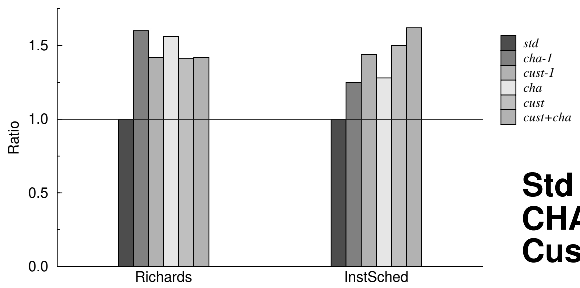
Before:

```
i := s->area();
```

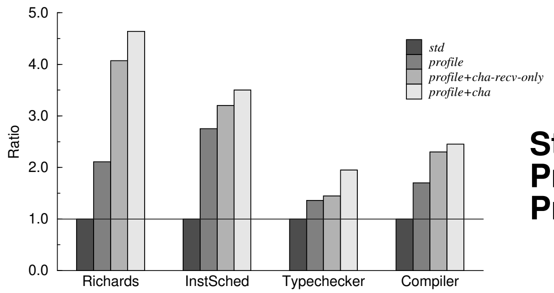
After receiver class prediction:

```
i :=
  s->class == Rect ? s->Rect::area() :
  s->class == Circle ? s->Circle::area() :
  s->area();
```

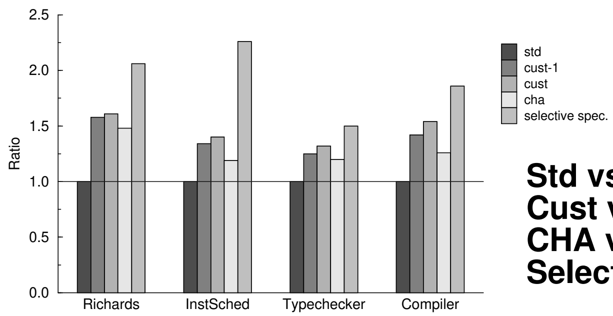
Some experimental results



Std vs. CHA vs. Cust



Std vs. Prof. vs. Prof. + CHA



Std vs. Cust vs. CHA vs. Selective