U.PORTO FEUP FACULDADE DE ENGENHARIA UNIVERSIDADE DO PORTO
DEI DEPARTAMENTO DE ENGENHARIA INFORMÁTICA

Compilers
Design and Implementation

Run-Time Environments

Implementing Object-Oriented Languages

Recapitulation

- What Have We Learned?
  - AR is a Run-Time structure to hold ***State*** regarding the execution of a procedure
  - AR can be allocated in Static, Stack or even Heap
  - Links to allow Call-Return and Access to Non-local Variables
  - Symbol-table plays important role

- Not Yet Done with Procedures
  - Saving Context before call and restoring after the call
  - Need to understand how to generate code for body

Implementing Object-Oriented Lang.

Mapping "message" or names to methods

- Static mapping, known at compile-time (Java, C++)
  - Fixed offsets & indirect calls
- Dynamic mapping, unknown until run-time (Smalltalk)
  - Look up name in class' table of methods

Want uniform placement of standard services *(**NEW, PRINT**, …)*

This is really a Data-Structures Problem

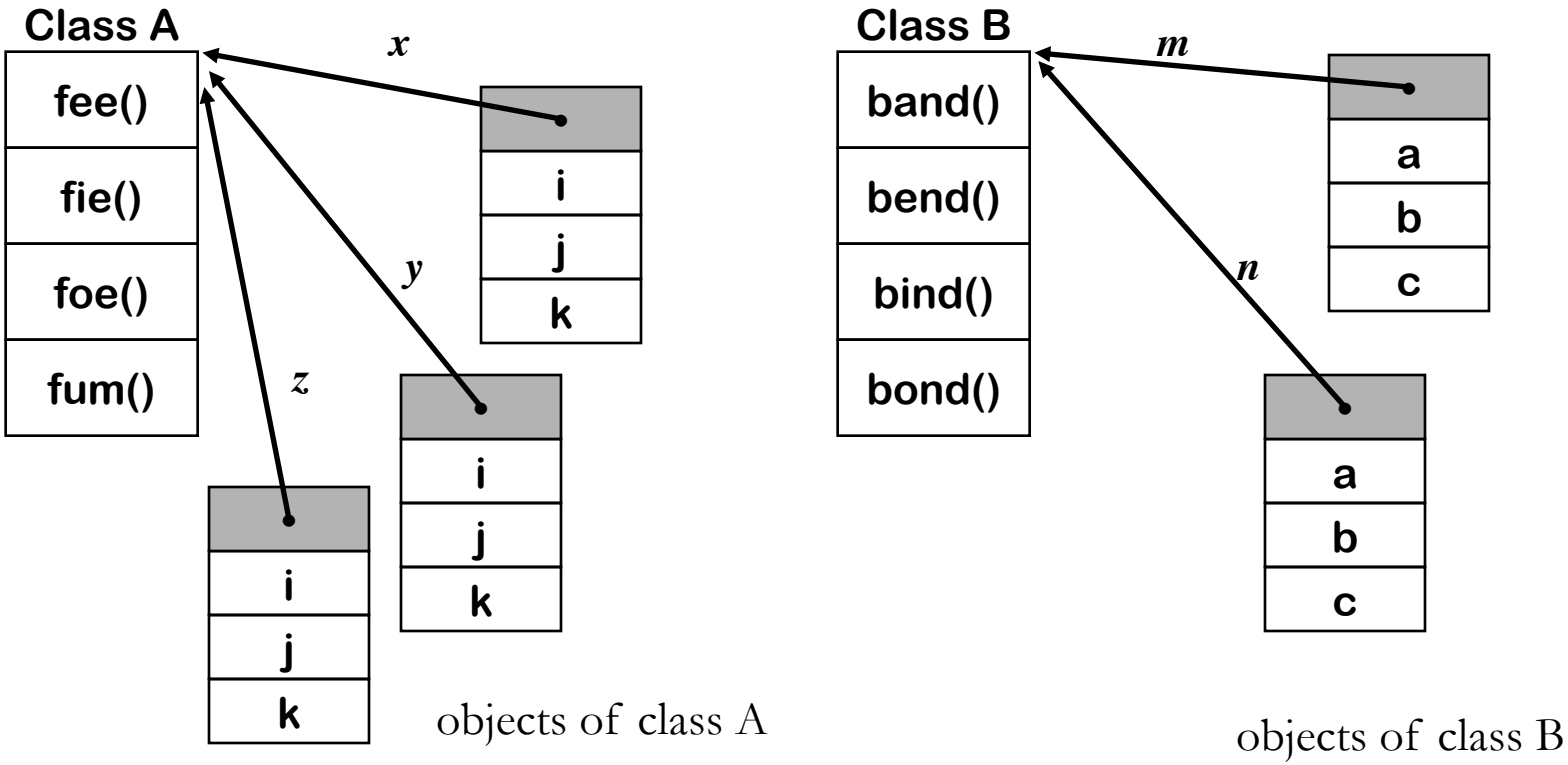
- Build a Table of Function Pointers
- Use a Standard Invocation Sequence

Basics and Terminology

1. **Object** An object is an abstraction with one or more members. Those members can be data items, code that manipulates those data items, or other objects. Each object has internal state—data whose lifetimes match the object's lifetime.
2. **Class** A class is a collection of objects with the same abstract structure and characteristics. A class defines the set of data members in each instance of the class and defines the code members (methods) that are local to that class. Some methods are public, or externally visible, others are private, or invisible outside the class.
3. **Inheritance** Inheritance refers to a relationship among classes that defines a partial order on the name scopes of classes. Each class may have a superclass from which it inherits both code and data members. If a is the superclass of b, b is a subclass of a. Some languages allow a class to have multiple superclasses.
4. **Receiver** Methods are invoked relative to some object, called the method's receiver. The receiver is known by a designated name, such as *this* or *self*, inside the method.

Implementing Object-Oriented Lang.

With static, compile-time mapped classes



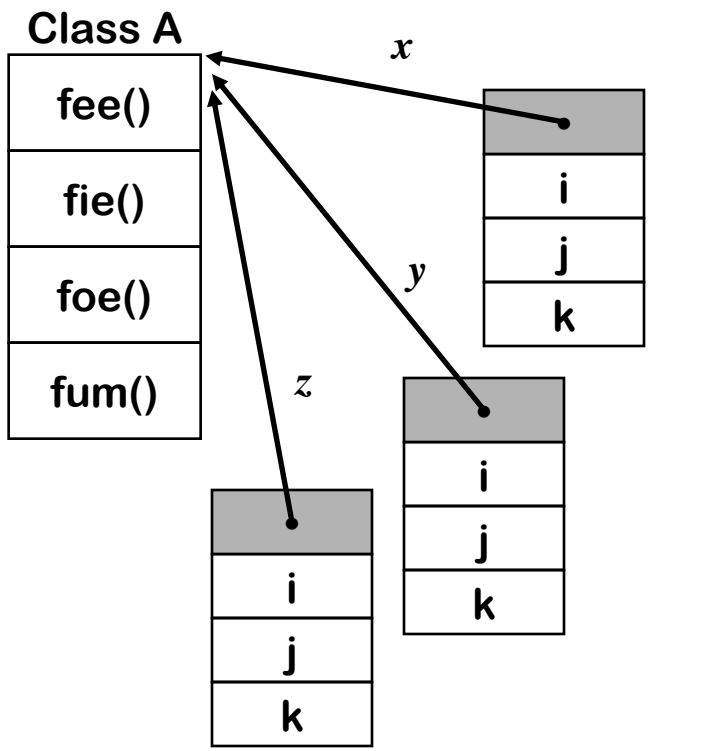
Message Dispatch becomes an indirect call through a function table

Implementing Object-Oriented Lang.

Method Code:

```
void A:fee(){
  …
  self->i = 0;
  …
}
```

```
t0 = &self + 4;
*t0 = 0;
```



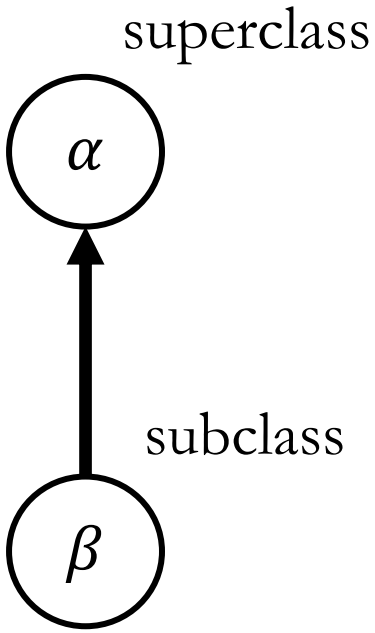
objects of class A

Method Accesses Object Data as Offsets from the **self** Reference

(analogous to access of struct fields in C)

Inheritance (simple)

- If α is a superclass of β, then β is a subclass of α and any method defined in α must operate correctly on an object of class β, if it is visible in β.
- The converse is not true; a method declared in class β cannot be applied to an object of its superclass α, as the method from β may need fields present in an object of class β that are absent from an object of class α

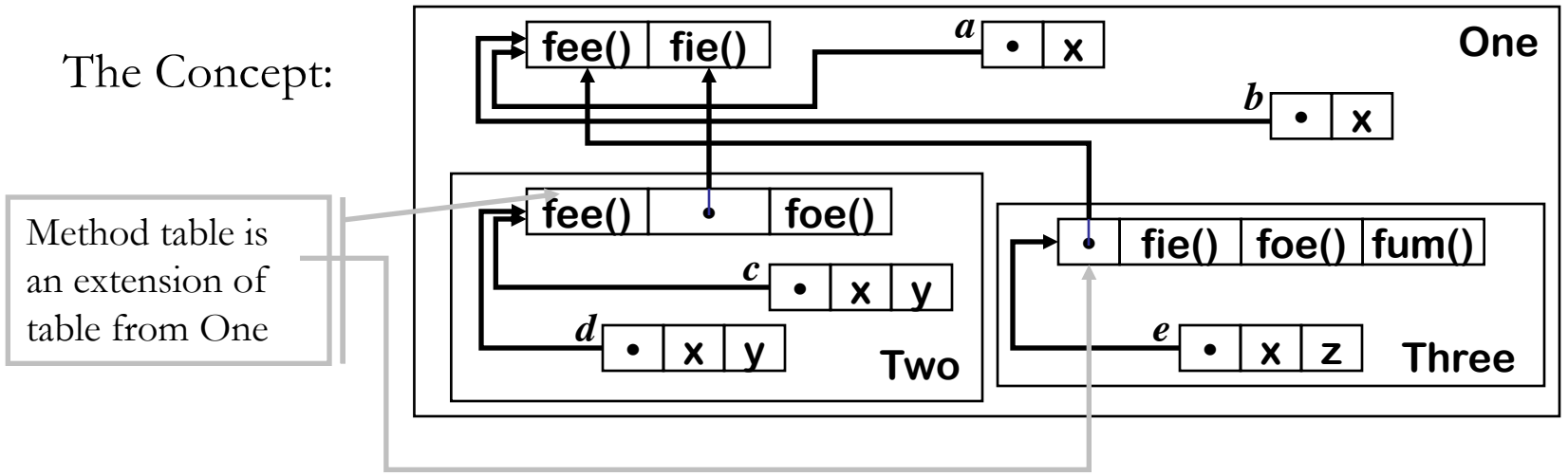


β derives from α

The Inheritance Hierarchy

To simplify object creation, we allow a class to inherit methods from an ancestor, or *superclass*. The descendant class is called the *subclass* of its ancestor.

The Concept:



Principle:

- B *subclass* A $\Rightarrow$ d $\in$ B can be used wherever a $\in$ A is expected
  - B has all the methods defined in its super class (in this case class A)
  - B may override a method definition from A
- Subclass provides all the interfaces of superclass

The Inheritance Hierarchy

Two distinct Implementation Philosophies

Static Class Structure

- Can map name to code at compile time
- Leads to 1-level jump vector
- Copy superclass methods
- Fixed offsets & indirect calls
- Less flexible & expressive

Dynamic Class Structure

- Cannot map name to code at compile time
- Multiple jump vector (1/class)
- Must search for method
- Run-time lookups caching
- Much more expensive to run

Impact on name space

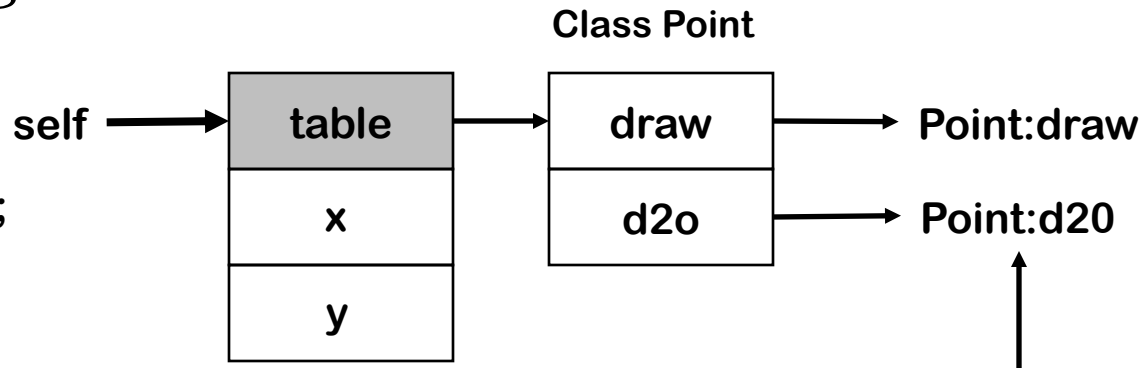
- Method can see instance variables of self, class, & superclasses
- Many different levels where a value can reside

In essence, OOL differs from ALL in the shape of its name space ***AND*** in the mechanism used to bind names to implementations

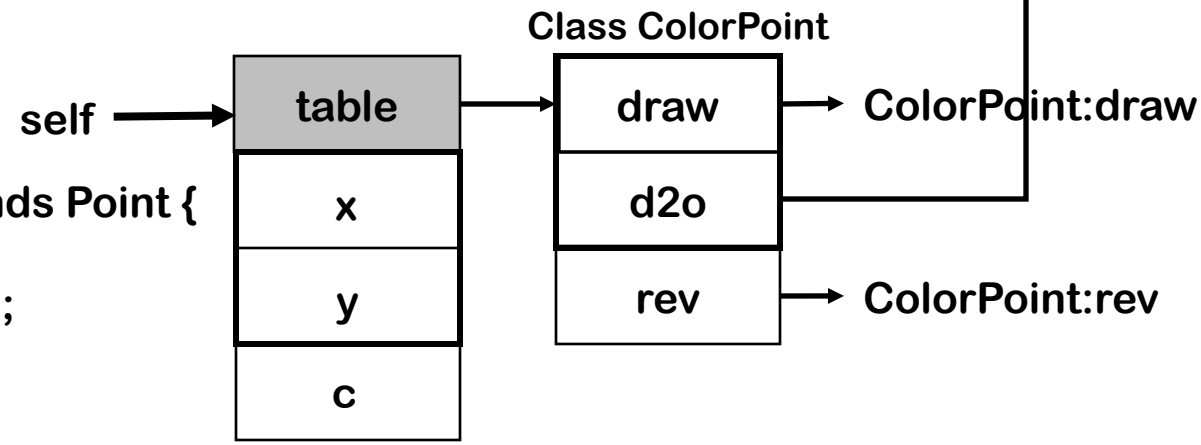
Single Inheritance & Dynamic Dispatch

- Use **prefixing** of tables

```
Class Point {
        int x, y;
        public void draw();
        public void d2o();
}
```




```
Class ColorPoint extends Point {
        Color c;
        public void draw();
        public void rev();
}
```



Multiple Inheritance

The Idea:

- Allow more flexible sharing of methods & attributes
- Relax the inclusion requirement

  If B is a subclass of A, it need not implement all of A's methods
- Need a linguistic mechanism for specifying partial inheritance

Problems when C inherits from both A & B

- C's method table can extend A or B, but not both
  - Layout of an object record for C becomes tricky
- Other classes, say D, can inherit from C & B
  - Adjustments to offsets become complex
- Say, both A & B might provide fum() — which is seen in C ?
  - C++ produces a "syntax error" when fum() is used

Multiple Inheritance Example

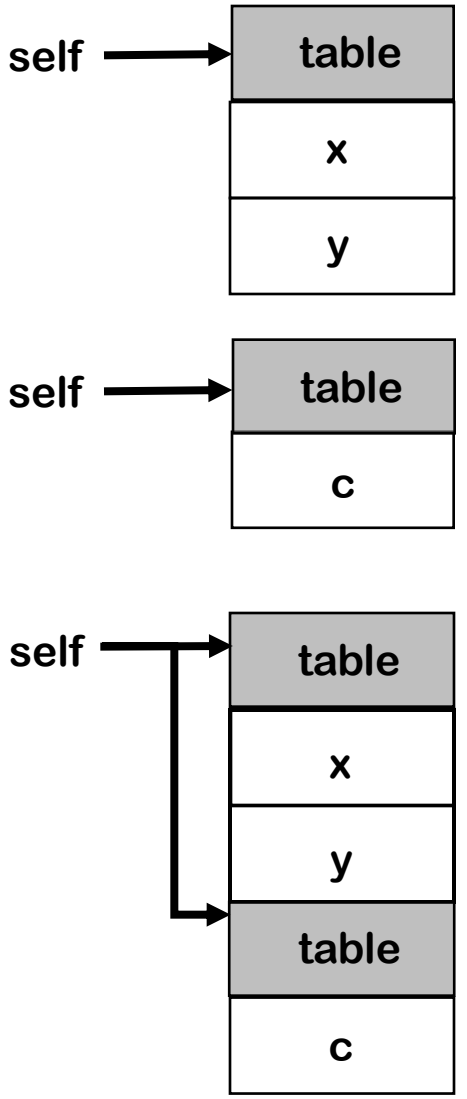
- Use **Prefixing** of Storage

```
Class Point {
        int x, y;
}

Class CThing {
        Color c;
}

Class CPoint extends  Point,  CThing {

}
```



Issue: does casting work properly?

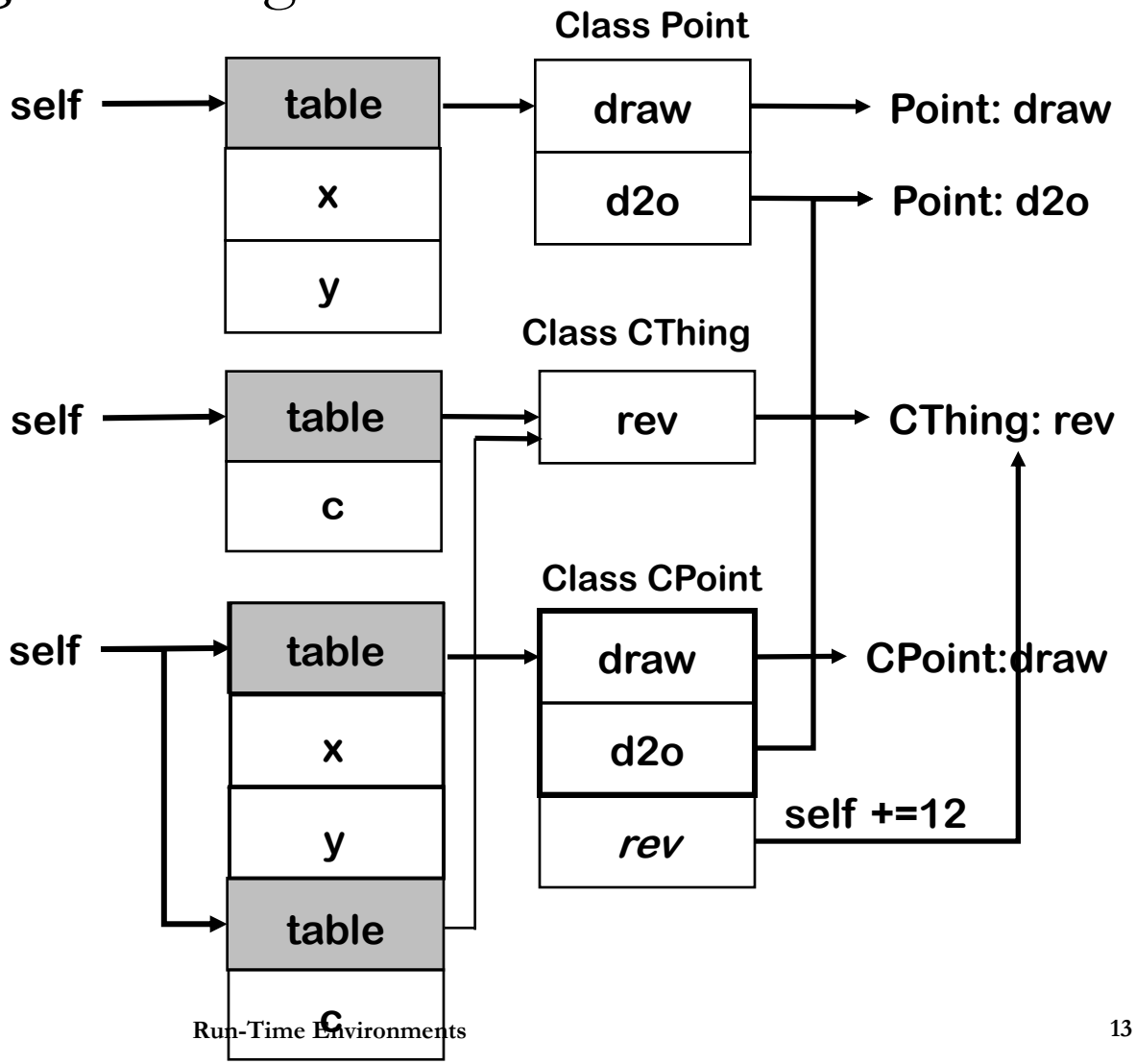
Multiple Inheritance Example

- Use **Prefixing** of Storage

```
Class Point {
        int x, y;
        void draw();
        void d2o();
}

Class CThing {
        Color c;
        void rev();
}

Class Cpoint extends
        Point, CThing {
        void draw()
}
```

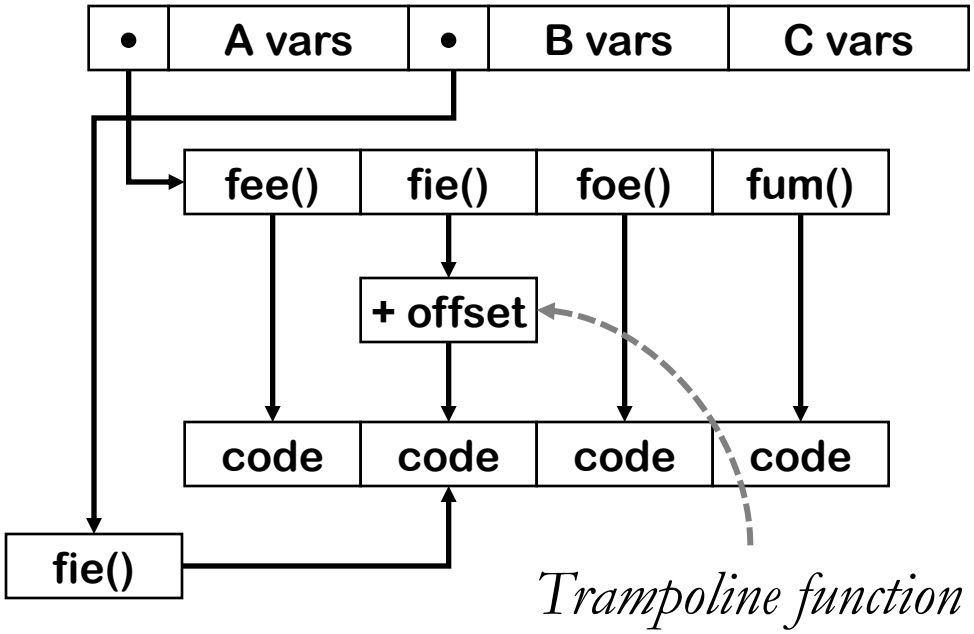


Casting with Multiple Inheritance

- Usage as Point:
  - No extra action (prefixing does everything)
- Usage as CThing:
  - Increment self by 12
- Usage as CPoint:
  - Lay out data for CThing at self + 12
  - When calling the rev method
    - Call in table points to a *trampoline* function that adds 12 to self, then calls rev
    - Ensures that rev, which assumes that self points to a CThing data area, gets the right data (at the correct layout offsets…)

Multiple Inheritance (Example)

Assume that C inherits fee() from A, fie() from B, & defines both foe() and fum()



This implementation

- Uses *trampoline* functions
- Optimizes well with inlining
- Adds overhead where needed (Zero offsets go away)
- Folds inheritance into data structure, rather than linkage

Assumes static class structure

For dynamic, why not rebuild on a change in structure?

Implementing Object-Oriented Lang.

So, what can an Executing Method see?

- The Object's own Attributes & Private Variables
- The Attributes & Private Variables of Classes that define it
  - *May be many such classes, in many combinations*
  - *Class variables are visible to methods that inherit the class*
- Object defined in the Global Name Space (or scope)
  - Objects may contain other objects
- Objects that contain their definition
  - *A class as an instance variable of another class, ...*

An Executing Method might reference any of these

Making this work requires compile-time elaboration for static case and run-time elaboration for dynamic case

Making it run quickly takes care, planning, and trickery…

Summary

- Support for Object-Oriented Languages:
  - Mostly focus on Message Dispatching
  - Finding the "correct" function in the Class Hierarchy
  - Adjusting Object Layout and Class References

- An Invocation Stack is still needed....