The Procedure Abstraction
Part V: Support for OOLs

COMP 412
Fall 2005

What about Object-Oriented Languages?

What is an OOL?

- A language that supports "object-oriented programming"

How does an OOL differ from an ALL? (ALGOL-Like Language)

- Data-centric name scopes for values & functions
- Dynamic resolution of names to their implementations

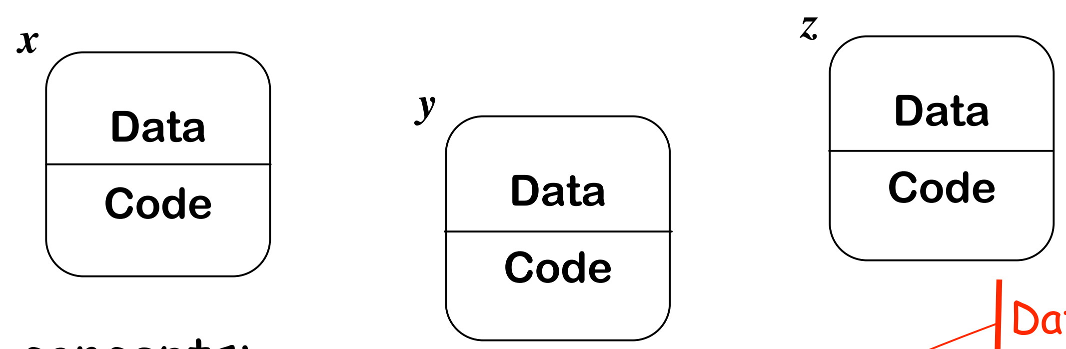
How do we compile OOLs ?

- Need to define what we mean by an OOL
- Term is almost meaningless today —
  - Smalltalk to C++ to Java
- We will focus on Java and C++
- Differences from an ALL lie naming and addressability

Object-Oriented Languages

An object is an abstract data type that encapsulates data, operations and internal state behind a simple, consistent interface.

The Concept:



Elaborating the concepts:

- Each object needs local storage for its internal state
  - Attributes are static (*lifetime of object*)
  - External access is through procedures
- Some methods are public, others are private
  - Locating a procedure by name is more complex than in an ALL
- Object's internal state leads to complex behavior

Data members, variables

Code members, methods

OOLs & the Procedure Abstraction

What is the shape of an OOL's name space?

- Local storage in objects (*beyond attributes*)
- Some storage associated with methods
  - Local values inside a method
  - Static values with lifetimes beyond methods
- Methods shared among multiple objects

In some OOLs, everything is an object.

In others, variables co-exist with objects & inside objects.

Classes

- Objects with the same ~~state~~ members are grouped into a *class*
  - Same code, same data, same naming environment
  - Class members are static & shared among instances of the class
- Allows abstraction-oriented naming
- Should foster code reuse in both source & implementation

Implementing Object-Oriented Languages

So, what can an executing method see?

- The object's own members
  - Smalltalk terminology: *instance variables*
- The members of the class that defines it
  - Smalltalk terminology: *class variables and methods*
- Any object defined in the global name space (*scope*)
  - Objects may contain other objects (!?!)

An executing method might reference any of these

An OOL resembles an ALL, with a wildly different name space

- Scoping is relative to hierarchy in the data of an OOL
- Scoping is relative to hierarchy in the code of an ALL

Implementing Object-Oriented Languages

So, what can an executing method see?

- The object's own members
  - Smalltalk terminology: *instance variables*
- The members of the class that defines it
  - Smalltalk terminology: *class variables and methods*
- Any object defined in the global name space (*scope*)
  - Objects may contain other objects (!?!)

An executing method might reference any of these

A final twist:

- Most OOLs support a hierarchical notion of inheritance
- Some OOLs support multiple inheritance
  - More than one path through the inheritance hierarchy

Java Name Space

Code within a method M for object O of class C can see:

- Local variables declared within M (*lexical scoping*)
- All instance variables & class variables of C
- All public and protected variables of any superclass of C
- Classes defined in the same package as C or in any explicitly imported package
  - public class variables and public instance variables of imported classes
  - package class and instance variables in the package containing C
- Class declarations can be nested!
  - These member declarations hide outer class declarations of the same name (*lexical scoping*)
  - Accessibility: public, private, protected, package

Java Name Spaces

We will use and extend this example

```
Class Point {
    public int x, y;
    public void draw();
}
Class ColorPoint extends Point {                // inherits x, y, & draw() from Point
    Color c;                                    // local data
    public void draw() {…}                      // override (hide) Point's draw
    public void test() { y = x; draw(); }       //  local code
}
Class C {                                       // independent of Point & ColorPoint
    int x, y;                                   // local data
    public void m()                             // local code
    {
        Point p = new ColorPoint();             // uses ColorPoint, and, by inheritance
        y = p.x;                                // the definitions from Point
        p.draw();
    }
}
```

Java Symbol Tables

To compile code in method M of object O with class C, the compiler needs:

- Lexically scoped symbol table for block and class nesting
  - Just like ALL — inner declarations hide outer declarations
- Chain of symbol tables for inheritance
  - Need mechanism to find the class and instance variables of all superclasses
- Symbol tables for all global classes (package scope)
  - Entries for all members with visibility
  - Need to construct symbol tables for imported packages and link them into the structure in appropriate places

Java Symbol Tables

To find the address associated with a variable reference in method M for an object O within a class C, the compiler must

- For an unqualified use (i.e., x):
  - Search the scoped symbol table for the current method
  - Search the chain of symbol tables for the class hierarchy
  - Search global symbol table (current package and imported)
    - Look for class (or interface)
  - In each case check visibility attribute of x
- For a qualified use (i.e.: Q.x):
  - Find Q by the method above
  - Search from Q for x
    - Must be a class or instance variable of Q or some class it extends
  - Check visibility attribute of x

Think back to "sheaf of tables" implementation

Implementing Object-Oriented Languages

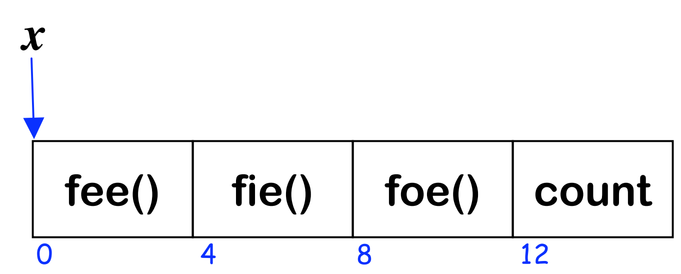
Two critical issues in OOL implementation:

- Object representation
- Mapping a method invocation name to a method implementation

These both are intimately related to the OOL's name space

Object Representation

- Static, private storage for attributes & instance variables
  - Heap allocate object records or "instances"
- Need consistent, fast access
  - Known, constant offsets from start
- Provision for initialization in NEW



OOL Storage Layout (Java)

Class variables

- Static class storage accessible by global name (*class C*)
  - Accessible via linkage symbol &_C or pointer chain from object
  - Nested classes are handled like blocks in ALLs
  - Method code put at fixed offset from start of class area

Object Representation

x

fee()	fie()	foe()	count
0	4	8	12

- Object storage is heap allocated
  - Fields at fixed offsets from start of object storage
- Methods
  - Code for methods is stored with the class
  - Methods accessed by offsets in class' code vector (or table)
    - Allows method references inline
  - Method local storage in object (no calls) or on stack

"leaf" routine in an ALL

AR as in an ALL

OOL Storage Layout

A minor problem

- The compiler must generate code for all these methods
  - Offsets must be consistent up and down the class hierarchy
  - If x is at offset 4 in an instance of Point, it must be at offset 4 in instances of classes that extend Point (e.g., ColorPoint)
- The compiler needs this consistency to generate code
  - Largely an issue of storage layout

OOL Storage Layout

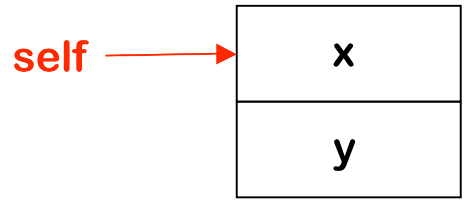
To map names into addresses at runtime

- An ALL converts the name into a static coordinate
  - Coordinate for variable turns into actions with runtime data structures that support addressability (*access links or display*)
  - Coordinate for procedure name turns into a relocatable mangled label for direct use in a load or jump
- Can we resolve names in an OOL with the same tricks?
  - Static coordinates?
  - Runtime links through AR s or a display-like structure
- Variable access must follow inheritance, not invocations
- Procedure calls are too frequent to allow excess overhead
  - Following a chain of pointers would be disastrously expensive
  - Reducing cost of “dispatch” is a key performance issue

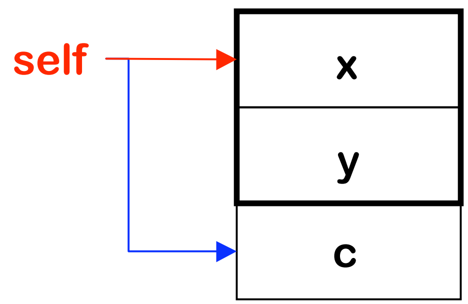
Variable Storage with Single Inheritance

- Use prefixing of storage

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




```
Class ColorPoint extends Point {
        Color c;
}
```



Does casting work properly?

Implementing Object-Oriented Languages

Mapping message names to methods

- Static mapping, known at compile-time (C++)
  - Fixed offsets & indirect calls
- Dynamic mapping, unknown until run-time (Smalltalk)
  - Lookup by textual name in class' table of methods
  - Walk up the (single) inheritance tree

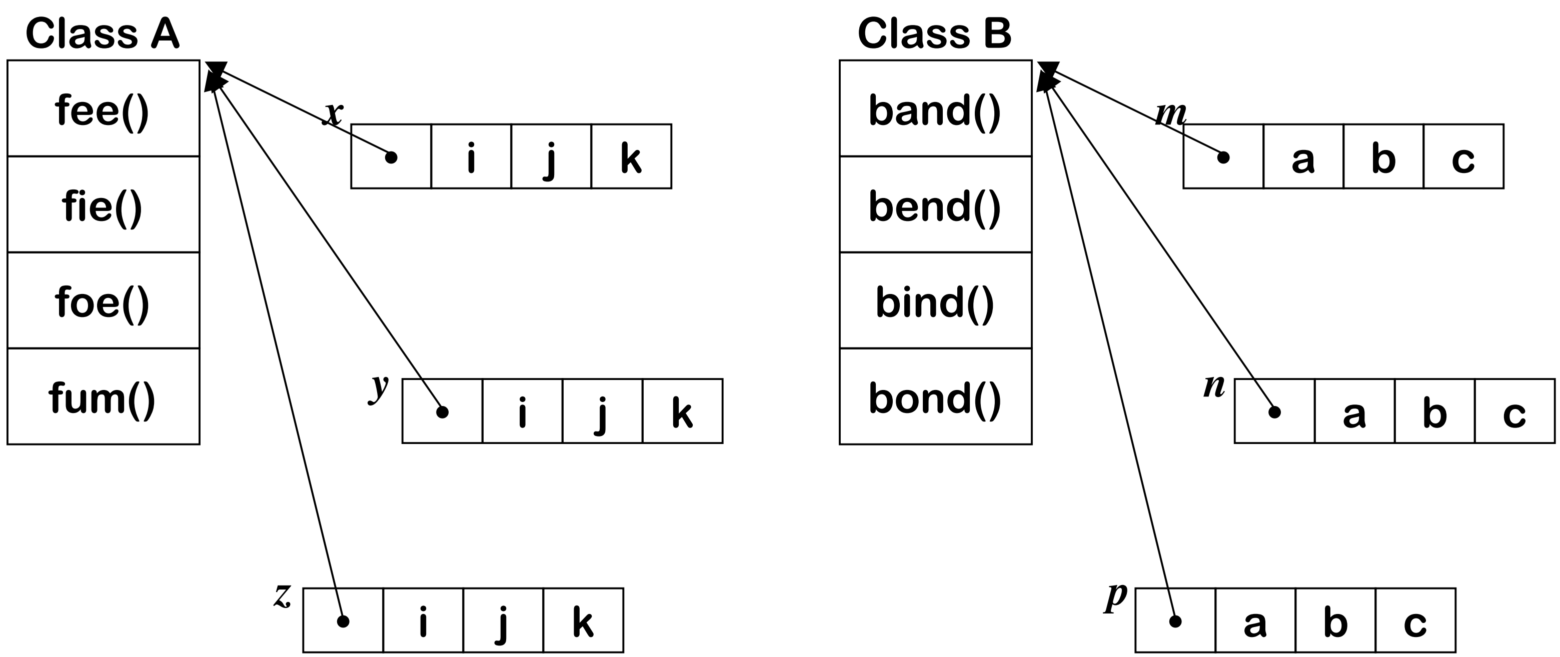
Want uniform placement of standard services ***(NEW, PRINT, …)***

This is really a data-structures problem

- Build a vector of function pointers (*code vector*)
- Use a standard calling sequence

Implementing Object-Oriented Languages

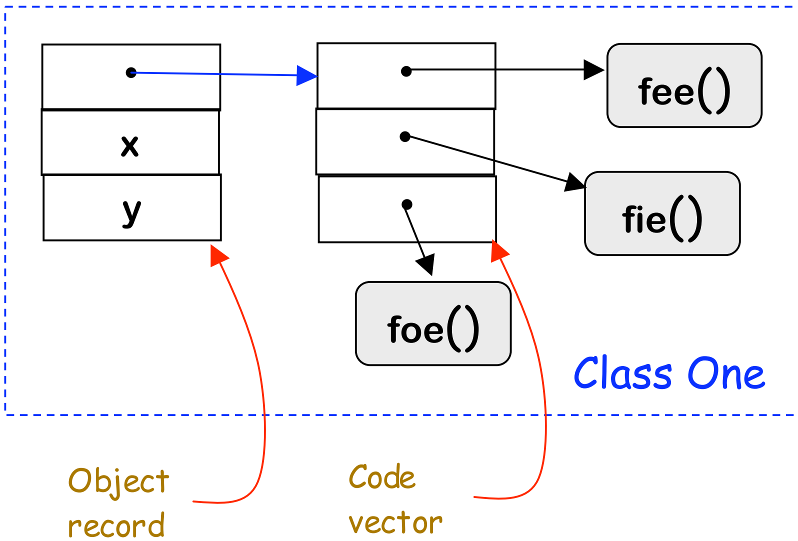
With static, compile-time mapped classes (*no inheritance*)



Message dispatch becomes an indirect call through a function table

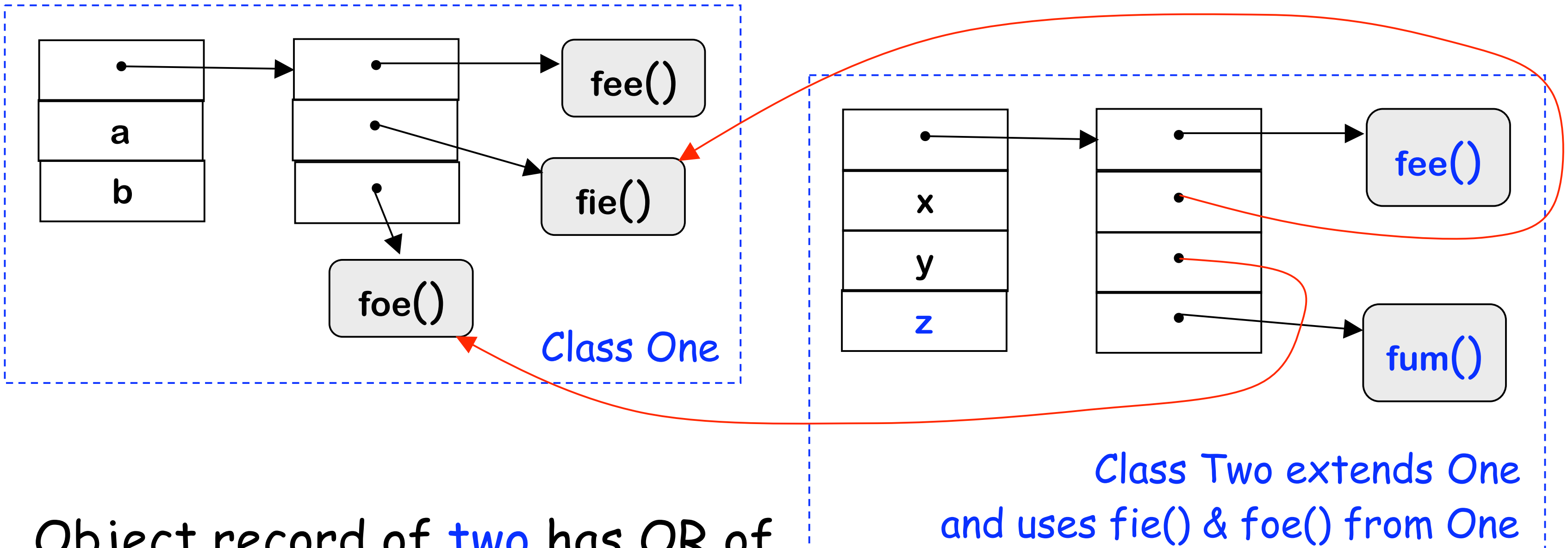
The Single Inheritance Hierarchy

The Concept:



The Single Inheritance Hierarchy

The Concept of Single Inheritance:



Object record of two has OR of one as its prefix

Code vector of two has CV of one as its prefix

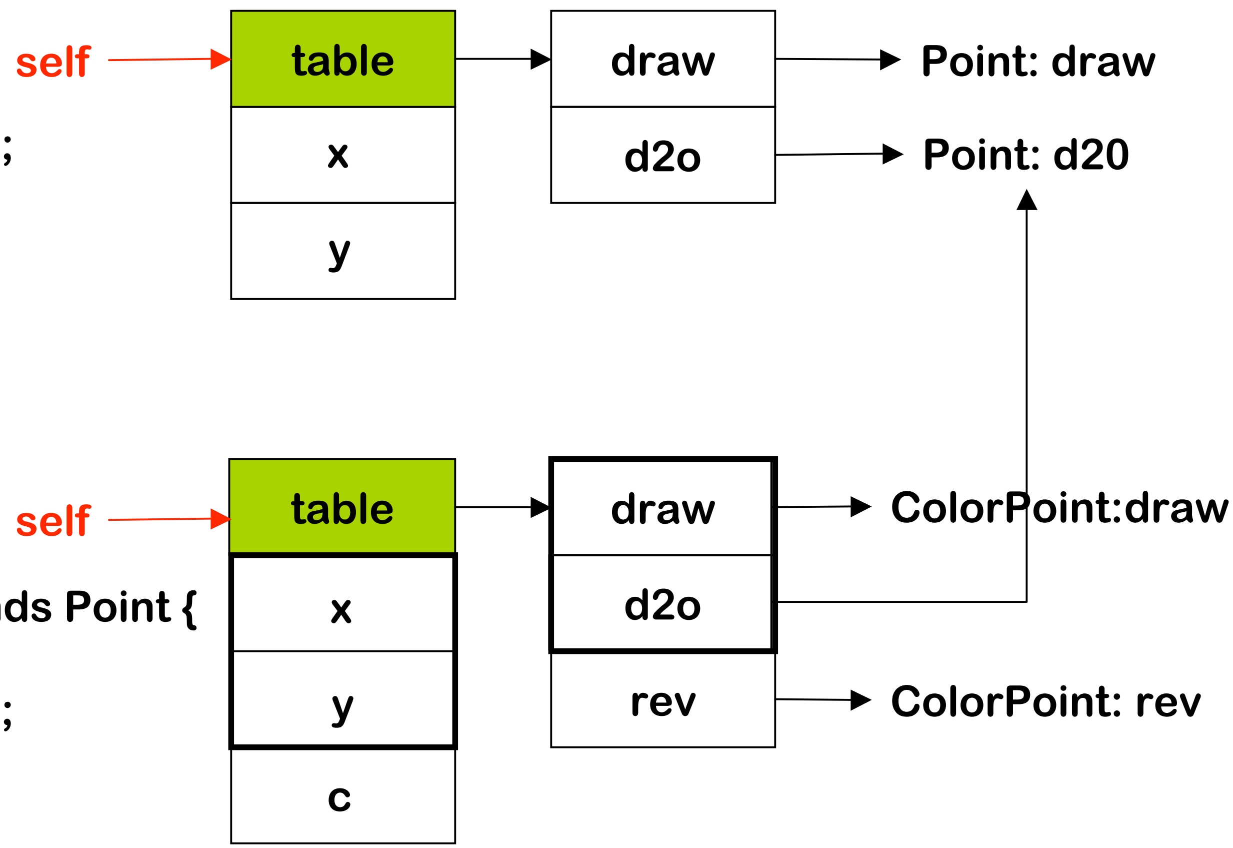
– Direct references to code bodies defined for one

Dynamic Dispatch with Single Inheritance

- To handle inheritance, prefix the code vectors. too

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

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



The Single Inheritance Hierarchy

Two distinct philos

Runtime changes (Java)

- Variable storage by prefixes
- Limit changes in instantiated classes
- Use 1-level code vectors and rebuild on hierarchy change
- Fixed offsets & 1-level of indirection on calls

Static class struc

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

structure

- ame to code me
- o vectors (1/class)
- 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

STOP

Multiple Inheritance

The idea

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

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

Problems when C inherits from both A & B

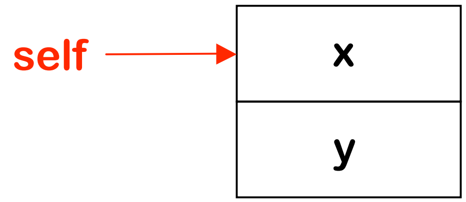
- C's method table can extend A or B, but not both
  - Layout of an object record for C becomes tricky
- Other classes, say D, can inherit from C & B
  - Adjustments to offsets become complex
- Both A & B might provide fum() — which is seen in C ?
  - C++ produces a "syntax error" when fum() is used

Need a better way to say "inherit"

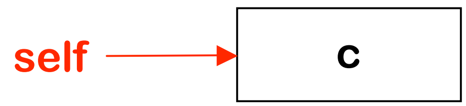
Variable Storage with Multiple Inheritance

- Use prefixing of storage

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

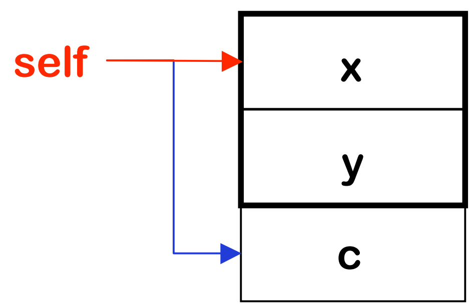



```
Class ColoredThing {
        Color c;
}
```




```
Class ColorPoint extends
        Point,  ColoredThing {

}
```



Does casting work properly?

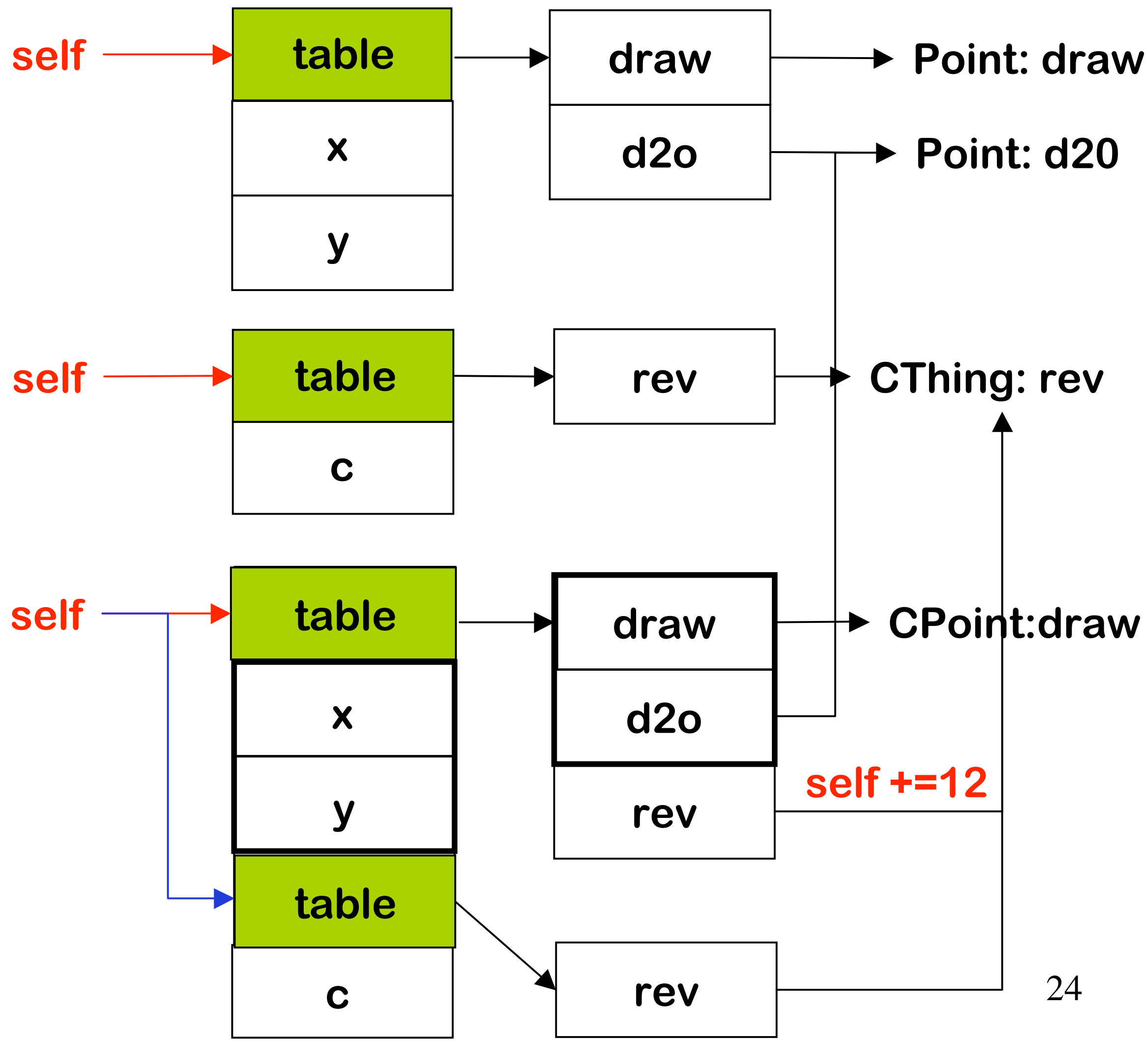
Multiple Inheritance Example

- Use prefixing of storage & code vectors

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

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

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



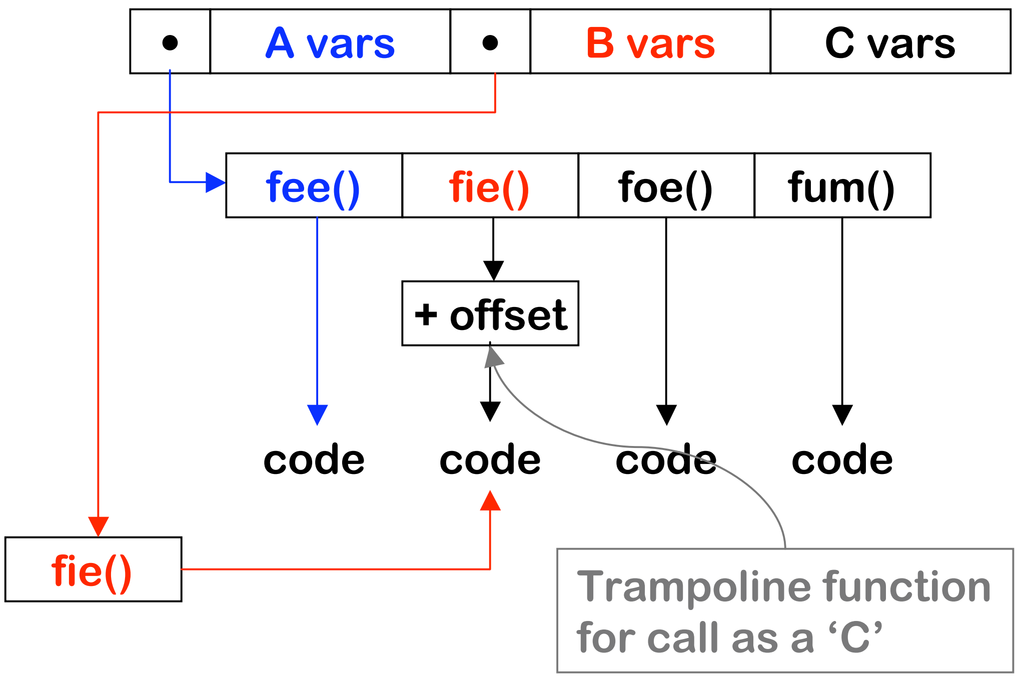
Casting with Multiple Inheritance

- 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 + 16
  - When calling rev
    - 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

Multiple Inheritance (Example)

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

Object record for an instance of C



This implementation

- Uses a trampoline function
- Optimizes well with inlining
- Overhead only incurred where it is really necessary
- Folds inheritance into data structure, rather than linkage

Assumes a static class structure

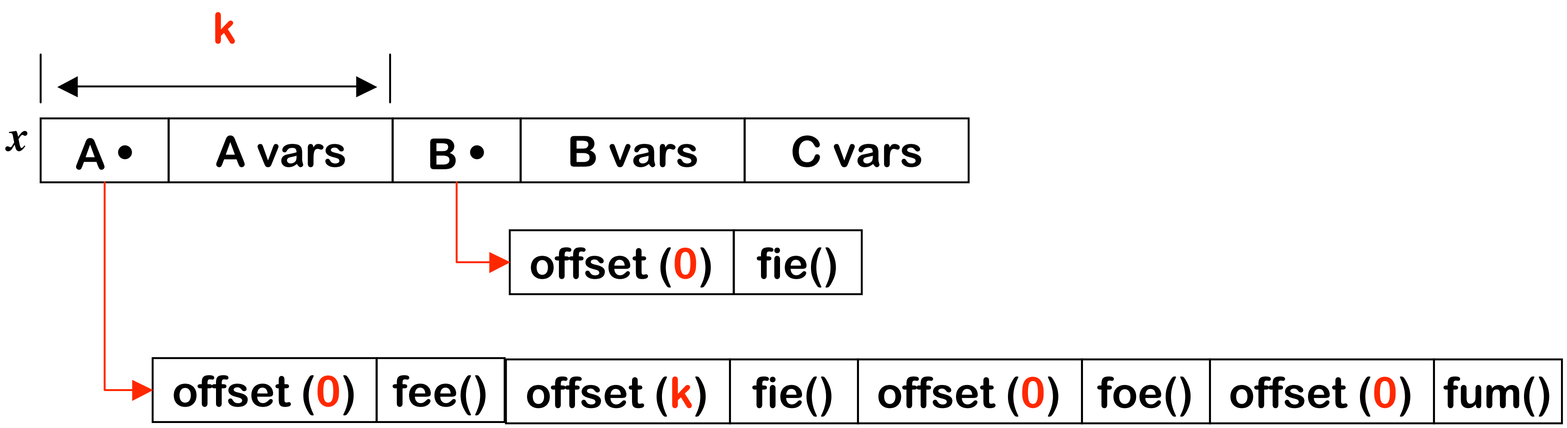
Rebuild it on a change in the inheritance hierarchy

Extra Slides Start Here

Multiple Inheritance with Offsets (Example)

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

Object record for an instance of C



To make this work, calls must add offset to self

Works, but adds overhead to each method invocation