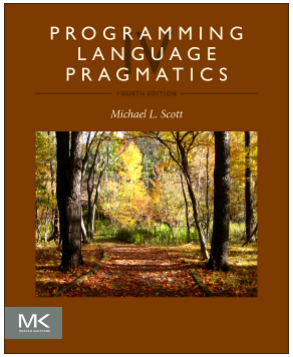
Implementation of Objects

17-363/17-663: Programming Language Pragmatics



Reading: PLP chapter 10

Acknowledgments: some presentation ideas
from Craig Chambers

HW 6 Thoughts

- The main challenge in HW6 is probably just writing tree traversals in OCaml
- We assigned a checkpoint (due Thursday, October 26) to make sure you get started
- The checkpoint is a small portion of the overall work but we hope it will help you get over this "hump."

HW 6 Mystery Explained

- When we studied composite types, we learned that records have two subtyping rules:

$$\frac{\overline{\tau} \leq \overline{\tau'}}{\{\, \overline{f : \tau} \,\} \leq \{\, \overline{f : \tau'} \,\}}\ \textit{S-depth}$$

$$\frac{}{\{\, \overline{f : \tau}, \overline{g : \tau'} \,\} \leq \{\, \overline{f : \tau} \,\}}\ \textit{S-width}$$

- But in the μTS specification, there is only the *S-width* rule. Why?

HW 6 Mystery Explained

- μTS has no *S-width* rule. Why?

$$\frac{\overline{\tau} \leq \overline{\tau'}}{\{\ \overline{f : \tau}\ \} \leq \{\ \overline{f : \tau'}\ \}}\ \textit{S-depth}$$

$$\frac{}{\{\ \overline{f : \tau}, \overline{g : \tau'}\ \} \leq \{\ \overline{f : \tau}\ \}}\ \textit{S-width}$$

- μTS interfaces are a combination of 3 things:
  - Records – *because there are several fields with names*
  - Recursive types – *because the interface type can be used in its own definition*
  - Pointer types – *because the fields are mutable*
    - Remember – $\tau_1 * \leq \tau_2 *$ only if $\tau_1 = \tau_2$
  - Fun fact: TypeScript interfaces are tagged unions too!
    - But not in the μTS language you are implementing

HW 6 Mystery Explained

- μTS has no *S-width* rule. Why?

$$\frac{\overline{\tau} \leq \overline{\tau'}}{\{\ \overline{f : \tau}\ \} \leq \{\ \overline{f : \tau'}\ \}}\ \textit{S-depth}$$

$$\frac{}{\{\ \overline{f : \tau}, \overline{g : \tau'}\ \} \leq \{\ \overline{f : \tau}\ \}}\ \textit{S-width}$$

- Our μTS rule is similar to the rules of Typescript
- Interestingly, Flow does support depth subtyping!
  - Flow is a different type system for JavaScript
  - Why? Flow lets you designate some fields as immutable
    - Can't write to those fields after initialization
    - Depth subtyping applies *only* to immutable fields
      - These fields are still implemented with pointers, but don't have to follow the invariant subtyping rules that pointers do

Object-Oriented Programming

- Analogy to the real world is central to the OO paradigm - you think in terms of *real-world* objects that interact to get things done
- Many OO languages are strictly sequential, but the model adapts well to parallelism as well
- Strict interpretation of the term
  - uniform data abstraction - everything is an object
  - inheritance
  - dynamic method binding

Object-Oriented Programming

- Lots of conflicting uses of the term out there object-oriented ***style*** available in many languages
  - data abstraction crucial
  - inheritance required by most users of the term OO
  - centrality of dynamic method binding a matter of dispute

Object-Oriented Programming

- SMALLTALK is, historically, the canonical object-oriented language
  - It has all three of the characteristics listed above
  - It's based on the thesis work of Alan Kay at Utah in the late 1960's
  - It went through 5 generations at Xerox PARC, where Kay worked after graduating
  - Smalltalk-80 is the current standard

Object-Oriented Programming

- Modula-3
  - single inheritance
  - all methods virtual
  - no constructors or destructors
- Java, C#
  - interfaces, *mix-in* inheritance
  - all methods virtual
- Scala
  - Multi-paradigm, classes, functions, traits
- JavaScript
  - Prototype-based, dynamically typed

Object-Oriented Programming

- Ada 95
  - –*tagged* types
  - –single inheritance
  - –no constructors or destructors
  - –*class-wide* parameters:
    - •methods static by default
    - •can define a parameter or pointer that grabs the object-specific version of all methods
      - –base class doesn't have to decide what will be virtual
  - –notion of child packages as an alternative to friends

Object-Oriented Programming

- Is C++ object-oriented?
  - Uses all the right buzzwords
  - Has (multiple) inheritance and generics (templates)
  - Allows creation of user-defined classes that look just like built-in ones
  - Has all the low-level C stuff to escape the paradigm
  - Has friends
  - Has static type checking

Object-Oriented Programming

- In the same category of questions:
  - Is Prolog a logic language?
  - Is Common Lisp functional?
- However, to be more precise:
  - Smalltalk is really pretty purely object-oriented
  - Prolog is primarily logic-based
  - Common Lisp is largely functional
  - C++ can be used in an object-oriented style

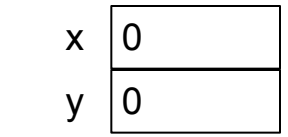
Object Models

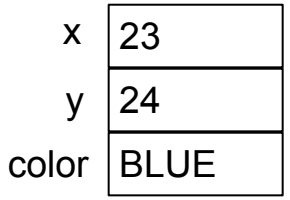
- An *object model* denotes the data and metadata representation used by a language implementation
- Tradeoffs in implementing object models:
  - Complexity
  - Performance
  - Memory usage
- Common features
  - An object is usually (at least one) contiguous block of memory
    - Sometimes several related blocks are used
  - Objects usually needs metadata– a “tag” or “header”
    - More information if more dynamic, has reflection, or is garbage collected
- We’ll start with object models for statically typed single-inheritance OO languages like Java and C#

Prefixing - Implementing Inheritance

- Prefixing: layout of subclass has layout of superclass as a prefix

```
class Point {
   int x;
   int y;
}
class ColorPoint extends Point {
  Color color;
}
```






```
// OK, ColorPoint is a subtype of Point
Point p = new ColorPoint(0, 1, green);
// subclasses of Point have x and y in the same place
int manhattanDistance = p.x + p.y;
```

Example due to Craig Chambers

Implementing Method Calls

Possible Strategies

1. Each object knows its type; search the inheritance hierarchy
   - Very slow

2. Use a hashtable
   - Can be a cache for strategy #1
   - Still slow, but was used in early Smalltalk systems

3. Store function pointers in objects, as if they were fields
   - Invocation is fast & constant time: load and indirect jump
   - Con: objects are big!

   - Observation: in this strategy, all objects of the same class will store the same function pointers. Can we factor them out?

Virtual Method Tables (vtables)

```
class foo {
    int a;
    double b;
    char c;
public:
    virtual void k( ...
    virtual int l( ...
    virtual void m();
    virtual double n( ...
    ...
} F;
```

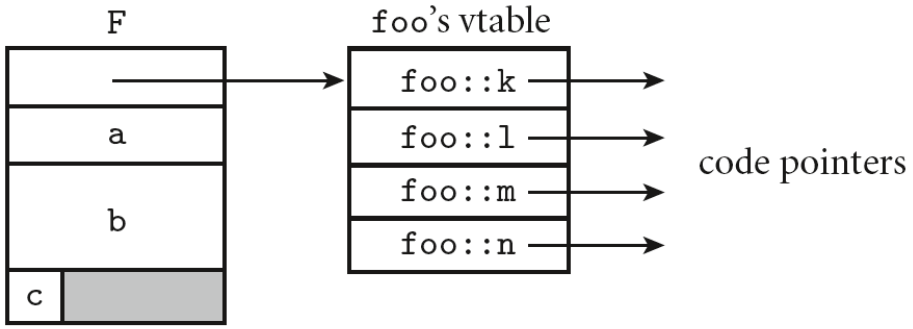


Figure 10.3 **Implementation of virtual methods.** The representation of object `F` begins with the address of the vtable for class `foo`. (All objects of this class will point to the same vtable.) The vtable itself consists of an array of addresses, one for the code of each virtual method of the class. The remainder of `F` consists of the representations of its fields.

- The assembly pseudocode generated for `f->m()` is:

```
r1 := f
r2 := *r1                    -- vtable address
r2 := *(r2 + (3-1) × 4) -- assuming 4=sizeof(address)
call *r2
```

Method Overriding

```
class bar : public foo {
    int w;
public:
    void m() override;
    virtual double s( ...
    virtual char *t( ...
    ...
} B;
```

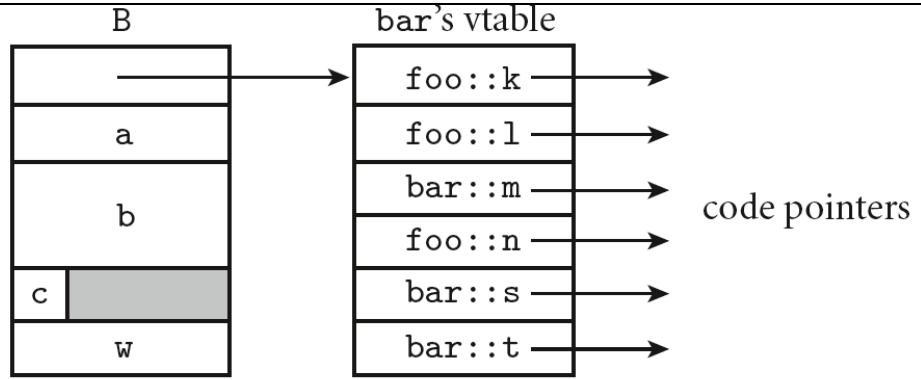


Figure 10.4 **Implementation of single inheritance.** As in Figure 10.3, the representation of object `B` begins with the address of its class's vtable. The first four entries in the table represent the same members as they do for `foo`, except that one—`m`—has been overridden and now contains the address of the code for a different subroutine. Additional fields of `bar` follow the ones inherited from `foo` in the representation of `B`; additional virtual methods follow the ones inherited from `foo` in the vtable of class `bar`.

Dynamic Type Casts

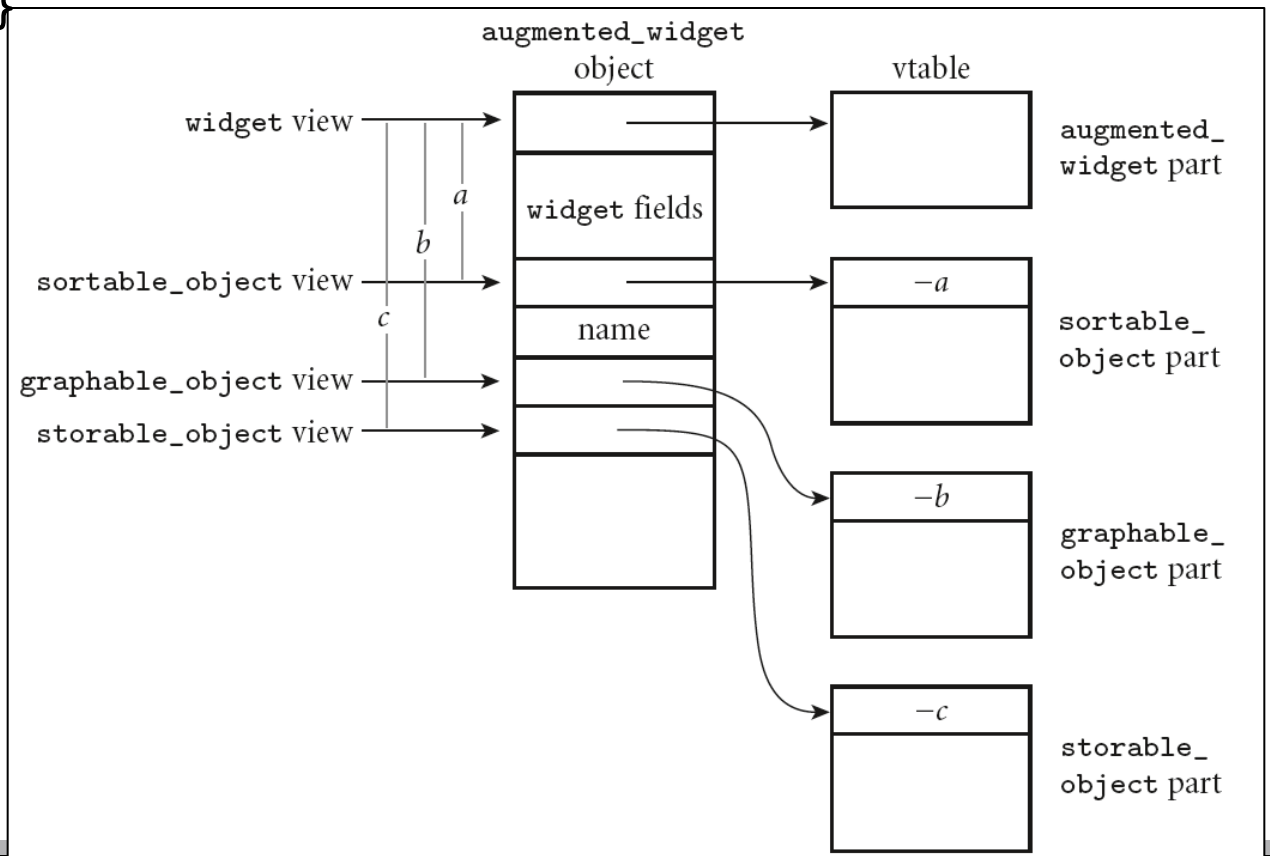
- Note that if you can query the type of an object, then you need to be able to get from the object to run-time type info
  - The standard implementation technique is a type info at the beginning of the vtable
  - In C++, the class only has a vtable if the class has virtual functions
    - That's why dynamic_cast is disallowed on a pointer whose static type doesn't have virtual functions
  - Other approaches: intervals, Cohen display

Implementing Methods: `this`

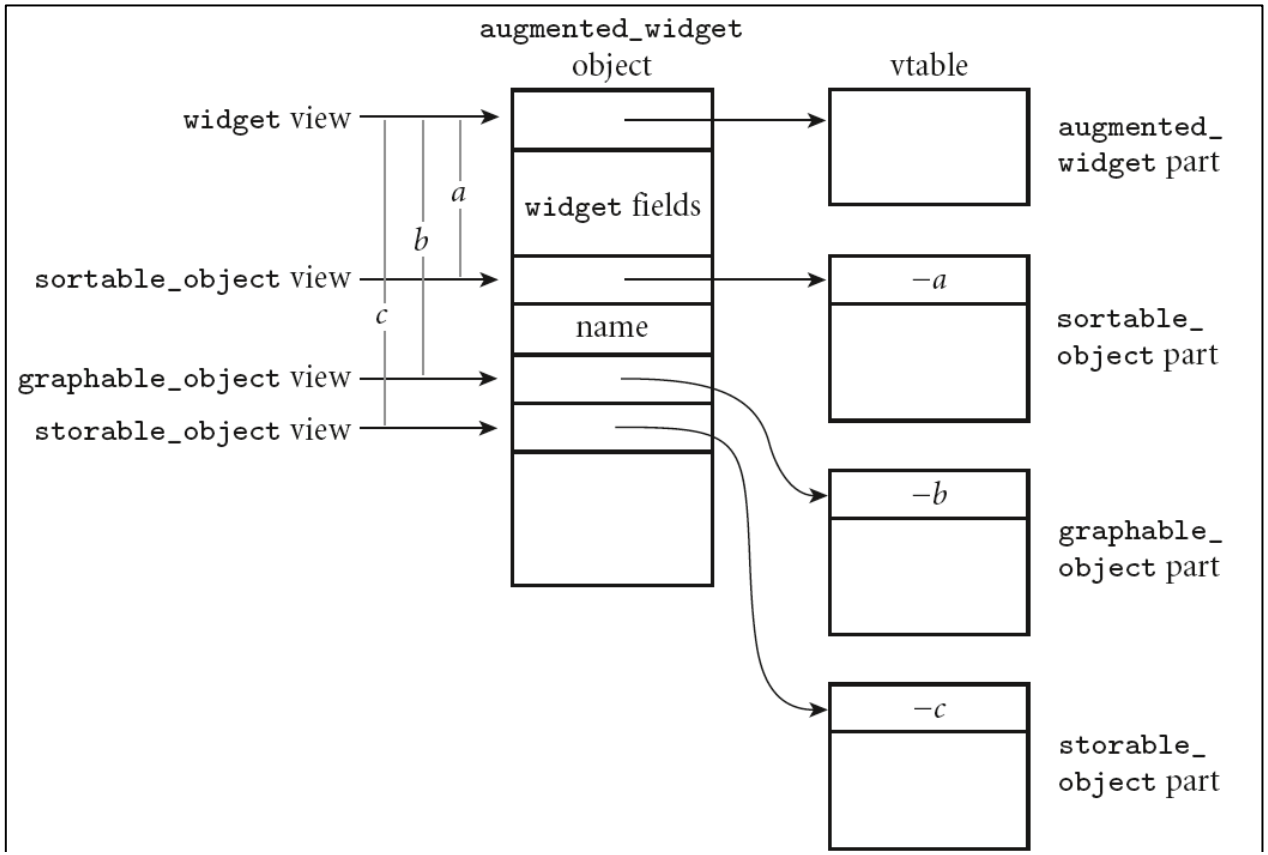
- Methods are passed an extra, hidden, initial parameter: *this* (called *self* in Smalltalk and some other languages)
  - Allows the method to access the fields of the object and call other methods
  - Usually a pointer to the start of the object storage in memory

Multiple Interface Inheritance

```
class widget { ... }
class named_widget       extends widget
      implements sortable_object { ... }
class augmented_widget extends named_widget
      implements graphable_object, storable_object
{ ... }
```



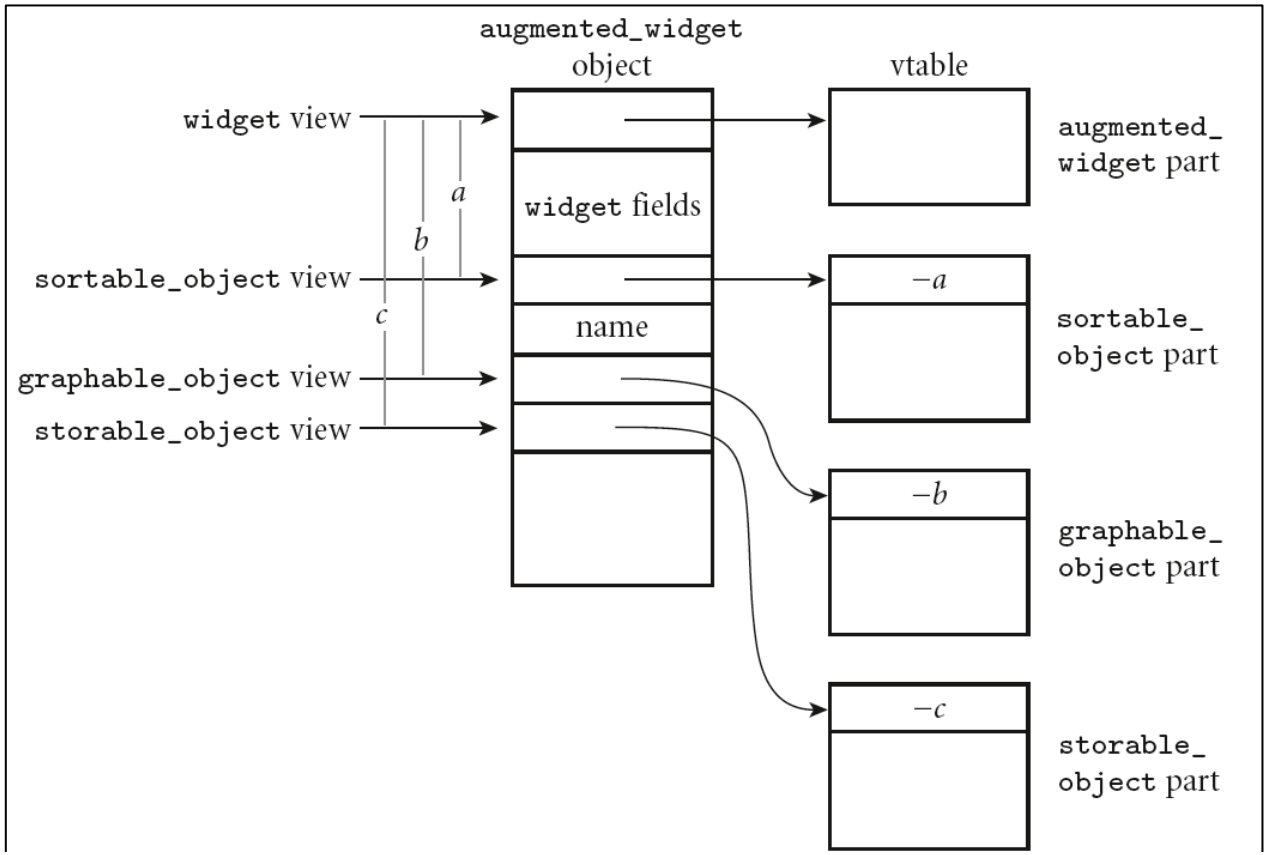
Multiple Interface Inheritance



- Consider a cast from `augmented_widget` to `sortable_object`:

```
r2 := r1 + a
```

Multiple Interface Inheritance



- Consider a call to an interface method of `sortable_object`

```
r2 := *r1          -- vtable address
r3 := *r2          -- this correction
r3 += r1           -- add correction to old address
call *(r2 + 4)     -- call (assumes first method in
```

Object model practice

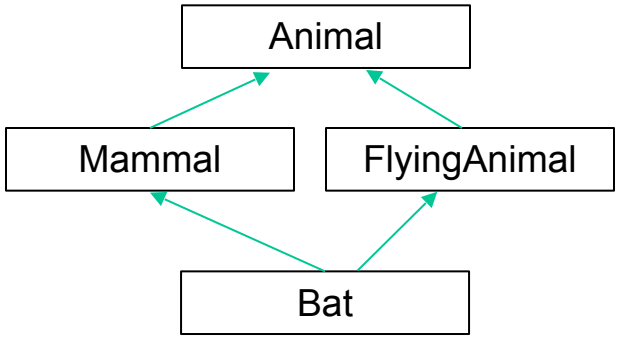
- Draw the layout of the object created at the end of this code. Show all virtual function tables.

```
interface Pingable {
  public void ping();
}
class Counter implements Pingable {
  int count = 0;
  public void ping() {
    ++count;
  }
  public int val() {
    return count;
  }
}

Counter c = new Counter();
```

Real Multiple Inheritance

Two approaches:



- "non-virtual inheritance" – A C++ hack
  - Just include state from both inherited classes
  - Works like multiple interface inheritance
  - If there's a diamond in the hierarchy, you get some fields twice
    - Good luck fixing bugs if the duplicate fields have inconsistent values!
  - Fast, simple, and works if there are no diamonds, or if the diamond classes have no state

- The right way (C++ virtual inheritance)
  - Essentially treat fields like methods – look up their location in a vtable
  - Slower, but has reasonable semantics

JavaScript's Object Model

- Each object has multiple dynamically-typed properties
  - Indexed by strings
  - Can be added or deleted dynamically
  - When a property is not found, the object's *prototype* is consulted
    - The prototype is the value of the property __proto__
      - This property can be a mutable object!

- The vtable strategy doesn't apply!
- Instead, start with a map from property name to value
  - Implemented as a list of pairs, or a hashtable
  - Slow!

Optimizing JavaScript

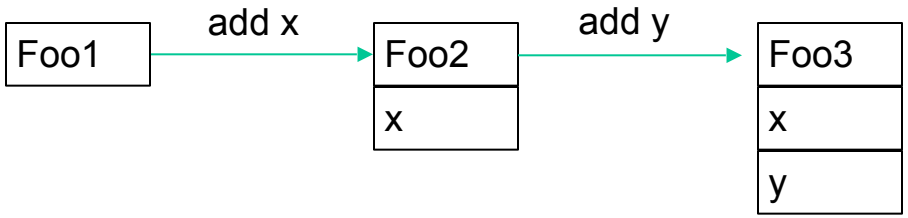
- Start with a map from property name to value
  - Implemented as a list of pairs, or a hashtable
  - Slow!

- Observation: most objects fall into one of a few “shapes”
- Used “hidden classes” (aka “shapes” or “maps”)
  - Every object has a pointer to an immutable map describing object’s properties
  - No need for a hashtable for most objects
  - Adding or removing a property changes the pointer to the map

Hidden Classes

- Hidden classes form a tree with transitions
- Example:

```
function Foo(x, y) {
      this.x = x;
      this.y = y;
}
var x = new Foo(33, 44);
```



- Each time a property is added, the hidden class is updated
- Deleting a property in LIFO order reverses the process
- Delete a different property?
  - Typically go to hashtable strategy (known as “dictionary mode” in V8)
    - otherwise too many hidden classes are generated

Inline Caches

- Consider looking up field x in the statement:

```
var f = o.x;
```

- An *inline cache* stores K entries, where an entry can be of the form:

  entry = {shape, offset}

- The access searches through the entries, looking for a matching shape
  - The hashtable is a backup

- Code for the inline cache access looks like:

```
lookup(o: Object, ic: InlineCache, propertyName: string)
{
    for (i = 0; i < K; i++) {
      if (o.shape == ic.entries[i].shape)
        return o.properties[ic.entries[i].offset];
    }
    // ic might be updated in this call
    return o.hashtable.lookup(propertyName, ic);
}
```

Mix-In Inheritance

- Classes can inherit from only one “real” parent
- Can “mix in” any number of interfaces, simulating multiple inheritance
- Interfaces appear in Java, C#, Go, Ruby, etc.
  - contain only abstract methods, no method bodies or fields
- Has become dominant approach, superseding true multiple inheritance

True Multiple Inheritance

- In C++, you can say

```
class professor : public
teacher, public researcher {
            ...
      }
```

  Here you get all the members of teacher ***and*** all the members of researcher
  - If there's anything that's in both (same name and argument types), then calls to the member are ambiguous; the compiler disallows them

True Multiple Inheritance

- You can of course create your own member in the merged class

```
professor::print () {
    teacher::print ();
    researcher::print (); ...
}
```

Or you could get both:

```
professor::tprint () {
    teacher::print ();
}
professor::rprint () {
    researcher::print ();
}
```

True Multiple Inheritance

- Virtual base classes: In the usual case if you inherit from two classes that are both derived from some other class B, your implementation includes two copies of B's data members
- That's often fine, but other times you want a ***single*** copy of B
  - For that you make B a virtual base class