

CIS 341: Compilers

Lecture 23

The Plan

- Today:
 - Continue with objects & compilation strategies
- Project 5: Full OAT compiler: objects and extended typechecking
 - Available online soon
 - Due: November 7th at 11:59 pm

Multiple Inheritance

- C++: a class may declare more than one superclass.

- Semantic problem: Ambiguity

```
class A { int m(); }  
class B { int m(); }  
class C extends A,B {...} // which m?
```

- Same problem can happen with fields.
- In C++, fields and methods can be duplicated when such ambiguity arises (though explicit sharing can be declared too)

- Java: a class may implement more than one interface.

- No semantic ambiguity: if two interfaces contain the same method declaration, then the class will implement a single method

```
interface A { int m(); }  
interface B { int m(); }  
class C implements A,B {int m() {...}} // only one m
```

General Approaches

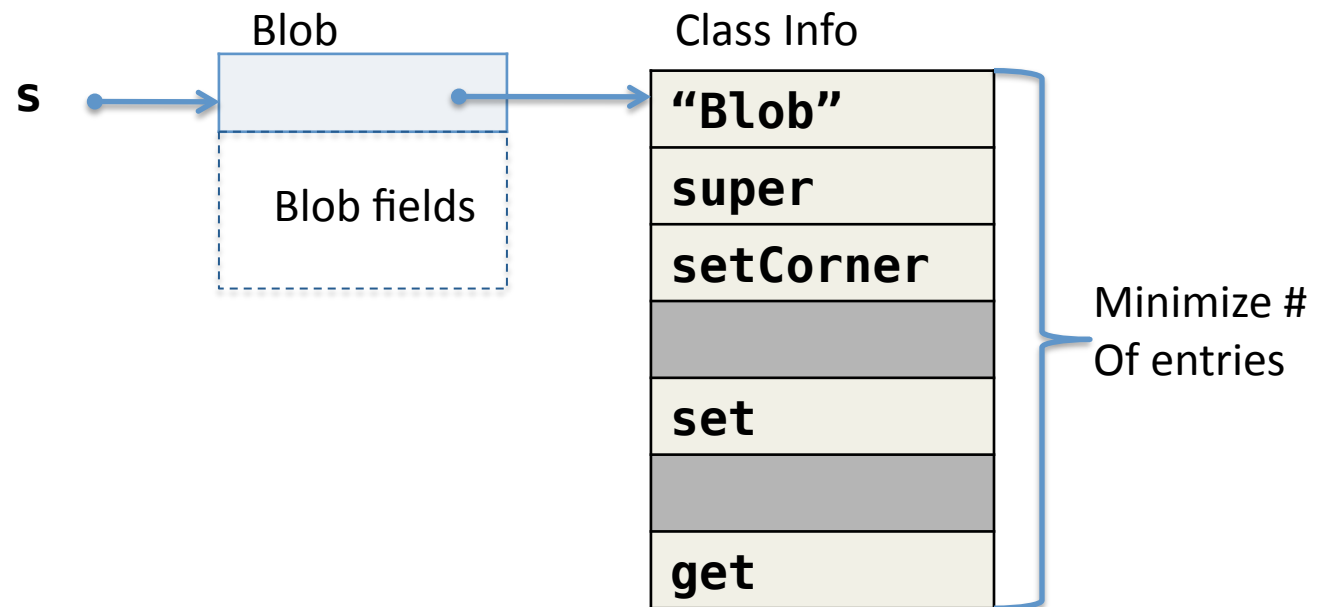
- Can't directly identify methods by position anymore.
- Option 1: Use a level of indirection:
 - Map method identifiers to code pointers (e.g. index by method name)
 - Use a hash table
 - May need to do search up the class hierarchy
- Option 2: Give up separate compilation
 - Use “sparse” dispatch vectors, or binary decision trees
 - Must know then entire class hierarchy
- Option 3: Allow multiple D.V. tables (C++)
 - Choose which D.V. to use based on static type
 - Casting from/to a class may require run-time operations
- Note: many variations on these themes
 - Different Java compilers pick different approaches...

Option 2: Sparse D.V. Tables

- Give up on separate compilation...
- Now we have access to the whole class hierarchy.
- So: ensure that no two methods in the same class are allocated the same D.V. offset.
 - Allow holes in the D.V. just like the hash table solution
 - Unlike hash table, there is never a conflict!
- Compiler needs to construct the method indices
 - Graph coloring techniques can be used to construct the D.V. layouts in a reasonably efficient way (to minimize size)
 - Finding an optimal solution is NP complete!

Example Object Layout

- Advantage: Identical dispatch and performance to single-inheritance case
- Disadvantage: Must know entire class hierarchy



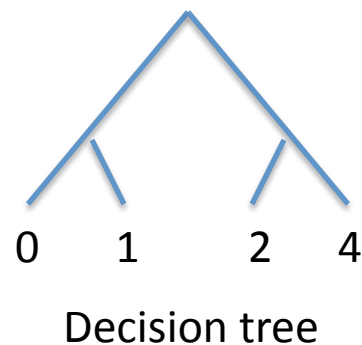
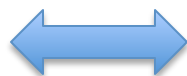
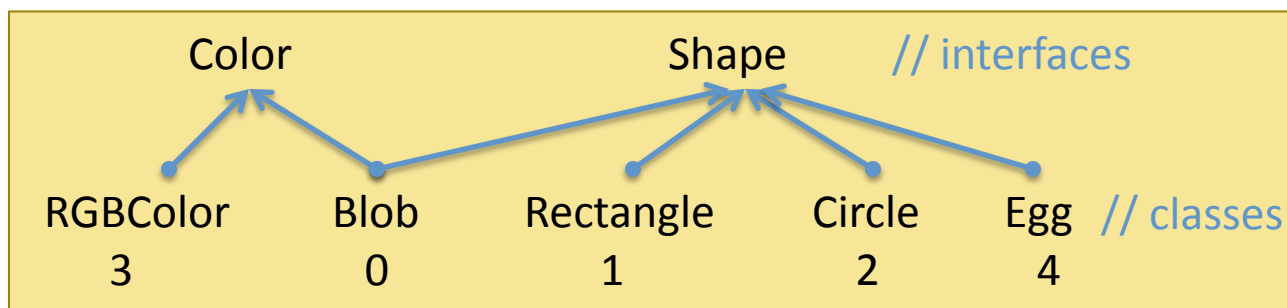
Option 2: Binary Search Trees

- Idea: Use conditional branches not indirect jumps
- Each object has a class index (unique per class) as first word
 - Instead of D.V. pointer (no need for one!)
- Method invocation uses range tests to select among n possible classes in $\lg n$ time
 - Direct branches to code at the leaves.

```
Shape x;  
x.SetCorner(...);
```



```
Mov eax, [[x]]  
Mov ebx, [eax]  
Cmp ebx, 1  
Jle __L1  
Cmp ebx, 2  
Je __CircleSetCorner  
Jmp __EggSetCorner  
__L1:  
Cmp ebx, 0  
Je __BlobSetCorner  
Jmp __RectangleSetCorner
```



Search Tree Tradeoffs

- Binary decision trees work well if the distribution of classes that may appear at a call site is skewed.
 - Branch prediction hardware eliminates the branch stall of ~10 cycles (on X86)
- Can use profiling to find the common paths for each call site individually
 - Put the common case at the top of the decision tree (so less search)
 - 90%/10% rule of thumb: 90% of the invocations at a call site go to the same class
- Drawbacks:
 - Like sparse D.V.'s you need the whole class hierarchy to know how many leaves you need in the search tree.
 - Indirect jumps can have better performance if there are >2 classes (at most one mispredict)

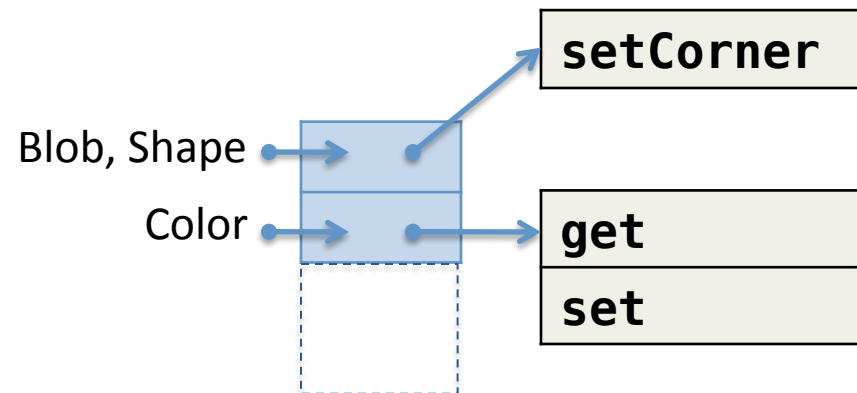
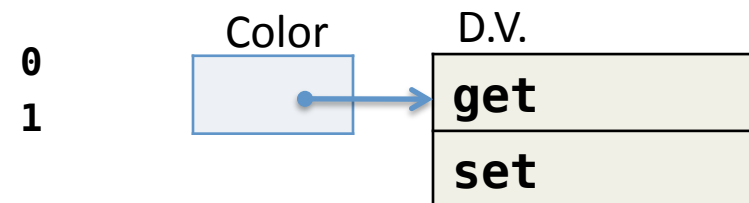
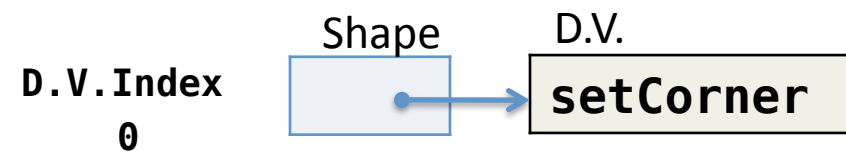
Option 3: Multiple Dispatch Vectors

- Duplicate the D.V. pointers in the object representation.
- Static type of the object determines which D.V. is used.

```
interface Shape {  
    void setCorner(int w, Point p);  
}
```

```
interface Color {  
    float get(int rgb);  
    void set(int rgb, float value);  
}
```

```
class Blob implements Shape, Color {  
    void setCorner(int w, Point p) {...}  
    float get(int rgb) {...}  
    void set(int rgb, float value) {...}  
}
```

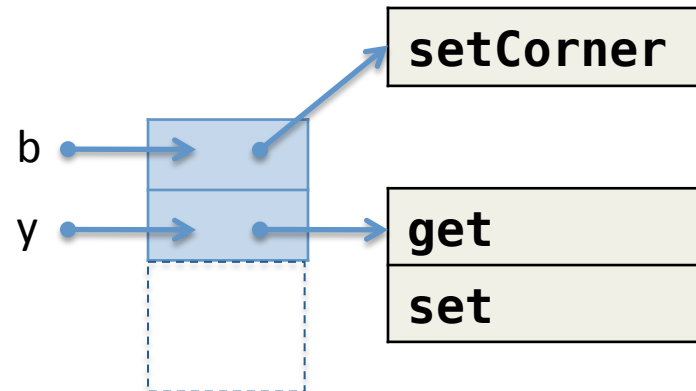


Multiple Dispatch Vectors

- A reference to an object might have multiple “entry points”
 - Each entry point corresponds to a dispatch vector
 - Which one is used depends on the statically known type of the program.

```
Blob b = new Blob();  
Color y = b;    // implicit cast!
```

- Compile
Color y = b;
As
Mov y, [[b]] + 4



Multiple D.V. Summary

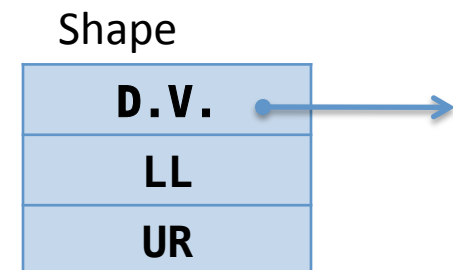
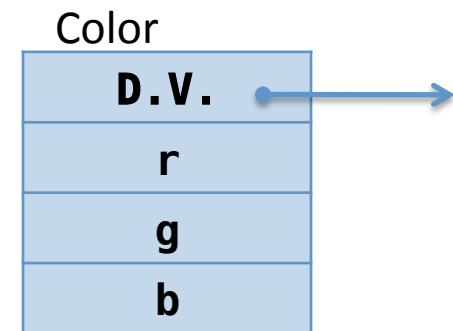
- Benefit: Efficient dispatch, same cost as for multiple inheritance
- Drawbacks:
 - Cast has a runtime cost
 - More complicated programming model... hard to understand/debug?

- What about multiple inheritance and fields?

Multiple Inheritance: Fields

- Multiple supertypes (Java): methods conflict (as we saw)
- Multiple inheritance (C++): fields can also conflict
- Location of the object's fields can no longer be a constant offset from the start of the object.

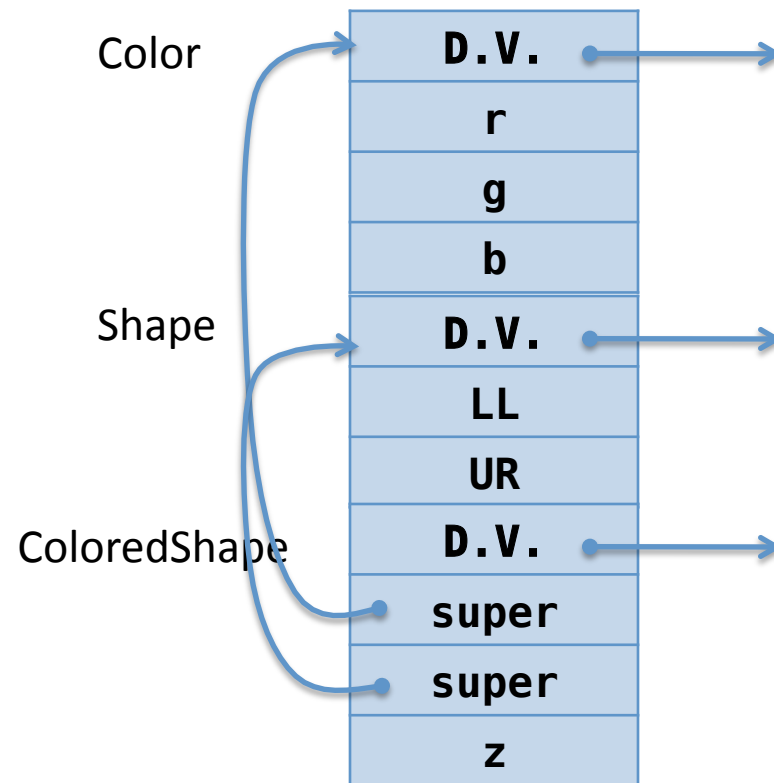
```
class Color {  
    float r, g, b; /* offsets: 4,8,12 */  
}  
class Shape {  
    Point LL, UR; /* offsets: 4, 8 */  
}  
class ColoredShape extends  
Color, Shape {  
    int z;  
}
```



ColoredShape ??

C++ approach:

- Add pointers to the superclass fields
 - Need to have multiple dispatch vectors anyway (to deal with methods)
- Extra indirection needed to access superclass fields
- Used even if there is a single superclass
 - Uniformity



Compiling Static Methods

- Java supports *static* methods
 - Methods that belong to a class, not the instances of the class.
 - They have no “this” parameter (no receiver object)
- Compiled exactly like normal top-level procedures
 - No slots needed in the dispatch vectors
 - No implicit “this” parameter
- They’re not really methods
 - They can only access static fields of the class

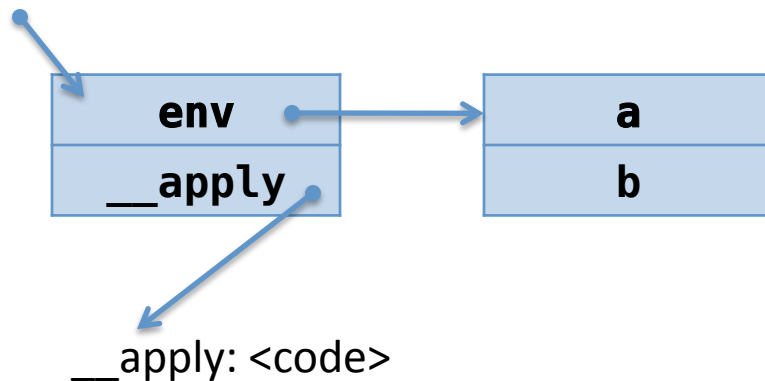
Compiling Constructors

- Java, C++ classes can declare constructors that create new objects.
 - Initialization code may have parameters supplied to the constructor
 - e.g. **new Color(r,g,b);**
- Modula-3: object constructors take no parameters
 - e.g. **new Color;**
 - Initialization would typically be done in a separate method.
- Constructors are compiled just like static methods, except:
 - The “this” variable is initialized to a newly allocated block of memory big enough to hold D.V. pointer + fields according to object layout
 - The D.V. pointer is initialized
 - The return value of the constructor is the (newly created) “this” pointer.

Observe: Closure \approx Single-method Object

- Free variables \approx Fields
- Environment pointer \approx “this” parameter
- Closure for function: \approx Instance of this class:

```
fun (x,y) ->  
  x + y + a + b
```



```
class C {  
  int a, b;  
  int apply(x,y) {  
    x + y + a + b  
  }  
}
```

