CS 516: COMPILERS

Lecture 18

Topics

- Compiling Classes and Objects

COMPILING
CLASSES AND OBJECTS

Code Generation for Objects

- ***Everything can be done during compilation from AST to IR***
  - Classes: To support them, we will have to:
    - Generate data structure types
      - For objects that are instances of the class and for the class tables
    - Generate the class tables for dynamic dispatch
  - Methods:
    - Method body code is similar to functions/closures
    - Method calls require *dispatch*
  - Fields:
    - Issues are the same as for records
    - Generating access code
  - Constructors:
    - Object initialization
  - Dynamic Types:
    - Checked downcasts
    - “instanceof” and similar type dispatch

Multiple Implementations

- The same interface can be implemented by multiple classes:

```
interface IntSet {
    public IntSet insert(int i);
    public boolean has(int i);
    public int size();
}
```

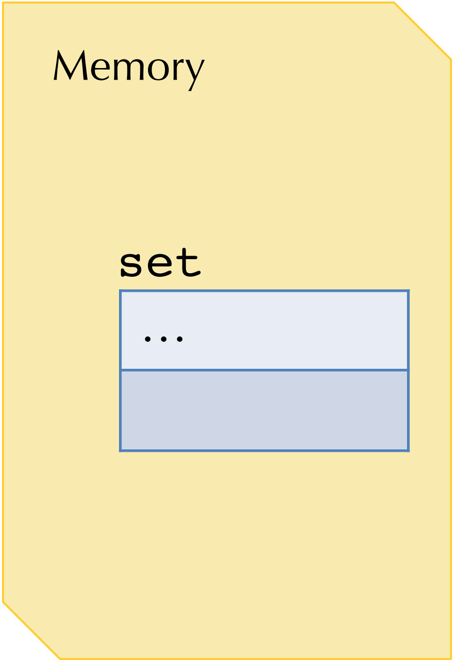
```
class IntSet1 implements IntSet {
  private List<Integer> rep;
  public IntSet1() {
    rep = new LinkedList<Integer>();}

  public IntSet1 insert(int i) {
     rep.add(new Integer(i));
    return this;}

  public boolean has(int i) {
    return rep.contains(new Integer(i));}

  public int size() {return rep.size();}
}
```

```
class IntSet2 implements IntSet {
  private Tree rep;
  private int size;
  public IntSet2() {
    rep = new Leaf(); size = 0;}

  public IntSet2 insert(int i) {
    Tree nrep = rep.insert(i);
    if (nrep != rep) {
      rep = nrep; size += 1;
    }
    return this;}

  public boolean has(int i) {
    return rep.find(i);}

  public int size() {return size;}
}
```

The Dispatch Problem

- Consider a client program that uses the IntSet interface:

```
IntSet set = …;
int x = set.size();
```

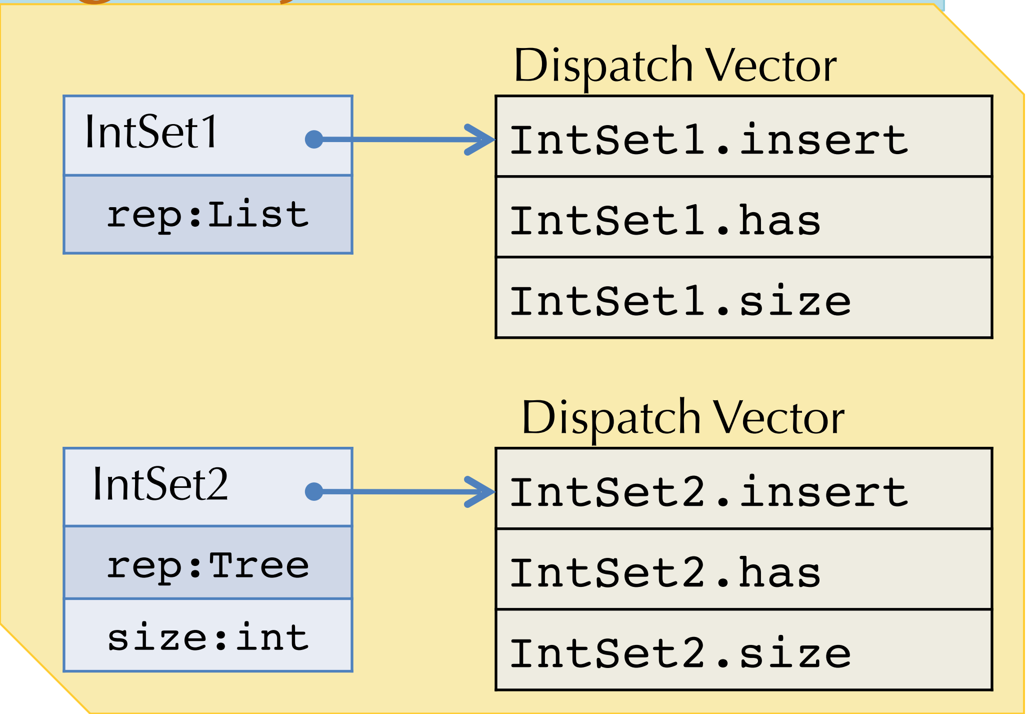


- Which code to call?
  - `IntSet1.size` ?
  - `IntSet2.size` ?

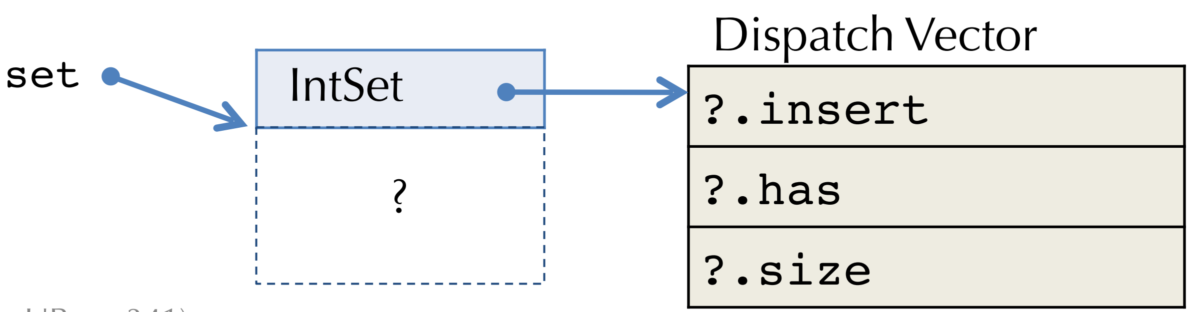
- Client code doesn't know the answer.
  - So objects must "know" which code to call.
  - Invocation of a method must indirect through the object.

Compiling Objects

- Objects contain a pointer to a *dispatch vector* (also called a *virtual table* or *vtable*) with pointers to method code.



- Code receiving `set:IntSet` only knows that `set` has an initial dispatch vector pointer and the layout of that vector.



Method Dispatch (Single Inheritance)

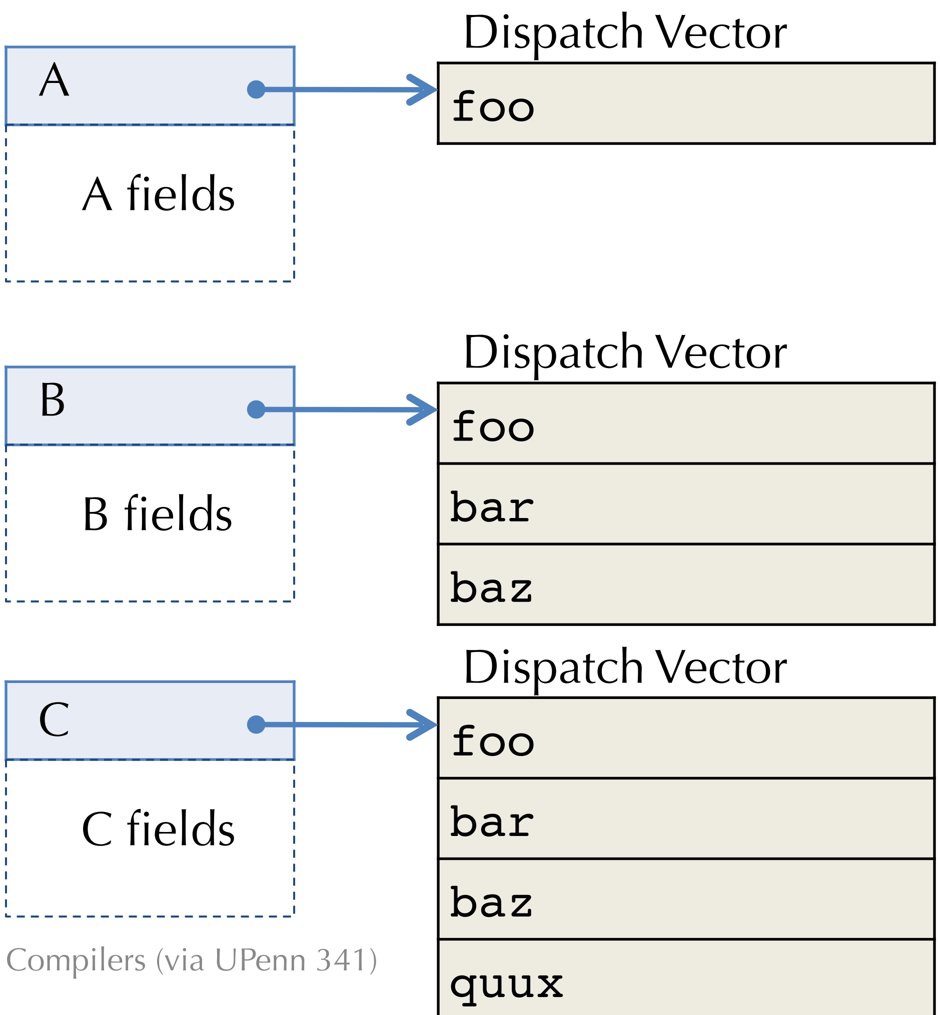
- Idea: every method has its own small integer index.
- Index is used to look up the method in the dispatch vector.

	Index
`interface A {`	
`  void foo();`	0
`}`	
`interface B extends A {`	
`  void bar(int x);`	1
`  void baz();`	2
`}`	
`class C implements B {`	
`  void foo() {…}`	0
`  void bar(int x) {…}`	1
`  void baz() {…}`	2
`  void quux() {…}`	3
`}`	

Inheritance / Subtyping:
C <: B <: A

Dispatch Vector Layouts

- Each interface and class gives rise to a dispatch vector layout.
- **Inheritance:** Inherited methods have *identical dispatch indices* in the subclass. (Width subtyping)



Representing Classes in LLVM

- During typechecking, create a *class hierarchy we denote* H
  - Part of the context: G;H;L
  - Maps each class to its interface:
    - Superclass
    - Constructor type
    - Fields
    - Method types
      (plus whether they inherit & which class they inherit from)

- Compile the class hierarchy to *produce*:
  - An LLVM IR struct type for each *object instance*
  - An LLVM IR struct type for each vtable (a.k.a. class table)
  - Global definitions that implement the class tables

Example OO Code

```
class A {
  new (int x)()                     // constructor
  { int x = x; }

  void print() { return; }          // method1
  int blah(A a) { return 0; }       // method2

}

class B <: A {
  new (int x, int y, int z)(x){
    int y = y;
    int z = z;
  }

  void print() { return; }      // overrides A
}

class C <: B {
  new (int x, int y, int z, int w)(x,y,z){
    int w = w;
  }

  void foo(int a, int b) {return;}
  void print() {return;}      // overrides B
}
```

Example OO Hierarchy in LLVM

```
class A {
  new (int x)()                    // constructor
  { int x = x; }

  void p
  i    pl

}

cla    B
  n    (i
       t
    int
  }

  void p
}

class C <
  new (i
    int v
  }

  void f
  void p
}
```

types

object instance types

class table types

```
%Object = type { %_class_Object* }
%_class_Object = type {  }

%A = type { %_class_A*, i64 }
%_class_A = type { %_class_Object*, void (%A*)*, i64 (%A*, %A*)* }

%B = type { %_class_B*, i64, i64, i64 }
%_class_B = type { %_class_A*, void (%B*)*, i64 (%A*, %A*)* }

%C = type { %_class_C*, i64, i64, i64, i64 }
%_class_C = type { %_class_B*, void (%C*)*, i64 (%A*, %A*)*, void (%C*, i64, i64)* }

@_vtbl_Object = global %_class_Object {  }

@_vtbl_A = global %_class_A { %_class_Object* @_vtbl_Object,
                              void (%A*)* @print_A,
                              i64 (%A*, %A*)* @blah_A }

@_vtbl_B = global %_class_B { %_class_A* @_vtbl_A,
                              void (%B*)* @print_B,
                              i64 (%A*, %A*)* @blah_A }

@_vtbl_C = global %_class_C { %_class_B* @_vtbl_B,
                              void (%C*)* @print_C,
                              i64 (%A*, %A*)* @blah_A,
                              void (%C*, i64, i64)* @foo_C }
```

Class tables

- (structs containing function ptrs)
- Used dynamically
- Globals
- Will be reused

The rest of this lecture will explain…

Example OO Hierarchy in LLVM

```
class A {
  new (int x)()           // constructor
  { int x = x; }

  void p
  i    pl

}

cla    B
  n    (i
       t
    int
  }

  void pi
}

class C
  new (i
    int v
  }

  void fo
  void pi
}
```

Fields

object instance types

class table types

types

```
%Object = type { %_class_Object* }
%_class_Object = type {  }

%A = type { %_class_A*, i64 }
%_class_A = type { %_class_Object*, void (%A*)*, i64 (%A*, %A*)* }

%B = type { %_class_B*, i64, i64, i64 }
%_class_B = type { %_class_A*, void (%B*)*, i64 (%A*, %A*)* }

%C = type { %_class_C*, i64, i64, i64, i64 }
%_class_C = type { %_class_B*, void (%C*)*, i64 (%A*, %A*)*, void (%C*, i64, i64)* }

@_vtbl_Object = global %_class_Object {  }

@_vtbl_A = global %_class_A { %_class_Object* @_vtbl_Object,
                              void (%A*)* @print_A,
                              i64 (%A*, %A*)* @blah_A }

@_vtbl_B = global %_class_B { %_class_A* @_vtbl_A,
                              void (%B*)* @print_B,
                              i64 (%A*, %A*)* @blah_A }

@_vtbl_C = global %_class_C { %_class_B* @_vtbl_B,
                              void (%C*)* @print_C,
                              i64 (%A*, %A*)* @blah_A,
                              void (%C*, i64, i64)* @foo_C }
```

Class tables

- (structs containing function ptrs)
- Used dynamically
- Globals
- Will be reused

The rest of this lecture will explain…

Example OO Hierarchy in LLVM

```
class A {
  new (int x)()          // constructor
  { int x = x; }
```

Fields

object instance types

class table types

types

```
%Object = type { %_class_Object* }
%_class_Object = type {  }

%A = type { %_class_A*, i64 }
%_class_A = type { %_class_Object*, void (%A*)*, i64 (%A*, %A*)* }

%B = type { %_class_B*, i64, i64, i64 }
%_class_B = type { %_class_A*, void (%B*)*, i64 (%A*, %A*)* }

%C = type { %_class_C*, i64, i64, i64, i64 }
%_class_C = type { %_class_B*, void (%C*)*, i64 (%A*, %A*)*, void (%C*, i64, i64)* }

@_vtbl_Object = global %_class_Object {  }

@_vtbl_A = global %_class_A { %_class_Object* @_vtbl_Object,
                              void (%A*)* @print_A,
                              i64 (%A*, %A*)* @blah_A }

@_vtbl_B = global %_class_B { %_class_A* @_vtbl_A,
                              void (%B*)* @print_B,
                              i64 (%A*, %A*)* @blah_A }

@_vtbl_C = global %_class_C { %_class_B* @_vtbl_B,
                              void (%C*)* @print_C,
                              i64 (%A*, %A*)* @blah_A,
                              void (%C*, i64, i64)* @foo_C }
```

Method types

Class tables
- (structs containing function ptrs)
- Used dynamically
- Globals
- Will be reused

The rest of this lecture will explain…

Method Arguments

- Methods bodies are compiled just like top-level procedures…
- … except that they have an implicit extra argument: `this` or `self`
  - Historically (Smalltalk), these were called the "receiver object"
  - Method calls were thought of a sending "messages" to "receivers"

A method in a class...

```
class IntSet1 implements IntSet {
   …
  IntSet1 insert(int i) { <body> }
}
```

… is compiled like this (top-level) procedure:

```
IntSet1 insert(IntSet1 this, int i) { <body> }
```

- Note 1: the type of "`this`" is the class containing the method.
- Note 2: references to fields inside <body> are compiled like `this.field`

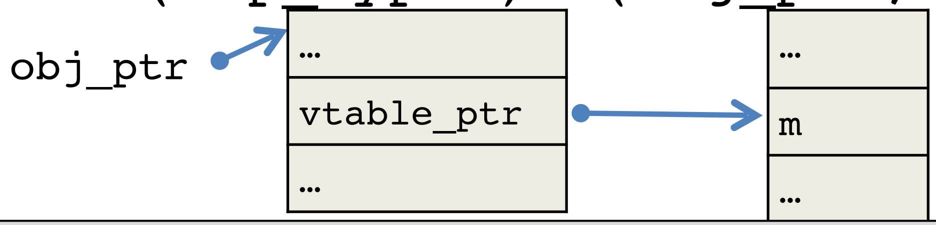
LLVM Method Invocation Compilation

- Consider method invocation:

$$[\![G;H;L \vdash \texttt{e.m(}\texttt{e}_1\texttt{,}\ldots\texttt{,}\texttt{e}_n\texttt{):t}]\!]$$

1. Compile $[\![G;H;L \vdash \texttt{e : C}]\!]$ to get a (pointer to) an object value of class type C. Call this value `obj_ptr`
2. Use **`getelementptr`** to extract **`vtable`** pointer from obj_ptr
3. Load the **`vtable`** pointer
4. Use **`getelementptr`** to extract function pointer from vtable
    1. using the information about C in H
5. Load the function pointer
6. Call through the function pointer, passing 'obj_ptr' for this:

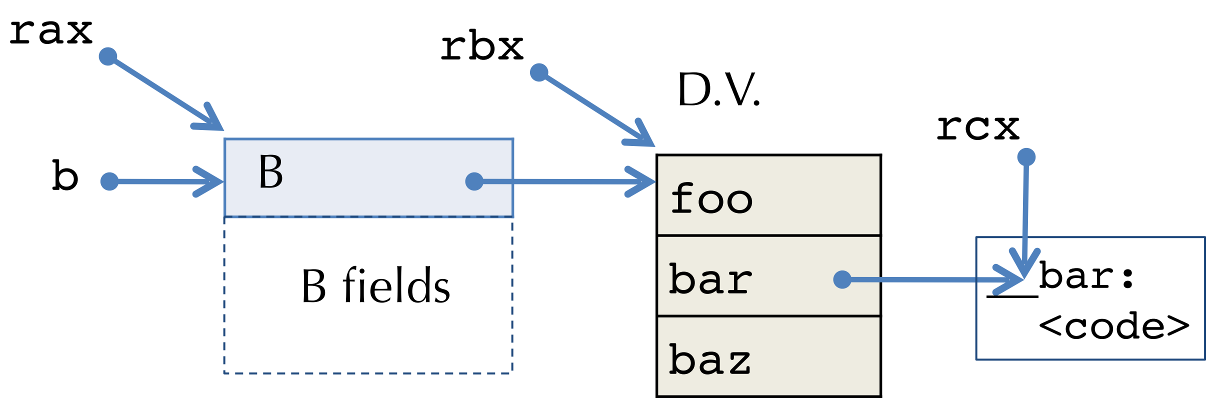
$$\texttt{call (cmp\_typ t) m(obj\_ptr, } [\![\texttt{e}_1]\!], \ldots, [\![\texttt{e}_n]\!]\texttt{)}$$



Note: Function calls may require bitcast to account for subtyping: arguments may be subtype of expected "formal" type.

X86 Code For Dynamic Dispatch

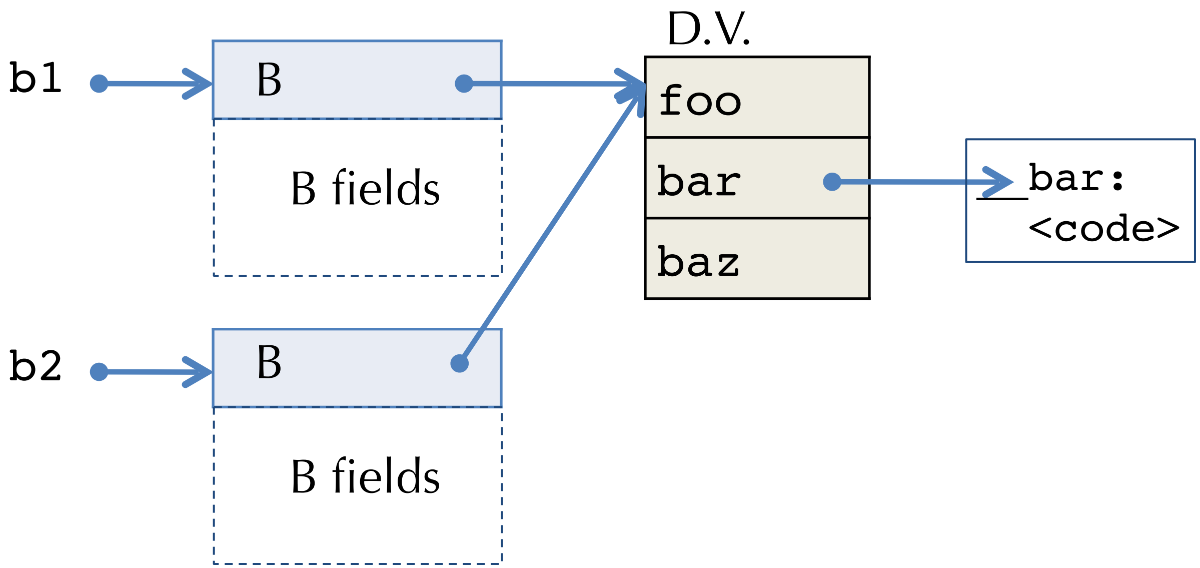
- Suppose `b : B`
- What code for `b.bar(3)`?
  - `bar` has index 1
  - Offset = 8 * 1




```
movq ⟦b⟧, %rax          // get b
movq [%rax], %rbx       // D.V. is the first spot in the memory layout
movq [rbx+8], %rcx      // method is D.V. + offset
movq %rax, %rdi         // Need "this" pointer
movq 3, %rsi            // Method argument
call %ecx               // Indirect call
```

Sharing Dispatch Vectors

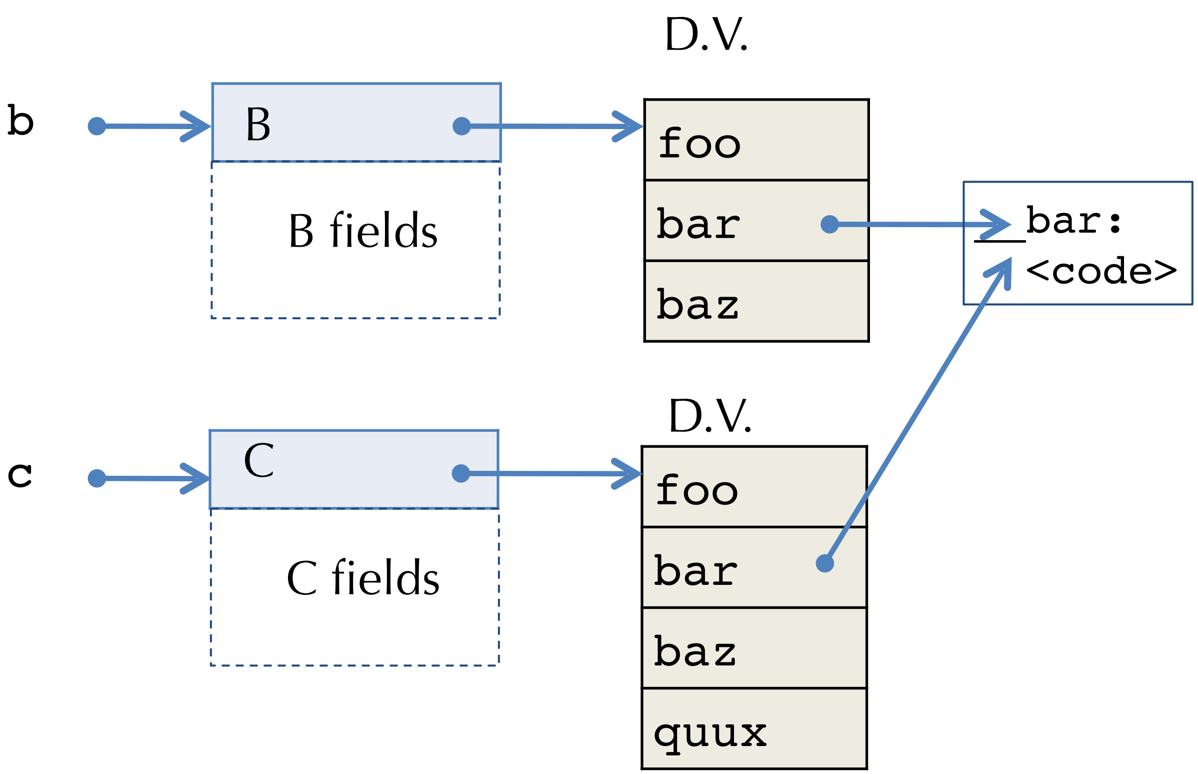
- All instances of a class may share the same dispatch vector.
  - Assuming that methods are immutable.
- Code pointers stored in the dispatch vector are available at link time – dispatch vectors can be built once at link time.



- One job of the object constructor is to fill in the object's pointer to the appropriate dispatch vector.
- Note: The address of the D.V. *is* the run-time representation of the object's type.

Inheritance: Sharing Code

- Inheritance: Method code "copied down" from the superclass
  - If not overridden in the subclass
- Works with separate compilation – superclass code not needed.



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

- OAT, 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:
  1. The "this" variable is initialized to a newly allocated block of memory big enough to hold D.V. pointer + fields according to object layout
  2. Constructor code initializes the fields
     - What methods (if any) are allowed?
  3. The D.V. pointer is initialized
     - When? Before/After running the initialization code?

Checked Casts

```
int sum(int[]? arr) {
    var z = 0;
    if?(int[] a = arr) {
      for(var i = 0; i<length(a); i = i + 1;) {
        z = z + a[i];
      }
    }
    return z;
}

int program (int argc, string[] argv) {
    var x = 0;
    x = x + sum(new int[]{1,2,3});
    x = x + sum(int[] null);
    return x;
}
```

Compiling Checked Casts

- How do we compile downcast:

```
if? (t x = exp) { … } else { … }
```

- Reason by cases: t must be either null, ref or ref? (can’t be just int or bool)
- **If t is null:** eg, `if?(null x = exp)`
  - The static type of exp must be ref? for some ref.
  - Emit code: if exp == null take the true branch, otherwise take the false branch
- **If t is string or t[]:** eg, `if?(int[] a = exp)`
  - The static type of exp must be the corresponding string? or t[]?
  - Emit code: if exp == null take the false branch, otherwise take the true branch
- **If t is C:** eg, `(C x = exp)`
  - The static type of exp must be D or D? (where C <: D)
  - If exp == null take the false branch, otherwise:
  - emit code to walk up the class hierarchy starting at D, looking for C
  - If found, then take true branch else take false branch
- **If t is C?:** eg, `(C? x = exp)`
  - The static type of exp must be D? (where C <: D)
  - If exp == null take the true branch, otherwise:
  - Emit code to walk up the class hierarchy starting at D, looking for C
  - If found, then take true branch else take false branch

"Walking up the Class Hierarchy"

- A non-null object pointer refers to an LLVM struct with a type like:

```
%B = type { %_class_B*, i64, i64, i64 }
```

- The first entry of the struct is a pointer to the vtable for Class B
  - This pointer *is* the dynamic type of the object.
  - It will have the value @vtbl_B
- The first entry of the class table for B is a pointer to its superclass:

```
@_vtbl_B = global %_class_B { %_class_A* @_vtbl_A,
                              void (%B*)* @print_B,
                              i64 (%A*, %A*)* @blah_A }
```

- Therefore, to find out whether an unknown type X is a subtype of C:
  - Assume C is not Object (ruled out by "silliness" checks for downcast)

  LOOP:
  - If X == @_vtbl_Object then NO, X is not a subtype of C
  - If X == @_vtbl_C then YES, X is a subtype of C
  - If X = @_vtbl_D, so set X to @_vtbl_E where E is D's parent and goto LOOP