

# Lecture #19: More Special Effects—Exceptions and OOP

## Administrivia

- First autograder ran Wednesday night. I'll add more tests for the final run.
- Test #2 on Tuesday, 14 April (in class).

# Exceptions and Continuations

- Exception-handling in programming languages is a very limited form of continuation.
- Execution continues after a function call that is still active when exception raised.
- Java provides mechanism to return a value with the exception, but this adds no new complexity.

# Approach I: Do Nothing

- Some say keep it simple; don't bother with exceptions.
- Use return code convention:

Example: C library functions often return either 0 for OK or non-zero for various degrees of badness.

- Problems:

# Approach I: Do Nothing

- Some say keep it simple; don't bother with exceptions.
- Use return code convention:

Example: C library functions often return either 0 for OK or non-zero for various degrees of badness.

- Problems:
  - Forgetting to check.
  - Code clutter.
  - Clumsiness: makes value-returning functions less useful.
  - Slight cost in always checking return codes.

## Approach II: Non-Standard Return

- First idea is to modify calls so that they look like this:

```
    call _f
    jmp  OK
    code to handle exception
OK:
    code for normal return
```

- To throw exception:
  - Put type of exception in some standard register or memory location.
  - Return to instruction *after* normal return.
- Awkward for the ia32 (above). Easier on machines that allow returning to a register+constant offset address [why?].
- Exception-handling code decides whether it can handle the exception, and does another exception return if not.
- Problem: Requires small distributed overhead for every function call.

# Approach III: Stack manipulation

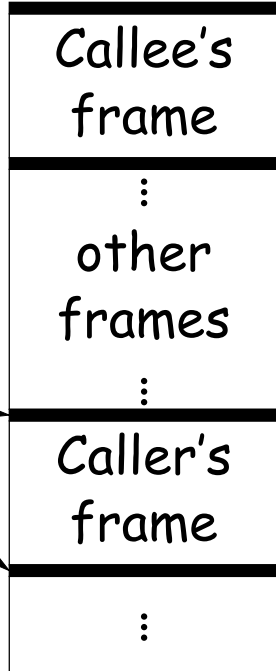
- C does not have an exception mechanism built into its syntax, but uses library routines:

```
jmp_buf catch_point;
```

```
void Caller () {  
    if (setjmp (catch_point) == 0) {  
        normal case, which eventually  
        gets down to Callee  
    } else {  
        handle exception  
    }  
}
```

catch\_point:

Caller's  
FP, SP,  
addr of  
setjmp call  
& others



```
void Callee () {  
    ...  
    // Throw exception:  
    longjmp (catch_point, 42);  
    ...  
}
```

## Approach III: Stack manipulation

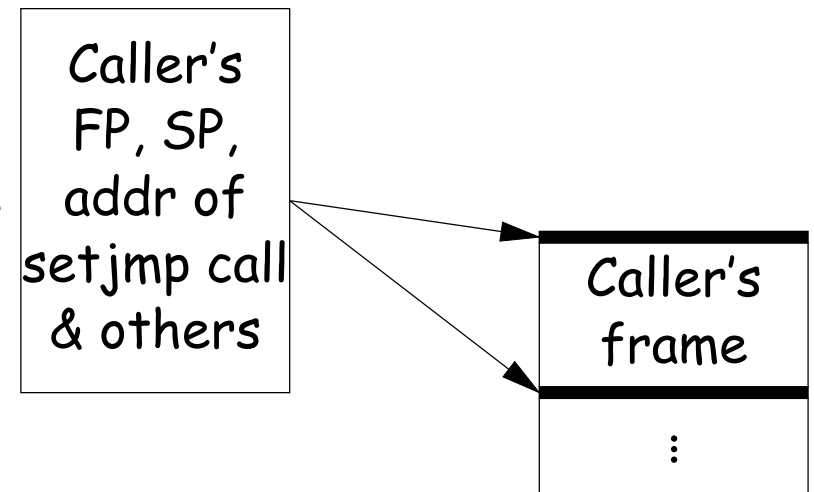
- *C* does not have an exception mechanism built into its syntax, but uses library routines:

```
jmp_buf catch_point;
```

```
void Caller () {  
    if (setjmp (catch_point) == 0) {  
        normal case, which eventually  
        gets down to Callee  
    } else {  
        handle exception  
    }  
}
```

```
void Callee () {  
    ...  
    // Throw exception:  
    longjmp (catch_point, 42);  
    ...  
}
```

catch\_point:



When `longjmp` called, restore stack as indicated by `catch_point` and return to the end of the `setjmp` call.

## Approach III: Discussion

- On exception, call to `setjmp` appears to return twice, with two different values.
- Does not require help from compiler,
- But implementation is architecture-specific.
- Overhead imposed on every `setjmp` call.
- If used to implement `try` and `catch`, therefore, would impose cost on every `try`.
- Subtle problems involving variables that are stored in registers:
  - The `jmp_buf` typically has to store such registers, but
  - That means the value of some local variables may revert unpredictably upon a `longjmp`.



## Approach IV: PC tables

- Sun's Java implementation uses a different approach.
- Compiler generates a table mapping instruction addresses (program counter (PC) values) to exception handlers for each function.
- If needed, compiler also leaves behind information necessary to return from a function ("unwind the stack") when exception thrown.
- To throw exception E:
  - while (current PC doesn't map to handler for E)  
unwind stack to last caller
- Under this approach, a try-catch incurs no cost unless there is an exception, but
- Throwing and handling the exception more expensive than other approaches, and
- Tables add space.

# New Topic: Dynamic Method Selection and OOP

- “Interesting” language feature introduced by Simula 67, Smalltalk, C++, Java: the *virtual function* (to use C++ terminology).
- Problem:
  - Arrange classes in a hierarchy of types.
  - Instance of subtype “is an” instance of its supertype(s).
  - In particular, inherits their methods, but can override them.
  - A *dynamic effect*: Cannot in general tell from program text what body of code executed by a given call.
- Implementation difficulty (as usual) depends on details of a language’s semantics.
- Some things still static:
  - Names of functions, numbers of arguments are (usually) known
  - Compiler can handle overloading by inventing new names for functions. E.g., G++ encodes a function  $f(\text{int } x)$  in class  $Q$  as  $\_ZN1Q1fEi$ , and  $f(\text{int } x, \text{int } y)$  as  $\_ZN1Q1fEii$ .

# I. Fully Dynamic Approach

- Regular Python has a completely dynamic approach to the problem:

```
class A:
    x = 2; def f (self): return 42

a = A (); b = A ()
print a.x, a.f()    # Prints 2 42
a.x = lambda (self, z): self.w * z
a.f = 13; a.w = 5
print a.x(3), a.f, a.w    # Prints 15 13 5
print b.x(3), b.f, b.w    # Error
print A.x                # Prints 2
A.x = lambda (self): 19
A.f = 2
A.v = 1
c = A ()
print c.x (), c.f, c.v    # Prints 19, 2, 1
```

# Characteristics of Dynamic Approach

- Each class instance is independent. Contents of class definition merely used for initialization.
- New attributes can be added freely to instances or to class.
- In other variants of this approach, there are no classes at all, only instances.
- Get new instances by cloning an existing object.
- Then can add new attributes.

# Implementing the Dynamic Approach

- Simple strategy: just put a dictionary in every instance, and in class.
- Create an instance by making fresh copy of class's dictionary.
- All checking at runtime.
- All objects (or pointers) carry around dynamic type

# Pros and Cons of Dynamic Approach

- Extremely flexible
- Conceptually simple
- Implementation easy
- Space overhead: every instance has pointers to all methods
- Time overhead: lookup on each call
- No static checking

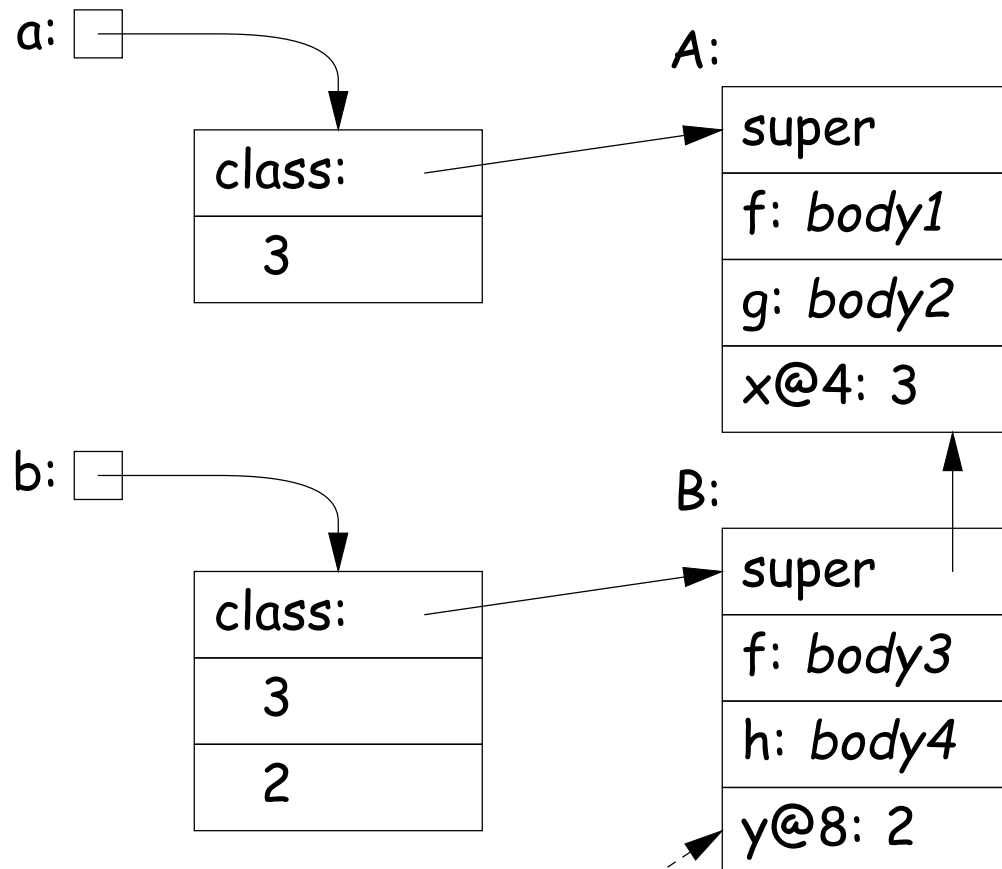
## II. Straight Single Inheritance, Dynamic Typing

- Each class has fixed set of methods and instance variables
- Methods have fixed definition in each class.
- Classes can inherit from single superclass.
- Otherwise, types of parameters, variables, etc., still dynamic
- Basically technique in Smalltalk, Objective C.

# Implementing the Smalltalk-like Approach

- Instances need not carry around copies of function pointers.
- Instead, each *class* has a data structure mapping method names to functions, and instance-variable names to offsets from the start of the object.

```
class A:  
    def f (...): body1  
    def g (...): body2  
    x = 3  
class B(A):  
    def f (...): body3  
    def h (...): body4  
    y = 2  
  
a = A ()  
b = B ()
```



"y is stored at offset 8 from start of instance"



# Pros and Cons of Smalltalk Approach

- Only need to store modifiable things—instance variables—in instances.
- Data structure can be a bit faster at accessing than fully dynamic method
- But still, not much static checking possible, and
- Some lookup of method names required.

# Single Inheritance with Static Types

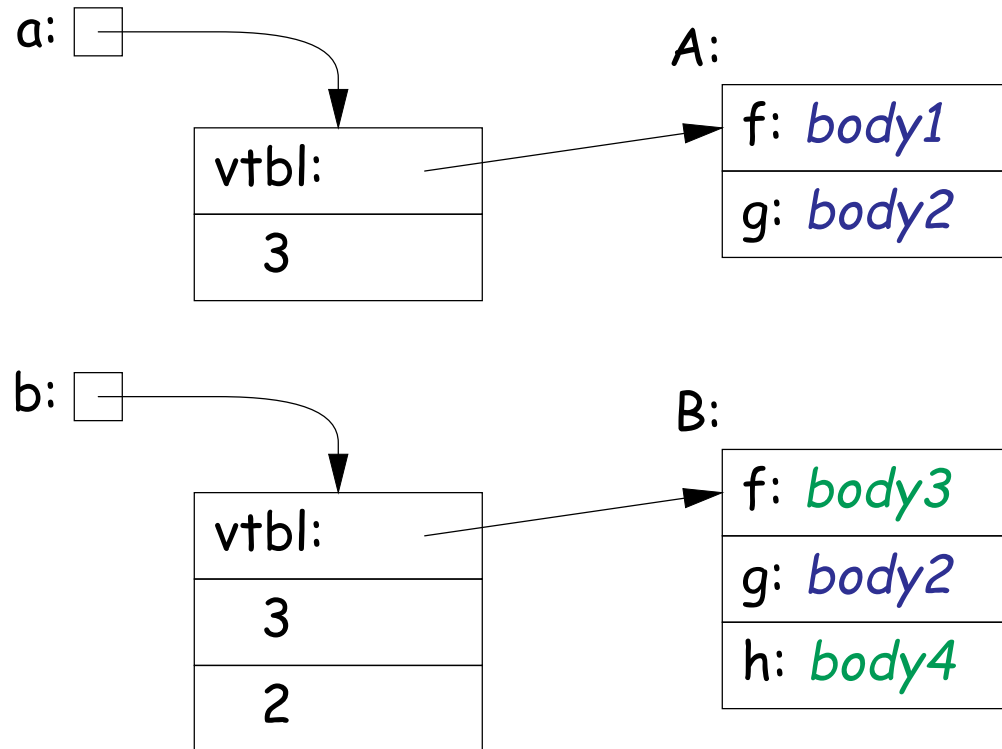
- Consider Java without interfaces. Type can inherit from at most one immediate superclass.
- For an access,  $x.w$ , insist that compiler knows a supertype of  $x$ 's dynamic type that defines  $w$ .
- Insist that all possible overridings of a method have compatible parameter lists and return values.
- Use a technique similar to previous one, but put entries for all methods (whether or not overridden) in each class data structure.
- Such class data structures are called "virtual tables" or "vtables" in C++ parlance.

# Implementation of Simple Static Single Inheritance

```
class A {  
    void f () { body1 }  
    void g () { body2 }  
    int x = 3  
}
```

```
class B extends A {  
    void f () { body3 }  
    void h () { body4 }  
    int y = 2  
}
```

```
-----  
a = new A ()  
b = new B ()
```



- No need to store offsets of x and y; compiler knows where they are.
- Also, compiler knows where to find 'f', 'g', 'h' virtual tables.
- **Important:** offsets of variables in instances and of method pointers in virtual tables are *known constants*, the *same for all subtypes*.
- So compiler knows how to call methods of b even if static type is A!

# Interfaces

- Java allows *interface inheritance* of any number of interface types (introduces no new bodies).
- This complicates life: consider

```
class A {
    int x;
    public f () { ... }
}

class B {
    int y;
    g () { ... }
    h () { ... }
    public f () { ... }
}

interface C {
    f ();
}

/*-----*/
class A2 extends A
    implements C
{...}

class B2 extends B
    implements C
{ ... }

/*-----*/
void f (C y) { y.f () } // How can this work?
```

- We can compile A and B without knowledge of C, A2, B2.
- How can we make the virtual table of A2 and B2 compatible with each other so that f is at same known offset regardless of whether dynamic type of C is A2 or B2? (Above isn't hardest example!)

# Interface Implementation I: Brute Force

- One approach is to have the system assign a different offset *globally* to each different function signature

(Functions  $f(\text{int } x)$  and  $f()$  have different function signatures)

- So in previous example, the virtual tables can be:

A:	B:	C:
0: unused	0: ptr to B.g	0: unused
4: unused	4: ptr to B.h	4: unused
8: ptr to A.f	8: ptr to B.f	8: unused
A2:	B2:	
0: unused	0: ptr to B.g	
4: unused	4: ptr to B.h	
8: ptr to A.f	8: ptr to B.f	

- No slowing of method calls.
- But, Total size of tables gets big (some optimization possible).
- And, must take into account all classes before laying out tables.

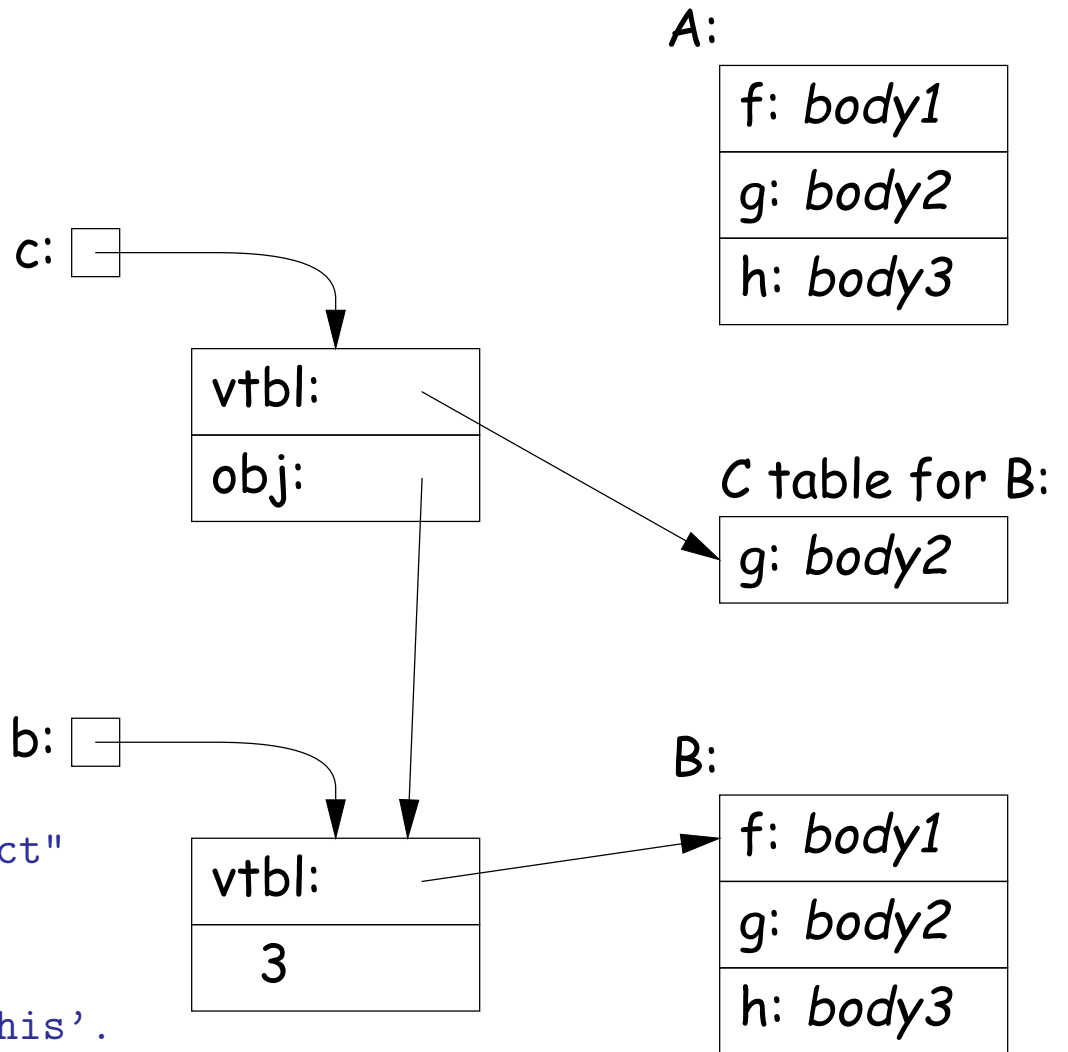
Complicates dynamic linking.

# Interface Implementation II: Make Interface Values Different

- Another approach is to represent values of static type  $C$  (an interface type) differently.
- Converting value  $x2$  of type  $B2$  to  $C$  then causes  $C$  to point to a two-word quantity:
  - Pointer to  $x2$
  - Pointer to a cut-down virtual table containing just the  $f$  entry from  $B2$  (at offset 0).
- Means that converting to interface requires work and allocates storage.

# Interface Implementation II, Illustrated

```
class A {  
    void f () { body1 }  
    void g () { body2 }  
    void h () { body3 }  
    int x = 3;  
}  
  
interface C { void g (); }  
  
class B extends A  
    implements C { }  
  
B b = new B ();  
C c = b;  
    // Create "interface object"  
c.g ();  
    // Get g from c.vtbl, ...  
    // ...and use c.obj as 'this'.
```



# Improving Interface Implementation II

- How can we avoid doing allocation to create value of interface type *C*?
- One method: extend the virtual table of all types to include an *interface vector*.
- Each entry in this vector identifies an interface the type implements, plus the table (e.g. "C table for B" in last slide).
- To implement '*C c = b*' from last slide, just copy pointer *b*, as for the usual cases when assigning to a variable whose type is a super-type of the value assigned.
- To implement '*c.g()*' from last slide, find the "C table" in the interface vector for object pointed to be *c* and fetch the entry for *g*. Just call as usual.
- Question for the reader: How best to design the interface vector?
  - Want fetching of *c.g* to be fast,
  - So best to avoid having to actually perform a search at execution time. How?



# Full Multiple Interitance

- Java allows multiple inheritance only via interfaces.
- Important point: *interfaces don't have instance variables.*
- Instance variables basically mess everything up for multiple inheritance, assuming we want to keep constant offsets to instance variables.

```
class A {  
    int x = 19;  
    void f () { ... x ... h() ... }  
    void h () {... }  
}
```

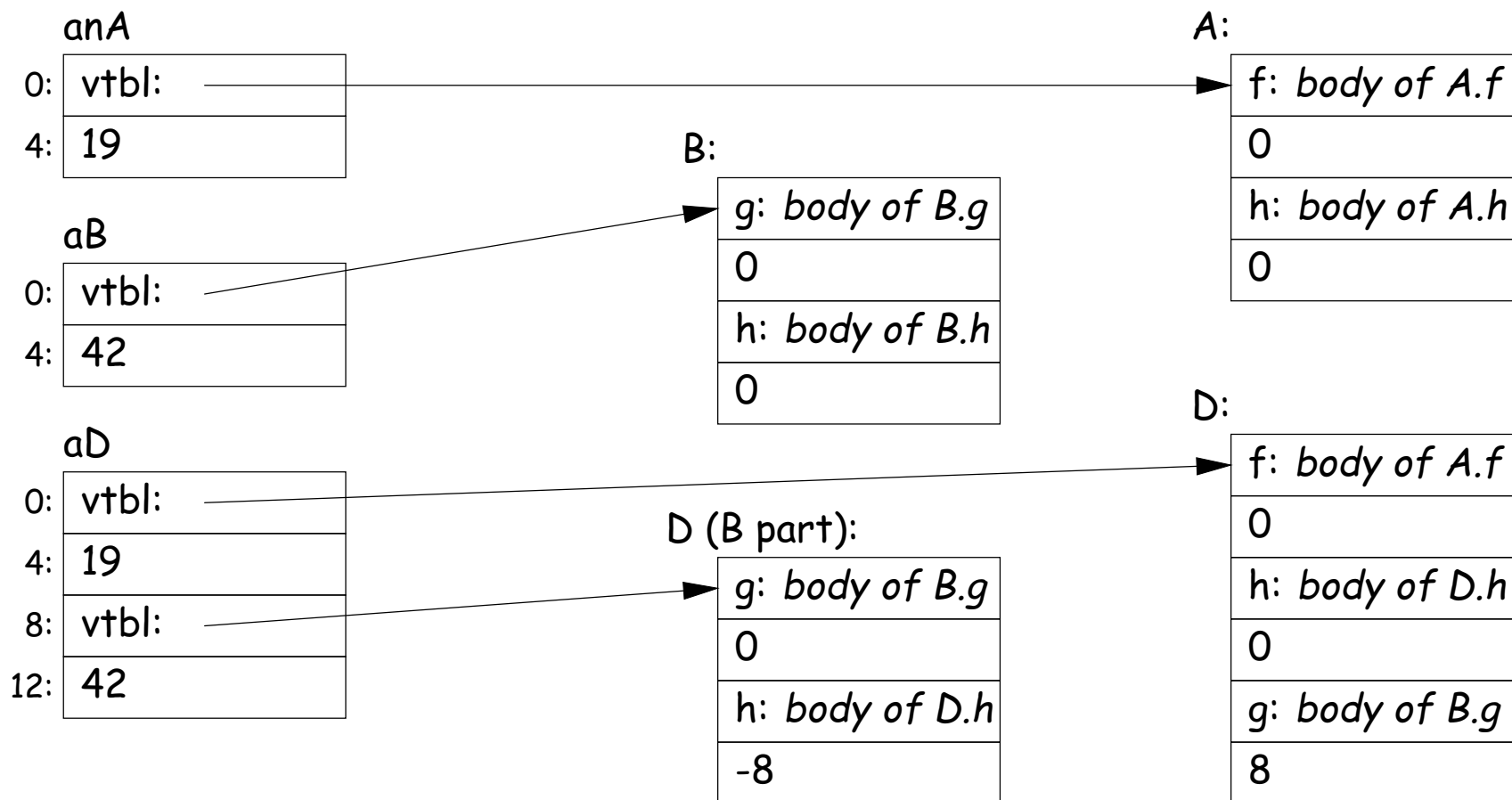
```
class B {  
    int y = 42;  
    void g () { ... y ... h() ... }  
    void h () {... }  
}
```

```
class D extends A, B {  
    // Where do x and y go?  
    void h () {... }  
}
```

- If `aD` is a `D`, then `aD.f` expects that 'this' points to an `A`, `aD.g` expects that it points to a `B`, but `aD.h` expects it to point to a `D`.
- How can these all be true??

# Implementing Full Multiple Inheritance I

- Idea is to extend the contents of the virtual table with an offset for each method.
- Offset tells how to adjust the 'this' pointer before calling.
- For the classes from the last slide:



# Implementing Full Multiple Inheritance I (contd.)

- To call `aD.g`,
  - Fetch function address of `g` from `D` table.
  - Call it, but first add 8 to pointer value of `aD` so as to get a pointer to the "B part" of `aD`.
- When `aD.g` eventually calls `h` (actually `this.h`),
  - 'this' refers to the "B part" of `aD`.
  - Its virtual table is "D (B part)" in the preceding slide.
  - Fetching `h` from that table gives us `D.h`, ...
  - ...which we call, after first adding the -8 offset from the table to "this."
  - Thus, we end up calling `D.h` with a "this" value that points to `aD`, as it expects.

# Implementing Full Multiple Inheritance II

- First implementation slows things down in all cases to accommodate unusual case.
- Would be better if only the methods inherited from B (for example) needed extra work.
- Alternative design: use stubs to adjust the 'this' pointer.
- Define  $B.g_1$  to add 8 to the 'this' pointer by 8 and then call  $B.g$ ; and  $D.h_1$  to subtract 8 and then call  $D.h$ :

