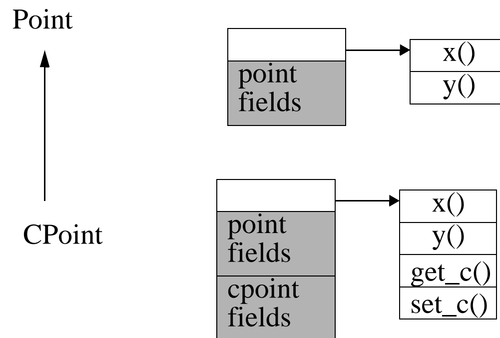
Object Layout and Dispatch for Multiple Inheritance

- This lecture is based on Myers's Master thesis "Fast Object Operations in a Persistent Programming System"
  - I will present all the arguments exactly as they are in the thesis, and then we will discuss them critically (hint: don't agree with everything I say)
  - This is an excellent intro to issues of object layout, dispatch mechanisms, multiple inheritance vs. interface inheritance, etc.
  - Warning: the presentation is condensed. We should spend a long time on each slide

Single Inheritance Object Layout



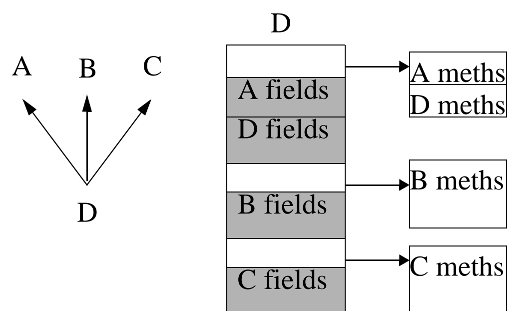
- Indirection for dynamic dispatch
- Superclass object embedded into subclass object, superclass dispatch table (v-table) a prefix of subclass dispatch table
  - subclass objects can always be viewed as superclass objects (makes things very simple)

Background

- Recall: C++ has true multiple inheritance, Java has single inheritance, but a class can conform to multiple interfaces. This is also the case for Theta, the language discussed in this thesis
  - alternatively: "a class has a single superclass but multiple supertypes"
- Assumptions for evaluation:
  - we want to maintain separate compilation: when we compile a class we don't know the entire inheritance hierarchy (especially, its subclasses)
  - objects are much more numerous than classes. Some extra space overhead per class is negligible if it saves space per object

Multiple Inheritance Object Layout

- True multiple inheritance: layout becomes convoluted



- The subclass object can be treated as a superclass object *only for the "primary superclass"*

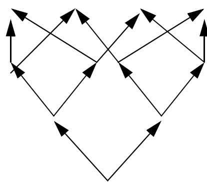
- This means that method dispatch is more complicated: methods are pre-compiled for a certain object layout
  - an inherited method will need to view a subclass object as a superclass object
  - an overridden method will need to view a superclass subobject as a subclass object
- This means that every method dispatch will incur some “offset adjustment” overhead (which sometimes can be eliminated)
  - an inherited method from a non-primary superclass, accessed through a subclass object, will need the `this` pointer adjusted to point to the superclass (where the method was inherited from) subobject
  - an overridden method, originally defined in a non-primary superclass, accessed through that superclass’s subobject, will need `this` adjusted to point to the subclass object
  - (example with `point`, `rect`, `box`, and an `area` method in `rect` and `box`)

Offset Adjustment Optimizations

- Neat optimization for the common case:
  - for the few methods that do need adjustment, put a pointer to a stub method in the dispatch vector. The stub method first adjusts the this pointer, then jumps to the right method
  - this way, the objects that don’t need offset adjustment incur no overhead
- Even better but complicated: rewrite the method code to use the correct offsets when it accesses fields. Point to the re-written version of the code from the dispatch table

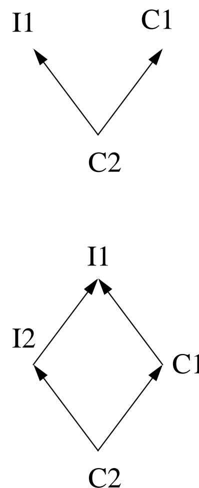
Space Overhead Problems

- The standard multiple inheritance layout may incur significant space overheads per object: one dispatch header word for every path to a root of the hierarchy
- Pathological example: exponential number of dispatch words (in the depth of the hierarchy)



- Myers argues for a new layout that has benefits for Java-like languages

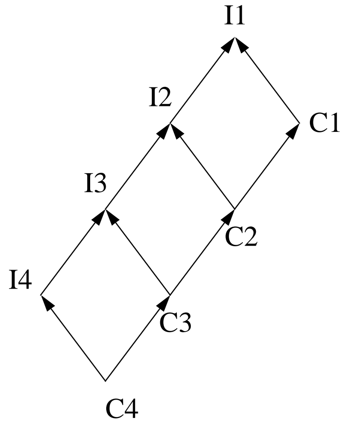
Layout Examples



- What are the layouts for these hierarchies
  - “I” denotes interface, “C” denotes class

More Complex

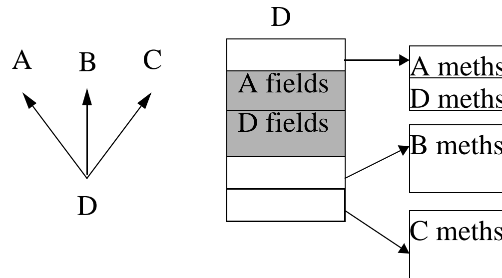
- Myers argues that the following hierarchy is common in Java-like languages (*motivating example*)



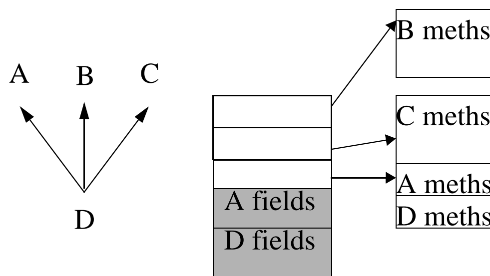
- What is the layout for that?

Myers's Bidirectional Object Layout

- Claimed to be 1) more compact; 2)amenable to faster dispatch; 3) better suited for persistent objects than the standard C++ multiple inheritance layout
  - only applicable to Java-like languages with single superclass
- Standard C++ layout for such languages:



- Bidirectional layout:



- Dispatch pointers grow upwards, fields grow downwards
- All objects have a single class dispatch header but perhaps multiple interface dispatch headers
- Offsets are simple:
  - for access through class pointer, no adjustment
  - for access through interface pointer, offset is same for every interface method (they are all "overridden", figuratively)

Optimizations: Type Headers

- Merging type headers: dispatch tables must be merge-able (distinct indices for distinct methods—e.g., I1 has offsets 0,2,4,6, I2 has offsets 1,3,5)
  - always true for sequential assignment of method offsets if a type is a subtype of another
  - several tricks to make independent dispatch tables merge-able: more sparse assignment of method indices
    - good for further reading if anyone is interested
  - simple trick: even spacing, beginning at original index in the range 1..N (randomly)
    - good when multiple supertypes are rare

Optimizations: Class + Type headers

Even with full merging of type headers, two separate header words (one for type header, one for class header) is still worse than single inheritance

- Merging class header with type header (3 common optimizations)
  - e.g., if class has no superclass, all methods can go after the supertype's methods
  - class methods can start at fixed offset (e.g., 100 slots from the beginning of the dispatch vector—this leaves room for 100 interface methods)
  - negative method indices for types
- If all previous optimizations are applied, most objects will have a single dispatch word, both for interface and for class accesses

Critique

- What's the contribution of the bidirectional object layout?
  - how much of the benefit can be obtained by applying the same optimizations to the standard multiple inheritance layout, given that the language is Java-like(no multiple inheritance but multiple interfaces) ?