Implementing Object-Oriented Languages

Key features:

- inheritance (possibly multiple)
- subtyping & subtype polymorphism
- message passing, dynamic binding

Subtype polymorphism is the key problem

- support uniform representation of data
  (analogous to boxing for polymorphic data)
  - e.g. every object has a class pointer, or a virtual fn table pointer, at a known offset
- organize layout of data to make instance variable access and method lookup & invocation fast
  - multiple inheritance complicates this
- perform static analysis to bound polymorphism
- perform transformations to reduce polymorphism

Implementing single inheritance

Key idea: **prefixing**

- layout of superclass is a prefix of layout of subclass

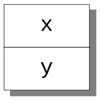
\+ instance variable access is just a load or store

\+ can add new instance variables in subclasses

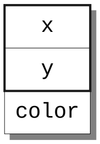
– cannot override or undefine instance variables

– multiple inheritance...

```
class Point {
  int x;
  int y;
}
```



```
class ColorPoint
    extends Point {
  Color color;
}
```



// OK: subclass polymorphism

`Point p = new ColorPoint(3,4,Blue);`

// OK: x and y have same offsets in all `Point` *subclasses*

`int manhattan_distance = p.x + p.y;`

Implementing dynamic dispatching (virtual functions)

Key idea: class-specific table of function pointers

- store pointer to table in each object
- assign table offset to method name,
  just as instance variable offsets are assigned
- exploit prefixing in function table layout:
  superclass's function table layout is a prefix of subclass's function table layout

Dynamically-dispatched call sequence:

`T obj = ...; ... obj.msg() ...` ⇒

load *(obj + offset$_{T::table}$), table
load *(table + offset$_{T::msg}$), method
call *method

\+ dynamic dispatching is fast & constant-time

\+ can add new methods, override old ones
(can undefine methods by leaving hole in function table)

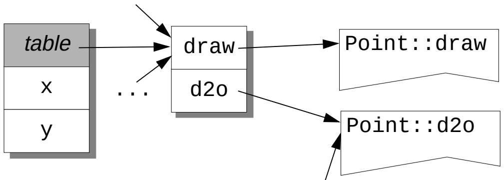
– extra word of memory per object

– loads, indirect calls can be slow on pipelined machine

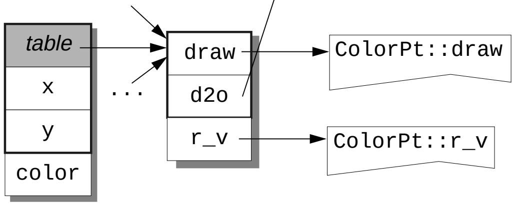
– no inlining

Example of function tables

```
class Point {
  int x;
  int y;
  void draw();
  int distance2origin();
}
```



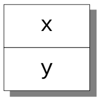
```
class ColorPoint extends Point {
  Color color;
  void draw();
  void reverse_video();
}
```



Multiple inheritance

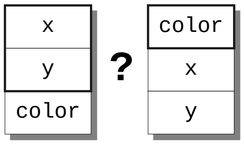
Problem: prefixing doesn't work with multiple inheritance

```
class Point {
  int x;
  int y;
}
```



```
class ColoredThing {
  Color color;
}
```

```
class ColorPoint
  extends Point,
          ColoredThing {
}
```



```
ColorPoint cp = new ColorPoint(3, 4, Blue);
Point p = cp; // OK
ColoredThing t = cp; // OK
// breaks:
ColorP cp2 = new ColorPoint(p.x, p.y, t.color);
```

Clever solution: embedding, not prefixing

Embed layout of each superclass **somewhere** in object

- e.g. concatenate layouts of immediate superclasses, then extend with subclass's instance variables

Trick: adjust object pointer to point to appropriate embedded object whenever use subclass polymorphism, based on static type of pointer

- perform pointer arithmetic on assignments, type casts, parameter & result passing if static type of l.h.s. differs from r.h.s.
  - must test for NULL pointer and not change, though
- constants to add/subtract determined by static types & analysis of inheritance graph, at compile-time

+ fast access to instance variables

– some type-casts (implicit or explicit) now do arithmetic

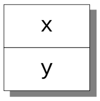
– interior pointers complicate garbage collection

– type-casts outside the class hierarchy break:

```
void* p = new ColorPoint(...);
ColoredThing t = (ColoredThing) p;
... t.color ... // returns p.x!
```

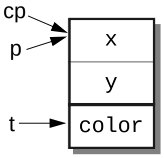
Example

```
class Point {
  int x;
  int y;
}
```



```
class ColoredThing {
  Color color;
}
```

```
class ColorPoint
  extends Point,
          ColoredThing {
}
```



```
ColorPoint cp = new ColorPoint(3, 4, Blue);
Point p = cp; // OK
ColoredThing t = cp; // OK: adds 8 to cp
// now this works:
ColorP cp2 = new ColorPoint(p.x, p.y, t.color);
// this works, too:
ColorP cp3 = (ColorPoint) t; // subtracts 8 from t
```

Multiple inheritance and dynamic dispatching

Extend embedding to implement dynamic dispatching, too

- replicate function tables for each immediate superclass, to avoid clashes in method offsets

– multiple function table pointers in object

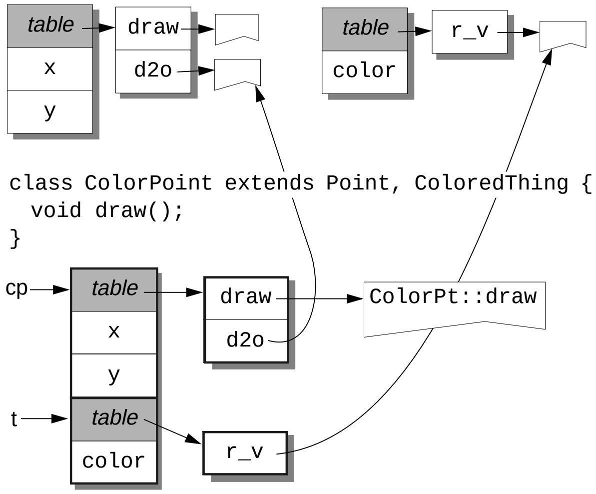
```
ColorPoint cp = new ColorPoint(3, 4, Blue);
... cp.distance2origin() ...; // works
... cp.draw() ...; // works

ColoredThing t = cp; // works; adds 12 to cp
... t.reverse_video() ...; // works
```

Example of embedded function tables

```
class Point {       class ColoredThing {
  int x;              Color color;
  int y;              void reverse_video() {
  void draw();          ... self.color ...
  int d2o();          }
}                   }
```

```
class ColorPoint extends Point, ColoredThing {
  void draw();
}
```



A problem

Simple embedding doesn't always work!

```
... cp.reverse_video() ...;
```

// breaks: accesses `self.x` instead of `self.color`
// inside `reverse_video` method

Analysis of the problem

Problem:
implicit cast of actual receiver to formal receiver ignored

- sometimes need to do pointer arithmetic

How to fix it?

- caller can't do it:
  doesn't know type of formal in callee
- callee can't do it:
  doesn't know type of actual in caller
- function tables can do it:
  caller's offset known, callee's offset known

An implementation strategy

Add an extra column to function table,
containing required pointer adjustment for receiver

Fetch and add adjustment to receiver pointer as part of call:

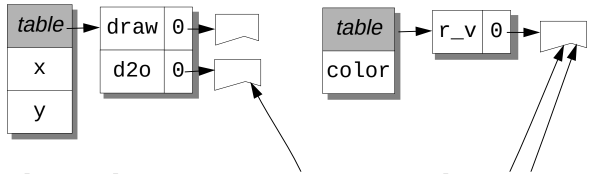
```
T obj = ...; ... obj.msg() ...    ⇒
  load *(obj + offset_{T::table}), table
  load *(table + offset_{T::msg}*2), method
  load *(table + offset_{T::msg}*2+4), delta
  add obj, delta, obj;
  call *method
```

- 5 instructions, not 3
  - costs even if multiple inheritance not used!
- some space cost
- only works for receiver;
  need some other mechanism if argument or result types change via method overriding

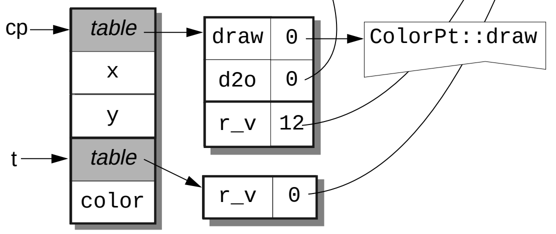
[Stroustrup 87]

Example of function tables with offsets

```
class Point {        class ColoredThing {
  int x;               Color color;
  int y;               void reverse_video() {
  void draw();           ... self.color ...
  int d2o();           }
}                    }
```



```
class ColorPoint extends Point, ColoredThing {
  void draw();
}
```



An alternate strategy

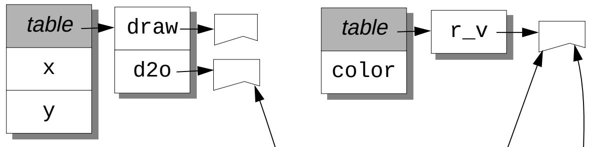
Insert "trampoline" function to perform updates where necessary

- dispatch sequence is same 3 instructions as for single inheritance case
- callee may be an adjustment function, which forwards to real function

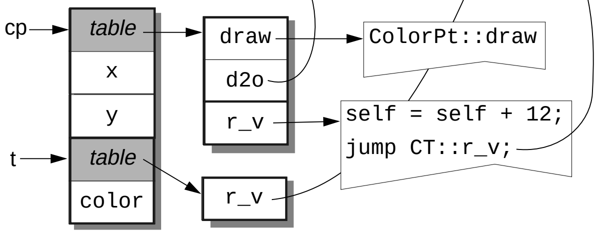
+ no cost for potential multiple inheritance where not used
+ invocations requiring adjustment may be faster, too

Example of function tables with trampolines

```
class Point {        class ColoredThing {
  int x;               Color color;
  int y;               void reverse_video() {
  void draw();           ... self.color ...
  int d2o();           }
}                    }
```

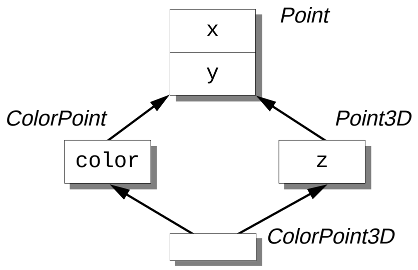


```
class ColorPoint extends Point, ColoredThing {
  void draw();
}
```



Multiple inheritance with shared superclasses

```
class Point {
  int x;
  int y;
}
class ColorPoint extends Point {
  Color color;
}
class Point3D extends Point {
  int z;
}
class ColorPoint3D extends ColorPoint,
                           Point3D {
}
```



First question: semantics

Question: how many `x` and `y` fields in a `ColorPoint3D` object?

- treat each path to `Point` as a distinct subcomponent?
- treat shared `Point` as a single subcomponent?

Answer depends mostly on whether subclassing is for
private implementation (maintain separate hidden copies)
or public classification (share a single visible copy)

Second question: implementation

If private, then replicate superclass along each path

- the default in C++
- need new language constructs to resolve ambiguity
- don't need new implementation techniques

If public, then have only one embedded copy of shared superclass

- `virtual` base class in C++,
  the default (only) case in most other OOPLs
- embedding won't work any more
  ⇒ need new implementation techniques

Implementing shared superclasses

Store offset of shared superclass's embedding in entry in function table

- load offset when accessing an instance variable in shared superclass (+3 instructions)
  - could put offset or pointer in object,
    trading space in object for faster access (+1 instruction)
  - could store offsets of shared indirect superclasses à la displays,
    trading space for faster access
- no effect when invoking method,
  other than trampoline stub functions

Cannot type-cast from shared superclass to subclass, in C++

- no fixed offset ⇒ depends on dynamic class of object

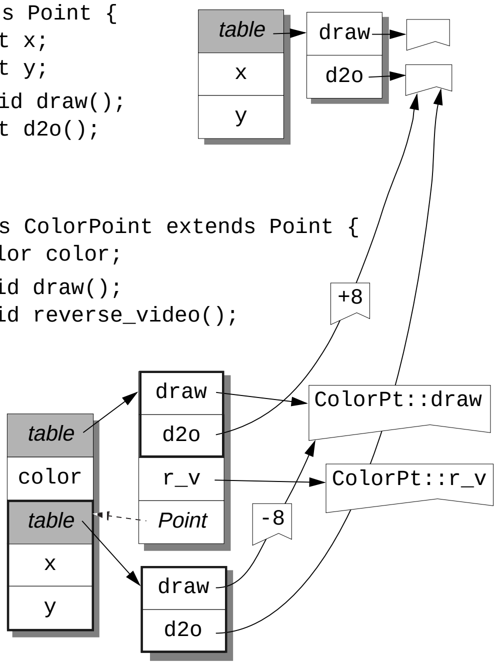
If sharing of common superclasses is the default,
significant overhead when accessing inherited instance vars

- is this implementation strategy good
  for languages other than C++?
  for C++?

Example of shared superclasses (part 1)

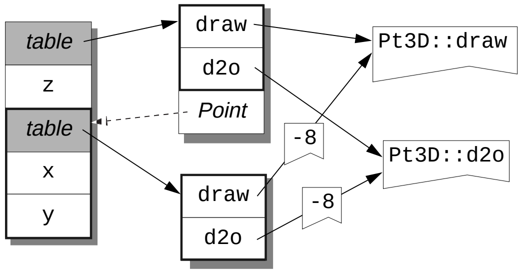
```
class Point {
  int x;
  int y;
  void draw();
  int d2o();
}

class ColorPoint extends Point {
  Color color;
  void draw();
  void reverse_video();
}
```



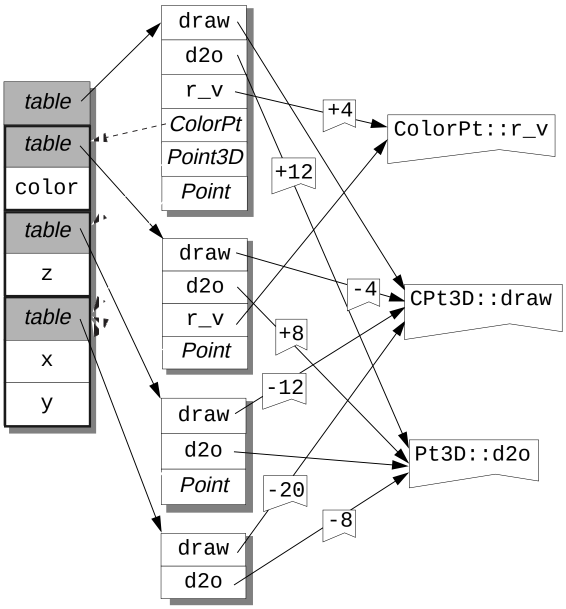
Example of shared superclasses (part 2)

```
class Point3D extends Point {
  int z;
  void draw();
  int d2o();
}
```



Example of shared superclasses (part 3)

```
class ColorPoint3D extends ColorPoint,
                           Point3D {
  void draw();
}
```



Restricted multiple inheritance

Some languages (e.g. Theta, Java, C#) support only single inheritance for implementation, but multiple inheritance from interfaces

If receiver of class type, then can apply prefixing and use regular single-inheritance sequence for method invocation and instance variable access

If receiver of interface type, then ...?

- may not have to consider instance variable access (e.g. in Theta, Java, C#)

Bidirectional object layout

Can adapt C++ MI layout rules, exploiting MI restrictions, using “bidirectional object layout” [Myers 95]

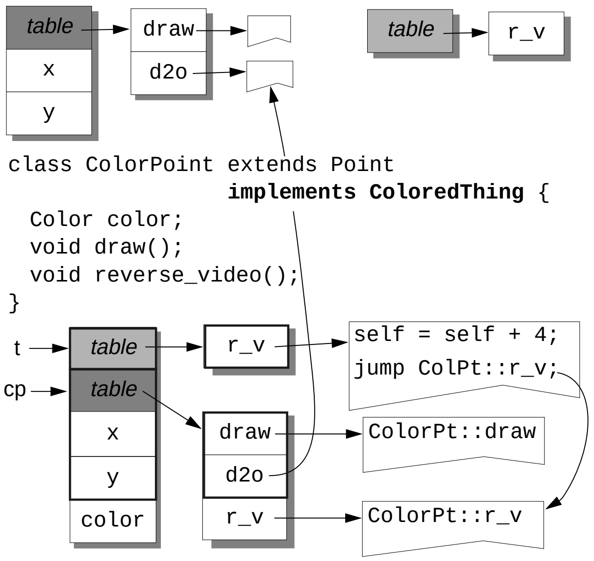
- only one superclass, whose layout is embedded in middle of subclass layout
  - main function table pointer for the class in middle
  - instance variables added to bottom end, prefixing style
  - function table pointers for interfaces added to top end, reverse prefixing style
- use interior pointers based on static type to select right function table pointer
  - pointer arithmetic only for interface types
  - no instance variables in interfaces, so no need for shared superclass offsets

+ fast instance variable access, message dispatching
+ simpler than regular C++ MI rules
+ can do compaction of function tables to reduce space cost
  - # of tables per object & size of tables
+ may even be used in C++ to optimize common case

Example

```
class Point {        interface ColoredThing {
  int x; int y;        void reverse_video();
  void draw();       }
  int d2o();
}
```

```
class ColorPoint extends Point
                  implements ColoredThing {
  Color color;
  void draw();
  void reverse_video();
}
```



Summary

language model	impl. strategy	instance var access	method invocation
single inheritance	prefixing	1 instruction	3 instructions
multiple inheritance	embedding/ offsets	1 instruction	5 instructions
	embedding/ trampolines	1 instruction	3 or 5 instructions
MI with shared superclasses	embedding/ trampolines/ class offsets	1, 2, or 4 instructions	3 or 5 instructions
restricted MI	bidirectional /trampolines	1 instruction	3 or 5 instructions

Notes:

- instruction sequences fare poorly on pipelined processors: data-dependent loads & jumps
- type-casts have run-time cost, with multiple inheritance
- multiple inheritance leads to interior pointers; bad for GC, debugging, object identity testing, ...
- multiple inheritance leads to multiple function table pointers, particularly with shared superclasses
- no inlining of dynamically-dispatched calls

Limitations of table-based techniques

Table-based techniques work well when:

- have static type information to use to map message/ instance variable names to offsets in tables/objects
  - not true in dynamically typed languages
- cannot extend classes with new operations except via subclassing
  - not true in languages with open classes (e.g. MultiJava [Clifton *et al.* 00]) or multiple dispatching (e.g. CLOS, Dylan, Cecil)
- cannot modify classes dynamically
  - not true in fully reflective languages (e.g. Smalltalk, Self, CLOS)
- memory loads and indirect jumps are inexpensive
  - increasingly less true with faster hardware

Dynamic table-based implementations

Standard implementation: global hash table in runtime system

- indexed by class × msg
- filled dynamically as program runs
- can be flushed after reflective operations
- + reasonable space cost
- + incremental
- – fair average-case dispatch time, poor worst-case time

Refinement: hash table per message name

- each call site knows statically which table to consult
- + faster dispatching

Alternative refinement: fixed-size hash table per class

- each call site knows statically which bucket offset to search, e.g. for invocations on Java interfaces [Alpern et al. 01]

Inline caching

Give each dynamically-dispatched call site its own small method lookup cache

- \+ call site knows its message name
- \+ cache is isolated from other call sites

Trick: use machine call instruction itself as a one-element cache

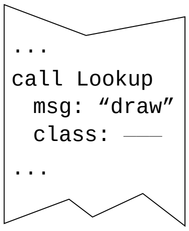
- initially: call runtime system's `Lookup` routine
- `Lookup` routine patches call instruction to branch to invoked method
  - record receiver class
- next time through, jump directly to expected target method
  - method checks whether current receiver class is same as last receiver class
  - if so, then cache hit (90-95% frequency, for Smalltalk)
  - if not, then call `Lookup` and rebind cache

- \+ fast dispatch sequence if cache hit (≈4 instructions plus call)
- \+ hardware call prefetching works well
- – exploits self-modifying code
- – low performance if not a cache hit

[Deutsch & Schiffman 84]

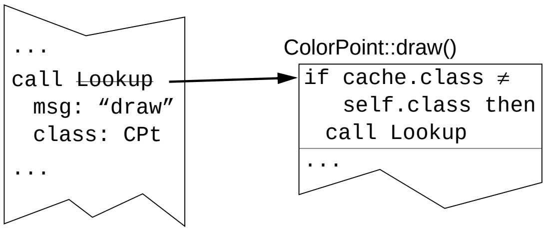
Example of inline caching

Initially:



```
...
call Lookup
 msg: “draw”
 class: ——
...
```

After caching target method:



...
call ~~Lookup~~
 msg: “draw”
 class: CPt
...

ColorPoint::draw()

```
if cache.class ≠
   self.class then
  call Lookup
...
```

Polymorphic inline caching (PIC)

Idea: support a multi-element cache by generating a call-site-specific dispatcher stub

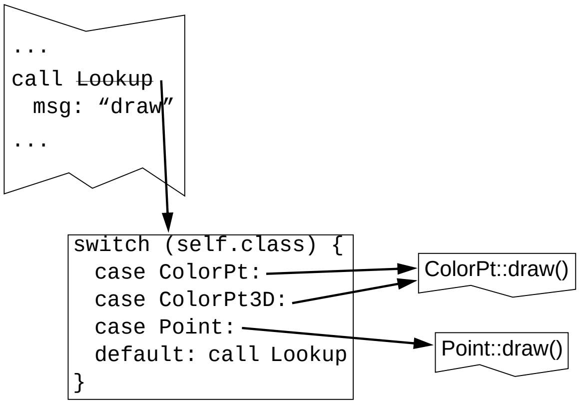
- \+ fast dispatching even if several classes are common
- – still slow performance if many classes equally common
- – some space cost

Foreshadowing:
dispatching stubs record dynamic profile data
of which receiver classes occur at which call sites

[Hölzle *et al.* 91]

Example of polymorphic inline caching

After a few receiver classes:



...
call ~~Lookup~~
 msg: “draw”
...

```
switch (self.class) {
  case ColorPt:
  case ColorPt3D:
  case Point:
  default: call Lookup
}
```

Implementing the dispatcher stub switch

In original PIC design, switch implemented with a linear chain of class identity tests

Alternatively, can implement with a binary search, exploiting ordering of integer class IDs or addresses

+ avoid worst-case behavior of long linear searches
+ a single test can direct many classes to same target method
– requires global knowledge to construct dispatchers

In traditional compilers, switch implemented with a jump table, akin to C++ dispatch tables

Can blend table-based lookups, linear search, and binary search [Chambers & Chen 99]

- exploit available static analysis of possible receiver classes, profile information of likely receiver classes
- construct dispatcher best balancing expected dispatching speed against dispatch space cost

---

Handling multiple dispatching

Languages with multimethods (e.g. CLOS, Dylan, Cecil) allow methods to dispatch on the run-time classes of any of the arguments

- call sites do not know statically which arguments may be dispatched upon

Implementation schemes:

- hash table indexed by *N* keys [Kiczales & Rodriguez 89]
- *N*-deep tree of hash tables, each indexed by 1 key [Dussud 89]
- *N*-deep DAG of 1-key dispatches [Chen & Turau 94, Chambers & Chen 99]
- compressed *N*+1-dimensional dispatch table [Amiel *et al.* 94, Pang *et al.* 99]

Probably more efficient to support multimethods directly than if simulated with double-dispatching [Ingalls 86] or visitor pattern [Gamma *et al.* 95]