These notes are not yet complete.

1 Inline caching

2 Sparse dispatch tables

3 Binary decision trees

These three topics are covered in the slides.

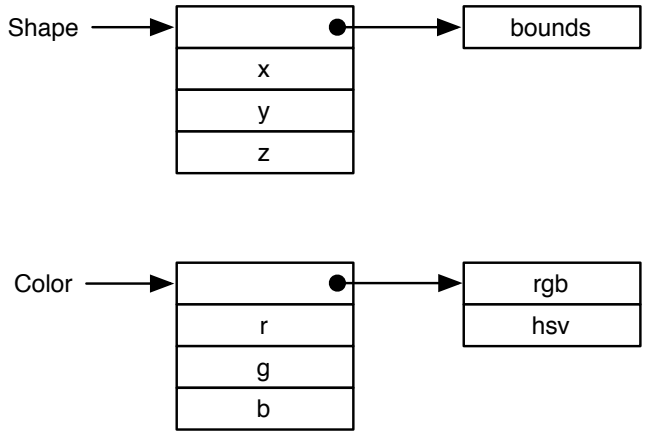
4 Using multiple dispatch tables for separate compilation

To deal with method index conflicts among superclasses, C++ may use multiple dispatch tables per object, and multiple references to the object. Which dispatch is to be used depends on which references to the object is used.

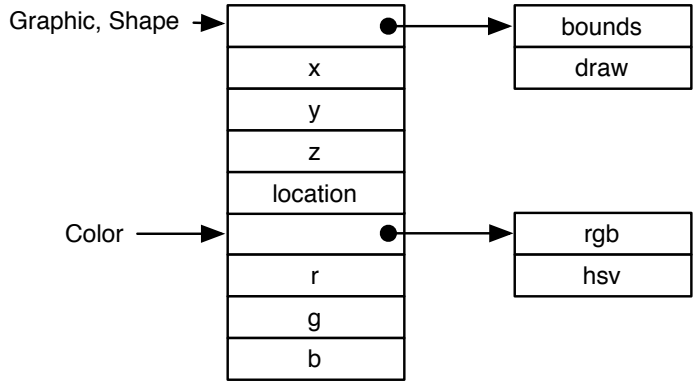
Different C++ implementations use different object layouts, but here is one possibility. Consider the following three classes:

```
class Shape {
  bounds()
  x,y,z: num
}
class Color {
  rgb()
  hsv()
  r,g,b: num
}
class Graphics extends Shape, Color {
  draw()
  location: int
}
```

For separate compilation the method indices for `Shape` and `Color` have to both be assigned independently. So both start methods in their dispatch table at zero:



We can merge both these layouts into a single object, but we need separate dispatch tables because `bounds()` and `rgb()` use the same method index:



There are two distinct "views" of the object, one as either a `Graphic` or a `Shape`, and one as a `Color`. To switch between these views, some computation is required. For example, we might subsumption to view a `Graphic` as a `Color`:

```
Graphic = new Graphic();
Color c = g;
```

We might expect that the assignment `c=g` involves no computation, but in fact it is necessary to add 40 to the address of `g`.

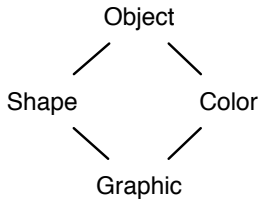
The result is fast dispatch in the usual case, but high per-object overhead, since we have two dispatch table pointers per object rather than just one. Supporting pointers to the interior of objects makes garbage collectors more complex and probably a little slower.

It's possible to put the methods of `Color` also into the `Graphic` dispath table, but since different class code expects different views of the receiver object, a *trampoline* is needed to bump the receiver pointer to the correct view.

This layout merges the dispatch tables for `Graphic` and `Shape`. In general, one can merge a class with that of one of its superclasses or implemented interfaces. There are more complex schemes for merging multiple dispatch tables more effectively, such as bidirectional dispatch tables. With a bidirectional layout, a class hierarchy that uses multiple inheritance only to allow a class to extend one other class and to implement one interface requires only a single dispatch table. This is possible by having the dispatch table grow in opposite directions.

5 Fields and multiple inheritance

Even the offsets to fields can conflict with with multiple inheritance. For eaxmple, consider this inheritance hierarchy:

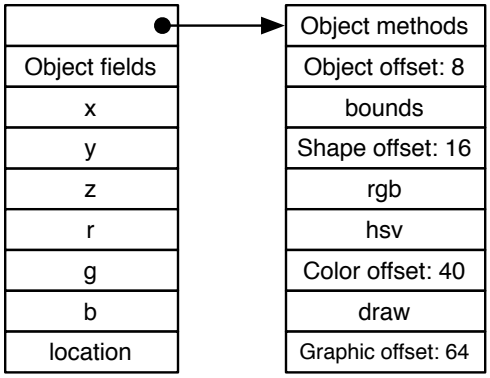


The code of both `Shape` and `Color` might need access to the fields of `Object`. But in a `Graphic` object, those fields can't be located at the same offset from the `Shape` and `Color` fields as in the `Shape` and `Color` object layouts.[1]

[1] Actually, C++ offers a version of "non-virtual" inheritance in which the fields *are* located at the same offset, but at the cost of duplicating the `Object` fields, which has strange semantics.

One way to solve the problem is to introduce internal pointers within the object between different views of the same object. This gives fast access to the fields of the current class view, and imposes no space or time overhead when inheritance is not being used. However, it has high per-object overhead even when single inheritance is being used. And internal pointers are a demanding feature that probably make the garbage collector slower.

A probably better idea is to store the offsets to fields in the dispatch table. Each field has a dispatch table index that can be consulted to find the field. Dispatch table indices can be assigned using graph coloring or by using multiple dispatch tables. For example, the following figure shows how the object layout might look assuming that dispatch table indices are assigned using graph coloring, so that there is a single dispatch table.



The sequence to access a field is more expensive than the usual indexed load. Before multiple inheritance, an access like `o.f` could be implemented as `mov`k_f`(o)`, t, where k_f is a known constant offset. With this object layout, accesses are more complex:

```
mov (o), t2
mov m_f(t2), t3
mov k_f(t2, t3), t1
```

The offset m_f is the location in the dispatch table of the field offset; the offset k_f is the offset within the subobject of the particular field. Since the offset `t3` is a constant, CSE can be used to avoid fetching it more than once.

This approach has much lower space overhead than using internal pointers, and access to fields from other class views are faster. However, access to fields of the current class view is slower, and there is a performance penalty even when inheritance is not being used.