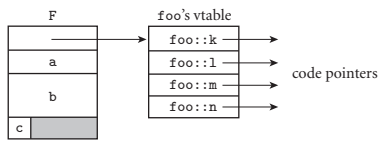
==========
Implementation of classes

Data members of classes are implemented just like structs (records). With (single) inheritance, derived classes have extra fields at the end. A pointer to the parent and a pointer to the child contain the same address -- the child just knows that the struct goes farther than the parent does.

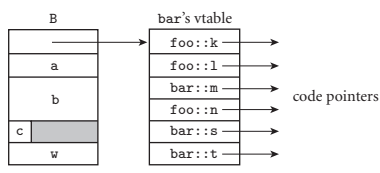
Non-virtual functions require no extra space at run time; the compiler just calls the appropriate version, based on type of variable. Member functions are passed an extra, hidden, initial parameter: 'this' (called 'current' in Eiffel and 'self' in Smalltalk).

Virtual functions are the only thing that requires any trickiness. They are implemented by creating a dispatch table ("vtable") for the class and putting a pointer to that table in the data of the object.Objects of a derived class have a different vtable. In that table, functions defined in the parent come first, though some of the pointers point to overridden versions. You could put the whole vtable table in the object itself. That would save a little time, but potentially waste a ***lot*** of space.

```
class foo {
    int a;
    double b;
    char c;
public:
    virtual void k( ...
    virtual int l( ...
    virtual void m();
    virtual double n( ...
    ...
} F;
```




```
class bar : public foo {
    int w;
public:
    void m() override;
    virtual double s( ...
    virtual char *t( ...
    ...
} B;
```



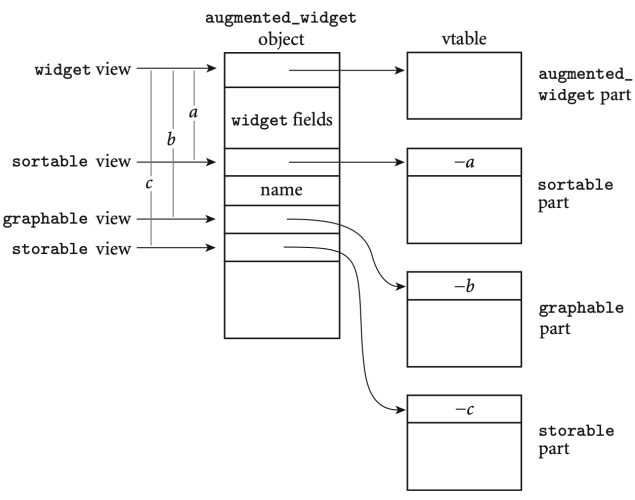

The C++ philosophy is to avoid run-time overhead whenever possible. (Sort of the legacy from C). That's why non-virtual functions are the default. Most other OO languages have much more run-time support.

Note that if you can query the type of an object, then you need to be able to get from the object to run-time type info. The standard implementation technique is to put a pointer to the type info at the beginning of the vtable. Of course you only have a vtable in C++ if your class has virtual functions. That's why you can't do a dynamic_cast on a pointer whose static type doesn't have virtual functions.

---
Mix-in inheritance

Simpler to implement than true multiple inheritance. Each class can have one "real" parent and an arbitrary number of ***interfaces***, each of which is fully abstract: no data members (other than statics); no non-pure-virtual methods. Now you create an extra vtable for each interface your object supports, and you embed pointers to these vtables among the data of each object. Each interface vtable begins with a field that gives the offset back from the vtable pointer to the beginning of the object in which that pointer appears:

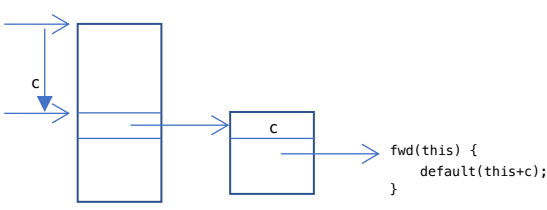
```
class widget {...

interface sortable {...
interface graphable {...
interface storable {...

class named_widget extends widget implements sortable { ...

class augmented_widget extends named_widget
                    implements graphable, storable {...
```




The `augmented_widget` part of the vtable includes the (non-interface) methods of `widget` and `named_widget`.

If the compiler needs to pass an `augmented_widget` to a method that expects a `graphable`, it passes the `graphable` view. (Likewise `sortable` or `storable`.) The method assumes the thing it was passed begins with a vtable pointer. It dereferences this pointer to find the vtable, then pulls the offset out of the first word of the vtable and subtracts it from the provisional '`this`' it was passed, to get the "real" '`this`', which it can pass to other methods.

Classic Java also allows static fields in Interfaces.
Starting with Java 8, Interfaces can have
- static methods
  - Straightforward: no access to `this`
- default methods
  - Designed to allow extension of an interface without rewriting all existing uses of that interface.
  - Implementation is a little tricky.
    - no access to members other than the methods and static fields of the interface itself
    - ***does*** need access to vtable, however: for each class that needs the default code, the compiler generates a static, class-specific forwarding routine that accepts the concrete-class-specific `this` parameter, adds back in the `this` correction that the regular calling sequence just subtracted out, and passes the resulting pointer-to-vtable-pointer to the default method.




---
For true multiple inheritance, see the PLP companion site.