9 Data Abstraction and Object Orientation

9.5 Multiple Inheritance

EXAMPLE 9.50
Deriving from two base classes (reprise)

Recall our simple example in C++:

```
class student : public person, public gp_list_node { ...
```

To implement multiple inheritance, we must be able to generate both a "`person` view" and a "`gp_list_node` view" of a `student` object on demand—for example, when assigning a reference to a `student` object into a `person` or `gp_list_node` variable. For one of the base classes (`person`, say) we can do the same thing we did with single inheritance: let the data members of that base class lie at the beginning of the representation of the derived class, and let the virtual methods of that base class lie at the beginning of the vtable. Then when we assign a reference to a `student` object into a `person` variable, code that manipulates the `person` variable will just use a prefix of the data members and the vtable. ■

EXAMPLE 9.51
(Nonrepeated) multiple inheritance

For the other base class (`gp_list_node`) things get more complicated: we can't put *both* base classes at the beginning of the derived class. One possible solution is shown in Figure © 9.7. It is based loosely on the implementation described by Ellis and Stroustrup [ES90, Chap. 10]. Because the `gp_list_node` fields of a `student` follow the `person` fields, the assignment of a reference to a `student` object into a variable of type `gp_list_node*` requires that we adjust our "view" by adding the compile-time constant offset *d*.

The vtable for a `student` is broken into two parts. The first part lists the virtual methods of the derived class and the first base class (`person`). The second part lists the virtual methods of the second base class. (We have already introduced a method, `print_mailing_label`, defined in class `person`. We may similarly imagine that `gp_list_node` defines a virtual method `debug_print` that is supposed to dump a printable representation of the contents of the node to standard output.) Generalization to three or more base classes is straightforward; see Exercise © 9.19.

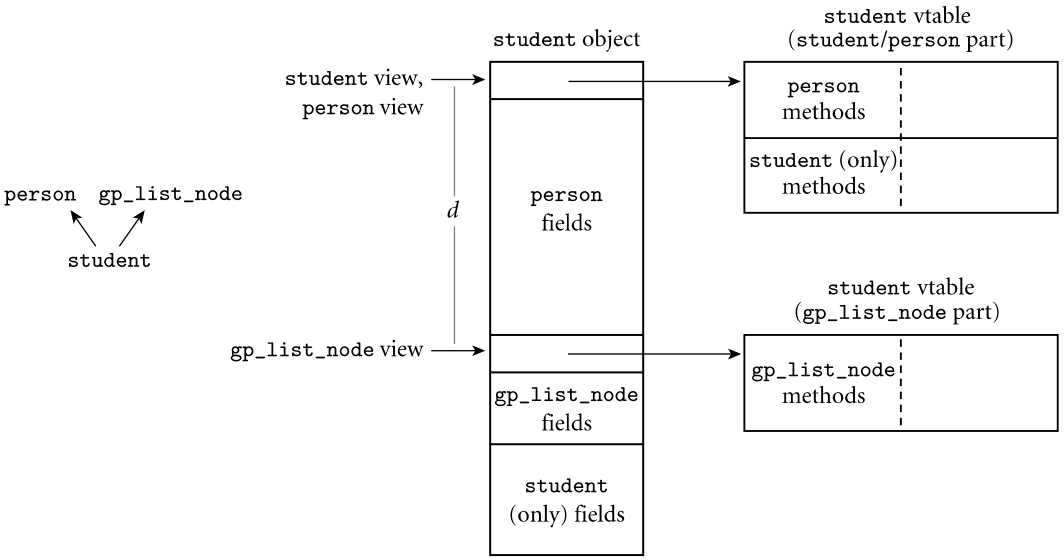


Figure 9.7 Implementation of (nonrepeated) multiple inheritance. The size *d* of the `person` portion of the object is a compile-time constant. We access the `gp_list_node` portion of the vtable by adding *d* to the address of a `student` object before indirecting. Likewise, we create a `gp_list_node` view of a `student` object by adding *d* to the object's address. Each vtable entry consists of both a method address and a "`this` correction" value equal to the signed distance between the view through which the vtable was accessed and the view of the class in which the method was defined.

Every data member of a `student` object has a compile-time-constant offset from the beginning of the object. Likewise, every virtual method has a compile-time-constant offset from the beginning of one of the parts of the vtable. The address of the `person`/`student` portion of the vtable is stored in the beginning of the object. The address of the `gp_list_node` portion of the vtable is stored at offset *d*. Note that both parts of the vtable are specific to class `student`. In particular, the `gp_list_node` part of the vtable is *not* shared by objects of class `gp_list_node`, because the contents of the tables will be different if `student` has overridden any of `gp_list_node`'s virtual methods. ■

EXAMPLE 9.52

Method invocation with multiple inheritance

To call the virtual method `print_mailing_label`, originally defined in `person`, we can use a code sequence similar to the one shown in Section 9.4.3 for single inheritance. To call a virtual method originally defined in `gp_list_node`, we must first add the offset *d* to our object's address, in order to find the address of the `gp_list_node` portion of the vtable. Then we can index into this `gp_list_node` vtable to find the address of the appropriate method to call. But we are left with one final problem: what is the appropriate value of `this` to pass to the method?

As a concrete example, suppose that `student` does not override `debug_print` (even though it probably ought to). If our object is of class `student`, we should pass a `gp_list_node` view of it to `debug_print`: the address of the object, plus *d*. If, however, our object is of some class (`transfer_student`, perhaps)

that does override `debug_print`, then we should pass a `transfer_student` view to `debug_print`. If we are accessing our object through a variable (a reference or a pointer) whose methods are dynamically bound, then we can't tell at compile time which one of these cases applies. Worse yet, we may not even know how to generate a `transfer_student` view if we have to: class `transfer_student` may not have been invented when this part of our code was compiled, so we certainly don't know how far into it the `gp_list_node` fields appear! ■

EXAMPLE 9.53
`This` correction

A common solution is for vtable entries to consist of a *pair* of fields. One is the address of the method's code; the other is a "`this` correction" value, to be added to the view through which we found the vtable. Returning to Figure 9.7, the "`this` correction" field of the vtable entry for `debug_print` would contain $-d$ if `debug_print` was overridden by `student`, and zero otherwise. In the `gp_list_node` part of the vtable for the (yet to be written) class `transfer_student`, the "`this` correction" field might contain some other value $-e$. In general, the "`this` correction" is the distance between the view of the class in which the method was *declared* (and through which we accessed the vtable) and the view of the class in which the method was *defined* (and which will therefore be expected by the subroutine's implementation).

If variable `my_student` contains a reference to (a student view of) some object at run time, and if `debug_print` is the third virtual method of `gp_list_node`, then the code to call `my_student.debug_print` would now look something like this:

```
r1 := my_student                  -- student view of object
r1 := r1 + d                      -- gp_list_node view of object
r2 := *r1                         -- address of appropriate vtable
r3 := *(r2 + (3−1) × 8)           -- method address
r2 := *(r2 + (3−1) × 8 + 4)       -- this correction
r1 := r1 + r2                     -- this
call *r3
```

Here we have assumed that both method addresses and `this` corrections are four-bytes long. On a typical machine this code is three instructions (including one memory access) longer than the code required with single inheritance, and five instructions (including three memory access) longer than a call to a statically identified method. ■

9.5.1 Semantic Ambiguities

EXAMPLE 9.54
Methods found in more than one base class

In addition to implementation complexities (only some of which we have discussed so far), multiple inheritance introduces potential semantic problems. Suppose that both `gp_list_node` and `person` define a `debug_print` method. If we have a variable `s` of type `student*` and we call `s->debug_print`, which version of the method should we get? In CLOS and Python, we get the version

from the base class that appeared first in the derived class's header. In Eiffel, we get a static semantic error if we try to define a derived class with such an ambiguity. In C++, we can define the derived class, but we get a static semantic error if we attempt to use a member whose name is ambiguous. In Eiffel we can use the feature renaming mechanism to get rid of naming conflicts when defining a derived class. In C++ we must redefine the ambiguous member explicitly:

```
void student::debug_print() {
    person::debug_print();
    gp_list_node::debug_print();
}
```

Here we have chosen to call the `debug_print` routines of both base classes, using the `::` scope resolution operator to name them. We could of course have chosen to call just one, or to write our own code from scratch. We could even arrange for access to both routines by giving them new names:

```
void student::debug_print_person() {
    person::debug_print();
}
void student::debug_print_list_node() {
    gp_list_node::debug_print();
}
```

EXAMPLE 9.55

Overriding an ambiguous method

Things are a little messier if either or both of the identically named base class methods are virtual, and we want to override them in the derived class. Following Stroustrup [Str97, Sec. 25.6], we can solve the problem by interposing an "interface" class between each base class and the derived class:

```
class person_interface : public person {
    virtual void debug_print_person() = 0;
    void debug_print() { debug_print_person(); }
        // overrides person::debug_print
};
```

DESIGN & IMPLEMENTATION

The cost of multiple inheritance

The implementation we have described for multiple inheritance, using `this` corrections in vtables, has the unfortunate property of increasing the overhead of all virtual method invocations, even in programs that do not make use of multiple inheritance. This sort of mandatory overhead is something that language designers (and the designers of systems languages in particular) generally try to avoid; as a matter of principle, complex special cases should not reduce the efficiency of the simpler common case. Fortunately, there are other implementations of multiple inheritance (see Exercise 9.25) in which the cost of modifying `this` is paid only when the correction is nonzero.

```
class list_node_interface : public gp_list_node {
    virtual void debug_print_list_node() = 0;
    void debug_print() { debug_print_list_node(); }
        // overrides gp_list_node::debug_print
};
class student : public person_interface, public list_node_interface {
public:
    void debug_print_person() { ...
    void debug_print_list_node() { ...
    ...
};
```

We leave it as an exercise (9.20) to show what happens if we assign a `student` object into a variable `p` of type `person*` and then call `p->debug_print()`. ■

EXAMPLE 9.56
Repeated multiple inheritance

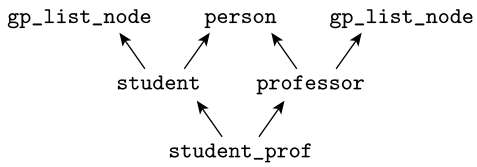
A more serious ambiguity arises when a class *D* inherits from two base classes, *B* and *C*, both of which inherit from some common base class *A*. In this situation, should an object of class *D* contain one instance of the data members of class *A* or two? The answer would seem to be program-dependent. For example, suppose in our administrative computing system that we would like to keep all professors in the same department on a linked list. Like class `student`, we might want class `professor` to inherit from both `person` and `gp_list_node`:

```
class professor : public person, public gp_list_node { ...
```

Furthermore, suppose that professors occasionally take courses as nonmatriculated students. In this case we might want a new class that supports both sets of operations:

```
class student_prof : public student, public professor { ...
```

Class `student_prof` inherits from `person` and `gp_list_node` twice, through both `student` and `professor`. If we think about it, we probably want a `student_prof` to have *one* instance of the data members of class `person`—one name, one university ID number, one mailing address—and *two* instances of the data members of class `gp_list_node`—two predecessors and two successors, one set for linking into the list of nonmatriculated students and another for linking into the faculty list for some department:



The `gp_list_node` case—separate copies from each branch of the inheritance tree—is known as *replicated inheritance*. The `person` case—a single copy from both branches of the tree—is known as *shared* inheritance. Both are forms of *repeated inheritance*. ■

EXAMPLE 9.57
Shared inheritance in C++

Replicated inheritance is the default in C++. Shared inheritance is the default in Eiffel. Shared inheritance can be obtained in C++ by specifying that a base class is `virtual`:

```
class student : public virtual person, public gp_list_node { ...
class professor : public virtual person, public gp_list_node { ...
```

In this case the members of class `person` are shared when inherited over multiple paths, while the members of class `gp_list_node` are replicated. ■

EXAMPLE 9.58
Replicated inheritance in Eiffel

Replicated inheritance of individual features can be obtained in Eiffel through the renaming mechanism described in Section 9.2.2:

```
class student inherit person; gp_list_node ...
class professor inherit person; gp_list_node ...

class student_prof
inherit
    student
        rename
            prev as prev_student,
            next as next_student
        end;
    professor
        rename
            prev as prev_prof,
            next as next_prof
        end
feature
    ...
end -- class student_prof
```

Features inherited with different final names are replicated; features inherited with the same final name are shared. Multiple inheritance in CLOS is always shared, unless the user interposes interface classes as shown above explicitly; there is no other renaming mechanism. ■

9.5.2 Replicated Inheritance

EXAMPLE 9.59
Using replicated inheritance

Replicated inheritance introduces no serious implementation problems beyond those of nonrepeated multiple inheritance. As shown in Figure 9.8, an object (in this case of class `D`) that inherits a base class (`A`) over two different paths in the inheritance tree has two copies of `A`'s data members in its representation, and a set of entries for the virtual methods of `A` in each of the parts of its vtable. Creation of a `B` view of a `D` object (e.g., when assigning a pointer to a `D` object into a `B*` variable) would not require the execution of any code. Creation of a `C` view (e.g., when assigning into a `C*` variable) would require the addition of offset *d*.

Because of ambiguity, we cannot access `A` members of a `D` object by name. We can access them, however, if we assign a pointer to a `D` object into a `B*` or `C*`

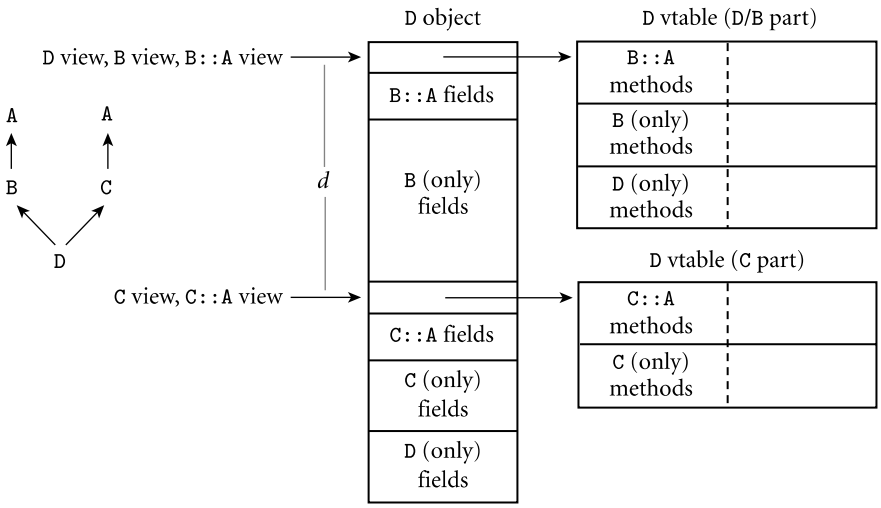


Figure 9.8 **Implementation of replicated multiple inheritance.** Each base class contains a complete copy of class `A`. As in Figure 9.7, the vtable for class `D` is split into two parts, one for each base class, and each vtable entry consists of a (method address, **`this`** correction) pair.

variable. Similarly, a pointer to a `D` object cannot be assigned into an `A` pointer directly: there would be no basis on which to choose the `A` for which to create a view. We can, however, perform the assignment through a `B*` or `C*` intermediary:

```
class A { ...
class B : public A { ...
class C : public A { ...
class D : public B, public C { ...
...
A* a;   B* b;   C* c;   D* d;
a = d;  // error; ambiguous
b = d;  // ok
c = d;  // ok
a = b;  // ok; a := d's B's A
a = c;  // ok; a := d's C's A
```

As described in Example 9.53, vtable entries will need to consist of (method address, `this` correction) pairs. ■

9.5.3 Shared Inheritance

Shared inheritance introduces a new opportunity for ambiguity and additional implementation complexity. As in the previous subsection, assume that `D` inher-

EXAMPLE 9.60
Overriding methods with shared inheritance

its from B and C, both of which inherit from A. This time, however, assume that A is shared:

```
class A {
public:
    virtual void f();
    ...
};
class B : public virtual A { ...
class C : public virtual A { ...
class D : public B, public C { ...
```

The new ambiguity arises if B or C overrides method f, declared in A: which version (if any) does D inherit? C++ defines a reference to f to be unambiguous (and therefore valid) if one of the possible definitions *dominates* the others, in the sense that its class is a descendant of the classes of all the other definitions. In our specific example, D can inherit an overridden version of f from either B or C. If both of them override it, however, any attempt to use f from within D's code will be a static semantic error. Eiffel provides comparatively elaborate mechanisms for controlling ambiguity. A class that inherits an overridden method over more than one path can specify which one it wants. Alternatively, through renaming, it can retain access to all versions. ■

EXAMPLE 9.61
Implementation of shared inheritance

To implement shared inheritance we must recognize that because a single instance of A is a part of both B and C, we cannot make the representations of both B and C contiguous in memory. In Figure 9.9, in fact, we have chosen to make neither B nor C contiguous. We insist, however, that the representation of every B, C, or D object (and every B, C, or D view of an object of a derived class) contain the address of the A part of the object at a compile-time-constant offset from the beginning of the view. To access a data member of A, we first indirect through this address, and then apply the offset of the member within A. To call the nth virtual method declared in A, we execute the following code.

```
r1 := my_D_view                  -- original view of object
r1 := *(r1 + 4)                  -- A view
r2 := *r1                        -- address of A part of vtable
r3 := *(r2 + (n − 1) × 8)        -- method address
r2 := *(r2 + (n − 1) × 8 + 4)    -- this correction
r1 := r1 + r2                    -- this
call *r3
```

This code sequence is the same number of instructions in length as our sequence for nonvirtual base classes (Example 9.53), but involves one more memory access (to indirect through the A address). The code will work with any D view of any object, including an object of a class derived from D, in which the D and A views might be more widely separated. The constant 4 in the second line assumes four-byte addresses, with the address of D's A part located immediately after D's initial vtable address. In an object with more than one virtual base class, the ad-

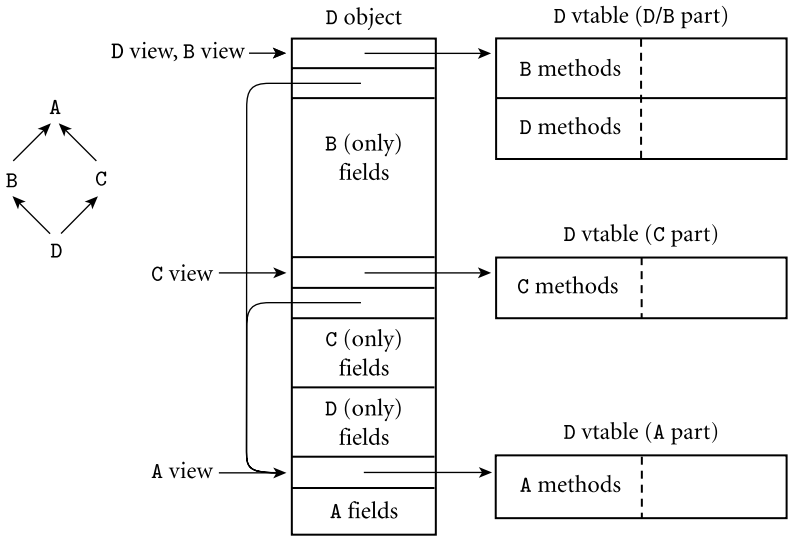


Figure 9.9 Implementation of shared multiple inheritance. Objects of class `B`, `C`, and `D` contain the address of their `A` components at a compile-time constant offset (in this case, immediately after the vtable address). As in Figures 9.7 and 9.8, `this` corrections for virtual methods in vtable entries are relative to the view of the class in which the method was declared (i.e., through which the vtable was accessed).

dress of the part of the object corresponding to each such base would be found at a different offset from the beginning of the object.

The implementation strategy of Figure 9.9 works in C++ because we always know when a base class is `virtual` (shared). For data members and virtual methods of nonvirtual base classes, we continue to use the (cheaper) lookup algorithms of Figures 9.7 and 9.8. In Eiffel, on the other hand, a feature that is inherited via replication at one level of the class hierarchy may be inherited via sharing later on. As a result, Eiffel requires a somewhat more elaborate implementation strategy (see Exercise 9.26).

We can avoid the extra level of indirection when accessing virtual methods of virtual base classes in C++ if we are willing to replicate portions of a class's vtable. We explore this option in Exercise 9.27.

9.5.4 Mix-In Inheritance

Before leaving the topic of multiple inheritance, we return briefly to the notion of a base class composed entirely of abstract methods, as mentioned in passing in Section 9.4.2. Such a class is called an *interface* in Java. It has neither data mem-

bers nor implementations of its methods.[1] It is therefore immune to most of the semantic ambiguities and implementation complexities of multiple inheritance.

Inheritance from one "real" base class and an arbitrary number of interfaces is known as *mix-in* inheritance—the virtual methods of the interface are "mixed into" the methods of the derived class. It may be stretching things a bit to speak of "inheriting" an interface, since the derived class must provide a definition for each of the interface's methods. Interfaces do, however, facilitate code reuse through polymorphism. If a formal parameter of a subroutine is declared to have an interface type, then any class that implements (inherits from) that interface can be passed as the corresponding actual parameter. The classes of objects that can legitimately be passed need not have a common class ancestor.

EXAMPLE 9.62
Mixing interfaces into a derived class

As an example, suppose that we have been given general purpose Java code that will sort objects according to some textual field, display a graphic representation of an object within a web browser window (hiding and refreshing as appropriate), and store references to objects by name in a dictionary data structure. Each of these capabilities would be represented by an interface. If we have already developed some complicated class of objects `widget`, we can make use of the general purpose code by mixing the appropriate interfaces into classes derived from `widget`, as shown in Figure 9.10. ■

EXAMPLE 9.63
Compile-time implementation of mix-in inheritance

As noted in Section 9.4.3, Java implementations usually look methods up by name at run time. In this case, the methods of an interface can simply be added to the method dictionary of any class that implements the interface. To implement mix-in inheritance without run-time method lookup, one simple approach is to augment the representation of objects of the class with the addresses of vtables for the implemented interfaces, as shown in Figure 9.11. Additional vtable pointers, like additional data members, are added to the end of the representation of objects of the base class to create the representation of the derived class. If interfaces and data members are added at several levels of the class hierarchy, then vtable pointers and data members may be interspersed at arbitrary offsets within objects. ■

CHECK YOUR UNDERSTANDING

42. Give a few examples of the semantic ambiguities that arise when a class has more than one base class.

43. Explain the distinction between replicated and shared multiple inheritance. When is each desirable?

44. Explain how even nonrepeated multiple inheritance introduces the need for multiple *views* of (the implementation of) an object, and for "`this` correction" fields in vtables.

---

1 Java actually does allow an interface to have data members, but such members are always constants; their values must be specified in the interface declaration.

```
public class widget { ...
}
interface sortable_object {
    String get_sort_name();
    bool less_than(sortable_object o);
    // All methods of an interface are automatically public.
}
interface graphable_object {
    void display_at(Graphics g, int x, int y);
    // Graphics is a standard library class that provides a context
    // in which to render graphical objects.
}
interface storable_object {
    String get_stored_name();
}
class named_widget extends widget implements sortable_object {
    public String name;
    public String get_sort_name() {return name;}
    public bool less_than(sortable_object o) {
        return (name.compareTo(o.get_sort_name()) < 0);
        // compareTo is a method of the standard library class String.
    }
}
class augmented_widget extends named_widget
        implements graphable_object, storable_object {
    ...         // more data members
    public void display_at(Graphics g, int x, int y) {
        ...     // series of calls to methods of g
    }
    public String get_stored_name() {return name;}
}
...
class sorted_list {
    public void insert(sortable_object o) { ...
    public sortable_object first() { ...
    ...
}
class browser_window extends Frame {
    // Frame is the standard library class for windows.
    public void add_to_window(graphable_object o) { ...
    ...
}
class dictionary {
    public void insert(storable_object o) { ...
    public storable_object lookup(String name) { ...
    ...
}
```

Figure 9.10 Interface classes in Java. By implementing the `sortable_object` interface in `named_widget` and the `graphable_object` and `storable_object` interfaces in `augmented_widget`, we obtain the ability to pass objects of those classes to and from such routines as `sorted_list.insert`, `browser_window.add_to_window`, and `dictionary.insert`.

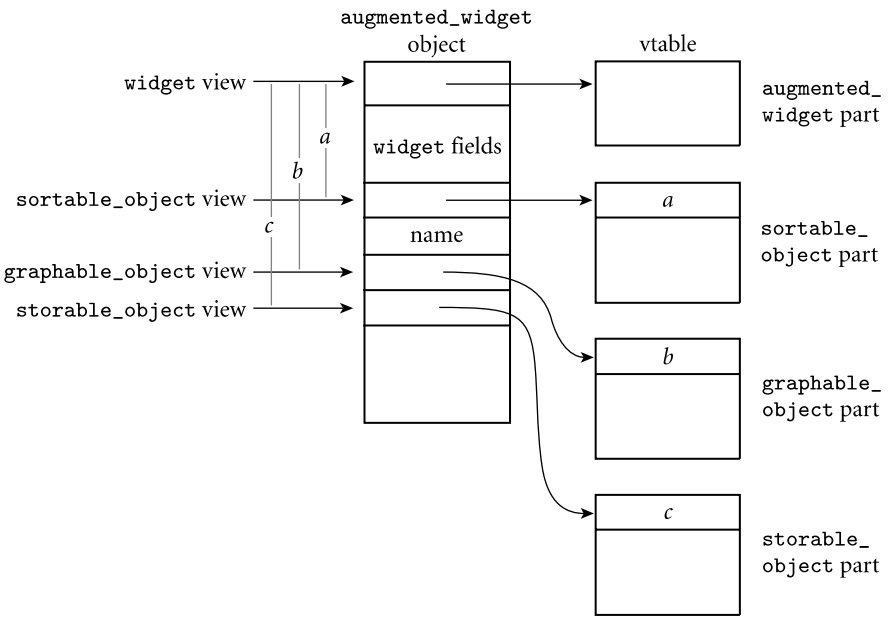


Figure 9.11 **Implementation of mix-in inheritance.** Objects of class `augmented_widget` contain four vtable addresses, one for the class itself (as in Figure 9.4), and three for the implemented interfaces. The view of the object that is passed to interface routines points directly at the relevant vtable pointer. The vtable then begins with a single `this` correction, used by all of its methods to regenerate a pointer to the object itself.

45. Explain how shared multiple inheritance introduces the need for an additional level of indirection when accessing fields of certain parent classes.

46. What is an *interface*, as defined by Java or C#? How is it related to *mix-in* style inheritance?

47. Why is mix-in inheritance simpler to implement than other styles of multiple inheritance?