10 Object Orientation

10.6 True Multiple Inheritance

EXAMPLE 10.56
Deriving from two base classes (reprise)

Recall our administrative computing example in C++:

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

To implement multiple inheritance, we must be able to generate both a "`person` view" and a "`system_user` view" of a `student` object on demand, for example when assigning a reference to a `student` object into a `person` or `system_user` 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 10.57
(Nonrepeated) multiple inheritance

For the other base class (`system_user`), 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 C-10.8. It is based loosely on the implementation described by Ellis and Stroustrup [ES90, Chap. 10]. Because the `system_user` fields of a `student` follow the `person` fields, the assignment of a reference to a `student` object into a variable of type `system_user*` 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 `system_user` defines a virtual method `print_stats` that is supposed to dump account statistics to standard output.) Generalization to three or more base classes is straightforward; see Exercise C-10.23.

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

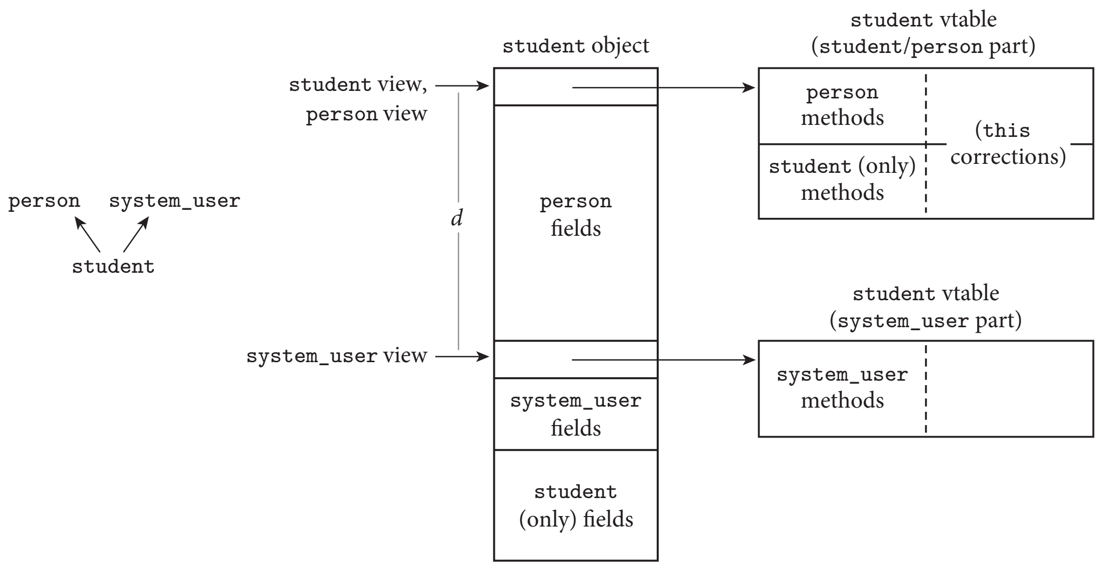

Figure 10.8 Implementation of (nonrepeated) multiple inheritance. The size d of the person portion of the object is a compile-time constant. We access the system_user portion of the vtable by adding d to the address of a student object before indirecting. Likewise, we create a system_user view of a student 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.

the person/student portion of the vtable is stored in the beginning of the object. The address of the system_user portion of the vtable is stored at offset d. Note that both parts of the vtable are specific to class student. In particular, the system_user part of the vtable is *not* shared by objects of class system_user, because the contents of the tables will be different if student has overridden any of system_user's virtual methods. ■

EXAMPLE 10.58
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 10.4.2 for single inheritance. To call a virtual method originally defined in system_user, we must first add the offset d to our object's address, in order to find the address of the system_user portion of the vtable. Then we can index into this system_user 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 print_stats (though it certainly could). If our object is of class student, we should pass a system_user view of it to print_stats: the address of the object, plus d. If, however, our object is of some class (transfer_student, perhaps) that does override print_stats, then we should pass a transfer_student view to print_stats. 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 `system_user` fields appear!

EXAMPLE 10.59
`this` correction

A common solution is for each vtable entry 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 C-10.8, the "`this` correction" field of the vtable entry for `print_stats` would contain $-d$ if `print_stats` was overridden by `student`, and zero otherwise. In the `system_user` 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 `print_stats` is the third virtual method of `system_user`, then the code to call `my_student.print_stats` would look something like this:

```
r1 := my_student                 -- student view of object
r1 := r1 + d                     -- system_user 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, that `this` is to be passed in `r1`, and that there are no other arguments. 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 accesses) longer than a call to a statically identified method.

DESIGN & IMPLEMENTATION

10.9 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 C-10.28) in which the cost of modifying `this` is paid only when the correction is nonzero.

10.6.1 Semantic Ambiguities

EXAMPLE 10.60
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 `system_user` and `person` define a `print_stats` method. If we have a variable `s` of type `student*` and we call `s->print_stats`, 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. To resolve the ambiguity, we can use the feature renaming mechanism in Eiffel to give different names to the inherited methods. In C++ we must redefine the conflicting method explicitly:

```
void student::print_stats() {
    person::print_stats();
    system_user::print_stats();
}
```

Here we have chosen to call the `print_stats` 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::print_person_stats() {
    person::print_stats();
}
void student::print_user_stats() {
    system_user::print_stats();
}
```

EXAMPLE 10.61
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 [Str13, Sec. 21.3.3], we can solve the problem by interposing an intermediate class between each base class and the derived class:

```
class person_interface : public person {
public:
    virtual void print_person_stats() = 0;
    void print_stats() { print_person_stats(); }
        // overrides person::print_stats
};
class system_user_interface : public system_user {
public:
    virtual void print_user_stats() = 0;
    void print_stats() { print_user_stats(); }
        // overrides system_user::print_stats
};
```

```
class student : public person_interface, public system_user_interface {
public:
    void print_person_stats() { ...
    void print_user_stats() { ...
    ...
};
```

We leave it as an exercise (C-10.24) to show what happens if we assign a `student` object into a variable `p` of type `person*` and then call `p->print_stats()`. ■

EXAMPLE 10.62 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 that professors, like students, are all given accounts in our administrative computing system. Then, like class `student`, we might want class `professor` to inherit from both `person` and `system_user`:

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

But now suppose that some professors take courses on occasion 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 from `system_user` twice, once each through `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 `system_user`—separate user accounts (with separate user ids, disk quotas, etc.) for the student and professor roles:

$$
\begin{tikzcd}
\texttt{system\_user} & & \texttt{person} & & \texttt{system\_user} \\
& \texttt{student} \arrow[ul] \arrow[ur] & & \texttt{professor} \arrow[ul] \arrow[ur] & \\
& & \texttt{student\_prof} \arrow[ul] \arrow[ur] & &
\end{tikzcd}
$$

The `system_user` 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 10.63 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 system_user { ...
class professor : public virtual person, public system_user { ...
```

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

EXAMPLE 10.64
Replicated inheritance in Eiffel

Replicated inheritance of individual features can be obtained in Eiffel by means of renaming:

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

class student_prof
inherit
    student
        rename
            user_id as student_user_id,
            disk_quota as student_disk_quota
        end;
    professor
        rename
            user_id as prof_user_id,
            disk_quota as prof_disk_quota
        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 in Example C-10.61 explicitly; there is no other renaming mechanism. ■

10.6.2 Replicated Inheritance

EXAMPLE 10.65
Using replicated inheritance

Replicated inheritance introduces no serious implementation problems beyond those of nonrepeated multiple inheritance. As shown in Figure C-10.9, 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*` 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:

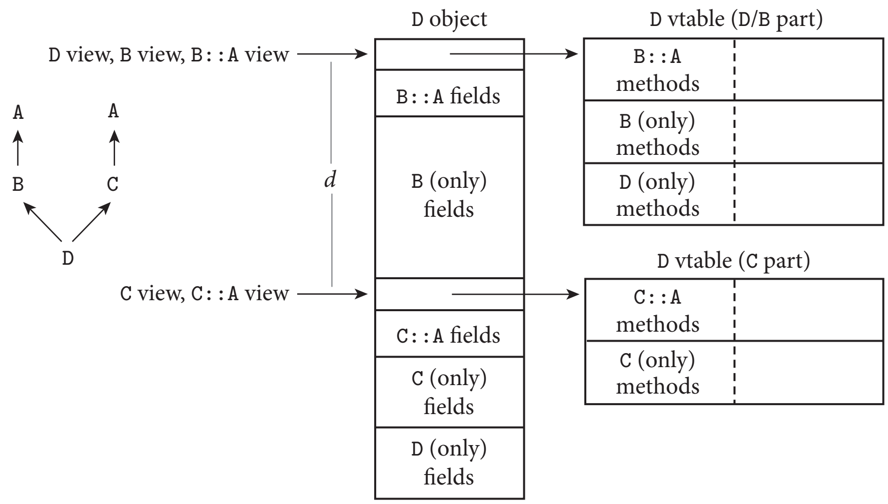

Figure 10.9 **Implementation of replicated multiple inheritance.** Each base class contains a complete copy of class `A`. As in Figure C-10.8, the vtable for class `D` is split into two parts, one for each base class, and each vtable entry consists of a ⟨method address, `this` correction⟩ pair.

```
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 C-10.59, vtable entries will need to consist of ⟨method address, `this` correction⟩ pairs. ■

10.6.3 Shared Inheritance

EXAMPLE 10.66

Overriding methods with shared inheritance

Shared inheritance introduces a new opportunity for ambiguity and additional implementation complexity. As in the previous subsection, assume that `D` inherits from `B` and `C`, both of which inherit from `A`. This time, however, assume that `A` is shared:

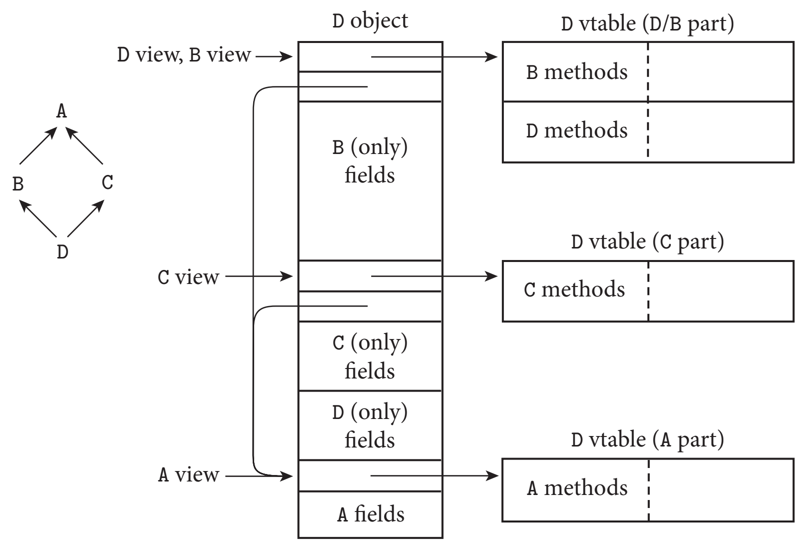

Figure 10.10 **Implementation of shared multiple inheritance.** Objects of class B, C, and D contain the address of their A components at a compile-time constant offset (in this case, immediately after the vtable address). As in Figures C-10.8 and C-10.9, `this` corrections for virtual methods in vtable entries are relative to the view of the class in which the method was declared (i.e., through which the vtable was accessed).

```
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 the version it wants. Alternatively, through renaming, it can retain access to all versions.

EXAMPLE 10.67
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 C-10.10, 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 *n*th 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 C-10.59), 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 4-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 address 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 C-10.10 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 C-10.8 and C-10.9. 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 C-10.29).

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 C-10.30.

✓ CHECK YOUR UNDERSTANDING

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

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

47. Explain how even nonrepeated multiple inheritance introduces the need for "`this` correction" fields in individual vtable entries.

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

49. Explain why true multiple inheritance is harder to implement than interface inheritance, traits, or mix-ins.

10 Object Orientation

10.7.1 The Object Model of Smalltalk

Smalltalk is heavily integrated into its programming environment. In fact, unlike all of the other languages mentioned in this book, a Smalltalk program does not consist of a simple sequence of characters. Rather, Smalltalk programs are meant to be viewed within the *browser* of a Smalltalk implementation, where font changes and screen position can be used to differentiate among various parts of a given program unit. Together with the contemporaneous Interlisp and Pilot/Mesa projects at PARC, the Smalltalk group shares credit for developing the now ubiquitous concepts of bit-mapped screens, windows, menus, and mice.

Smalltalk uses an untyped reference model for all variables. Every variable refers to an object, but the class of the object need not be statically known. As described in Section 10.3.1, every Smalltalk object is an instance of a class descended from a single base class named `Object`. All data are contained in objects. The most trivial of these are simple immutable objects such as `true` (of class `Boolean`) and `3` (of class `Integer`).

EXAMPLE 10.68
Operations as messages in Smalltalk

Operations are all conceptualized as *messages* sent to objects. The expression `3 + 4`, for example, indicates sending a `+` message to the (immutable) object `3`, with a reference to the object `4` as argument. In response to this message, the object `3` creates and returns a reference to the (immutable) object `7`. Similarly, the expression `a + b`, where `a` and `b` are variables, indicates sending a `+` message to the object referred to by `a`, with the reference in `b` as argument. If `a` happens to refer to `3` and `b` refers to `4`, the effect will be the same as it was in the case of the constants. ■

EXAMPLE 10.69
Mixfix messages

As described in Section 6.1, multiargument messages have multiword ("mixfix") names. Each word ends with a colon; each argument follows a word. The expression

```
myBox displayOn: myScreen at: location
```

sends a `displayOn: at:` message to the object referred to by variable `myBox`, with the objects referred to by `myScreen` and `location` as arguments. ■

EXAMPLE 10.70
Selection as an `ifTrue: ifFalse:` message

Even control flow in Smalltalk is conceptualized as messages. Consider the selection construct:

```
n < 0
    ifTrue: [abs <- n negated]
    ifFalse: [abs <- n]
```

This code begins by sending a `< 0` message (a `<` message with `0` as argument) to the object referred to by `n`. In response to this message, the object referred to by `n` will return a reference to one of two immutable objects: `true` or `false`. This reference becomes the value of the `n < 0` expression.

Smalltalk evaluates expressions left-to-right without precedence or associativity. The value of `n < 0` therefore becomes the recipient of an `ifTrue: ifFalse:` message. This message has two arguments, each of which is a *block*. A block in Smalltalk is a fragment of code enclosed in brackets. It is an immutable object, with semantics roughly comparable to those of a lambda expression in Lisp. To execute a block we send it a `value` message.

When sent an `ifTrue: ifFalse:` message, the immutable object `true` sends a `value` message to its first argument (which had better be a block) and then returns the result. The object `false`, on the other hand, in response to the same message, sends a `value` message to its second argument (the block that followed `ifFalse:`). The left arrow (`<-`) in each block is the assignment operator. Assignment is not a message; it is a side effect of evaluation of the right-hand side. As in expression-based languages such as Algol 68, the value of an assignment expression is the value of the right-hand side. The overall value of our selection expression will be the value of one of the blocks, namely a reference to `n` or to its additive inverse, whichever is non-negative. For the sake of convenience, Boolean objects in Smalltalk also implement `ifTrue:`, `ifFalse:`, and `ifFalse: ifTrue:` methods. ■

EXAMPLE 10.71
Iterating with messages

Iteration is modeled in a similar fashion. For enumeration-controlled loops, class `Integer` implements `timesRepeat:` and `to: by: do:` methods:

```
pow <- 1.
10 timesRepeat:
    [pow <- pow * n]

sum <- 0.
1 to: 100 by: 2 do:
    [:i | sum <- sum + (a at: i)]
```

The first of these code fragments calculates $\texttt{n}^{10}$. In response to a `timesRepeat:` message, the integer k sends a `value` message to the argument (a block) k times. The second code fragment sums the odd-indexed elements of the array referred to by `a`. In response to a `to: by: do:` message, the integer k behaves as one might expect: it sends a `value:` message to its third argument (a block) $\lfloor (t - k + b)/b \rfloor$ times, where t is the first argument and b is the second argument. Note the colon at the end of `value:`. The plain `value` message is unary; the `value:` message has an argument; it is understood by blocks that have a (single) formal parameter. In our loop example, the integer `1` sends the messages `value: 1`, `value: 3`, `value: 5`, and so on to the block `[:i | sum <- sum + (a at: i)]`. The `:i |` at the beginning of the block is its formal parameter. The `at:` message is understood by arrays. For iteration with a step size of one, integers also provide a `to: do:` method. ■

EXAMPLE 10.72
Blocks as closures

Because it is an object, a block can be referred to by a variable:

```
b <- [n <- n + 1].          " b is now a closure"
c <- [:i | n <- n + i].     " so is c"
...
b value.                    " increment n by 1"
c value: 3.                 " increment n by 3"
```

A block with two parameters expects a `value: value:` message. A block with j parameters expects a message whose name consists of the word `value:` repeated j times. Comments in Smalltalk are double-quoted (strings are single-quoted). ■

EXAMPLE 10.73
Logical looping with messages

For logically controlled loops, Smalltalk relies on the `whileTrue:` message, understood by blocks:

```
tail <- myList.
[tail next ~~ nil]
    whileTrue: [tail <- tail next]
```

This code sets `tail` to the final element of `myList`. The double-tilde (`~~`) operator means "does not refer to the same object as." The method `next` is assumed to return a reference to the element following its recipient. In response to a `whileTrue:` message, a block sends itself a `value` message. If the result of that message is a reference to `true`, the block sends a `value` message to the argument of the original message and repeats. Blocks also implement a `whileFalse:` method. ■

EXAMPLE 10.74
Defining control abstractions

The blocks of Smalltalk allow the programmer to construct almost arbitrary control-flow constructs. Because of their simple syntax, Smalltalk blocks are even easier to manipulate than the lambda expressions of Lisp. In effect, a `to: by: do:` message turns iteration "inside out," making the body of the loop a simple message argument that can be executed (by sending it a `value` message) from within the body of the `to: by: do:` method. Smalltalk programmers can define similar methods for other container classes, obtaining all the power of iterators (Section 6.5.3) and much of the power of `call_with_current_continuation` (Section 9.4.3):

```
myTree inorderDo: [:node | whatever ]
```

■

It is worth noting that the uniform object model of computation in Smalltalk does not necessarily imply a uniform implementation. Just as Clu implementations implement built-in immutable objects as values, despite their reference semantics (Section 6.1.2), a Smalltalk implementation is likely to use the usual machine instructions for computer arithmetic, rather than actually sending messages to integers. In a similar vein, the most common control-flow constructs (`ifTrue: ifFalse:`, `to: by: do:`, `whileTrue:`, etc.) are likely to be recognized by a Smalltalk interpreter, and implemented with special, faster code.

We end this subsection by observing that recursion works at least as well in Smalltalk as it does in other imperative languages. The following is a recursive implementation of Euclid's algorithm:

EXAMPLE 10.75
Recursion in Smalltalk

```
gcd: other                                  "other is a formal parameter"
    (self = other)
        ifTrue:   [↑self].                  "end condition"
    (self < other)
        ifTrue:   [↑self gcd: (other - self)]       "recurse"
        ifFalse:  [↑other gcd: (self - other)]      "recurse"
```

The up-arrow (↑) symbol is comparable to the `return` of C or Algol 68. The keyword `self` is comparable to `this` in C++. We have shown the code in mixed fonts, much as it would appear in a Smalltalk browser. The header of the method is identified by bold face type.

✓ CHECK YOUR UNDERSTANDING

50. Name the three projects at Xerox PARC in the 1970s that pioneered modern GUI-based personal computers.
51. Explain the concept of a *message* in Smalltalk.
52. How does Smalltalk indicate multiple message arguments?
53. What is a *block* in Smalltalk? What mechanism does it resemble in Lisp?
54. Give three examples of how Smalltalk models control flow as message evaluation.
55. Explain how type checking works in Smalltalk.

10 Object Orientation

10.9 Exercises

10.23 Suppose that class D inherits from classes A, B, and C, none of which share any common ancestor. Show how the data members and vtable(s) of D might be laid out in memory. Also show how to convert a reference to a D object into a reference to an A, B, or C object.

10.24 Consider the person_interface and system_user_interface classes described in Example C-10.61. If student is derived from person_interface and system_user_interface, explain what happens in the following method call:

```
student s;
person *p = &s;
...
p->print_stats();
```

You may wish to use a diagram of the representation of a student object to illustrate the method lookups that occur and the views that are computed. You may assume an implementation akin to that of Figure C-10.9, without shared inheritance.

10.25 Given the inheritance tree of Example C-10.62, show a representation for objects of class student_prof. You may want to consult Figures C-10.8, C-10.9, and C-10.10.

10.26 Given the memory layout of Figure C-10.8 and the following declarations:

```
student& sr;
system_user& ur;
```

show the code that must be generated for the assignment

```
ur = sr;
```

(Pitfall: Be sure to consider null pointers.)

10.27 Standard C++ provides a "pointer-to-member" mechanism for classes:

```
class C {
public:
    int a;
    int b;
} c;
int C::*pm = &C::a;
    // pm points to member a of an (arbitrary) C object
...
C* p = &c;
p->*pm = 3;     // assign 3 into c.a
```

Pointers to members are also permitted for subroutine members (methods), including virtual methods. How would you implement pointers to virtual methods in the presence of C++-style multiple inheritance?

10.28 As an alternative to using ⟨method address, `this` correction⟩ pairs in the vtable entries of a language with multiple inheritance, we could leave the entries as simple pointers, but make them point to code that updates `this` in-line, and then jumps to the beginning of the appropriate method. Show the sequence of instructions executed under this scheme. What factors will influence whether it runs faster or slower than the sequence shown in Example C-10.59? Which scheme will use less space? (Remember to count both code and data structure size, and consider which instructions must be replicated at every call site.)

Pursuing the replacement of data structures with executable code even further, consider an implementation in which the vtable itself consists of executable code. Show what this code would look like and, again, discuss the implications for time and space overhead.

10.29 In Eiffel, shared inheritance is the default rather than the exception. Only renamed features are replicated. As a result, it is not possible to tell when looking at a class whether its members will be inherited replicated or shared by derived classes. Describe a uniform mechanism for looking up members inherited from base classes that will work whether they are replicated *or* shared. (Hint: Consider the use of dope vectors for records containing arrays of dynamic shape, as described in Section 8.2.2. For further details, consult the compiler text of Wilhelm and Maurer [WM95, Sec. 5.3].)

10.30 In Figure C-10.10, consider calls to virtual methods declared in `A`, but called through a `B`, `C`, or `D` object view. We could avoid one level of indirection by appending a copy of the `A` part of the vtable to the `D/B` and `C` parts of the vtable (with suitably adjusted `this` corrections). Give calling sequences for this alternative implementation. In the worst case, how much larger may the vtable be for a class with n ancestors?

10.31 Consider the Smalltalk implementation of Euclid's algorithm, presented at the end of Section C-10.7.1. Trace the messages involved in evaluating `4 gcd: 6`.

10 Object Orientation

10.10 Explorations

10.39 Figure out how multiple inheritance is implemented in your local C++ compiler. How closely does it follow the strategy of Sections C-10.6.2 and C-10.6.3? What rationale do you see for any differences?

10.40 Learn how multiple inheritance is implemented in Perl and Python (you might begin by reading Section 14.4.4). Describe the differences with respect to Sections C-10.6.2 and C-10.6.3. Discuss the advantages and drawbacks of dynamic typing in object-oriented languages.