What is an Object-Oriented Programming Language?

Kathleen Fisher and John C. Mitchell
Computer Science Dept., Stanford University, Stanford, CA 94305
{kfisher,mitchell}@cs.stanford.edu

June 13, 1995

Abstract

This document is a set of rough notes on basic features of object-oriented programming languages. It is based on the first section of the paper "Notes on typed object-oriented programming," *Proc. Theoretical Aspects of Computer Software,* Springer LNCS 789, 1994, pages 844–885.

1 Introduction

"Object orientation" is both a language feature and a design methodology. This paper is primarily about language features. However, some aspects of object-oriented design are important for understanding the power and usefulness of object-oriented languages. In general, object-oriented design is concerned with the way that programs may be organized and constructed. Objects provide a program-structuring tool whose importance seems to increase with the size of the programs we build.

Roughly speaking, an object consists of a set of operations on some hidden, or encapsulated, data. A characteristic of objects is that they provide a uniform interface to a variety of system components. For example, an object can be as small as a single integer or as large as a file system or output device. Regardless of its size, all interactions with an object occur via simple operations that are called "message sends" or "member function invocations." The use of objects to hide implementation details and provide a "black box" interface is useful for the same reasons that data and procedural abstraction are useful.

Although this paper is about language features, not methodology, we describe object-oriented design briefly since this design paradigm is one of the reasons for the success of object-oriented programming. The following list of steps is taken from [Boo91], one of many current books on object-oriented design.

1. Identify the objects at a given level of abstraction.
2. Identify the semantics (intended behavior) of these objects.
3. Identify the relationships among the objects.
4. Implement the objects.

This is an iterative process based on associating objects with components or concepts in a system. The process is iterative because an object is typically implemented using a number of "sub-objects," just as in top-down programming a procedure is typically implemented by a number of finer-grained procedures.

The data structures used in the early examples of top-down programming (see [Dij72]) were very simple and remained invariant under successive refinements of the program. Since these refinements involved simply replacing procedures with more detailed versions, older forms of structured programming languages, such as Algol, Pascal, and C, were adequate. When solving more complex tasks, however, it is often the case that both the procedures and the data structures of a program need to be refined in parallel. Object-oriented languages support this joint refinement of function and data.

2 Basic Concepts

Not suprisingly, all object-oriented languages have some notion of an "object," which is essentially some data and a collection of methods that operate on that data. There are two flavors of object-oriented languages: class-based and delegation-based. These flavors correspond to two different ways of defining and creating objects. In class-based languages, such as Smalltalk [GR83] and C++ [ES90], the implementation of an object is specified by its *class*. In such languages, objects are created by *instantiating* their classes. In delegation-based languages, such as Self, objects are defined directly from other objects by adding new methods via *method addition* and replacing old methods via *method override*. In the remainder of the paper, we will focus on the more common class-based languages.

Although there is some debate as to what exactly constitutes an object-oriented programming language (besides merely having objects), there seems to be general agreement that such a language should provide the following features: dynamic lookup, subtyping, inheritance, and encapsulation. Briefly, a language supports dynamic lookup if when a message is sent to an object, the method body to execute is determined by the run-time type of the object, not its static type. Subtyping means that if some object ob_1 has all of the functionality of another object ob_2, then we may use ob_1 in any context expecting ob_2. Inheritance is the ability to use the definition of simpler objects in the definitions of more complex ones. Encapsulation means that access to some portion of an object's data is restricted to that object (or perhaps to its descendants). We explore these features in more detail in the following subsections.

2.1 Dynamic lookup

In any object-oriented language, there is some way to invoke the methods associated with an object. In Smalltalk, this process is called "sending a message to an object," while in C++ it is "calling a member function of an object." To give a neutral syntax, we write

$$\texttt{receiver} \Leftarrow \texttt{operation}$$

for invoking `operation` on the object `receiver`. For expositional clarity, we will use the Smalltalk terminology for the remainder of this section.

Sending messages is a dynamic process: the method body corresponding to a given message is selected according to the run-time identity of the receiver object. The fact that this selection is dynamic is essential to object-oriented programming. Consider, for example, a simple graphics program that manipulates "pictures" containing many different kinds of shapes: squares, circles, triangles, etc. Each square object "knows" how to `draw` a square, each circle "knows" how to `draw` a circle, etc. When the program wants to display a given picture, it sends the `draw` message to each shape in the picture. At compile-time, the most we know about an object in the picture is that it is some kind of a shape and hence has some `draw` method. At run-time, we can find the appropriate `draw` method for each shape by querying that shape for its version of the `draw` method. If the shape is a square, it will have the square `draw` method, etc. [1]

There are two main views for what sending a message means operationally. In the first view, each object contains a "method table" that associates a method body with each message defined for that object. When a message is sent to an object at run-time, the corresponding method is retrieved from that object's method table. As a result, sending the same message to different objects may result in the execution of different code. In the example above, a square shape draws a square in response to the `draw` message, while a circle draws a circle. This behavior is called *dynamic lookup,* or, variously, dynamic binding, dynamic dispatch, and run-time dispatch. Both C++ and Smalltalk support this model of message sending.

The second view of message sending treats each message name as an "overloaded" function. When a message `m` is sent to an object `ob`, `ob` is treated as the first argument to an overloaded function named `m`. Unlike the traditional overloading of arithmetic operators, the appropriate code to execute when `m` is invoked is selected according to the run-time type of `ob`, not its static type. In this view, the methods of an object are not actually part of the object. Each object consists solely of its state. The methods from all the objects in a program are collected together by name. For example, the circle and square objects from above would simply contain their local state, i.e., the circle might contain its center and radius, the square its corner points. The `draw` methods from each would be collected into some "method repository". When the `draw` message is sent to some object `ob`, the dynamic type of `ob` is determined and the appropriate `draw` code selected from the repository. If `ob` were a circle, the circle `draw` method would be executed, etc. In this view, we again get the important characteristic that sending the same message to different objects can result in the execution of different code. Languages such as CLOS [Ste84] and Dylan [App92] support this model of message sending. A theoretical study appears in [CGL92].

The second view is somewhat more flexible than the first. In particular, in the second approach it is possible to take more than the first argument into account in the selection of the appropriate method body to execute. For example, if we write

$$\texttt{receiver} \Leftarrow \texttt{operation}(\texttt{arguments})$$

for invoking an operation with a list of arguments, then the actual code invoked can depend on the receiver alone (as explained above), or on the receiver and one or more arguments. When the selection of code depends only on the receiver, it is called *single dispatch;* when it also depends on one or more arguments, it is called *multiple dispatch.* Multiple dispatch is useful for implementing operations such as equality, where the appropriate comparisons to use depend on the dynamic type of both the receiver object and the argument object.

Although multiple dispatch is in some ways more general than the single dispatch found in C++ and Smalltalk, there seems to be some loss of encapsulation. This apparent loss arises because in order to define a function on different kinds of arguments, that function must typically have access to the internal data of each function argument. For example, suppose we wanted to define a `same_center` method that compares the centers of any two shapes and returns true if

[1] In C++ , only member functions designated *virtual* are selected dynamically. Non-virtual member functions are selected according to the static type of the receiver object. Needless to say, this distinction is the source of some confusion.

they match. Using multiple dispatch, we can write such a function by giving one version of the method for each pair of shapes we wish to consider: circle and circle, circle and square, square and circle, etc. Notice that this `same_center` method does not conceptually belong to any one of the shapes, and yet it must have access to the internal data of each shape object in order to do any meaningful comparisons. This external access of object internals violates the standard notions of encapsulation for object-oriented languages. It is not clear that this loss of encapsulation is inherent to multiple dispatch. However, current multiple dispatch systems do not seem to offer any reasonable encapsulation of private or local data for objects.

2.2 Subtyping

The basic principle associated with subtyping is *substitutivity:* if `A` is a subtype of `B`, then any expression of type `A` may be used without type error in any context that requires an expression of type `B`. We will write "`A` <: `B`" to indicate that `A` is a subtype of `B`.

The primary advantage of subtyping is that it permits uniform operations over various types of data. For example, subtyping makes it possible to have heterogeneous data structures containing objects that belong to different subtypes of some common base type. Consider as an example a queue containing various bank accounts to be balanced. These accounts could be savings accounts, checking accounts, investment accounts, etc., but each is a subtype of `bank_account` so balancing is done in the same way for each. This uniform treatment is generally not possible in strongly typed languages without subtyping.

Subtyping in an object-oriented language also allows functionality to be added with minimal modification to the system. If objects of a type `B` lack some desired behavior, then we may wish to replace objects of type `B` with objects of another type `A` that have the desired behavior. In many cases, the type `A` will be a subtype of `B`. By designing the language so that substitutivity is allowed, one may add functionality in this way without any other modification to the original program.

An example illustrating this use of subtyping occurs in building a series of prototypes of an airport scheduling system. In an early prototype, one would define a class `airplane` with methods such as `position`, `orientation`, and `acceleration` that would allow a control tower object to affect the approach of an airplane. In a later prototype, it is likely that different types of airplanes would be modeled. If one adds classes for Boeing 757's and Beechcrafts, these would be subtypes of `airplane`, containing extra methods and fields reflecting features specific to these aircraft. By virtue of the subtyping relation, all Beechcrafts are instances of `airplane` and the general control algorithms that apply to all airplanes can be used for Beechcrafts without modification or recompilation.

2.3 Inheritance

Inheritance is a language features that allows new classes to be defined as increments to existing ones. It is an implementation technique. For every object or class of objects defined using inheritance, there is an equivalent definition that does not use inheritance, obtained by expanding the definition so that inherited code is duplicated. The importance of inheritance is that it saves the effort of duplicating (or reading duplicated) code, and that when one class is implemented by inheriting from another, changes to one affect the other. This has a significant and sometimes debated impact on program maintenance and modification.

Using a neutral notation, we can illustrate by a simple example the form of inheritance that appears in most object-oriented languages. The two classes below define objects with private data v and public methods f and g. The class B is defined by inheriting the declarations of A, redefining the function g, and adding a private variable w.

```
class A =
    private
        val v =  ...
    public
        fun f(x)  = ... g(...) ...
        fun g(y)  = ... original definition ...
    end;

class B = extend A with
    private
        val w = ...
    public
        fun g(y)  = ... new definition ...
    end;
```

The simplest, but not most efficient, implementation of inheritance is to incorporate the relationship between classes explicitly in the run-time representation of objects, as is done in Smalltalk. For the example classes A and B above, this implementation is shown in Figure 1. This figure shows data structures representing the

A Class: stores pointers to the A Template and A Method Dictionary.

A Template: gives the names and order of data associated with each A object.

A Method Dictionary: contains pointers to the names and code for methods defined in the A Class.

B Class: stores pointers to the B Template, B Method Dictionary and base class A.

B Template: gives the names and order of data associated with each B object.

B Method Dictionary: contains pointers to the names and code for methods defined in the B Class.

The figure also shows an A object a and a B object b. Both of these objects contain pointers to their class and storage for their data.

We can see how this data-structure allows us to find the correct methods to execute at run-time by tracing the evaluation of the expression $b \Leftarrow f()$. The sequence of events is:

1. We find the method dictionary for B objects by following b's class pointer to the B Class and then accessing the class's method dictionary.

2. We search the B method dictionary for method name f.

3. Since f is not there, we follow the B Class's base class pointer to the A Class and then access the A method dictionary.

4. We find the function f in the A method dictionary.

5. When the body of f refers to g, we begin the search for the g method with the b object, guaranteeing that we find the g function defined in the B Class.

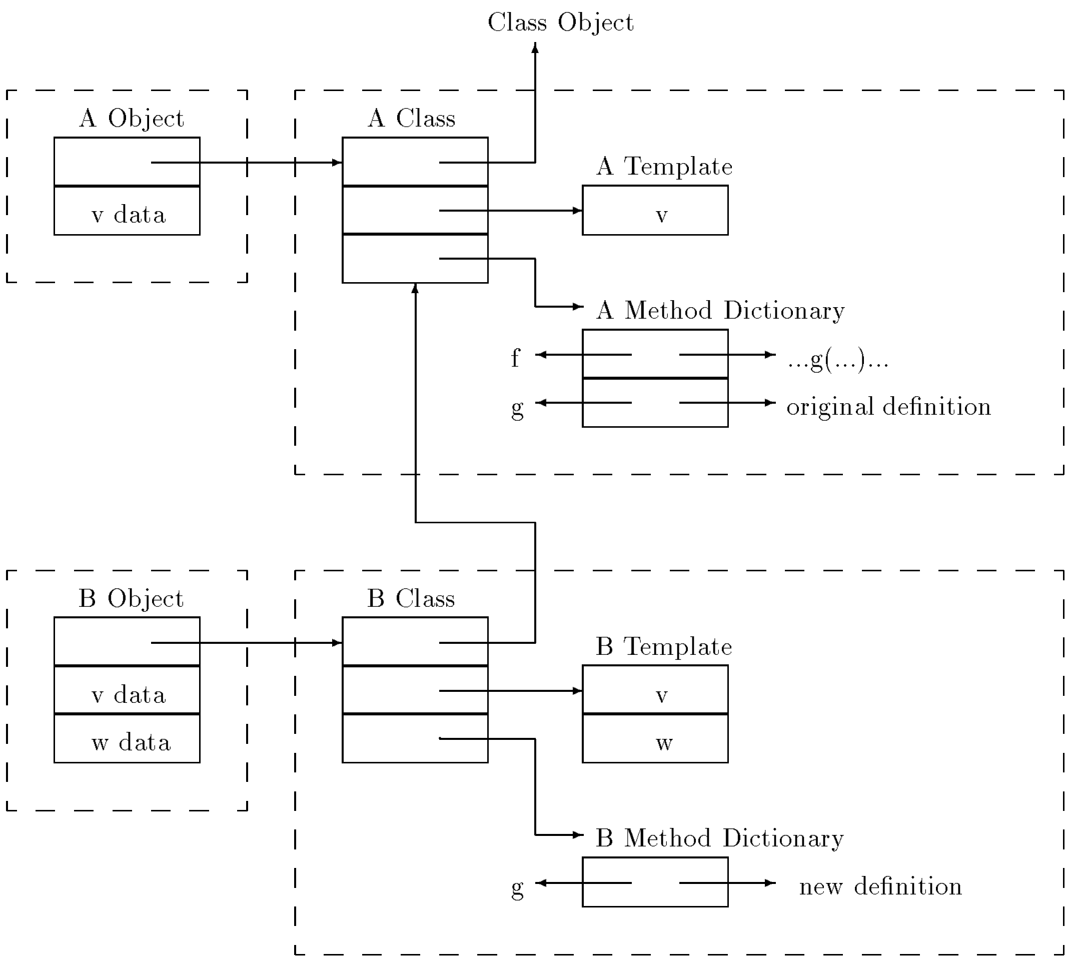

Figure 1: Smalltalk-style representation of `B` object inheriting from class `A`.

This implementation may be optimized in several ways. The first is to cache recently-found methods. Another possibility is to expand the method tables of derived classes to include the method tables of their base classes. This expansion eliminates the upward search through the method dictionaries of more than one class. Since the dictionaries contain only pointers to functions, this duplication does not involve a prohibitive space overhead. C++ makes this optimization.

A more significant optimization may be made in typed languages such as C++ , where the set of possible messages to each object can be determined statically. If method dictionaries, or *virtual function tables* (vtables) in C++ terminology, can be constructed so that all subtypes of a given class `A` store pointers to the common methods in the same relative positions in their respective vtables, then the offset of a method within any vtable can be computed at compile-time. This optimization reduces the cost of method lookup to a simple indirection without search, followed by an ordinary function call. In untyped languages such as Smalltalk, this optimization is not possible because at compile-time, all we know about an object is that it *is* an object. In general, we do not know what messages it understands, let alone where the corresponding methods are stored.

Figure 2 shows a schematic C++ representation of the example classes `A` and `B` given above. This figure contains an `A` object `a` and a `B` object `b`. Each of these objects stores its instance variables and has a pointer to its class's vtable. The `A` vtable contains pointers to the methods defined in the `A` class, while the `B` vtable contains pointers to all the methods defined in `B` and to those defined in `A` but not redefined in `B`. (The expression `A::f` denotes the `f` function defined in the `A` Class and the "`&`" denotes C++ 's address-of operator) By duplicating the `f` method pointer in the `B` vtable, we do not have to access the `A` vtable when manipulating a `B` object.

We may see how this data structure works by tracing the evaluation of the expression `b`$\Leftarrow$`f()`. The sequence of events is essentially[2]:

1. We find the vtable for `B` objects by following `b`'s vtable pointer.

2. At compile-time, we may determine that the `f` method is the first entry in the `B` vtable, so we retrieve the `f` method from the vtable without searching.

3. When the body of `f` refers to `g`, we retrieve the `g` method from `b`'s vtable, guaranteeing that we use the `g` function defined in the `B` class.

For more information, see [ES90, Section 10.7c].

2.4 Encapsulation

Objects are used in most object-oriented programming languages to provide encapsulation barriers similar to those given by abstract data types (ADT's). However, because object-oriented languages have inheritance, object-oriented encapuslation can be more complex than simple abstract data types. In particular, there are two "clients" of the code in a given ADT: the implementor, who "lives" inside the encapsulation barrier, and the general client, who "lives" outside and may only interact with the ADT via its interface. A graphic representation of this relationship appears in Figure 3. Because of inheritance, there are three "clients" of the code in a given object definition, not two. The additional "client," the inheritor, uses the given object definition via inheritance to implement new object definitions. Because object definitions have two external clients, there are two interfaces to the "outside": the *public* interface lists what the general client may see, while the

[2] The actual process is somewhat more complicated because of multiple inheritance. See [ES90, Chapter 10] for more details.

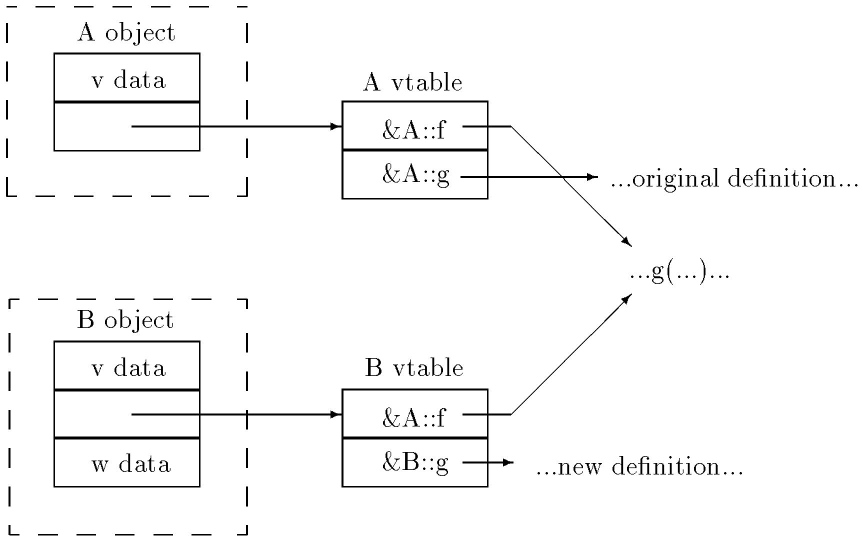

Figure 2: A C++-style representation of `A` and `B` objects where class `B` inherits from class `A`.

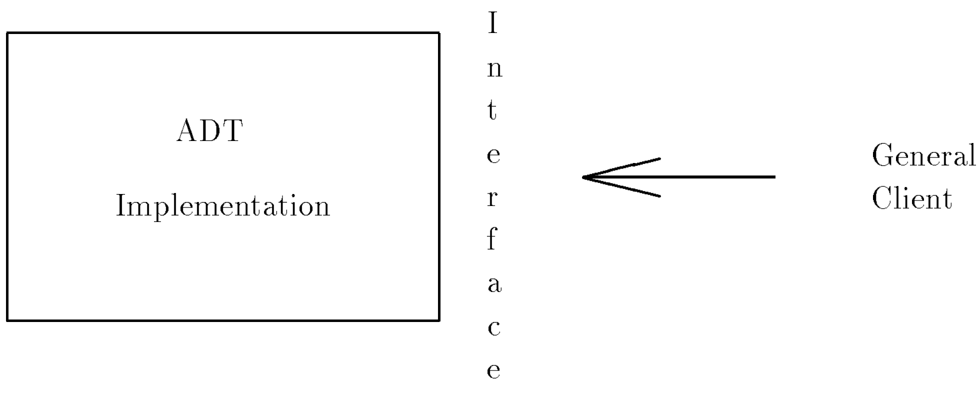

Figure 3: In ADT-style encapsulation, the general client interacts with the ADT implementation through a single interface.

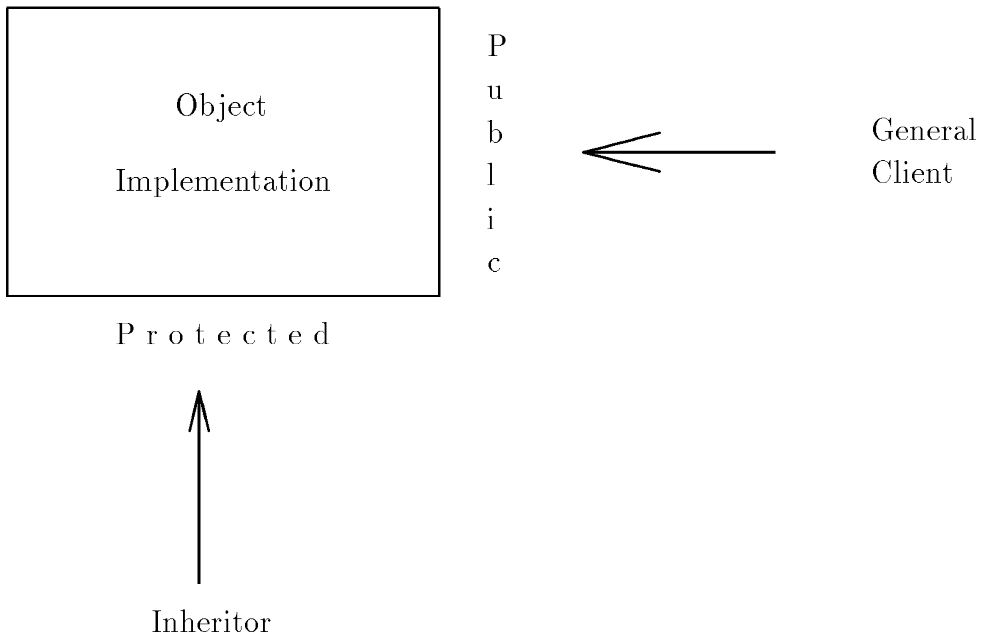

Figure 4: In object-oriented encapsulation, the general client interacts with the object implementation via the public interface, while the inheritors interact via the protected interface.

protected interface lists what inheritors may see. (This terminology comes from C++ .) A graphic representation appears in Figure 4. It is typically the case that the public interface is a subset of the protected one. In Smalltalk, these interfaces are generated automatically: the public interface lists the methods of an object, while the protected interface lists its methods and its instance variables. In C++ , the programmer explicitly declares which components of an object are public, which are protected, and which are *private*, visible only in the object definition itself.

The encapsulation provided by object-oriented languages helps insure that programs can be written in a modular fashion and that the implementation of an object can be changed without forcing changes in the rest of the system. In particular, as long as the public interface of an object remains unchanged, modifications to its implementation do not force general clients to change their code. Similarly, if implementation modifications preserve an object's protected interface, inheritors need not update their code, either [3].

3 ADT's vs. objects

The encapuslation benefits provided by objects are the same as those realized by abstract data types. However, because object-oriented languages provide dynamic lookup, subtyping, and inheritance in addition to encapsulation, objects may be used more flexibly than ADT's. The importance of these added features becomes apparent when we wish to use related data abstractions in similar ways. We illustrate this point with the following example involving queues.

[3] In both cases, however, they may have to recompile.

A typical language construct for defining an abstract data type is the ML `abstype` declaration, which we use below to define a queue ADT.

```
exception Empty;

abstype queue = Q of int list
with
     fun mk_Queue()      = Q(nil)
     and is_empty(Q(l)) = l=nil
     and add(x,Q(l))     = Q(l @ [x])
     and first (Q(nil))  = raise Empty
     |   first (Q(x::l)) = x
     and rest (Q(nil))   = raise Empty
     |   rest (Q(x::l))  = Q(l)
     and length (Q(nil)) = 0
     |   length (Q(x::l))= 1 + length (Q(l))
end;
```

In this example, a queue is represented by a list. However, only the functions given in the declaration may access the list. This restriction allows the invariant that list elements appear in first-in/first-out order to be maintained, regardless of how queues are used in client programs.

A drawback of the kind of abstract data types used in ML and other languages such as CLU [LSAS77, L+81] and Ada [US 80] becomes apparent when we consider a program that uses both queues and priority queues. For example, suppose that we are simulating a system with several "wait queues," such as a bank or hospital. In a teller line or hospital billing department, customers are served on a first-come, first-served basis. However, in a hospital emergency room, patients are treated in an order that takes into account the severity of their injuries or ailments. Some aspects of this kind of "wait queue" are modeled by the abstract data type of priority queues, shown below:

```
abstype pqueue = Q of int list
with
     fun mk_PQueue()      = Q(nil)
     and is_empty(Q(l)) = l=nil
     and add(x,Q(l))    =
         let fun insert(x,nil) = [x:int]
         |   insert(x,y::l) = if x<y then x::y::l else y::insert(x,l)
         in Q(insert(x,l)) end
     and first (Q(nil))   = raise Empty
     |   first (Q(x::l))  = x
     and rest (Q(nil))    = raise Empty
     |   rest (Q(x::l))   = Q(l)
     and length (Q(nil)) = 0
     |   length (Q(x::l))= 1 + length (Q(l))
end;
```

For simplicity, like the queues above, this queue is defined only for integer data. Although the priority of a queue element may come from any ordered set, we use the integer value as the priority, with lower numbers given higher priority.

Note that the signature of priority queues, the list of available methods and their associated types, is the same as for ordinary queues: both have the same number of operations, and each operation has the same type, except for the difference between the type names `pqueue` and `queue`. However, if we declare both queues and priority queues in the same scope, the second declarations of `is_empty, add, first, rest,` and `length` hide the first. This name clashing requires us to rename them, say as `q_is_empty, q_add, q_first, q_rest, q_length` and `pq_is_empty, pq_add, pq_first, pq_rest, pq_length`.

In a hospital simulation (or real-time hospital management) program, we might occasionally like to treat priority queues and ordinary queues uniformly. For example, we might wish to count the total number of people waiting in any line in the hospital. To write this code, we would like to have a list of all the queues (both priority and ordinary) in the hospital and go down the list asking each queue for its length. But if the `length` operation is different for queues and priority queues, we have to decide whether to call `q_length` or `pq_length`, even though the correct operation is uniquely determined by the data. This shortcoming of ordinary abstract data types is eliminated in object-oriented programming languages by a combination of subtyping and dynamic lookup.

Another drawback of traditional abstract data types becomes apparent when considering the implementation of the priority queue above. Although the priority queue's version of the `add` function is different from the queue's version, the other five functions have identical implementations. In an object-oriented language, we may use inheritance to define `pqueue` from `queue` (or vice versa), giving only the new `add` function.

4 Object-oriented vs. conventional program organization

Because object-oriented languages have subtyping, inheritance, and dynamic lookup, programs written in an object-oriented style are organized quite differently from those written in a traditional style. In this section, we illustrate some of the differences between object-oriented and "conventional" program organizations via an extended example. We give two versions of a program that manipulates several kinds of geometric shapes. One version uses classes; the other does not.

Without classes, we use records (or `struct`'s) to represent each shape. For each operation on shapes, we have a function that tests the type of shape passed as an argument and branches accordingly. We illustrate this program structure using a C program, with each shape represented as a `struct` (analogous to a Pascal or ML record). The code appears in Appendix A. We will refer to this program as the "typecase" version, since each function is implemented by a case analysis on the types of shapes. For brevity, the only shapes are circles and rectangles.

We can see the advantage of object-oriented programming by rewriting the program so that each object has the shape-specific operations as methods. This version appears in Appendix B.

Some observations:

- We can see the difference between the two program organizations in the following matrix. For each function, `center, move, rotate` and `print`, there is code for each geometric shape, in this case `circle` and `rectangle`. Thus we have eight different pieces of code.

class	*function*			
	center	move	rotate	print
circle	`c_center`	`c_move`	`c_rotate`	`c_print`
rectangle	`r_center`	`r_move`	`r_rotate`	`r_print`

In the "typecase" version, these functions are arranged by column, while in the class-based program, they are arranged by row. Each arrangement has some advantages when it comes to program maintenance and modification. In the object-oriented approach, adding a new shape is straightforward. The code detailing how the new shape should respond to the existing operations all goes in one place: the class definition. Adding a new operation is more complicated, since the appropriate code must be added to each of the class definitions, which could be spread throughout the system. In the "typecase" version, the reverse situation is true: adding a new operation is relatively easy, but adding a new shape is difficult.

- There is a loss of encapsulation in the typecase version, since the data manipulated by `rotate`, `print` and the other functions has to be publicly accessible. In contrast, the object-oriented solution encapsulates the data in the `circle` and `square` objects. Only the methods of these objects may access this data.

- The "typecase" version cannot be statically type-checked in C. It could be type-checked in a language with a built-in "typecase" statement which tests the type of an struct directly. An example of such a language feature is the Simula `inspect` statement. Adding such a statement would require that every struct be tagged with its type, a process which requires about the same amount of space overhead as making each struct into an object.

- In the typecase version, "subtyping" is used in an ad hoc manner. We coded circle and rectangle so that they have a shared field in their first location. This is a hack to implement a tagged union that could be avoided in a language providing disjoint (as opposed to C unchecked) unions.

- The complexity of the two programs is roughly the same. In the "typecase" version, there is the space cost of an extra data field (the type tag) and the time cost, in each function, of branching according to type. In the "object" version, there is a hidden class or `vtbl` pointer in each object, requiring essentially the same space as a type tag. In the optimized C++ approach, there is one extra indirection in determining which method to invoke, which corresponds to the switch statement in the "typecase" version. (Although in practice a single indirection will frequently be more efficient than a switch statement.) A Smalltalk-like implementation would be less efficient in general, but for methods that are found immediately in the subclass method dictionary (or via caching), the run-time efficiency may be comparable.

A similar example appears in [Str86, Sections 7.2.7–8].

5 Advanced topics

5.1 Inheritance is not subtyping

Perhaps the most common confusion surrounding object-oriented programming is the difference between subtyping and inheritance. One reason subtyping and inheritance are often confused is that some class mechanisms combine the two. A typical example is C++ , where `A` will be recognized by the compiler as a subtype of `B` only if `B` is a public parent class of `A`. Combining these mechanisms is an elective design decision, however; there seems to be no inherent reason for linking subtyping and inheritance in this way.

We may see the differences between inheritance and subtyping most clearly by considering an example. Suppose we are interested in writing a program that requires `dequeue`s, `stack`s, and queues. One way to implement these three classes is first to implement dequeue and then to implement stack and queue by appropriately restricting (and perhaps renaming) the operations of dequeue. For example, stack may be obtained from dequeue by limiting access to those operations that add and remove elements from one end of the dequeue. Similarly, we may obtain queue from dequeue by restricting access to those operations that add elements at one end and remove them from the other. This method of defining stack and queue by inheriting from dequeue is possible in C++ through the use of private inheritance. (We are not recommending this style of implementation; we use this example simply to illustrate the differences between subtyping and inheritance.) Note that although stack and queue inherit from dequeue, they are not subtypes of dequeue. To see this point, consider a function f that takes a dequeue d as an argument and then adds an element to both ends of d. If stack or queue were a subtype of dequeue, then function f should work equally well when given a stack s or a queue q. However, adding elements to both ends of either a stack or a queue is not legal; hence, neither stack nor queue is a subtype of dequeue. In fact, the reverse is true. Dequeue is a subtype of both stack and queue, since any operation valid for either a stack or a queue would be a legal operation on a dequeue. Thus, inheritance and subtyping are different relations: we defined stack and queue by inheriting from dequeue, but dequeue is a subtype of stack and queue, not the other way around.

A more detailed comparison of the two mechanisms appears in [Coo92], which analyzes the inheritance and subtyping relationships between Smalltalk's collection classes. In general, there is little relationship between the two relations. See [Sny86] for more examples.

5.2 Object types

There are two forms of types we might give to objects. The first is a type that simply gives the interface to its objects. The second is an interface plus some implementation information. In the first case, the elements of a type will be all objects that have a given interface. We call such types "interface types". In the second case, a type will contain only those elements that also have a certain representation. The type that C++ gives to an object is of the second form, since all objects of the same type are guaranteed to have the same implementation.

Since the first form of type is more basic, we begin by discussing it. The following example uses the syntax of Rapide, an experimental language designed for prototyping software and mixed software/hardware systems [BL90, MMM91, KLM94, KLMM94].

```
type Point is interface
   x_val : Int;
   y_val : Int;
   distance : Point -> Int;
end interface;
```

Objects of type Point must have two integer methods, called x_val and y_val, and a method called distance. This distance method requires only one argument, since the method belongs to a particular point and therefore may compute the distance between the point passed as an actual parameter and the particular point to which the method belongs. In other words, the intended use of the distance method of a point object p is to compute the distance between p and another point object q, by a call of the form p ⇐ distance(q). Of course, since the interface gives only the names of methods and their types, the distance method is not actually forced to compute the distance between two points. If we wish to specify that distance must compute distance, then a more expressive form of specification must be added to the interface. One significant feature of this type interface for Points is that the type name Point appears within it. Hence interface types seem to be recursively-defined types.

To discuss object types in general, we introduce the syntax $\{\!|\mathtt{m}_1 : \mathtt{A}_1, \ldots, \mathtt{m}_k : \mathtt{A}_k|\!\}$ for the interface type specifying methods $\mathtt{m}_1, \ldots, \mathtt{m}_k$ of types $\mathtt{A}_1, \ldots, \mathtt{A}_k$, respectively. Using this notation, we may recursively define the type Point as

$$\mathtt{Point} = \{\!|\mathtt{x\_val} : \mathtt{Int}, \mathtt{y\_val} : \mathtt{Int}, \mathtt{distance} : \mathtt{Point} \rightarrow \mathtt{Int}|\!\}$$

Objects that have this interface type are guaranteed to have integer x_val and y_val methods. They are also guaranteed to have a method distance that returns an integer whenever it is given another object with the Point interface. Objects with this interface are *not* required to have any particular implementation. For example, an object that stores a point in polar coordinates and implements x_val and y_val as functions that convert the stored polar coordinates into their cartesian counterparts may be given this interface type, just as the obvious cartesian implementation may. It is also the case that objects with the Point interface may have more methods than just those listed in the interface. For example, the polar point object described above must have some fields storing the polar coordinates of the point. These fields are not reflected in the Point interface.

If the type of an object is its interface, then subtyping for object types is "compatibility" or "conformance" of interfaces. More specifically, if one interface provides all of the methods of another with compatible types, then every object of the first type should be acceptable in any context expecting an object of the second type. This kind of subtyping is of the form:

$$\{\!|\mathtt{x} : \mathtt{Point},\ \mathtt{c} : \mathtt{Color}|\!\} <: \{\!|\mathtt{x} : \mathtt{Point}|\!\}$$

which we call "width" subtyping. (We use the sumbol $<:$ to denote the subtype relation between types.) This subtyping "judgement" says that we may consider any object that has the interface $\{\!|x : Point,\ c : Color|\!\}$ to have the interface $\{\!|x : Point|\!\}$ as well. In other words, we may put an object with interface $\{\!|x : Point,\ c : Color|\!\}$ into any context expecting an object with interface $\{\!|x : Point|\!\}$ and be guaranteed that no type errors will result. We may see the justification for this guarantee by considering what a context C[ob] may ask of its argument object ob. Since C expects to be given an object with the $\{\!|x : Point|\!\}$ interface, all it "knows" about its argument object is that it has an x method that returns a Point object. Hence all it may do with ob is ask for its x method and then treat the result as a Point. Since any object with the $\{\!|x : Point,\ c : Color|\!\}$ interface has an x method that returns an Point object, giving such an object to our context can not result in any type errors.

It is also generally possible to specialize the type of one or more methods to a subtype. This subtyping, which we call "depth" subtyping, is of the form:

$$\{\!|\mathtt{x} : \mathtt{ColorPoint}|\!\} <: \{\!|\mathtt{x} : \mathtt{Point}|\!\}$$

if we assume that $\mathtt{ColorPoint} <: \mathtt{Point}$. This subtyping judgement says that we may consider any object with interface $\{\!|x : ColorPoint|\!\}$ to have interface $\{\!|x : Point|\!\}$ as well. In other words, we may put any object with interface $\{\!|x : ColorPoint|\!\}$ into any context expecting a $\{\!|x : Point|\!\}$ object and not produce any type errors. As above, we may see the justification for this guarantee by considering what such a context might ask of its argument. Because it expects an object with interface $\{\!|x : Point|\!\}$, all it "knows" about its argument object is that it has an x method that returns a Point object, and hence that is all it may ask for. If we give such a context some object cp with interface $\{\!|x : ColorPoint|\!\}$, the context may only ask cp for its x value, at which point cp returns something with interface ColorPoint. Because we know that $\mathtt{ColorPoint} <: \mathtt{Point}$, we are guaranteed that this result object may be safely treated as a `Point` object. Hence no type errors may result from putting a $\{\!|x : ColorPoint|\!\}$ object into a context expecting a $\{\!|x : Point|\!\}$ object.

Combining these two forms of subtyping, we have

$$\{\!|\mathtt{x} : \mathtt{ColorPoint},\ \mathtt{c} : \mathtt{Color}|\!\} <: \{\!|\mathtt{x} : \mathtt{Point}|\!\}$$

An alternative form of object type is an interface type with some additional guarantees about the form of the implementations of objects given that type. The types that C++ gives to its objects have this flavor. If we know that a particular object `ob` has type `B`, then we know `ob` has all of the methods and the associated types that are listed in the class `B`. We are also guaranteed that the implementation of `ob` is an extension (perhaps trivial) of the implementation given by class `B`.

These implementation guarantees are important for objects with binary operations (those that take another object of the same type as an argument), and they permit more efficient implementations of objects. For these types, subtyping must take into account both interface subtyping and compatibility of implementations. Since the implementation of an object is intended to be hidden, the second form of type should not give any explicit information about the implementation. Instead, it appears that "implementation types" are properly treated as a form of partially-abstract types. This is a current research topic, with some of the basic ideas explained in [CW85, KLM94, PT93] using bounded existential types.

5.3 Method specialization

It is relatively common for one or more methods of an object to take objects of the same type as parameters or return objects of the same type as results. For example, consider points with the following interface.

```
type Point is interface
   x    : Int
   move : Int -> Point
   eq   : Point -> Bool
end interface;
```

(For simplicity, we drop the `y` coordinate and work with one-dimensional points.) The `move` method of a point `p` returns a `Point`. Similarly, the `eq` method takes as a parameter an object of `Point` type.

When colored points are defined in terms of points, it is desirable that the types of the methods be specialized to return or use colored points instead of points. Otherwise, we effectively lose type information about the object we are dealing with whenever we send the `move` method, and we are restricted to using only point methods when comparing colored points for equality. If it is possible to inherit a `move` method defined for points in such a way that the resulting method on colored points has type $\mathtt{Int} \rightarrow \mathtt{Colored\_Point}$, then we say that *method specialization* occurs. This form of method specialization is called "mytype" specialization because the type that changes is the type of the object that contains the methods [Bru92, Bru93]. It is also meaningful to specialize types other than the type of the object itself when defining a derived class.

Method specialization is generally not provided in existing typed object-oriented languages, but it is common to take advantage of method specialization (in effect) in untyped object-oriented languages. Therefore, if we are to devise typed languages to support useful untyped programming idioms, we need to devise type systems that support method specialization.

Acknowledgements: Thanks to Brian Freyburger and Steve Fisher for several insightful discussions of C++ , to Andy Hung for the drawings in Section 2.3 and to Luca Cardelli and Sandeep Singhal for comments on drafts of this paper.

References

[App92] Apple Computer. *Dylan: an object-oriented dynamic language.* Apple Computer, 1992.

[BL90] F. Belz and D.C. Luckham. A new approach to prototyping Ada-based hardware/software systems. In *Proc. ACM Tri-Ada'90 Conference*, December 1990.

[Boo91] G. Booch. *Object-Oriented Design with Applications.* Benjamin Cummings, Redwood City, CA, 1991.

[Bru92] K. Bruce. The equivalence of two semantic definitions of inheritance in object-oriented languages. In *Proc. Mathematical Foundations of Programming Language Semantics*, pages 102–124, Berlin, 1992. Springer LNCS 598.

[Bru93] K. Bruce. Safe type checking in a statically-typed object-oriented programming language. In *Proc 20th ACM Symp. Principles of Programming Languages*, pages 285–298, 1993.

[CGL92] Castagna, Ghelli, and Longo. A calculus for overloaded functions with subtyping. In *1992 ACM Conf. Lisp and Functional Programming*, pages 182–192, 1992.

[Coo92] W.R. Cook. Interfaces and specifications for the Smalltalk-80 collection classes. In *ACM Conf. Object-oriented Programming: Systems, Languages and Applications*, pages 1–15, 1992.

[CW85] L. Cardelli and P. Wegner. On understanding types, data abstraction, and polymorphism. *Computing Surveys*, 17(4):471–522, 1985.

[Dij72] E.W. Dijkstra. Notes on structured programming. In O.J. Dahl, E.W. Dijkstra, and C.A.R. Hoare, editors, *Structured Programming.* Academic Press, 1972.

[ES90] M. Ellis and B. Stroustrop. *The Annotated C^{++} Reference Manual.* Addison-Wesley, 1990.

[GR83] A. Goldberg and D. Robson. *Smalltalk–80: The language and its implementation.* Addison Wesley, 1983.

[KLM94] Dinesh Katiyar, David Luckham, and John Mitchell. A type system for prototyping languages. In *Proc. 21-st ACM Symp. on Principles of Programming Languages*, 1994.

[KLMM94] Dinesh Katiyar, David Luckham, S. Meldal, and John Mitchell. Polymorphism and subtyping in interfaces. In *ACM Workshop on Interface Definition Languages*, 1994.

[L^+81] B. Liskov et al. *CLU Reference Manual.* Springer LNCS 114, Berlin, 1981.

[LSAS77] B. Liskov, A. Snyder, R. Atkinson, and C. Schaffert. Abstraction mechanisms in clu. *Comm. ACM*, 20:564–576, 1977.

[MMM91] J.C. Mitchell, S. Meldal, and N. Madhav. An extension of Standard ML modules with subtyping and inheritance. In *Proc. 18th ACM Symp. on Principles of Programming Languages*, pages 270–278, January 1991.

[PT93] Benjamin C. Pierce and David N. Turner. Statically typed friendly functions via partially abstract types. Technical Report ECS-LFCS-93-256, University of Edinburgh, LFCS, April 1993. Also available as INRIA-Rocquencourt Rapport de Recherche No. 1899.

[Sny86] A. Snyder. Encapsulation and inheritance in object-oriented programming languages. In *Proc. ACM Symp. on Object-Oriented Programming Systems, Languages, and Applications*, pages 38–46, October 1986.

[Ste84] G.L. Steele. *Common Lisp: The language.* Digital Press, 1984.

[Str86] B. Stroustrop. *The C^{++} Programming Language.* Addison-Wesley, 1986.

[US 80] US Dept. of Defense. *Reference Manual for the Ada Programming Language.* GPO 008-000-00354-8, 1980.

A Shape program: Typecase version

```
#include <stdio.h>
#include <stdlib.h>


   /*
    * We use the following enumeration type to ``tag'' shapes.
    * The first field of each shape struct stores what particular
    * kind of shape it is.
    */
enum ShapeTag {Circle, Rectangle};


   /*
    * The following struct Pt and functions newPt and copyPt are
    * used in the implementations of the Circle and Rectangle
    * shapes below.
    */
struct Pt {
   float x;
   float y;
};

struct Pt* newPt(float xval, float yval) {
   struct Pt* p = (struct Pt *)malloc(sizeof(struct Pt));
   p->x = xval;
   p->y = yval;
   return p;
};

struct Pt* copyPt(struct Pt* p) {
   struct Pt* q = (struct Pt *)malloc(sizeof(struct Pt));
   q->x = p->x;
   q->y = p->y;
   return q;
};


   /*
    * The Shape struct provides a flag that is used to get some static
    * type checking in the operation functions (center, move, rotate,
    * and print) below.
    */
struct Shape {
   enum ShapeTag tag;
};
```

```
   /*
    * The following Circle struct is our representation of a circle.
    * The first field is a type tag to indicate that this struct
    * represents a circle.  The second field stores the circle´s
    * center point and the third field holds its radius.
    */
struct Circle {
   enum ShapeTag  tag;
   struct Pt*     center;
   float          radius;
};


   /*
    * The function newCircle creates a Circle struct from a given
    * center point and radius.  It sets the type tag to ``Circle.´´
    */
struct Circle* newCircle(struct Pt* cp, float r) {
   struct Circle* c = (struct Circle*)malloc(sizeof(struct Circle));
   c->center=copyPt(cp);
   c->radius=r;
   c->tag=Circle;
   return c;
};


   /*
    * The function deleteCircle frees resources used by a Circle.
    */
void deleteCircle(struct Circle* c) {
   free (c->center);
   free (c);
};


   /*
    * The following Rectangle struct is our representation of a rectangle.
    * The first field is a type tag to indicate that this struct
    * represents a rectangle.  The next two fields store the rectangles
    * top-left and bottom-right corner points.
    */
struct Rectangle {
   enum ShapeTag  tag;
   struct Pt*     topleft;
   struct Pt*     botright;
};


   /*
    * The function newRectangle creates a rectangle in the location
    * specified by parameters tl and br.  It sets the type tag to
    * ``Rectangle.´´
```

```
    */
struct Rectangle* newRectangle(struct Pt* tl, struct Pt* br)  {
   struct Rectangle* r = (struct Rectangle*)malloc(sizeof(struct Rectangle));
   r->topleft=copyPt(tl);
   r->botright=copyPt(br);
   r->tag=Rectangle;
   return r;
};


   /*
    * The function deleteRectangle frees resources used by a Rectangle.
    */
void deleteRectangle(struct Rectangle* r) {
   free (r->topleft);
   free (r->botright);
   free (r);
};


   /*
    * The center function returns the center point of whatever shape
    * it is passed.  Because the computation depends on whether the
    * shape is a Circle or a Rectangle, the function consists of a
    * switch statement that branches according to the type tag stored
    * in the shape s.  If the tag is Circle, for instance, we know
    * the parameter is really a circle struct and hence that it has
    * a ``center´´ component which we can return.  Note that we need
    * to insert a typecast to instruct the compiler that we have a
    * circle and not just a shape.  Note also that this program
    * organization assumes that the type tags in the struct are
    * set correctly.  If some programmer incorrectly modifies a type tag
    * field, the program will no longer work and the problem cannot
    * be detected at compile time because of the typecasts.
    */
struct Pt* center (struct Shape* s) {
   switch (s->tag) {
    case Circle: {
       struct Circle* c = (struct Circle*) s;
       return copyPt(c->center);
    };
    case Rectangle: {
       struct Rectangle* r = (struct Rectangle*) s;
       return newPt((r->botright->x - r->topleft->x)/2,
                    (r->botright->x - r->topleft->x)/2);
    };
   };
};


   /*
    * The move function receives a Shape parameter s and moves it
    * dx units in the x-direction and dy units in the y-direction.
```

```
    * Because the code to move a Shape depends on the kind of shape,
    * this function inspects the Shape´s type tag field within a switch
    * statement.  Within the individual cases, typecasts are used to
    * convert the generic shape parameter to a Circle or Rectangle as
    * appropriate.
    */
void move (struct Shape* s,float dx, float dy) {
   switch (s->tag) {
    case Circle: {
         struct Circle* c = (struct Circle*) s;
         c->center->x    += dx;
         c->center->y    += dy;
      };
      break;
    case Rectangle: {
       struct Rectangle* r = (struct Rectangle*) s;
       r->topleft->x      += dx;
       r->topleft->y      += dy;
       r->botright->x     += dx;
       r->botright->y     += dy;
    };
   };
};


   /*
    * The rotate function rotates the shape s ninety degrees.  Like
    * the center and move functions, this code uses a switch statement
    * that checks the type of shape being manipulated.
    */
void rotate (struct Shape* s) {
   switch (s->tag) {
    case Circle:
         /* Rotating a circle is not a very interesting operation! */
      break;
    case Rectangle: {
         struct Rectangle* r = (struct Rectangle*)s;
         float d = ((r->botright->x - r->topleft->x) -
                    (r->topleft->y - r->botright->y))/2.0;
         r->topleft->x  += d;
         r->topleft->y  += d;
         r->botright->x -= d;
         r->botright->y -= d;
      };
      break;
   };
};


   /*
    * The print function outputs a description of its Shape parameter.
    * This function again selects its processing based on the type tag
    * stored in the Shape struct.
```

```
    */
void print (struct Shape* s) {
   switch (s->tag) {
    case Circle: {
         struct Circle* c = (struct Circle*) s;
         printf("circle at  %.1f  %.1f  radius %.1f \n",
                c->center->x, c->center->y, c->radius);
      };
      break;
    case Rectangle: {
         struct Rectangle* r = (struct Rectangle*) s;
         printf("rectangle at  %.1f  %.1f   %.1f  %.1f \n",
                r->topleft->x, r->topleft->y,
                r->botright->x, r->botright->y);
      };
      break;
   };
};


   /*
    * The body of this program just tests some of the above functions.
    */
void main() {
   struct Pt* origin = newPt(0,0);
   struct Pt* p1     = newPt(0,2);
   struct Pt* p2     = newPt(4,6);

   struct Shape* s1  = (struct Shape*)newCircle(origin,2);
   struct Shape* s2  = (struct Shape*)newRectangle(p1,p2);

   print(s1);
   print(s2);

   rotate(s1);
   rotate(s2);

   move(s1,1,1);
   move(s2,1,1);

   print(s1);
   print(s2);

   deleteCircle((struct Circle*)s1);
   deleteRectangle((struct Rectangle*)s2);

   free(origin);
   free(p1);
   free(p2);
};
```

B Shape program: Object-oriented version

```
#include <stdio.h>

   // (The following is a running C++ program, but it does not represent
   // an ideal C++ implementation.  The code has been kept simple so
   // that it can be understood by readers who are not well-versed in C++).


   // The following class Pt is used by the shape objects below.  Since
   // Pt is a class in this version of the program, the ``newPt´´ and
   // ``copyPt´´ functions may be implemented as class member functions.
   // For readability, we have in-lined the function definitions and
   // named both of these functions ``Pt´´; these overloaded functions
   // are differentiated by the types of their arguments.
class Pt {
 public:
   Pt(float xval, float yval) {
      x = xval;
      y=yval;
   };

   Pt(Pt* p) {
      x = p->x;
      y = p->y;
   };

   float x;
   float y;
};


   // Class shape is an example of a ``pure abstract base class,´´
   // which means that it exists solely to provide an interface to
   // classes derived from it.  Since it provides no implementations
   // for the methods center, move, rotate, and print, no ``shape´´
   // objects can be created.  Instead, we use this class as a base
   // class.  Our circle and rectangle shapes will be derived from
   // it.  This class is useful because it allows us to write
   // functions that expect ``shape´´ objects as arguments.  Since
   // our circles and rectangles are subtypes of shape, we may pass
   // them to such functions in a type-safe way.
class Shape {
public:
  virtual Pt* center()=0;
  virtual void move(float dx, float dy)=0;
  virtual void rotate()=0;
  virtual void print()=0;
};


   // Class Circle consolidates the center, move, rotate, and print
   // functions for circles.  It also contains the object constructor
```

```
   // ``Circle,´´ corresponding to the function ``newCircle´´ and the
   // object destructor ``~Circle, corresponding to the function
   // ``deleteCircle´´ from the typecase version.  Note that the
   // compiler guarantees that the Circle´s methods are only called on
   // objects of type Circle.  The programmer does not need to keep an
   // explicit tag field in the object.
class Circle : public Shape {
 public:
   Circle(Pt* cn, float r) {
      center_ = new Pt(cn);
      radius_ = r;
   };

   virtual ~Circle() {
      delete center_;
   };

   virtual Pt* center() {
      return new Pt(center_);
   };

   void move(float dx, float dy) {
      center_->x += dx;
      center_->y += dy;
   };

   void rotate() {
         /* Rotating a circle is not a very interesting operation! */
   };

   void print() {
      printf("circle at  %.1f  %.1f  radius %.1f \n",
             center_->x, center_->y, radius_);
   };

 private:
   Pt*   center_;
   float radius_;
};


   // Class Rectangle consolidates the center, move, rotate, and print
   // functions for rectangles.  It also contains the object constructor
   // ``Rectangle,´´ corresponding to the function ``newRectangle´´ and the
   // object destructor ``~Rectangle, corresponding to the function
   // ``deleteRectangle´´ from the typecase version.  Note that the
   // compiler guarantees that the Rectangle´s methods are only called on
   // objects of type Rectangle.  The programmer does not need to keep an
   // explicit tag field in the object.
class Rectangle : public Shape {
 public:
   Rectangle(Pt* tl, Pt* br) {
      topleft_  = new Pt(tl);
```

```
      botright_ = new Pt(br);
   };

   virtual ~Rectangle() {
      delete topleft_;
      delete botright_;
   };

   Pt* center() {
      return new Pt((botright_->x - topleft_->x)/2,
                    (botright_->x - topleft_->x)/2);
   };

   void move(float dx,float dy) {
      topleft_->x  += dx;
      topleft_->y  += dy;
      botright_->x += dx;
      botright_->y += dy;
   };

   void rotate() {
      float d = ((botright_->x - topleft_->x) -
                 (topleft_->y - botright_->y))/2.0;
      topleft_->x  += d;
      topleft_->y  += d;
      botright_->x -= d;
      botright_->y -= d;
   };

   void print () {
      printf("rectangle coordinates  %.1f  %.1f   %.1f  %.1f \n",
             topleft_->x, topleft_->y,
             botright_->x, botright_->y);
   };

 private:
   Pt* topleft_;
   Pt* botright_;
};


   /*
    * The body of this program just tests some of the above functions.
    */
void main() {
   Pt* origin = new Pt(0,0);
   Pt* p1     = new Pt(0,2);
   Pt* p2     = new Pt(4,6);

   Shape* s1 = new Circle(origin, 2 );
   Shape* s2 = new Rectangle(p1, p2);

   s1->print();
```

```
   s2->print();

   s1->rotate();
   s2->rotate();

   s1->move(1,1);
   s2->move(1,1);

   s1->print();
   s2->print();

   delete s1;
   delete s2;

   delete origin;
   delete p1;
   delete p2;
}
```