34

Emulating object technology in non-O-O environments

Fortran, Cobol, Pascal, C, Basic, PL/I and even assembly language still account for a large part of the software being written or updated today. Clearly, a project using one of these languages will not be able to draw the full benefits of object technology, as this would require a notation such as the one we have studied in this book, and the supporting compiler, environment and libraries. But people who are required to use pre-O-O tools, often because of non-technical constraints, can still gain inspiration from object technology and use some of its concepts to improve the quality of their software development.

This chapter presents the techniques of *object emulation* that may enable you to approximate some of object technology. It will particularly examine the case of Fortran, Pascal and C. (Ada and other encapsulation languages were discussed in the preceding chapter; the following one covers O-O languages such as Simula, Smalltalk, Objective-C, C++ and Java.) This presentation will be directly applicable if you must use one of these languages. But it extends further:

- If you use another non-O-O language not on this list, such as Basic or Cobol, you should not have too much trouble transposing the concepts.
- Even if you are able to use an O-O language, the following discussion can give you a better grasp of the innovations of object technology and of the supporting implementation techniques (which often make use, internally, of older languages).

34.1 LEVELS OF LANGUAGE SUPPORT

In assessing how programming languages succeed in supporting object-oriented concepts, we may distinguish three broad categories (ignoring the lowest level, mostly containing assembly languages, which does not even support a routine construct):

- The **functional** level comprises languages whose unit of decomposition is the *routine*, a functional abstraction capturing a processing step. Data abstraction is handled, if at all, through definitions of data structures, either local to a routine or global.
- Languages at the **encapsulation** level provide a way to group a set of routines and data declarations in a syntactical unit, called a *module* or *package*; typically each unit can be compiled separately. This was discussed in some detail for Ada.

- Then we find **object-oriented** languages. This is not the place to be fussy about what exactly it takes to deserve this label — chapter 2 defined a set of criteria, and of course all of part C was devoted to analyzing O-O mechanisms in detail —, but we should at the very least expect some support for classes, inheritance, polymorphism and dynamic binding.

See [Wegner 1987].

> For the second category, encapsulation languages, which supports a data abstraction mechanism but no classes, inheritance, polymorphism or dynamic binding, you will find that the literature commonly uses the term **object-based**, introduced in an article by Peter Wegner. Because the English words *based* and *oriented* do not readily evoke the conceptual difference between encapsulation techniques and O-O languages, "object-based" is a little hard to justify, especially to newcomers. Although either terminology is acceptable once you have defined the conventions, I have in the end decided to stick here to the phrases "encapsulation languages" and "object-oriented languages", which more clearly conjure up the conceptual difference.

While we are on the subject of terminology: the term "functional language" is ambiguous since other parts of the literature apply it to a class of languages, based on mathematical principles and often deriving directly or indirectly from Lisp, which use side-effect-free functions instead of imperative constructs such as procedures and assignments. To avoid any confusion, the present book always uses the term *applicative* to denote this programming style. The word *function* in our use of "functional language" is to be contrasted with *object*, not (as when "functional" is a synonym for "applicative") with *procedure*. (To make a confusing situation worse, it is quite common to see "procedural" taken to mean "not object-oriented"! There is, however, no basis for such terminology; "procedural" normally means "imperative", as opposed to applicative; all the common O-O languages, including the notation of this book, are quite procedural.)

A general comment on O-O emulation. In its most basic form, object technology is "programming with abstract data types". You can apply a rudimentary form of the ideas, even at the functional level, by defining a set of strict methodological guidelines requiring every data access to go through routines. This assumes that you start from an object-oriented *design* that has defined ADTs and their features; then you will write a set of routines representing these features — *put*, *remove*, *item*, *empty* in our standard stack example — and require all client modules to go through these routines. This is a far cry from object technology proper, and can only work under the assumption that everyone in the team behaves; but, if you lack any kind of language support, it can be a start. We will call this technique the **disciplinary approach**.

34.2 OBJECT-ORIENTED PROGRAMMING IN PASCAL?

Pascal, introduced in 1970 by Niklaus Wirth, has been for many years the dominant language for teaching introductory programming in computing science departments, and has influenced many of the subsequent language designs. Pascal is definitely a functional language in the sense just defined.

Pascal proper

How much of the object-oriented approach can you implement in Pascal?

Not much. The Pascal program structure is based on a completely different paradigm. A Pascal program consists of a sequence of paragraphs, appearing in an immutable order: labels, constants, types, variables, routines (procedures and functions), and executable instructions. The routines themselves have the same structure, recursively.

This simple rule facilitates one-pass compilation. But it dooms any attempt at using O-O techniques. Consider what it takes to implement an ADT, such as the standard example of stacks represented by arrays: a few constants such as the array size, one or a few types such as the record type describing the stack implementation, a few variables such as the pointer to the stack top, and a few routines representing the operations on the abstract data type. In Pascal, these elements will be scattered all over the program: all the constants for various abstract data types together, all the types together and so on.

"Linguistic Modular Units", page 53.

The resulting program structure is the opposite of O-O designs. Using Pascal would contradict the Linguistic Modular Units principle, which expresses that any modular policy you choose must be supported by the available language constructs, for fear of damaging composability, decomposability and other modularity requirements.

So if we take Pascal as defined by its official standard, there is little we can do to apply O-O techniques this language beyond what was called the disciplinary approach above: imposing a strict methodological rule for data accesses.

Modular extensions of Pascal

Beyond standard Pascal, many commercially available versions remove the restrictions on the order of declarations and include support for some form of module beyond the routine, including separate compilation. Such modules may contain more than one routine, together with associated constants, types and routines. The resulting languages and products, more flexible and powerful than Pascal, are Pascal only by name; they are not standardized, and in fact resemble more an encapsulation language such as Modula-2 or Ada, to which the applicable discussion is that of the preceding chapter.

Object-oriented extensions of Pascal

Over the years a number of companies have offered object-oriented extensions of Pascal, loosely known as "Object Pascal". Two are particularly significant:

- Apple's version, originating from a language originally called *Clascal* and used for some of the software in Apple's Macintosh and its Lisa predecessor.
- Borland's version of Pascal, most recently adapted as the programming language for Borland's *Delphi* environment.

The preceding discussion does not really apply to such languages since — even more than with the modular extensions — their connection to the original Pascal is essentially their name, syntactic style, and statically typed approach. Borland Pascal, in particular, is an O-O language with exception handling. It does not, however, support any of the mechanisms of genericity, assertions, garbage collection and multiple inheritance.

34.3 FORTRAN

FORTRAN should virtually eliminate coding and debugging
FORTRAN Preliminary Report, IBM, November 1954

Cited in [Wexelblat 1981].

The oldest surviving programming language, Fortran remains widely used for scientific computation. Shockingly perhaps for people who went on from it to such "structured" languages as Pascal, you can in fact get a little more O-O frills in Fortran, although this is partly thanks to facilities that may be considered low-level and were intended for other goals.

The official name is FORTRAN, although the less obtrusive form is commonly used too.

Some context

Fortran was initially designed, as a tool for programming the IBM 704, by an IBM team under John Backus (later also instrumental in the description of Algol), with a first general release in 1957. Fortran II followed, introducing subroutines. Fortran IV solidified the language in 1966 (Fortran III, 704-specific, was not widely distributed), and was standardized by ANSI. The next revision process led to Fortran 77, actually approved in 1978, with better control structures and some simplifications. An even longer revision yielded Fortran 90 and Fortran 95, which have been diversely met and have not quite replaced their predecessors.

For most people with a computing science degree earned after the First World War, Fortran is old hat, and they would rather be caught reading the Intel 4044 User's Manual than admit they know anything about *FORMAT* and arithmetic *IF* instructions. In reality, however, quite a few programmed in Fortran at some stage, and many other people who are programmers by any objective criterion, even if their business card reads "theoretical physicist", "applied mathematician", "mechanical engineer" or even, in a few cases, "securities analyst", use Fortran as their primary tool day in and day out. Fortran remains in common use not only for maintaining old software but even for starting new projects.

To the outsider it sometimes seems that scientific programming — the world of Fortran — has remained aloof from much of the evolution in software engineering. This is partly true, partly not. The low level of the language, and the peculiar nature of scientific computing (software produced by people who, although scientists by training, often lack formal software education), have resulted in some software of less than pristine quality. But some of the best and most robust software also comes from that field, including advanced simulations of extremely complex processes and staggering tools for scientific visualization. Such products are no longer limited to delicate but small numerical algorithms; like their counterparts in other application areas, they often manipulate complex data structures, rely on database technology, include extensive user interface components. And, surprising as it may seem, they are still often written in Fortran.

The *COMMON* technique

A Fortran system is made of a main program and a number of routines (subroutines or functions). How can we provide a semblance of data abstraction?

The usual technique is to represent the data through a so-called *COMMON* block, a Fortran mechanism for making data accessible to any routine that cares to want it, and to implement each of the associated exported features (such as *put* etc. for stacks) through a separate routine. Here for example is a sketch of a *put* routine for a stack of real numbers:

A C at the first position on a line introduces a comment.

```
      SUBROUTINE RPUT (X)
      REAL X
C
C     PUSH X ON TOP OF REAL STACK
C
      COMMON /STREP/ TOP, STACK (2000)
      INTEGER TOP
      REAL STACK
C
      TOP = TOP + 1
      STACK (TOP) = X
      RETURN
      END
```

This version does not have any overflow control; clearly it should be updated to test for *TOP* going over the array size. (The next version will correct this.) The function to return the top element is

```
      INTEGER FUNCTION RITEM
C
C      TOP ELEMENT OF REAL STACK
C
      COMMON /STREP/ TOP, STACK (2000)
      INTEGER TOP
      REAL STACK

      RITEM = STACK (TOP)
      RETURN
      END
```

which would similarly need to test for underflow (empty stack). *REMOVE* and other features will follow the same pattern. What unites the different routines, making sure that they access the same data, is simply the name of the common block, *STREP*. (It is in fact possible, in different routines, to pretend that the same common block contains data of different types and sizes if the total memory occupied somehow coincides, although in a family-oriented book like this one it is probably preferable to avoid going into details that might not be entirely suitable for the younger members of the audience).

The limitations are obvious: this implementation describes **one abstract object** (one particular stack of reals), not an abstract data type of which the software can create arbitrarily many instances at run time, as with a class. The Fortran world is very static: you must dimension all the arrays (here to 2000, a number picked arbitrarily). Because there is no genericity, you should in principle declare a new set of routines for each type of stack; hence the names *RPUT* and *RITEM*, where the *R* stands for Real. One can work around some of these problems, but not without considerable effort.

The multiple-entry subroutine technique

The *COMMON*-based technique, as you will have noted, violates the Linguistic Modular Units principle. In a system's modular structure, the routines are physically independent although conceptually related. You can all too easily update one and forget the others.

It is in fact possible to improve on this situation (without removing some of the other limitations just listed) through a language trait legalized by Fortran 77: multiple entry points to a single routine.

This extension — which was probably introduced for different purposes, but may be redeemed for the "good cause" — enables Fortran routines to have entry points other than the normal routine header. Client routines may call these entry points as if they were autonomous routines, and the various entries may indeed have different arguments. Calling an entry will start execution of the routine at the entry point. All entries of a routine share the persistent data of the routine; a persistent data item, which in Fortran 77 must appear in a *SAVE* directive, is one whose value is retained from one activation of a routine to the next. Well, you see where we are driving: we can use this technique to define a module that encapsulates an abstract object, almost as we would in one of the encapsulation languages. In Ada, for example, we could write a package with a data structure declaration, such as a stack representation, and a set of routines that manipulate these data. Here we will simulate the package with a subroutine, the data structure with a set of declarations that we make persistent through a *SAVE*, and each Ada routine (each feature of the corresponding class in an O-O language) with an entry. Each such entry must be followed by the corresponding instructions and a *RETURN*:

```
ENTRY (arguments)
… Instructions …
RETURN
```

so that the various entry-delimited blocks are disjoint: control never flows from one block to the next. This is a restricted use of entry points, which in general are meant to allow entering a routine at any point and then continuing in sequence. Also note that clients will never call the enclosing subroutine under its own name; they will only call the entries.

The main difference with the preceding *COMMON*-based solution is that all the features of the underlying abstract data type now appear in the same syntactical unit. The second part of the facing page shows an example implementing an abstract object (stack of reals). The calls from a client will look like this:

```
        LOGICAL OK
        REAL X
C
        OK = MAKE ()
        OK = PUT (4.5)
        OK = PUT (–7.88)
        X = ITEM ()
        OK = REMOVE ()
        IF (EMPTY ()) A = B
```

Look at this text for just a second, from a distance; you could almost believe that it is the use of a class, or at least of an object, through its abstract, officially defined interface!

A Fortran routine and its entry points must be either all subroutines, or all functions. Here since *EMPTY* and *ITEM* must be functions, all other entries are also declared as functions, including *MAKE* whose result is useless.

A stack module emulation in Fortran

```
C     -- IMPLEMENTATION OF ONE
C     -- ABSTRACT STACK OF REALS
C
      INTEGER FUNCTION RSTACK ()
      PARAMETER (SIZE=1000)
C
C     -- REPRESENTATION
C
      REAL IMPL (SIZE)
      INTEGER LAST
      SAVE IMPL, LAST
C
C     -- ENTRY POINT DECLARATIONS
C
      LOGICAL MAKE
      LOGICAL PUT
      LOGICAL REMOVE
      REAL ITEM
      LOGICAL EMPTY
C
      REAL X
C
C     -- STACK CREATION
C
      ENTRY MAKE ()
         MAKE = .TRUE.
         LAST = 0
      RETURN
C
C        -- PUSH AN ITEM
C
      ENTRY PUT (X)
         IF (LAST .LT. SIZE) THEN
            PUT = .TRUE.
            LAST = LAST + 1
            IMPL (LAST) = X
      ELSE
         PUT = .FALSE.
      END IF
      RETURN
C     -- REMOVE TOP ITEM
C
      ENTRY REMOVE (X)
         IF (LAST .NE. 0) THEN
            REMOVE = .TRUE.
            LAST = LAST – 1
         ELSE
            REMOVE = .FALSE.
         END IF
      RETURN
C
C     -- TOP ITEM
C
      ENTRY ITEM ()
         IF (LAST .NE. 0) THEN
            ITEM = IMPL (LAST)
         ELSE
            CALL ERROR
     *         ('ITEM: EMPTY STACK')
         END IF
      RETURN
C
C     -- IS STACK EMPTY?
C
      ENTRY EMPTY ()
         EMPTY = (LAST .EQ. 0)
      RETURN
C
      END
```

This style of programming can be applied successfully to emulate the encapsulation techniques of Ada or Modula-2 in contexts where you have no choice but to use Fortran. It suffers of course from stringent limitations:

- No internal calls are permitted: whereas routines in an object-oriented class usually rely on each other for their implementations, an entry call issued by another entry of the same subroutine would be understood as an instance of recursion — anathema to Fortran, and run-time disaster in many implementations.
- As noted, the mechanism is strictly static, supporting only one abstract object. It may be generalized to allow for a fixed number of objects (by transforming every variable into a one-dimensional array, and adding a dimension to every array). But there is no portable support for dynamic object creation.
- In practice, it seems that some Fortran environments (two decades after Fortran 77 was published!) do not deal too well with multiple-entry subroutines; in particular debuggers do not always know how to keep track of multiple entries. Before applying this technique to a production development, check with the local Fortran guru to find out whether it is wise to rely on this facility in your environment.
- Finally, the very idea of hijacking a language mechanism for purposes other than its probable design objective raises dangers of confusion and errors.

34.4 OBJECT-ORIENTED PROGRAMMING AND C

Born in a log cabinet, C quickly rose to prominence. Although most people interested in both C and object technology have focused on the O-O extensions of C discussed in the next chapter (C++, Objective-C, Java), it remains interesting to see how C itself can be made to emulate O-O concepts, if only to understand the techniques that have made C so useful as a stepping stone towards the implementation of more advanced languages.

Some context

C was designed at AT&T's Bell Laboratories as a portable language for writing operating systems. The first version of Unix had used assembly language, but a portable version soon appeared necessary, and C was designed around 1970 to make it possible. It was derived from ideas found in BCPL, a language of the sixties which, like C, can be mentioned in the same breath as "high-level", "machine-oriented" and "portable": high-level thanks to control structures comparable to those of Algol or Pascal; machine-oriented because you can manipulate data at the most elementary level, through addresses, pointers and bytes; portable because the machine-oriented concepts are so defined as to cover a wide variety of computer types.

C's timing could not have been better. In the late seventies Unix became the operating system of choice for many universities, and C spread with it. Then in the eighties the microcomputer revolution burst out, and C was ready to serve as its *lingua franca* — more scalable than Basic, more flexible than Pascal. At the same time Unix also enjoyed some commercial success, and along with Unix still came C. In a few years, a boutique product became the dominant language in large segments of the computing industry, including much of where the action really was.

Anyone interested in the progress of programming languages — even people who do not care too much for the language itself — has a political debt to C, and sometimes a technical one as well:

- Politically, C ended the fossilized situation that prevailed in the programming language world until around 1980. No one in industry wanted to hear (particularly after the commercial failure of Algol) about anything else than the sacred troika, Fortran for science, Cobol for business and PL/I for true blue shops. Outside of academic circles and a few R&D departments, any attempt at suggesting other solutions was met with as much enthusiasm as if it were a proposal to introduce a third brand of Cola drink. C broke that mindset, making it acceptable to think of the programming language as something you choose from a reasonably broad and evolving catalog. (A few years later, C itself became so entrenched that in some circles the choices seemed to have gone from three to one, but it is the fate of successful subversives that they become the new Establishment.)

- Technically, the portability and machine-closeness of C have made it an attractive solution as a target language of compilers for higher-level languages. The first C++ and Objective-C implementations used this approach, and compilers for many other languages, often having no visible connection to C, have followed their example. The advantages for the compiler writers and their users are: portability, since you can have a single C-generating compiler for your language and use C compilers (available nowadays for almost any computer architecture) to take care of platform dependencies; efficiency, since you can rely on the extensive optimization techniques that have been implemented in good C compilers; and ease of integration with ubiquitous C-based tools and components.

With time, the contradiction between the two views of C — high-level programming language, and portable assembly language — has become more acute. Recent evolution of the ANSI standard for C (first published in 1990, following the earlier version known as *K&R* from the authors of the first C book, Kernighan and Ritchie) have made the language more typed — and hence less convenient for its use as a compiler's target code. It has even been announced that forthcoming versions will have a notion of class, obscuring the separation from C++ and Java.

[Kernighan 1978], [Kernighan 1988].

Although an O-O extension of C simpler than C++ and Java may be desirable, one can wonder whether this evolution is the right one for C; a hybrid C-based O-O language will always remain a strange contraption, whereas the idea of a simple, portable, universally available, efficiently compilable machine-oriented language, serving both as a target language for high-level compilers and as a low-level tool for writing very short routines to access operating system and machine-dependent facilities (that is to say, for doing the same thing that assembly language used to do for C, only at the next level) remains as useful as it ever was.

Basics

As with any other language, you can apply to C the "disciplinary" technique of restricted data access, requiring all uses of data structures to go through functions. (All routines in C are functions; procedures are viewed as functions with a "void" result type.)

Beyond this, the notion of file may serve to implement higher-level modules. Files are a C notion on the borderline between the language and the operating system. A file is a compilation unit; it may contain a number of functions and some data. Some of the functions may be hidden from other files, and some made public. This achieves encapsulation: a file may contain all the elements pertaining to the implementation of one or more abstract objects, or an abstract data type. Thanks to this notion of file, you can essentially reach the *encapsulation language* level in C, as if you had Ada or Modula-2. As compared to Ada, however, you will be missing genericity and the distinction between specification and implementation parts.

In practice, a commonly used C technique is rather averse to O-O principles. Most C programs use "header files", which describe shared data structures. Any file needing the data structures will gain access to them through an "include" directive (handled by the built-in C preprocessor) of the form

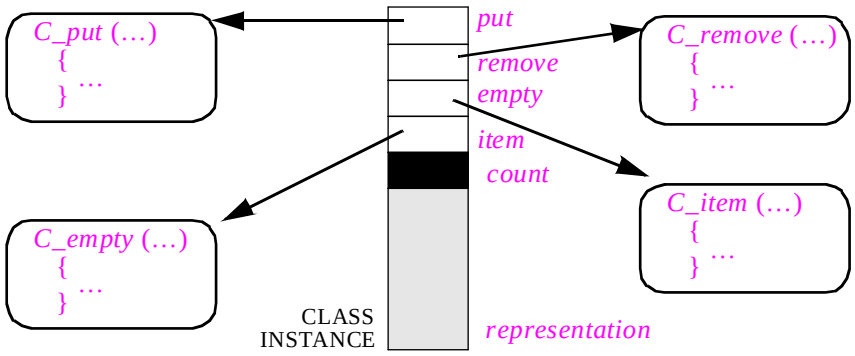
#include <*header.h*>

where *header.h* is the name of the header file (*.h* is the conventional suffix for such file names). This is conceptually equivalent to copying the whole header file at the point where the directive appears, and allows the including file to access directly the data structure definitions of the header file. As a result the C tradition, if not the language itself, encourages client modules to access data structures through their physical representations, which clearly contradicts the principles of information hiding and data abstraction. It is possible, however, to use header files in a more disciplined fashion, enforcing rather than violating data abstraction; they can even help you go some way towards defining interface modules in the style we studied for Ada in the preceding chapter.

Emulating objects

Beyond the encapsulation level, one of the more specialized and low-level features of C — the ability to manipulate pointers to functions — can be used to emulate fairly closely some of the more advanced properties of a true O-O approach. Although it is sufficiently delicate to suggest that its proper use is by compilers for higher-level languages rather than C programmers, it does deserve to be known.

In we take a superficial look at the notion of object as it exists in object technology, we might say that "every object has access to the operations applicable to it". This is a little naïve perhaps, but not altogether wrong conceptually. If, however, we take this view literally, we find that C directly supports the notion! It is possible for an instance of a "structure type" of C (the equivalent of record types in Pascal) to contain, among its fields, pointers to functions.



A C object with function references

For example, a C structure type *REAL_STACK* may be declared by the type definition

```
typedef struct
    {
    /* Exported features */
        void (*remove) ();
        void (*put) ();
        float (*item) ();
        BOOL (*empty) ();
    /* Secret features (implementation) */
        int count;
        float representation [MAXSIZE];
    }
    REAL_STACK;
```

The braces { } delimit the components of the structure type; *float* introduces real numbers; procedures are declared as functions with a **void** result type; comments are delimited by /* and */. The other asterisks * serve to de-reference pointers; the idea in the practice of C programming is that you add enough of them until things seem to work, and if not you can always try a & or two. If this still does not succeed, you will usually find someone who knows what to do.

Here the last two components are an integer and an array; the others are references to functions. In the declaration as written, the comments about exported and secret features apply to the emulated class, but everything is in fact available to clients.

Each instance of the type must be initialized so that the reference fields will point to appropriate functions. For example, if *my_stack* is a variable of this type and *C_remove* is a stack popping function, you may assign to the *remove* field of the *my_stack* object a reference to this function, as follows:

```
my_stack.remove = C_remove
```

In the class being emulated, feature *remove* has no argument. To enable the *C_remove* function to access the appropriate stack object, you must declare it as

```
C_remove (s)
        REAL_STACK s;
    {
        … Implementation of remove operation …
    }
```

so that a client may apply *remove* to a stack *my_stack* under the form

my_stack.*remove* (*my_stack*)

More generally, a routine *rout* which would have n arguments in the class will yield a C function *C_rout* with $n+1$ arguments. An object-oriented routine call of the form

$x.rout\ (arg_1, arg_2, \ldots, arg_n)$

will be emulated as

$x.C\_rout\ (x, arg_1, arg_2, \ldots, arg_n)$

Emulating classes

Exercise E34.3, page 1112.

The preceding technique will work to a certain extent. It can even be extended to emulate inheritance.

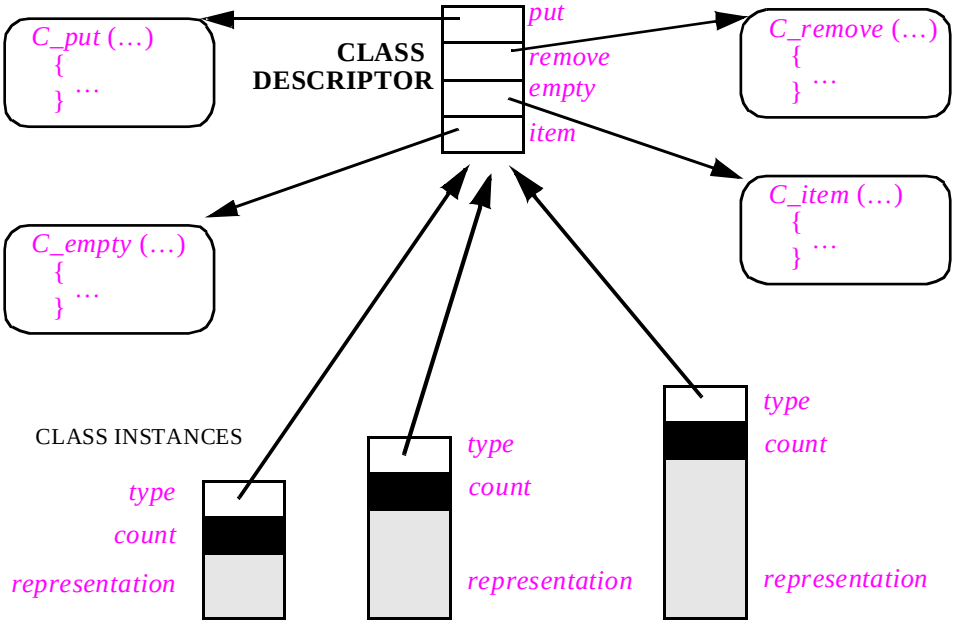
But it is inapplicable to any serious development: as illustrated in the figure of the preceding page, it implies that **every instance** of every class physically contains references to all the routines applicable to it. The space overhead would be prohibitive, especially with inheritance.

To bring this overhead down to an acceptable level, notice that the routines are the same for all instances of a class. So we may introduce for each class a run-time data structure, the **class descriptor**, containing references to the routines; we can implement it as a linked list or an array. The space requirements decrease dramatically: instead of one pointer per routine per object, we can use one pointer per routine *per class*, plus *one pointer per object* giving access to the class descriptor, as shown by the figure at the top of the following page.

Timewise we pay the price of an indirection: as shown in the figure, you have to go through the descriptor to find the function applicable to an object. The space economy and the simplification seem well worth this penalty.

There is no secret about it: the technique just sketched is what has made C useful as an implementation vehicle for object-oriented languages, starting with Objective-C and C++ in the early eighties. The ability to use function pointers, combined with the idea of grouping these pointers in a class descriptor shared by an arbitrary number of instances, yields the first step towards implementing O-O techniques.

This is only a first step, of course, and you must still find techniques for implementing inheritance (multiple inheritance in particular is not easy), genericity, exceptions, assertions and dynamic binding. To explain how this can be done would take another book. Let us, however, note one important property, deducible from what we have



C objects sharing a class descriptor

seen so far. Implementing dynamic binding, regardless of the details, will require run-time access to the type of each object, to find the proper variant of the feature *f* in a dynamically bound call *x*.*f* (…) (written here in O-O notation). In other words: in addition to its official fields, defined explicitly by the software developer through type declarations, each object will need to carry an extra internal field, generated by the compiler and accessible only to the run-time system, indicating the **type** of the object. Well, with the approach just defined, we already have a possible implementation of this type field — as a pointer to the class descriptor. This is the reason why the above figure uses the label *type* for such fields.

O-O C: an assessment

This discussion has shown that implementation techniques are available in C to emulate object-oriented ideas. But it does not mean that programmers should use these techniques. As with Fortran, the emulation does violence to the language. C's main strength is, as noted, its availability as a "structured assembly language" (a successor to BCPL and Wirth's PL/360), portable, reasonably simple and efficiently interpreted. Its basic concepts are very far from those of object-oriented design.

The danger in trying to force an object-oriented peg into a C hole is to get an inconsistent construction, impairing the software development process and the quality of the resulting products. Better use C for what it does well: small interfaces to low-level hardware or operating system facilities, and machine-generated target code; then when the time comes to apply object technology we should use a tool designed for that purpose.

34.5 BIBLIOGRAPHICAL NOTES

Techniques for writing Fortran packages based on the principles of data abstraction are described in [M 1982a]. They use routines sharing *COMMON* blocks, rather than multiple-entry routines. They go further in their implementation of object-oriented concepts than the techniques described in this chapter, thanks to the use of specific library mechanisms that provides the equivalent of dynamically allocated class instances. Such mechanisms, however, require a significant investment, and will have to be ported anew to each platform type.

I am indebted to Paul Dubois for pointing out that the multiple-entry Fortran technique, although definitely part of the standard, is not always supported well by current compilers.

[Cox 1990] (originally 1986) contains a discussion of C techniques for the implementation of object-oriented concepts.

The basic reference on the history of classical programming languages is a conference proceedings [Wexelblat 1981]; see [Knuth 1980] for the earliest efforts.

EXERCISES

E34.1 Graphics objects (for Fortran programmers)

Write a set of Fortran multiple-entry routines that implement basic graphics objects (points, circles, polygons). For a specification of the abstractions involved and the associated operations, you may rely on the GKS graphics standard.

E34.2 Genericity (for C programmers)

How would you transform the C emulation of a “real stack” class declaration into an emulated generic declaration, easy to adapt to stacks of any type *G* rather than just **float**?

E34.3 Object-oriented programming in C (term project)

Design and implement a simple object-oriented extension of C using the ideas of this chapter. You may write either a pre-processor, translating an extended version of the language into C, or a function package that does not change the language itself.

Approach the problem through three successive refinements:

- Implement first a mechanism allowing objects to carry their own references to available routines.
- Then see how to factor routine references at the class level.
- Finally, study how to add single inheritance to the mechanism.