18

Global objects and constants

Local knowledge is not always enough; components of a software system may need to access global information. It is easy to think of examples: a shared value, such as the size of available memory; an error window, to which all the components of an interactive system must be able to output messages; the gateway to a database or network.

In classical approaches, it is not difficult to provide for global objects; you just declare them as global variables, owned by the main program. In the modular style of design made possible by object-oriented techniques, there is neither a main program nor global variables. But even if our software texts do not include global variables our software executions may still need to share objects.

Such global objects pose a challenge to the method. Object technology is all about decentralization, all about modularity, all about autonomy. It has developed from the beginning of this presentation as a war of independence for the modules, each fighting for its freedom from the excesses of central authority. In fact, there is no central authority any more. How then do we satisfy the need for common institutions? In other words, how do we allow components to share data in a simple way, without jeopardizing their autonomy, flexibility and reusability?

It will not work, of course, to pass shared objects as arguments to the modules that need them. This would soon become clumsy if too many components need them. Besides, argument passing assumes that one module owns the value and then passes it on to others; in the case of a truly shared value no one module can claim ownership.

To find a better answer we will start from a well-known notion, which we need in object-oriented software construction just as much as we did in more traditional approaches: constants. What is, after all, a constant such as *Pi* if not a simple object shared by many modules? Generalizing this notion to more complex objects will provide a first step towards fully general constant and shared objects.

18.1 CONSTANTS OF BASIC TYPES

Let us start with a simple notation to denote constant values.

Using symbolic constants

A rule of software style, the **Symbolic Constant principle**, states that when an algorithm refers to a certain value — a number, a character, a string… — it should almost never use it directly. Instead, a declaration should associate a name with the value, so that the algorithm can use the name (known as a *symbolic* constant) rather than the value (known as a *manifest* constant). Two reasons justify this principle:

"Manifest and symbolic constants", page 884. See also "Modular continuity", page 44.

- Readability: someone who reads your software may not understand what the value 50 is doing in a certain algorithm; if instead you use the symbolic constant *US_states_count* everything is clear.
- Extendibility: in practice, with a few exceptions (such as the value of π, unlikely to change soon), the only constant thing about constants is change. To update the value of a constant it suffices, if you have been using symbolic constants, to change one declaration. This is much nicer than having to chase throughout the software for all the places that may have relied on the earlier value.

The principle permits using manifest constants (hence the word "almost" above) for zero elements of various operations, as in a loop **from** *i* := *1* **until** *i* > *n* … iterating over the elements of an array whose numbering follows the default convention of starting at 1. (But *n* should be symbolic, not manifest.)

Although few software developers apply the Symbolic Constant principle as systematically as they should, the benefits of declaring a symbolic constant are well worth the small extra effort. So we need a clear and simple way of defining symbolic constants in an O-O framework.

Constant attributes

A symbolic constant, like everything else, will be defined in a class.We will simply treat a constant value as an attribute which happens to have a fixed value, the same for all instances of the class.

For the syntax, we can reuse the keyword **is** which already serves to introduce routines; only here it will be followed by a value of the appropriate type, rather than by an algorithm. The following examples include one for each of the basic types *INTEGER*, *BOOLEAN*, *REAL* and *CHARACTER*:

```
Zero: INTEGER is 0
Ok: BOOLEAN is True
Pi: REAL is 3.1415926524
Backslash: CHARACTER is '\'
```

Backslash is of type *CHARACTER*, its value a single character. Constants of string type, denoting character strings of arbitrary length, will be discussed below.

As these examples illustrate, the recommended style convention for names of constant attributes is to start with a capital letter, with the rest in lower case.

A descendant may not redefine the value of a constant attribute.

Like other attributes, constant attributes are either exported or secret; if they are exported, clients of the class may access them through feature calls. So if *C* is the class containing the above declarations and *x*, declared of type *C*, has a non-void value, then *x*.*Backslash* denotes the backslash character.

Unlike variable attributes, constant attributes do not occupy any space at run time in instances of the class. So there is no run-time penalty for adding as many constant attributes as you need.

18.2 USE OF CONSTANTS

Here is an example showing how clients may use constant attributes defined in a class:

```
class FILE feature
        error_code: INTEGER;      -- Variable attribute
        Ok: INTEGER is 0
        Open_error: INTEGER is 1
        …
        open (file_name: STRING) is
                        -- Open file of name file_name
                        -- and associate it with current file object
                do
                        error_code := Ok
                        …
                        if "Something went wrong" then
                                error_code := Open_error
                        end
                end
        … Other features …
end
```

A client may call *open* and compare the resulting error code to any of the constants to test how the operation went:

```
f: FILE; …
f.open
if f.error_code = f.Open_error then
        "Appropriate action"
else
        …
end
```

Often, however, a group of constants is needed without being attached to any particular object. For example, a system performing physics computations may use some numerical constants; or a text editor may need character constants describing the character

keys associated with various commands. In such a case, the constants will still be grouped in a class (where else could they be?), but there will not be any instances of that class; it is simply used as parent for the classes that need to access the constants, as in

```
class EDITOR_CONSTANTS feature
    Insert: CHARACTER is 'i'
    Delete: CHARACTER is 'd'; -- etc.
    …
end

class SOME_CLASS_FOR_THE_EDITOR inherit
    EDITOR_CONSTANTS
    … Other possible parents …
feature …
    … Routines of the class have access to the constants
      declared in EDITOR_CONSTANTS …
end
```

A class such as *EDITOR_CONSTANTS* is used only to host a group of related constants, and its role as an "abstract data type implementation" (our working definition of the notion of class) is less obvious than in earlier examples. But it definitely serves a useful purpose. We will examine its theoretical justification in a later chapter.

See "FACILITY INHERITANCE", 24.9, page 847.

The scheme shown would not work without multiple inheritance, since *SOME_CLASS_FOR_THE_EDITOR* may need other parents, either for access to other constants or for more standard uses of inheritance.

18.3 CONSTANTS OF CLASS TYPES

Symbolic constants, allowing you to use identifiers to denote certain constant values, are not just useful for predefined types such as *INTEGER*; the need also arises for types that developers have defined, through classes. Here the solution is less obvious.

Manifest constants are inappropriate for class types

A typical example in which you may need to define a constant for a non-basic types is that of a class describing complex numbers:

```
class COMPLEX creation
    make_cartesian, make_polar
feature
    x, y: REAL
            -- Real and imaginary parts
```

```
        make_cartesian (a, b: REAL) is
                -- Initialize with real part a, imaginary part b.
            do
                x := a; y := b
            end
        … Other routines (x and y are the only attributes) …
    end
```

You may want to define the complex number *i*, with real part 0 and imaginary part 1. The first idea that comes to mind is a manifest constant notation such as

```
i: COMPLEX is "Expression specifying the complex number (0, 1)"
```

How can you write the expression after **is**? For simple types, the manifest constants were self-evident: *345* is a constant of type integer, *'A'* of type character. But no such predefined notation is available for developer-defined class types.

One could imagine a notation based on the attributes of the class; something like

Not a retained notation. For purposes of illustration only.

```
i: COMPLEX is COMPLEX (0, 1)
```

But such an approach (although present in some O-O languages) is incompatible with the principles of modularity which serve as the basis for object technology. It would mean requiring clients of *COMPLEX* to describe constants in terms of the implementation. This breaks information hiding. You could not add an attribute, even a secret one, without invalidating client code; neither could you re-implement an attribute such as *x* as a function (to switch internally to a polar representation).

Besides, how could you make sure that such manifest constants will satisfy the class invariant if there is one?

This last remark opens the way to a correct solution. An earlier chapter noted that it is the responsibility of the **creation procedures** to make sure that every object satisfies the invariant immediately upon creation. Creating objects in any other way (apart from the safe companion mechanism, *clone*) would lead to error situations. So we should look for a mechanism that, rather than manifest objects in the above style, will rely on the usual technique for object creation.

Once functions

We may view a constant object as a function. For example *i* could be defined within class *COMPLEX* itself as

```
    i: COMPLEX is
            -- Complex number with real part 0 and imaginary part 1
        do
            !! Result.make_cartesian (0, 1)
        end
```

This almost does the job, since the function will always return a reference to an object of the desired form. Since we rely on normal creation procedures, the invariant will be satisfied, so we will only produce consistent objects.

The result, however, is not exactly what we need: each client use of *i* in the client produces a new object, identical to all the others. This is a waste of time and space:

To get the proper behavior, we need a special kind of function: one which executes its body only the first time it is called. We can call this a **once function**. A once function is otherwise similar to a normal function; syntactically, it will be distinguished by the keyword **once**, replacing the usual **do**, to introduce the body:

```
i: COMPLEX is
        -- Complex number with real part 0 and imaginary part 1
    once
        !! Result.make_cartesian (0, 1)
    end
```

The only change

The first time a once function is called during a system's execution, it executes its body. In the example this creates an object representing the desired complex number, and returns a reference to that object. Every subsequent call executes no instruction at all, but terminates immediately, returning the result computed the first time around.

Regarding efficiency: a call to *i* other than the first should take only marginally longer than an attribute access.

The result computed by the first call to a once function is applicable to all instances of a class, in the general sense of the word "instance" covering instances of descendants as well, except of course for any descendant that redefines the function. As a consequence you can freely redefine functions from once to non-once and conversely. Here if a descendant *COMPLEX1* of *COMPLEX* redefines *i*, a call to *i* on an instance of *COMPLEX1* will use the redefined version (whether once or non-once); a call on a direct instance of *COMPLEX* or a descendant other than *COMPLEX1* will use the once function, that is to say the value computed by the first such call.

18.4 APPLICATIONS OF ONCE ROUTINES

The notion of once routine extends beyond examples such as *i* to more general applications: shared objects, global system parameters, initialization of common properties.

Shared objects

For reference types such as *COMPLEX*, as you may have noted, the "once" mechanism actually offers constant *references*, not necessarily constant *objects*. It guarantees that the body of the function is executed only once, to compute a result, which later calls will also return without further computation.

If the function returns a value of a reference type, its body will usually contain a creation instruction, as in the example of *i*. All calls will return a reference to the object

created by the first. Although the creation will never be executed again, nothing prevents callers from modifying the object through the reference. Therefore the mechanism provides **shared** objects rather than constant ones.

An example of a shared object, cited at the beginning of this chapter, is a window showing error messages in an interactive system. Assume we have decided that any component of the system that detects a user error may output a message to that window, through a call of the form

```
Message_window.put_text ("Appropriate error message")
```

Here *message_window* is of type *WINDOW*, with class *WINDOW* declared as

```
class WINDOW creation
    make
feature
    make (…) is
            -- Create window at size and position indicated by arguments.
        do … end
    text: STRING
        -- Text to be displayed in window
    put_text (s: STRING) is
            -- Make s the text to be displayed in window.
        do
            text := s
        end
    … Other features …
end -- class WINDOW
```

Obviously *Message_window* must be the same for all components of the system. This is achieved by declaring the corresponding feature as a once function:

```
Message_window: WINDOW is
        -- Window where error messages will be output
    once
        !! Result.make ("…Size and position arguments…")
    end
```

In this case the message window object must be shared by all its users, but it is not a constant object: each call to *put_text* changes the object by putting its own chosen text in it. The best place to declare *Message_window* is a class from which all system components needing access to the message window will inherit.

In the case of a shared object that denotes a constant, such as *i*, you may want to disallow calls of the form *i.some_procedure* that might change the fields. To achieve this, simply include clauses *i.x = 0* and *i.y = 1* in the class invariant.

Once functions returning results of basic types

Another application of once functions is to represent global values — "system parameters" — used by several classes in a system. Such values will usually be constant over a given system execution; they are initially computed from user input, or from some information obtained from the environment. For example:

- The components of a low-level system may need to know the available memory space, obtained from the environment at initialization time.
- A terminal handler may start by querying the environment about the number of terminal ports: once obtained, these data elements are then used by several modules of the application.

Such global values are similar to shared objects such as *Message_window*; but in general they are values of basic types rather than class instances. You may represent them through once functions. The scheme is:

```
Const_value: T is
        -- A system parameter computed only once
    local
        envir_param: T'        -- Any type (T or another)
    once
        "Get the value of envir_param from the environment"
        Result := "Some value computed from envir_param"
    end
```

Such once functions of basic types describe dynamically computed constants.

Assume the above declaration is in a class *ENVIR*. A class needing to use *Const_value* will get it simply by listing *ENVIR* among its parents. There is no need here for an initialization routine as might be used in classical approaches to compute *Const_value*, along with all other global parameters, at the beginning of system execution. As was seen in an earlier chapter, such a routine would have to access the internal details of many other modules, and hence would violate the criteria and principles of modularity: decomposability, few interfaces, information hiding etc. In contrast, classes such as *ENVIR* may be designed as coherent modules, each describing a set of logically related global values. The first component that requests the value of a global parameter such as *Const_value* at execution time will trigger its computation from the environment.

See "Modular decomposability", page 40.

Although *Const_value* is a function, components that use it may treat it as if it were a constant attribute.

The introduction to this chapter mentioned that none of the modules that use a shared value has more claim to own it than any of the others. This is especially true in the cases just seen: if, depending on the order of events in each execution of the system, any one among a set of modules may trigger the computation of the value, it would be improper to designate any single one among them as the owner. The modular structure reflects this.

Once procedures

> *The function close should only be called once. We recommend using a global variable in your application to check that close is not called more than once.*
>
> (From the manual for a commercial C library.)

The "once" mechanism is interesting not just for functions but for procedures as well.

A once procedure is appropriate when some facility used on a system-wide basis must be initialized, but it is not known in advance which system component will be the first to use the facility. It is like having a rule that whoever comes in first in the morning should turn on the heating.

A simple example is a graphics library providing a number of display routines, where the first display routine called in any system execution must set up the terminal. The library author could of course require every client to perform a setup call before the first display call. This is a nuisance for clients and does not really solve the problem anyway: to deal properly with errors, any routine should be able to detect that it has been called without proper setup; but if it is smart enough to detect this case, the routine might just as well do the setup and avoid bothering the client!

Once procedures provide a better solution:

```
check_setup is
        -- Perform terminal setup if not done yet.
    once
        terminal_setup -- Actual setup action
    end
```

Then every display routine in the library should begin with a call to *check_setup*. The first call will do the setup; subsequent ones will do nothing. Note that *check_setup* does not have to be exported; client authors do not need to know about it.

This is an important technique to improve the usability of any library or other software package. Any time you can remove a usage rule — such as "Always call procedure *xyz* before the first operation" — and instead take care of the needed operations automatically and silently, you have made the software better.

Arguments

Like other routines, once routines — procedures and functions — can have arguments. But because of the definition of the mechanism, these arguments are only useful in the call that gets executed first.

In the earlier analogy, imagine a thermostat dial which anyone coming into the building may turn to any marking, but such that only the first person to do so will set the temperature: subsequent attempts have no effect.

Once functions, anchoring and genericity

(This section addresses a specific technical point and may be skipped on first reading.)

Once functions of class types carry a potential incompatibility with anchored types and genericity.

Let us start with genericity. In a generic class *EXAMPLE* [*G*] assume a once function returning a value whose type is the formal generic parameter:

```
f: G is once ... end
```

Warning: not valid. See below.

and consider a possible use:

```
character_example: EXAMPLE [CHARACTER]
...
print (character_example.f)
```

So far so good. But you also try to do something with another generic derivation:

```
integer_example: EXAMPLE [INTEGER]
...
print (integer_example.f + 1)
```

The last instruction adds two integer values. Unfortunately, the first of them, the result of calling *f*, has already been computed since *f* is a once function; and it is a character, not an integer. The addition is not valid.

The problem is that we are sharing a value between different generic derivations which expect the type of that value to depend on the actual generic parameter.

A similar issue arises with anchored types. Assume a class *B* which adds an attribute to the features of its parent *A*:

```
class B inherit A feature
        attribute_of_B: INTEGER
end
```

Assume that *A* had a once function *f*, returning a result of anchored type:

```
f: like Current is once !! Result.make end
```

Warning: not valid. See below.

and that the first evaluation of *f* is in

```
a2 := a1.f
```

with *a1* and *a2* of type *A*. The evaluation of *f* creates a direct instance of *A*, and attaches it to entity *a2*, also of type *A*. Fine. But assume now that a subsequent use of *f* is

```
b2 := b1.f
```

where *b1* and *b2* are of type *B*. If *f* were a non-once function, this would not cause any problem, since the call would now produce and return a direct instance of *B*. Since here we have a once function, the result has already been computed through the first call; and that result is a direct instance of *A*, not *B*. So an instruction such as

```
print (b2.attribute_of_B)
```

will try to access a non-existent field in an object of type *A*.

The problem is that anchoring causes an implicit redefinition. Had *f* been explicitly redefined, through a declaration appearing in *B* under the form

f: *B* **is once** !! *Result*.*make* **end**

assuming that the original in class *A* similarly returned a result of type *A* (rather than **like** *Current*), then we would not have any trouble: direct instances of *A* use the *A* version, direct instances of *B* use the *B* version. Anchoring, of course, was introduced precisely to rid us of such explicit redefinitions serving type needs only.

These two cases are evidence of incompatibilities between the semantics of once functions (procedures are fine) and the results of either anchored or formal generic types.

One way out, suggested by the last observation on implicit *vs.* explicit redefinition, would be to treat such cases as we would explicit redefinitions: to specify that the result of a once function will be shared only within each generic derivation of a generic class, and, if the result is anchored, only within the direct instances of the class. The disadvantage of this solution, however, is that it goes against the expected semantics of once functions, which from a client's viewpoint should be the conceptual equivalent of a shared attribute. To avoid confusion and possible errors it seems preferable to take a more draconian attitude by banning such cases altogether:

Once Function rule

The result type of a once function may not be anchored, and may not involve any formal generic parameter.

18.5 CONSTANTS OF STRING TYPE

The beginning of this chapter introduced character constants, whose value is a single character. The example was

Backslash: *CHARACTER* **is** '\'

Often, classes will also need symbolic constants representing multi-character strings.The notation for manifest string constants will use double quotes:

[S1]

Message: *STRING* **is** "*Syntax error*"

Recall that *STRING* is a class of the library, not a simple type. So the value associated at run time with an entity such as *Message* is an object (an instance of *STRING*). As you may have guessed, the above declaration is a shorthand for the declaration of a once function, here of the form:

```
Message: STRING is
        -- String of length 12, with successive characters
        -- S, y, n, t, a, x, , e, r, r, o, r
    once
        !! Result.make (12)
        Result.put ('S', 1)
        Result.put ('y', 2)
        …
        Result.put ('r', 12)
    end
```

The creation procedure for strings takes as argument the initial expected length of the string; *put* (*c*, *i*) replaces the *i*-th character with *c*.

Such string values are therefore not constants but references to shared objects. Any class that has access to *Message* may change the value of one or more of its characters.

You can also use string constants as expressions, for argument passing or assignment:

```
Message_window.display ("CLICK LEFT BUTTON TO CONFIRM EXIT")
greeting := "Hello!"
```

18.6 UNIQUE VALUES

It is sometimes necessary to define an entity that has several possible values denoting possible cases. For example a read operation may produce a status code whose possible values are codes meaning “successful”, “error on opening” and “error on reading”.

A simple solution is to use a variable integer attribute

```
code: INTEGER
```

with a set of associated integer constants, such as

[U1]

```
Successful: INTEGER is 1
Open_error: INTEGER is 2
Read_error: INTEGER is 3
```

so that you can write conditional instructions of the form

[U2]

```
if code = Successful then …
```

or multi-branch instructions of the form

See “Multi-branch”, page 449.

[U3]

```
inspect
        code
when Successful then
        …
when …
end
```

It is tedious, however, to have to come up with the individual constant values. The following notation has the same practical effect as [U1]:

[U4]

Successful, Open_error, Read_error: *INTEGER* **is unique**

A **unique** value specification, coming in lieu of a manifest integer value in the declaration of a constant integer attribute, indicates that the value is chosen by the compiler rather than the developer. So the conditional instruction [U2] and the multi-branch [U3] are still applicable.

All **unique** values within a class are guaranteed to be positive and different; if they are declared together, as the three in [U4], they are also guaranteed to be consecutive. So if you want to express that *code* will only receive one of their values, you can include the invariant clause

code >= Successful; *code <= Read_error*

With this invariant, a descendant — which, as we know, may change the invariant only by strengthening it — may constrain the possible values of *code* further, for example to just two possibilities; it may not extend the set of possibilities.

You should only use Unique values to represent a fixed set of possible values. As soon as this set is open to variation, or the instructions in a structure such as [U3] are non-trivial, it is preferable to devise a set of classes which variously redefine some features, and then to rely on dynamic binding, satisfying the Open-Closed principle. More generally, do not use unique values for classification since the object-oriented method has better techniques. The preceding example is typical of good uses of the mechanism; others would be traffic light states (*green, yellow, red*: *INTEGER* **is unique**) or, as seen earlier, notes on the scale (*do, re, mi, …*: *INTEGER* **is unique**). But a declaration *savings, checking, money_market*: *INTEGER* **is unique** is probably a misuse if the various kinds of account have different features or different implementations of a common feature; here inheritance and redefinition will most likely provide a better solution.

These observations can be summed up as a methodological rule:

Discrimination principle

Use unique values to describe a fixed number of possible cases. For classification of data abstractions with varying features, use inheritance.

Although similar in some respects to the “enumerated types” of Pascal and Ada, unique declarations do not introduce new types, only integer values. The discussion section will explore the difference further.

18.7 DISCUSSION

In this discussion, the term "global object" refers both to global constants of basic types and to shared complex objects; their "initialization" includes object creation in the latter case.

Initializing globals and shared objects: language approaches

The principal problem addressed by this chapter is an instance of a general software issue: how to deal with global constant and shared objects, and particularly their initialization in libraries of software components.

Since the initialization of a global object should be done just once, the more general issue is how to enable a library component to determine whether it is the first to request a certain service.

This boils down to an apparently simple question: how to share a boolean variable and initialize it consistently. We can associate with a global object *p*, or any group of global objects that need to be initialized at the same time, a boolean indicator, say *ready*, which has value true if and only if initialization has been performed. Then we may include before any access to *p* the instruction

```
if not ready then
    "Create or compute p"
    ready := True
end
```

The initialization problem still applies to *ready*, itself a global object that must somehow be initialized to false before the first attempt to access it.

This problem has not changed much since the dawn of programming languages, and the early solutions are still with us. A common technique in block-structured languages such as Algol or Pascal is to use for *ready* a global variable, declared at the highest syntactical level. The main program will do the initialization. But this does not work for a library of autonomous modules which, by definition, is not connected to any main program.

In Fortran, a language designed to allow routines to be compiled separately (and hence to enjoy a certain degree of autonomy), the solution is to include all global objects, and in particular *ready* indicators, in a shared data area called a common block, identified by its name; every subroutine accessing a common block must include a directive of the form

```
COMMON /common_block_name/ data_item_names
```

There are two problems with this approach:

- Two sets of routines may use a common block of the same name, triggering a conflict if an application needs them both. Changing one of the names to remove the conflict may cause trouble since common blocks, by nature, are shared by many routines.
- How do we initialize the entities of a common block, such as our *ready* indicators? Because there is no default initialization rule, any data in a common block must be initialized in a special module called a "block data" unit. Fortran 77 allows named block data units, so that developers can combine global data from various contexts — provided they do not forget to include all the relevant block data units. A serious risk of accidental inconsistency exists.

The C solution is conceptually the same as in Fortran 77. The *ready* indicator should be declared in C as an "external" variable, common to more than one "file" (the C compilation unit). Only one file may contain the declaration of the variable with its initial value (false in our case); others will use an **extern** declaration, corresponding to Fortran's *COMMON* directive, to state that they need the variable. The usual practice is to group such definitions in special "header" files, with names conventionally ending with *.h*; they correspond to the block data units of Fortran. The same problems arise, partially alleviated by "Make" utilities which help programmers keep track of dependencies.

A solution would appear to be at hand with modular languages such as Ada or Modula 2 where routines may be gathered in a higher-level module, a "package" in Ada terms: if all the routines using a group of related global objects are in the same package, the associated *ready* indicators may be declared as boolean variables in that package, which will also contain the initialization. But this approach (also applicable in Fortran 77 and C using techniques described in chapter 18) does not solve the problem of initialization in autonomous library components. The more delicate question discussed in this chapter is what to do for global objects that must be shared between routines in **different** and independent modules. Ada and Modula provide no simple answer in this case.

In contrast, the "once" mechanism preserves the independence of classes, but allows context-dependent initializations.

Manifest string constants

The notation allows string constants (or more properly, as we have seen, shared objects) to be declared in manifest form, using double quotes: "…". A consequence of this policy is that the language definition, and any compiler, must rely on the presence of class *STRING* in the library. This is a compromise between two extreme solutions:

- *STRING* could have been a predefined basic type, as is the case in many languages. This, however, would have meant adding all string operations (concatenation, substring extraction, comparison etc.) as language constructs, making the language considerably more complex, even though only few applications require all these operations; some do not even need strings at all. Among the advantage of using a class is the ability to equip its operations with precise specifications through assertions, and to allow other classes to inherit from it.
- Treating *STRING* as just any other class would preclude manifest constants of the "…" form [S1], requiring developers always to enter the characters individually as in form [S2]. It might also prevent the compiler from applying optimizations for time-sensitive operations such as character access.

On the ARRAY case see "Efficiency considerations", page 327.

So *STRING*, like its companion *ARRAY*, leads a double life: predefined type when you need manifest constants and optimization, class when you need flexibility and generality. All this, of course, is part of the general effort to have a single, universal, consistent type system entirely based on the notion of class.

Unique values and enumerated types

Pascal and derivatives allow declaring a variable as

code: *ERROR*

where *ERROR* is declared as an "enumerated type":

type *ERROR* = (*Normal, Open_error, Read_error*)

Being declared of type *ERROR*, variable *code* may only take the values of this type: the three symbolic codes given.

We have seen how to obtain the equivalent effect in the O-O notation: define the symbolic codes as **unique** integer constants, and *code* as an integer attribute, possibly with an invariant clause stating that its value must lie between *Normal* and *Read_error*. The result at execution time is almost identical, since Pascal compilers typically implement values of an enumerated type by integers. (A good compiler may take advantage of the small number of possible values to represent entities such as *code* by short integers.)

The **unique** technique involves no new type. It seems indeed hard to reconcile the notion of enumerated type with object technology. All our types are based on classes, that is to say abstractly characterized by the applicable operations and their properties. No such characterization exists for enumerated types, which are mere sets of values. Enumerated types actually raise problems even in non-O-O languages:

- The status of the symbolic names is not clear. Can two enumerated types share one or more symbolic names (as *Orange* both in type *FRUIT* and in type *COLOR*)? Are they exportable and subject to the same visibility rules as variables?
- It is difficult to pass values of an enumeration type to and from routines written in other languages, such as C or Fortran, which do not support this notion. Since **unique** values are plain integers they cause no such problem.
- Enumerated values may require special operators. For example you will expect a **next** operator yielding the next value, but it will not be defined for the last enumeration element. You will also need an operator to associate an integer with every enumerated value (its index in the enumeration). To go the other way around requires more operators since we must know the bounds of the enumeration to restrict applicable integer values. The resulting syntactic and semantic complexity seems out of proportion with the mechanism's contribution to the language.

Uses of enumeration types in Pascal and Ada tend to be of the form

type *FIGURE_SORT* = (*Circle*, *Rectangle*, *Square*, …)

to be used in connection with variant record types of the form

```
FIGURE =
        record
                perimeter: INTEGER;
                … Other attributes common to figures of all types …
                case fs: FIGURE_SORT of
                        Circle: (radius: REAL; center: POINT);
                        Rectangle: … Attributes specific to rectangles …;
                        …
                end
        end
```

themselves used in **case** discrimination instructions:

```
procedure rotate (f: FIGURE)
        begin case f of
                Circle: … Appropriate actions to rotate a circle …;
                Rectangle: …;
        …
```

which we have learned to handle in a better way to preserve extendibility: by defining a different version of procedures such as *rotate* for each new variant, represented by a class.

When this most important application of enumerated types disappears, all that remains is the need, in some cases, to select integer codes having a fixed number of possible values. Defining them as integers avoids many of the semantic ambiguities associated with enumerated types; for example there is nothing mysterious about the expression *Circle + 1* if *Circle* is officially an integer. The only unpleasantness of integers would be to have to assign the values yourself; **unique** values solve that problem.

18.8 KEY CONCEPTS INTRODUCED IN THIS CHAPTER

- A challenging problem in any approach to software construction is how to allow for global data: objects that must be shared by various modular components, and initialized at run time by whatever component happens to need them first.
- A constant can be *manifest* (expressed as a self-describing representation of its value) or *symbolic* (expressed by a name).
- You can declare manifest constants of basic types as constant attributes, occupying no space in objects.
- Except for strings, developer-defined types have no manifest constants, which would damage information hiding and extendibility.
- A once routine, which differs from a normal function by one keyword, **once** instead of **do**, is evaluated only once during a system's execution: the first time any component of the system calls it. For a function, subsequent calls return the same value as the first; for a procedure, subsequent calls have no effect.
- Shared objects may be implemented as once functions. You can use the invariant to specify that they are constant.
- Use once procedures for operations to be performed only once over the execution of a system, such as initializations of global parameters.
- The type of a once function may not be anchored or generic.
- Constants of string types are treated internally as once functions, although they look like manifest constants written in double quotes.
- Enumerated types à la Pascal do not go well with the object-oriented method, but to represent codes with several possible values there is a need for "unique" attributes: symbolic constants of type *INTEGER*, whose value is chosen by the compiler rather than by the software writer.

18.9 BIBLIOGRAPHICAL NOTES

[Welsh 1977] and [Moffat 1981] study the difficulties raised by enumerated types.

Some of the techniques of this chapter were introduced in [M 1988b].

EXERCISES

E18.1 Emulating enumerated types with once functions

Show that in the absence of Unique types a Pascal enumerated type of the form

type *ERROR* = (*Normal*, *Open_error*, *Read_error*)

could be represented by a class with a once function for each value of the type.

E18.2 Emulating unique values with once functions

Show that in a language that does not support the notion of **unique** declaration it is possible to obtain the effect of

value: *INTEGER* **is unique**

by a declaration of the form

value: *INTEGER* **is once** … **end**

where you are requested to fill in the body of the once function and anything else that may be needed.

E18.3 Once functions in generic classes

Give an example of a once function whose result involves a generic parameter and, if not corrected, would yield a run-time error.

See "Once functions, anchoring and genericity", page 652.

E18.4 Once attributes?

Examine the usefulness of a notion of "once attribute", patterned after once routines. A once attribute would be common to all instances of the class. Issues to be considered include: how does a once attribute get initialized? Is the facility redundant with once functions without arguments and, if not, can you explain clearly under what conditions each facility is appropriate? Can you think of a good syntax for declaring once attributes?