7 MODULARITY AND DATA ABSTRACTION: ADA

7.1 HISTORY AND MOTIVATION

The Software Crisis and Reliable Programming

In the 1970s the recognition of a "software crisis," that is, that the costs of producing software were increasing without bound, led a number of computer scientists to search for a solution. For example, Dijkstra and others observed that the difficulty of producing a program seemed to increase with the square of the program's length, so it seemed that very large programs would be completely infeasible. It thus became necessary to find a means of writing programs that would result in their cost being a *linear* function of their length.

Parnas's Principles

One of the traditional methods used to control the complexity of a large program was *modularization*, the division of a program into a number of independent *modules*. When this is done, each module is like a small program that can be implemented independently of the other modules. Therefore, the work to implement the entire program is roughly the sum of the work required for each module, that is, linear in the program size. Similarly, each module can be debugged, understood, and maintained individually.

In 1971 and 1972, D. L. Parnas, at Carnegie-Mellon University, developed several principles to guide the decomposition of a program into modules. One of his principles is that there should be one module for each difficult design decision in the program. This means that the results of each decision can be hidden in the corresponding module; if this decision is later changed, only that module has to be modified. This is called Information Hiding and is formalized in the Principle already introduced in Chapter 2 (Section 2.5).

Note: Ada is a registered trademark of the Ada Joint Program Office, U.S. Government.

Abstract Data Types

A common design decision is the choice of data structure representation. For example, a stack can be represented as an array with a top index or as a linked list; a set can be represented as an array of values or a bit string. Since the choice of data structure representation is often a difficult design decision, in a well-modularized program there will be one module for each data structure. Any manipulation of the data structure must then be done through the procedures provided by the module because the representation of the data structure is hidden in the module. To put it another way, users of the module are required to use the *abstract* operations on the data structure (e.g., push and pop for stacks) because they are prohibited from using the *concrete* operations (e.g., subscripting or pointer operations). It is for this reason that a module that provides a set of abstract operations on a data structure (or class of similar data structures), is called an *abstract data type*. This corresponds to our earlier definition of an abstract data type: a set of data values together with a set of operations on those data values. Specific means for designing abstract data types and modules are discussed later.

Experimental Abstract Type Languages

By 1973 a number of programming language researchers had designed languages supporting abstract data types and modules. These languages, which are called *abstract type languages*, included Alphard, CLU, Mesa, Euclid, Modula, and Tartan. Many of them were based on the idea of a *class*, a construct first included in the language Simula in 1967. The experience gained from using these experimental languages was important in the later development of production abstract type languages, including Ada.

DoD Saw the Need for a New Language

In the mid-1970s the United States Department of Defense (DoD) identified the need for a state-of-the-art programming language to be used by all the military services in *embedded* (or *mission critical*) computer applications. These are applications in which a computer is embedded in and integrated with some larger system, for instance, a weapons system or a command, control, and communication system. (*Non*embedded systems include traditional offline scientific and data-processing applications.) At this time DoD was spending about $3 billion annually on software—most of it going for embedded systems. A significant factor in this high cost was limited portability and reuse of software resulting from the fact that over 400 programming languages and dialects were then in use for embedded applications. DoD recognized that its programming needs in the 1980s and beyond would not be satisfied by the programming languages then in use, and in 1975 it set up the Higher-Order Language Working Group (HOLWG) to study the development of a single language for these applications. It was estimated in 1976 that such a language would save $12–24 billion over the next 17 years.

A Series of Specifications Was Published

In the period 1975 to 1979, HOLWG published a series of specifications that the new language was required to meet. Each specification was more detailed than the previous, as suggested by their names:

- 1975 Strawman
- 1975 Woodenman
- 1976 Tinman
- 1978 Ironman
- 1979 Steelman

Each specification made more specific or froze some of the requirements of the preceding specification.

Information Hiding, Verification, and Concurrency

The specifications placed some general requirements on the design of the language, such as readability and simplicity. The specifications also defined more specific requirements, including a module facility to support information hiding, mechanisms to allow concurrent programming, and a design amenable to verification of program correctness. There were also many concrete requirements such as the character set and commenting conventions to be used.

There Were Several Competing Designs

In 1977 HOLWG studied 26 existing languages and concluded that none of these met the specifications. This led to a competitive language design effort that lasted from 1977 to 1979 and resulted in 16 proposals. Later evaluations resulted in the number being reduced to four, then two, and eventually one.

The Winner Is Named "Ada"

The winning language was designed by a CII-Honeywell-Bull team headed by Jean Ichbiah. In May 1979 HOLWG renamed this language "Ada" in honor of Augusta Ada, Countess of Lovelace, the daughter of the poet Lord Byron. Ada was a mathematician and Charles Babbage's (and, hence, the world's) first programmer. This continued a tradition of naming programming languages after mathematicians (e.g., Pascal, Euler, Euclid). In response to over 7000 comments and suggestions from language design experts in over 15 nations, Ada was revised and reached its final form in September 1980. It become a military and American National Standard in January 1983, and became mandatory for all mission critical (embedded) software in July 1984. It became an ISO standard in 1987.

Subsets and Supersets Were Not Permitted

The goal of Ada is to decrease embedded computer software costs by increasing portability and reuse of software. It was clear to the Department of Defense that such a goal could not be achieved if there were a number of mutually incompatible subsets and supersets of Ada in use. Therefore, DoD took the unprecedented action of registering the name "Ada" as a trademark. This provides the ability to control the use of this name and to guarantee that anything called "Ada" is the standard language. That is, subsets and supersets of Ada cannot legally be called "Ada." How does DoD decide whether a compiler does in fact implement the Ada language? For this DoD has set up a validation procedure, comprising over 2500 tests, which attempts to ensure that a candidate compiler implements no more and no less than the standard language. We will discuss later the controversy revolving around Ada subsets.

Ada Has Been Revised

As a result of experience with the original version of Ada, now known as Ada 83, as well as developments in programming languages and computer technology, a project began in 1988 to develop a new version, Ada 95. The early stages of the project produced a 1990 report specifying 41 requirements and 22 study topics to be addressed by the revision effort. The resulting revision, which was standardized in 1995, is a substantial one and includes ideas from fifth-generation object-oriented programming languages (Section 12.5). Ada 95 is mostly, but not entirely, upward compatible with Ada 83, so the programmer must be wary of incompatibilities in converting to the new version.

Ada 95 Comprises a Core Language and Special Needs Annexes

Because a language satisfying all the requirements would have been enormous, the prohibition against Ada versions was relaxed. Therefore the Ada 95 report describes a "core language," which must be implemented, and six "special needs annexes," which are optional extensions to the language for particular application areas (systems programming, information systems, real-time systems, numerical programming, distributed systems, and safety and security).

In the following, "Ada" will refer to both versions of the language unless otherwise specified.

7.2 DESIGN: STRUCTURAL ORGANIZATION

Figure 7.1 displays a small Ada module. We can see that its syntax is quite similar to Pascal's. The convention in Ada programs is to type keywords in lowercase and all other words in upper and lowercase. Ada's constructs can be divided into four categories:

1. Declarations
2. Expressions
3. Statements
4. Types

Expressions and statements are very similar to their counterparts in Pascal. Types are also similar except that they are more flexible and some of the problems of the Pascal type system have been corrected.

The most significant differences between Pascal and Ada appear in the declarations. The declarations in Ada can be classified:

1. Object
2. Type
3. Subprogram
4. Package
5. Task

```
package Tables is

   type Table is array (Integer range < >) of Float;

   procedure BinSearch (T: Table; Sought: Float;
         Location: out Integer; Found: out Boolean) is
      subtype Index is Integer range T'First..T'Last;
      Lower  : Index := T'First;
      Upper  : Index := T'Last;
      Middle : Index := (T'First + T'Last')/2;
   begin
      loop
         if T(Middle) = Sought then
            Location := Middle;
            Found := True;
            return;
         elsif Upper < Lower then
            Found := False;
            return;
         elsif T(Middle) > Sought then Upper := Middle - 1;
         else Lower := Middle + 1;
         end if;
         Middle := (Lower + Upper)/2;
      end loop;
   end BinSearch;
end Tables;
```

Figure 7.1 An Ada Package

The *object declarations* serve the same function as Pascal's constant and variable declarations. Subprogram declarations are similar to Pascal's function and procedure declarations although the name of the procedure is allowed to be one of Ada's build-in operators (+, −, =, >, etc.). This permits *overloading* of additional meanings onto these operators. In previous chapters we have seen that most languages overload the arithmetic operators. For example, '+' normally applies to both integers and reals. In Ada, users can extend this overloading so that '+' applies to their own data types, for example, complex numbers or matrices. We will see an example of this later.

Two of Ada's most important facilities are its *package* and its *task* declarations, both of which declare *modules*. Tasks are distinguished from packages in their ability to execute concurrently (in parallel) with other tasks. Modules (packages and tasks) are the basic components of which Ada programs are constructed.

Each module forms a disjoint environment that can communicate with other modules through well-defined *interfaces*. To accomplish this, the declaration of a module is broken down into two parts: a *specification*, which describes the interface to that module, and a *body* or *definition*, which describes how the module is implemented. Some of the other declarations can also be expressed in this two-part way. The specification of a package contains the specifications of the names (procedures, types, etc.) supplied by the package; the body of the package contains the bodies or definitions of these names. The purpose of this structure is to implement the information-hiding principles you read about earlier and that you will learn more about later in this chapter.

Ada is designed to permit a conventional compiled implementation. Typically, an Ada compiler would be divided into four subsystems:

1. Syntactic analyzer
2. Semantic analyzer
3. Optimizer
4. Code generator

The *syntactic analyzer* (parser) is conventional and only moderately more complicated than an analyzer for Pascal. Some interactive Ada systems replace the syntactic analyzer with a *syntax-directed editor*, which directly generates the parse tree to be used by the semantic analyzer. The *semantic analyzer* performs type checking, as in Pascal, and processes some of Ada's more complicated features, such as generic declarations and overloaded operators (discussed later). The complexity of these features makes Ada's semantic analyzer much larger and more complicated than Pascal's. The result of the semantic analyzer is a program tree that is passed to a conventional *optimizer* and *code generator*.

7.3 DESIGN: DATA STRUCTURES AND TYPING

The Numeric Types Are Generalized

Ada's integer types are essentially like those of Pascal, including the ability to use a *range constraint* to limit the set of permissible values; for example,

```
    type Coordinate is range -100 .. 100;
```

Arithmetic on integers is exact and essentially the same as that in FORTRAN, Algol-60, and Pascal.

*Ada goes far beyond Pascal's simple provision of a **real** data type: It provides two classes that provide approximate arithmetic on real numbers—the floating-point types and the fixed-point types.* We will investigate the floating-point types first since they are the more familiar.

The declaration

```
type Coefficient is digits 10 range -1.0e10 .. 1.0e10;
```

defines `Coefficient` to be a floating-point type with at least 10 digits of precision and able to accommodate numbers in the specified range. If the target computer provides arithmetic of several different precisions (such as single and double precision), then the compiler can select the appropriate precision on the basis of the floating-point constraint.

Ada specifies that each implementation must provide a predefined type `Float` that corresponds to the machine's usual precision. The types `Short_Float` and `Long_Float` may also be predefined if they are supported by the implementation, although the use of these types compromises program portability. This also contradicts HOLWG's goal of having no dialects of Ada.[1] Programmers are encouraged to use the floating-point constraint (i.e., the `digits` specification) rather than these predefined types so that their programs will be more machine independent. By using a `digits` specification such as the one above, programmers state the precision they *want* and leave it to the compiler to determine the machine representation that they *need.* This is impossible if programmers use machine-dependent types such as `Float` and `Long_Float`. Unfortunately, programmers frequently do not know the precision they need; further, there is an unfortunate tendency for programmers to write the precision specification that they know will get them a particular representation on a particular implementation. This was the experience in PL/I, where programmers wrote `BINARY FIXED(31)`, not because they wanted numbers of this precision, but because this will be represented as one 32-bit word on an IBM-360. This defeats the entire purpose of these machine-independent specifications.

Floating-point constraints illustrate the Preservation of Information Principle.

The Preservation of Information Principle

The language should allow the representation of information that the user might know and that the compiler might need.

In other words, if users know their requirements at the more abstract level (number of digits required), they should not be required to state them at the more concrete level (number of words required) since this puts into the program machine-dependent design decisions that are better left to the compiler.

Arithmetic on floating-point numbers is conventional: Operations are effectively performed at the maximum available precision and then rounded to the precision of the operands.

[1] The same comments apply to Ada's "optional" types `Short_Integer` and `Long_Integer`.

Whereas the floating-point types provide approximate arithmetic with a relative error bound, the fixed-point types provide it with an absolute error bound. Fixed-point arithmetic had been the rule on early computers; it fell into disuse in scientific applications after the introduction of floating-point hardware (see Chapter 1). It is still used in languages for commercial programming (e.g., COBOL), where an absolute bound on the error is required (e.g., one cent). Fixed-point numbers have been included in Ada because they are used by many of the peripheral devices incorporated in embedded computer systems (e.g., analog-to-digital converters). Fixed-point types are specified using a *fixed-point constraint,* for example,

```
type Dollars is delta 0.01 range 0.00 .. 1_000_000.00;
```

The `delta` specifies the absolute error bound; in this case, values of type `Dollars` will be exact multiples of 0.01. For instance, the number 16.75 would be stored as the binary equivalent of 1675 since $16.75 = 1675 \times 0.01$. The minimum number of bits required to store a fixed-point type is just the logarithm of the number of values to be represented, for example,

$$\log_2 [1 + (1000000 - 0)/0.01] \approx \log_2 10^8 \approx 26.6$$

Therefore, 27 bits are required. Converting an integer value to a fixed-point value requires division by the `delta` value; for example, `Dollars(2.0)` will result in the binary representation of $2/0.01 = 200$. If the `delta` is a power of 2, then it is a simple optimization to replace this operation by a left shift. For instance, if `Volt` is defined

```
type Volt is delta 0.125 range 0.0 .. 255.0;
```

then the conversion `Volt(20)` is accomplished by a left shift of three (since $0.125 = 1/8$). For this reason, the Ada definition permits the compiler to choose an actual `delta` that is less than the specified `delta` but a power of 2 to make the conversion more efficient.

The arithmetic rules for fixed-point types are more complicated than those for integers and floating-point types. This is particularly true for multiplication and division. For instance, if `VF` is a variable of fixed-point type `F` and `VI` is an `INTEGER` variable, then `VF*VI` and `VI*VF` are both type `F` and the assignment `VF := VF*VI` is permitted. However, if `VG` is a variable of any fixed type (including `F`), then `VF*VG` is considered to be of type "universal fixed," that is, a fixed-point number of maximum accuracy. It is then illegal to assign `VF := VF*VG` because the types do not match. An explicit type conversion must be used — `VF := F(VF*VG)`. Division obeys similar rules. These unintuitive rules are an almost unavoidable consequence of fixed-point arithmetic.

■ ***Exercise 7-1*:*** Either design a better system of fixed-point arithmetic or show that all of the reasonable alternates are inferior to Ada's system.

■ ***Exercise 7-2*:*** We have seen that Ada provides three basic sorts of numbers. Some other languages, such as APL, LISP, and BASIC, provide only one kind (essentially equivalent to floating point) and use it for all purposes. Discuss the relative advantages and disadvantages of these two approaches.

Constructors Are Based on Pascal's

The data structure constructors of Pascal have been carried forward into Ada. These include enumerators, arrays, records, and pointers (called *access* types in Ada). All of these have been varied from the Pascal model in an attempt to eliminate some of their problems. For example, a frequent cause of errors in Pascal programs was changing the discriminant (tag) of a variant record without initializing the corresponding fields. As we saw in Chapter 5, this left a loophole in the Pascal type system. Ada solves this problem by stating that the discriminant can be changed only by assigning a complete record value to the record, that is, by assigning to all of the fields of that variant in one operation. In all other situations, the discriminant is treated like a constant. This ensures that the fields that correspond to the discriminant's value are always initialized.

▩ ***Exercise 7-3*:*** Some languages (e.g., Algol-68) provide a *discriminated union* instead of variant records. A discriminated union type is defined by a declaration such as

```
type Person is union (Male, Female);
```

This means that objects of type `Person` may be either `Males` or `Females`, that is, that the set of data values subsumed by the type `Person` is the union of the values subsumed by the types `Male` and `Female`. It is called a *discriminated* union because the language uses a hidden *discriminant* (like Pascal's discriminant in a variant record) to tell whether a particular `Person` is a `Male` or a `Female`. This discriminant is automatically maintained by the system, therefore, it can never be wrong. Compare and contrast these two solutions to the same problem. Are there any security differences? How about readability and efficiency? Suggest how each could be improved or suggest a better alternative to both.

▩ ***Exercise 7-4*:*** Some languages (e.g., C) provide an *undiscriminated* or *free* union; that is, there is no discriminant, either hidden or visible. Discuss this feature in terms of the principles of good language design.

Name Equivalence Is Used

The Ada type system is stronger than Pascal's because of the consistent use of *name equivalence* (which was discussed in Chapter 5, Section 5.3). Some of the inconveniences and awkwardness of name equivalence have been mollified through the use of two new concepts—*subtype* and *derived type*—that are discussed later. In its simplest terms, name equivalence states that two objects are taken to have the same type only if they are associated with the same type name. For example, in

```
type Person is record ID, Age: Integer; end record;
type Auto is record ID, Age: Integer; end record;

X: Person;
Y: Auto;
```

the variables `X` and `Y` are of different types because the associated type names, `Person` and `Auto`, are different. The fact that the structural descriptions of these types are the same is irrelevant. This decision is based on the principle that if programmers restate the same type definition with a different name, it must be because they intend to use the types for different things (as is obvious in this case).

Ada extends the use of name equivalence to types that do not have a name, for instance,

```
X: record ID, Age: Integer; end record;
Y: record ID, Age: Integer; end record;
```

Again, X and Y have different types. The easiest way to understand this is to imagine that the compiler invented names for these two types (such as Person and Auto) and substituted them in the declarations of X and Y. We can see that one effect of name equivalence is to encourage programmers to name their types.

Name equivalence has been used in Ada for a number of reasons. One is the presumption mentioned above that if programmers repeat a type definition, they probably did it because the types are logically different. A second reason is that the alternative, *structural equivalence*, is not well defined. For example, in comparing two record types, is the order of the fields significant? Consider the following:

```
type R is record X: Float; N: Integer; end record;
type S is record N: Integer; X: Float; end record;
```

Can a variable of type R be assigned to a variable of type S? There are also other questions that must be answered: Are the names of the fields significant? For example, are these compatible types?

```
type R is record X: Float; N: Integer; end record;
type S is record A: Float; I: Integer; end record;
```

They are represented in the same way, but it is not obvious that it makes sense (in terms of the program's purpose) to assign one to the other.

There is no general agreement on which definition of structural equivalence is best, and almost all of the definitions are difficult for compilers to implement. Thus, name equivalence seems preferable.

■ ***Exercise 7-5*:*** Suppose we adopted the version of structural equivalence that ignores the order of fields. Describe the code a compiler would have to generate for a record assignment. Contrast this with the case in which the order of fields is significant.

■ ***Exercise 7-6*:*** Do you agree with the preferability of name equivalence? Defend either name or structural equivalence or propose and defend an alternative.

Subtypes Are Clarified

One of the reasons that name equivalence has not been adopted by previous languages is that it is often *too* restrictive. Consider the following declarations:

```
N: Integer;
type Index is range 1..100;
I: Index;
```

Since Index and Integer have different names, pure name equivalence would consider them different, unrelated types. Hence, the variables I and N have different types, and it is illegal to assign `I` to `N`, `N := I`. This is certainly unintuitive since the type `Index` is just a subset of the type `Integer`. The situation is even worse than this; it is no longer possible to write `I + 1` because the '+' operation is defined on `Integers`, not `Indexes`. This is clearly unacceptable.

Ada solves this problem by stating that a *constraint*, such as `range 1..100`, defines a *subtype* of some *base type*, `Integer` in this case. Subtypes inherit all of the operations defined on the base type, so `I + 1` is still legal. Subtypes are also *compatible* with the base type and other subtypes of the base type, so assignments like `N:=I` and even `I:=N` (with a run-time constraint check) are legal.

This implicit definition of subtypes is supplemented by an explicit mechanism for declaring subtypes. The declaration

```
subtype Index is Integer range 1..100;
```

explicitly declares `Index` to be a subtype of `Integer` that inherits all of the operations and properties of the base type. (Notice that we *must* specify `Integer` in the `subtype` declaration and that we *cannot* specify it in the `type` declaration. Syntactic irregularities like this confuse programmers.)

A subtype can be further constrained:

```
subtype Little_Index is Index range 1..10;
```

Then any object of type `Little_Index` will also be an `Integer`, an `Index`, and any other subtype of `Integer` (if it is in the appropriate range).

Why are there apparently two methods of introducing subtypes? It seems that the `subtype` declaration will handle all situations and hence that the subtype interpretation of certain `type` declarations is superfluous. The only explanation seems to be that these `type` declarations have no other useful meaning. In this situation Ada supplies a useless duplication of function.

Derived Types

Ada provides yet another facility for declaring types, *derived types*. An example of a derived type declaration is

```
type Percent is new Integer range 0..100;
```

This defines a new type, `Percent`, that is different from `Integer`, `Index`, and every other type. In particular, it is not possible to assign a `Percent` variable to an `Index` variable or vice versa. This seems useful since we would not want to use `Percents` and `Indexes` interchangeably; they mean different things. What makes a derived type different is that it inherits all of the operations, functions, and other attributes (built-in or user defined) of the type from which it is derived. It is as though every subprogram declaration containing `Integer` or `Integer range 0..100` were copied with these types replaced by `Percent`. Thus, the type `Percent` inherits an entire set of subprograms just like, but distinct from, those defined on `Integers`. This allows a user to define a new type that is *abstractly* different from the type it is derived from, yet still make use of all of the operations defined on the original type. In fact, it is possible to convert explicitly between derived types and their parent types. For instance, `Percent(N)` converts an `Integer` value `N` to a `Percent` value.

Constraints Replace Subranges

The Pascal subrange type constructor has been replaced in Ada by a more general facility—the *constraint.* A constraint is a mechanism for restricting the allowable set of data values in a type without restricting the operations applicable to those data types. Thus, a constraint defines a subtype of a given type. The simplest example of a constraint is the *range constraint* that we have already seen. For example, `Integer range 1..100` restricts the integers to numbers in the range 1–100 and `Character range 'A'..'Z'` restricts the characters to be alphabetic. Range constraints have the same implementation costs and benefits as Pascal subrange types. The Ada constructs are more general because expressions that must be evaluated at run-time are allowed in the constraint. In these cases some checking must be done at run-time that could otherwise be done at compile-time.

We have also seen *accuracy constraints* (e.g., `Float digits 10 range -1e6..1e6` and `Dollars delta 1 range 1..10`) that are applicable to the approximate numeric types. A third type of constraint is the *discriminant constraint.* Suppose `Person` is a variant record whose discriminant can take on two values—`Male` and `Female`. Then `Person(Male)` is an example of a discriminant constraint; this is the type of `Person` records in which the discriminant has the value `Male`. Again, what we have done is restrict the set of possible values; `Person(Male)` is a subtype of `Person`. If an expression of type `Person` is assigned to a variable of type `Person(Male)`, then a run-time check will be necessary to ensure that the discriminant has the value `Male`. This is exactly analogous to the check required by a range constraint.

Index Constraints Solve Pascal's Array Problem

Recall (Chapter 5, Section 5.3) that there was a serious problem resulting from the interaction of Pascal's array types and its strong typing facility. This problem is solved by the fourth type of constraint—the *index constraint.* Suppose we wish to write a general-purpose procedure to sum the elements of a real array. To do this we define a type `Vector` that is an array of `Float` numbers with the indices unconstrained:

```
type Vector is array (Integer range < >) of Float;
```

This declaration means that each `Vector` object is a `Float` array whose index type is some subrange of `Integer`. Therefore, to declare a `Vector` object, this range must be specified as in

```
Data: Vector(1..100);
Days: Vector(1..366);
```

`Data` and `Days` are `Vectors` of lengths 100 and 366, respectively. Ada avoids Pascal's problem by allowing programmers to use the unconstrained type in a formal parameter specification, for instance,

```
function Sum (V:Vector) return Float is ....
```

The Ada type system will allow both `Data` and `Days` to be passed to `Sum`; i.e., `Sum(Data)` and `Sum(Days)` are legal because they are of the same type, `Vector` (remember, a constraint defines a subtype and subtypes are compatible with their parent type). The compiler must pass the actual bounds of the `Vector` as hidden parameters to `Sum`. Within `Sum` it is possible to access these hidden parameters by `V'First` and `V'Last`. Further, `V'Range` stands for `V'First .. V'Last`, so the loop to sum the array could be written

```
for I in V'Range loop
  Total := Total + V(I);
end loop;
```

It is also possible to declare variables to be of type `Vector`, in which case the actual bounds of the `Vector` have to be stored along with its contents.

▩ ***Exercise 7-7:*** Discuss how index constraints simplify string manipulation.

Enumerations Can Be Overloaded

Recall that Pascal did not allow an overlap in the elements of enumeration types; for example,

```
type Primary is (Red, Blue, Green);
type Stop_Light is (Red, Yellow, Green);
```

would be illegal in Pascal. Ada does not have this restriction; the above type declarations are legal as they stand. This seems to introduce ambiguities into the program because `Red` can mean either the `Primary` value `Red` or the `Stop_Light` value `Red`; we say that the identifier `Red` is *overloaded* because it has two or more meanings. Ada uses context to determine which `Red` is meant. For example, if `C` has been declared to be a `Primary` variable, then `C := Red` is unambiguous; both the compiler and the human reader can see that it is the `Primary Red` that is meant. There are some circumstances (connected with overloaded procedures discussed later) in which the correct type cannot be determined from context; in these situations programmers are required to specify which they mean—`Primary(Red)` or `Stop_Light(Red)`. In some cases, even though the use of an enumeration literal is not ambiguous, it may be very difficult for both the human and the compiler to tell what is meant; these will become apparent in Section 8.1. Why do the designers of Ada allow this confusing and potentially ambiguous situation? One reason is convenience; it is normal in natural languages for one word to have several meanings, such as a primary color and the state of a stop light. Another reason results from the fact that in Ada character sets are considered enumeration types. For instance, a character set could be defined by

```
type DisCode is ('A', 'B', 'C', 'D', 'E', 'F', 'G', 'H',
  'I', 'J', 'K', 'L', 'M', 'N', 'O', 'P', 'Q', 'R', 'S',
  'T', 'U', 'V', 'W', 'X', 'Y', 'Z', '0', '1', '2', '3',
  '4', '5', '6', '7', '8', '9', '+', '-', '.');
```

Since the same character, say `'A'`, will normally appear in several different character sets, overloading of enumeration literals seems to be implied.

■ ***Exercise 7-8*:*** Do you agree with the designers of Ada on the issue of overloaded enumerations? Discuss alternates, such as (1) the Pascal solution, (2) other treatments of character sets, and (3) always requiring the type to be specified [e.g., `Primary(Red)`]. Suggest additional solutions to this problem.

7.4 DESIGN: NAME STRUCTURES

The Primitives Are Those of Pascal

The primitive name structures of Ada are based on the Pascal model; there are constant, variable, type, procedure, and function declarations, with improvements in almost all of these. There are also task and package declarations.

One of the simplest declarations is the variable declaration, which, in contrast to Pascal's, allows initialization, for example,

```
Approximation: Float := 1.0;
```

The consistent use of initialization eliminates a common error: using an uninitialized variable. It also causes a program to be more readable by making the initialization of the variable obvious. The initial value is not restricted to being a constant. It can be an expression of any complexity, which is evaluated when the block or procedure in which it occurs is entered. This is advantageous since it permits the same construct to be used for all initialization, regardless of whether or not the initial value is a constant.

The constant declaration is a modified form of a variable declaration. For instance,

```
Feet_Per_Mile: constant Integer := 5280;
PI: constant := 3.14159_26535_89793
```

A constant declaration is interpreted exactly like a variable declaration except that its value cannot be changed after its initialization at scope entry-time. It is therefore considerably more general than a Pascal constant because its value can be computed during the execution of the program and may differ in different instances of the scope. This is a useful facility and aids program maintenance. (Recall our discussion of `MaxData` in Section 5.4, p. 194.)

Notice in the example above that the type of the constant (`Pi`) is not specified in the declaration. Ada 83 allows the type to be omitted if it is a numeric type, and if the expression on the right "involves only literals, names of numeric literals, calls of the predefined function ABS, parenthesized literal expressions, and the predefined arithmetic operators."[2] This unusual feature is included to allow constants of type *universal integer* and *universal real* to be named. These types have the maximum possible precision and are not normally accessible to programmers. The benefit of this kind of declaration is to permit the programmer to name a type- and precision-independent numerical constant. The cost of this feature is the above-quoted unintuitive and difficult to remember rule. The corresponding rule (defining a static expression) in Ada 95 is even more complex, comprising several dozen rules.

[2] *Reference Manual for the Ada Programming Language*, July 1980, p. 3-3.

■ ***Exercise 7-9*:*** The Simplicity and Regularity Principles are violated by the restrictions on the definition of universal integer and universal real constants. Try to develop an alternative solution that is simpler and more regular.

Specifications and Definitions

In our discussion of the structural organization of Ada, we saw that information hiding was supported by the ability to divide declarations into two parts:

1. One that defines the interface
2. One that provides an implementation

Most of the declarations in Ada can be broken into these two parts. For instance, a constant can be *specified* by

```
Max_Size: constant Integer;
```

This specifies that Max_Size is the name of an Integer constant but does not define its value. A "deferred" constant like this would usually be used as part of the specification of a package; this way a package can provide a constant without defining its value to be part of the interface. This constant can be *implemented*, that is, given a value, by a conventional constant definition:

```
Max_Size: constant Integer := 256;
```

This definition would normally appear in the private part of the package. We will see the way in which these specifications and definitions are used in our discussion of packages below.

Subprograms Can Be Specified

Since subprograms (procedures and functions) form most of the interface to a package, subprogram specifications are very important. For instance, the interface specification

```
procedure BinSearch (T: Table; Sought: Float;
                     Location: out Integer; Found: out Boolean);
```

tells the reader and the compiler that `BinSearch` is a procedure with four parameters: The first two are input parameters of type `Table` and `Float`, respectively, and the third and fourth are output parameters of type `Integer` and `Boolean`, respectively. This is the *interface* between the procedure `BinSearch` and its callers, that is, the information that must be known to both the users and the implementer of this procedure. It is essentially a contract between the users and the implementer. A specification such as this would usually appear as part of the interface specification of a package.

A definition of `BinSearch` that meets the above specification appears in Figure 7.1. We can see that the definition repeats the specification; this provides useful redundancy and helps readability. It should be clear that this entire structure supports information hiding by embedding the important design decisions in the body of the procedure.

Global Variables Considered Harmful

In the early 1970s, many programming language researchers were beginning to question the block structure of Algol-60 and other second- and third-generation languages. While it was agreed that such languages had many desirable characteristics, they were also seen to cause problems in large programs. Some of these were described by Wulf and Shaw (at Carnegie-Mellon University) in a paper in *Sigplan Notices* (1973) called "Global Variable Considered Harmful." In this paper they identified four problems with block structure:

- Side effects
- Indiscriminate access
- Vulnerability
- No overlapping definitions

We describe each of these problems below.

Side Effects

Computer scientists had recognized for many years the danger of *side effects*, a change to a nonlocal variable by a procedure or function. For example, suppose we have the following Algol-60 procedure:

```
integer procedure Max (x, y);   integer x, y;
  begin
    count := count + 1;
    Max := if x > y then x else y;
  end;
```

This procedure computes the maximum of two integers; it also has a side effect of incrementing the variable `count` (which we assume has been defined in an outer block). We may suppose that the programmer's intention is to determine the number of times the `Max` procedure is invoked. What is the matter with such a side effect? It makes it very difficult to determine the effects of a procedure from the form of a call of the procedure. For example, if we see

```
length := Max (needed, requested);
```

it is immediately obvious that this call on `Max` involves the variables `needed`, `requested`, and `length`. These are clearly part of the *interface* to the `Max` procedure. We would never guess that this procedure modifies `count` without looking at an implementation of the procedure. To see the problems this can cause, imagine we were looking through a program to find all the places `count` was modified because it was connected with a bug. We would very likely overlook the line shown above because it modifies `count` without ever mentioning it. Furthermore, if the procedure `Max` is predefined in some library, it may not even be possible to look at its definition. This is a potential maintenance program. Furthermore, it introduces some semantic problems. Consider the invocation,

```
count := 10;
length := Max (count, 10);
```

The procedure Max actually modifies one of its actual parameters. The exact value returned depends on whether Max increments count before or after it tests its parameters. Also, length will be set to 11 if the parameters are passed by name (as in our definition) and to 10 if passed by value. All of these problems arise because the global variable count is a hidden part of the interface to Max. It is really both an input and output parameter but does not appear in the parameter list.

FORTRAN allowed side effects through use of output parameters and COMMON blocks. For the most part, however, they could be avoided by avoiding COMMON. Unfortunately, side effects are a natural consequence of block structure since being nested inside a block implies that all variables declared in that block are visible, and hence alterable. We summarize the problems of side effects as follows:

Side effects result from hidden access to a variable.

■ ***Exercise 7-10*:*** Describe some programming situations in which side effects are useful. How could the same thing be accomplished without them? Discuss the trade-offs.

Indiscriminate Access

Closely related to side effects is the problem of *indiscriminate access*, that is, that programmers cannot prevent inadvertent access to variables. We will consider an example of this. Suppose we wanted to provide a stack to be used in an Algol-60 program. We would probably structure our program like this:

```
begin
  integer array S[1:100];
  integer TOP;

  procedure Push(x); integer x;
    begin   TOP := TOP + 1;   S[TOP] := x;   end;

  procedure Pop(x); integer x;
    begin   Pop := S[TOP];    TOP := TOP-1;   end;

  TOP := 0;
  ... uses of Push and Pop ...
end
```

The variable S, which is the stack, must be declared in the same block as Push and Pop so that it is visible from the bodies of Push and Pop. For the Push and Pop procedures to be visible to their users, they must be declared in a block that contains all uses. This means that S is visible to users of Push and Pop and that these users may inadvertently (or intentionally!) use or alter the value of S without going through the Push and Pop procedures. This situation is pictured in Figure 7.2.

There is no way to arrange the declarations in a block-structured language so that indiscriminate access cannot occur. The problem with this kind of direct access is that it creates a maintenance problem. There is no guarantee that all users of the stack go through the Push and Pop procedures, and there may be uses that depend on the details of the way the

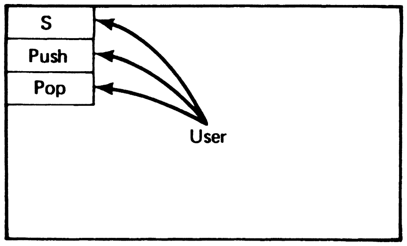

Figure 7.2 Indiscriminate Access

stack is implemented. Therefore, it will not be possible to change this implementation (e.g., to make it more efficient or correct a bug) without chasing down every reference to `S`. If we were guaranteed that all users of the stack went through `Push` and `Pop`, then we would only have to modify these to change the implementation of the stack; maintenance would be greatly simplified. Unfortunately, there is no way to accomplish this in a block-structured language. We summarize the problem of indiscriminate access:

The problem of *indiscriminate access* is the inability to prevent access to a variable.

■ ***Exercise 7-11:*** Show that indiscriminate access to `S` cannot be prevented by suitable arrangement of the block structure.

Vulnerability

We saw that the problem of indiscriminate access was that under certain circumstances it was impossible to prevent access to a variable. *Vulnerability* is the dual problem: Under certain circumstances it is impossible to *preserve* access to a variable. The basic problem of vulnerability is that new declarations can be interposed between the definition and use of a variable. Let's see what this means. Suppose we have a very large Algol program that has this structure:

```
begin
   integer x;
   . . . . . many lines of code . . . . .
   begin
      . . . . . many lines of code . . . . .
      ...x := x + 1;...
      . . . . . . . . . . . . . . . . . . . . . .
   end;
   . . . . .
end;
```

We will suppose that there are so many lines of code between the definition and use of x that they fill many pages. Let's further suppose that in the process of maintaining this program we decide that we need a new local variable in the inner block; we pick x, not realizing that it is already used in that block. This is the result of our modification:

```
begin
   integer x;
   .....many lines of code.....
   begin real x; comment NEW DECLARATION;
      .....many lines of code.....
      ...x := x + 1;...
      ......................
   end;
   .....
end;
```

We can see what has happened; access to the outer declaration of **integer** x has been blocked and the statement x := x + 1 now refers to the new **real** variable x. The new declaration of x has been interposed between the original definition of x and its use. This is illustrated in Figure 7.3. We can state the problem of vulnerability in the following way: A program segment ('x := x + 1' in this case) cannot control the assumptions under which it executes (the integer declaration of x, in this case). Summarizing,

Vulnerability means a program segment cannot preserve access to a variable.

No Overlapping Definitions

The last problem with block structure that we will discuss is that it does not permit *overlapping definitions.* The need for these arises from attempts to modularize large systems. Suppose we have a large software system composed of four modules P1, P2, P3, P4: these may be procedures or blocks. Also suppose that we want P1 and P2 to communicate through a shared data area (say, an array) DA and that we want P2, P3, and P4 to communicate through a shared data area DB. This situation is illustrated in Figure 7.4.

Since DA must be visible to both P1 and P2, it must be declared at the same or surrounding level to these modules. Similarly, DB must be declared at the same or surrounding level to P2, P3, and P4. Therefore, our program must be structured as shown in Figure 7.5. We can see that P1 has access to DB, and P3 and P4 have access to DA. This access is not needed and spreads knowledge of implementation decisions where it is not needed. This can

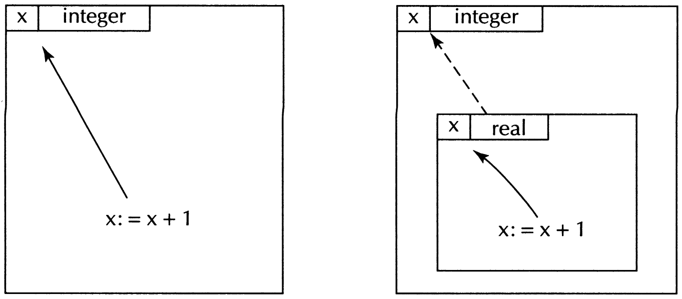

Figure 7.3 Vulnerability

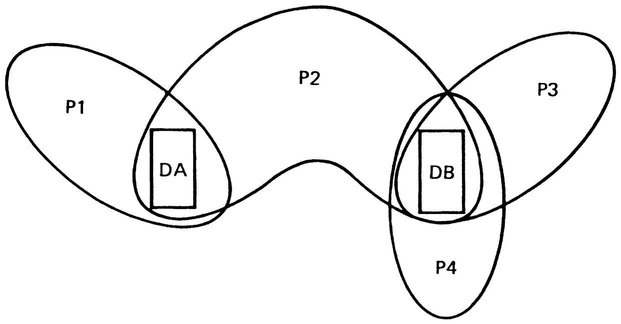

Figure 7.4 Overlapping Definitions

create both a maintenance and a security problem. The problem of no overlapping definitions is summarized as follows:

No overlapping definitions means we cannot control shared access to variables.

Attributes of an Alternative

Wulf and Shaw identified several attributes that they thought an alternative to block structure should satisfy:

1. The default should *not* be to extend the scope of a variable to inner blocks. That is, there should be no *implicit inheritance* of access to variables from enclosing blocks. Side effects, indiscriminate access, and vulnerability can all be seen to be results of this implicit inheritance, although these problems are not solved by just eliminating this feature.

2. The right to access a name should be by the mutual consent of the creator and accessor of the name. That is, the creator (definer) of a name should be able to determine who can access the name and who cannot, and potential users of the name should never have the name imposed on them if they do not want it. This would solve the problems of indiscriminate access, vulnerability, and no overlapping definitions.

3. Access rights to a structure and its substructures should be decoupled. This means that the ability to access some structure (e.g., a stack) should not imply the ability to access

```
begin
   array DA[...];
   array DB[...];
   procedure P1;    ...;
   proceudre P2;    ...;
   procedure P3;    ...;
   procedure P4;    ...;
   ...
end
```

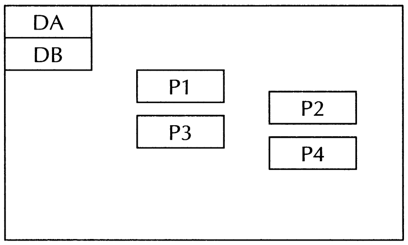

Figure 7.5 No Overlapping Definitions

the mechanism that implements it (e.g., the top pointer of the stack or the array containing its elements). This problem, which is related to indiscriminate access, can severely complicate maintenance.

4. It should be possible to distinguish different types of access. For example, it should be possible to give some users read-only access to a data structure and others read-write access. This helps to solve the side effect and vulnerability problems.

5. Declaration of definition, name access, and allocation should be decoupled. In block-structured languages these functions are usually closely connected. For instance, by declaring a variable in an Algol block (1) the name is defined by its appearance in the declaration, (2) name access is determined by its occurrence in the block since that block and all inner blocks implicitly inherit access to the variable, and (3) storage allocation and deallocation are determined since they will occur simultaneously with entry to and exit from the block. These are really three orthogonal (i.e., independent) functions. In a few cases, languages attempted to decouple these functions. Algol decouples name definition and access from allocation with its **own** variables; Pascal accomplishes the same with its dynamically allocated storage (`new` and `dispose`). Proper separation of these functions would help to solve most of the problems of block structure. We will see later that although Ada has not abandoned block structure, its packages and other related mechanisms eliminate many of the block's shortcomings.

Parnas's Principles

At about the same time that Wulf and Shaw were doing their work, Parnas enunciated two important principles of information hiding. In the introduction to this chapter, we discussed the general idea of information hiding: Each difficult design decision should be hidden inside a module. This rule determines what should be in each module. The two principles we will see next guide us in designing the interfaces between modules.

Parnas's Principles

1. One must provide the intended user with *all* the information needed to use the module correctly *and nothing more.*
2. One must provide the implementor with *all* the information needed to complete the module *and nothing more.*

Thus, the user of a module does not know how it is implemented and cannot write programs that depend on the implementation. This makes the module more maintainable since implementors know exactly what they can and cannot change without impacting the users. Similarly, implementors have no knowledge of the context of use of their module, except that provided in the interface. This simplifies maintenance of programs that use the module because programmers know what they can safely change and what they cannot. We will see that the Ada package construct directly supports Parnas's principles.

Packages Support Information Hiding

The Ada construct that supports the information-hiding principles and controls access to declarations is the *package*. The declaration of a package is broken down into two parts—an interface specification and a body. The interface specification defines the interface between the inside and the outside of the package; hence, it is that information about the package that must be known to the user; and that information about the way it will be used that must be known to the implementor. The package specification is effectively a contract between the user and the implementor of the package. A package specification has the following form:

```
package Complex_Type is
        . . . specification of public names . . .
end Complex_Type;
```

Between the brackets of the package specification (`package-end`), all the specifications of the public names (i.e., the names in the interface) are written. A partial specification of a package that provides complex arithmetic is shown in Figure 7.6.

Figure 7.6 shows that the package `Complex_Type` provides a type (`Complex`), a constant (`I`), and several functions (`Re`, `Im`), and that it overloads the arithmetic operators. The function definitions are specified in the usual way: The types of the parameters and the returned value are specified.

The type `Complex` is also listed in the interface, but it is defined to be a *private* type. This means that although the name `Complex` is visible (and hence may be used in object declarations, parameter specifications, etc.), the internal structure of `Complex` numbers is hidden from users of the package. If this were not done, it would be possible for users to access directly the components of `Complex` numbers without going through the `Re` and `Im` functions. This would interfere with later maintenance if the implementor decided to use a

```
package Complex_Type is
   type Complex is private;
   I: constant Complex;
   function "+" (X,Y : Complex) return Complex;
   function "-" (X,Y : Complex) return Complex;
   function "*" (X,Y : Complex) return Complex;
   function "/" (X,Y : Complex) return Complex;
   function Re  (X : Complex) return Float;
   function Im  (X : Complex) return Float;
   function "+" (X : Float; Y : Complex) return Complex;
   function "*" (X : Float; Y : Complex) return Complex;
private
   type Complex is
      record Re, Im : Float := 0.0; end record;
   I : constant Complex := (0.0, 1.0);
end Complex_Type;
```

Figure 7.6 Specification of Complex Arithmetic Package

different representation for `Complex` *numbers. Notice that there is an appendage to the package specification introduced by the word* `private`*. This private part of the package in-*cludes a definition of the `Complex` type that specifies its representation. What is this information doing in the specification? This is a concession that the Ada designers have been forced to make so that packages will not be too difficult to compile. Users of the `Complex_Type` package will want to declare objects of type `Complex`, which will require the compiler to allocate storage for these records; thus, the compiler must know the representation of `Complex` numbers. This is especially necessary if the program using `Complex_Type` and the package defining it are compiled separately; under these circumstances only the specification of `Complex_Type` is available to the compiler when it is compiling the program using the package.

We can see that this package also defines a public constant, `I`, defined as a *deferred* constant. Its value must be deferred because it depends on the representation of `Complex` numbers, which is private. The actual definition of the constant is given in the private part of the specification along with the type definition.

The package body, which is known only to the implementor, gives the definition of each name mentioned in the specification. It may also declare any local procedures, functions, types, and so on, needed by this implementation; all of these are private. Part of the implementation of `Complex_Type` is shown in Figure 7.7.

```
package body Complex_Type is

   function "+" (X,Y : Complex) return Complex is
   begin
      return (X.Re + Y.Re, X.Im + Y.Im);
   end;

   function "*" (X,Y : Complex) return Complex is
      RP: constant Float := X.Re*Y.Re - X.Im*Y.Im;
      IP: constant Float := X.Re*Y.Im + X.Im*Y.Re;
   begin
      return (RP,IP);
   end;

   function Re (X : Complex) return Float is
   begin return X.Re; end;

   function Im (X : Complex) return Float is
   begin return X.Im; end;

   function "+" (X : Float; Y : Complex) return Complex is
   begin
      return (X + Y.Re, Y.Im);
   end;
   -- other definitions
end Complex_Type;
```

Figure 7.7 Partial Implementation of a Complex Arithmetic Package

■ ***Exercise 7-12:*** Complete the definition of the package `Complex_Type`.

■ ***Exercise 7-13*:*** We have seen that the private part of a package specification mixes representation information important only to the implementor with the interface information needed by the user. Describe an alternative that does not mix things up this way but still allows a compiler to allocate storage for objects of private types. You may alter Ada's package declarations or describe an alternative method for the compiler to get the needed information.

Name Access Is by Mutual Consent

We have seen that the implementor of a package can control, by the placement of the declarations in the public part or the private part of the package, which names can be accessed by a user of the package. Anything placed in the specification is public and potentially accessible. A user gains access to the publics of a package with a `use` declaration, as shown:

```
declare
   use Complex_Type;
   X,Y : Complex;
   Z : Complex := 1.5 + 2.5*I;
begin
   X := 2.5 + 3.5*I;
   Y := X + Z;
   Z := Re(Z) + Im(X)*I;
   if X = Y then X := Y + Z;
   else X := Y*Z; end if;
end;
```

The `use` declaration makes all of the public names of the package visible throughout the block in which it appears. We can see that this permits using all of the types (`Complex`), functions (`Re`, `+`), constants (`I`), and so forth, as though they were built in. (In fact, the Ada language is defined as though the "built-in" types are defined in packages that are automatically `use`d for all programmers.) Thus, name access is by mutual consent: The package implementor determines which attributes are to be public and the package user decides whether to *import* the attributes of a particular package. In fact, Ada provides even more control to package users since, if they do not need all of the names defined by a package, they can select just the ones they want. This is done with a dot notation similar to Pascal's (e.g., `Complex_Type.I`) or by a variant of `use` that we will not discuss here.

A compilation comprises one or more "library items," which are often packages. Whether they are part of the same compilation or not, they are not mutually visible without explicit declaration. For one item to be visible to another, the latter must have a *context clause* mentioning the former. For example, a module needing to use complex numbers should be preceded by `with Complex_Type`. (Then the names can be accessed by the dot notation, e.g., `Complex_Type.Re`; they will be directly visible, without using the dot, if the `with` is followed by `use Complex_Type`.)

Packages as Libraries

We have just seen how to use a package to define an *abstract data type.* There are also many other ways that packages can be used to modularize programs; some of these are discussed in the following sections. One of the simplest, which is really a degenerate form of an abstract data type, is a *library*. Suppose we wished to define a `Plot` library that provided subprograms for plotting. This can easily be specified as a package:

```
package Plot is
   type Point is record X,Y : Float; end record;
   procedure Move_To (Location : Point);
   procedure Line (From, To : Point);
   procedure Circle (Center : Point; Radius : Float);
   procedure Fit (Data : array (Integer range < >) of Point);
   function Where return Point;
end Plot;
```

Then, if users wish to do some plotting in a module, they only have to include a `with Plot` request in the context clause preceding that module. Of course, the private part of the package may include the definitions of constants and subprograms that are needed by the implementation but are hidden from users. You can see that a library is just a package that does not contain any data structures.

Packages Permit Shared Data Areas

We have just looked at packages that contain procedures but no data structures; we will now look at the opposite—packages that contain data structures but no procedures. For example, suppose we wanted a buffer to be used for communicating characters between two subprograms. This could be done by the declaration

```
package Communication is
   In_Ptr, Out_Ptr : Integer range 0..99 := 0;
   Buffer : array (0..99) of Character := (0..99 => ' ');
end Communication;
```

(The declaration of `Buffer` makes it an array initialized to all blanks.) Given this definition of `Communication`, two procedures `P` and `Q` can use it for communication by including a `use` for the package:

```
with Communication; use Communication;
procedure P is
   use Communication;
begin
       .
       .
       .
   Buffer(In_Ptr) := Next;
```

```
    In_Ptr := (In_Ptr + 1) mod 100;
        .
        .
        .
end P;

with Communication; use Communication;
procedure Q is
    use Communication;
begin
        .
        .
        .
    C := Buffer(Out_Ptr);
        .
        .
        .
end Q;
```

This way of using packages is similar to the way labeled `COMMON` is used in FORTRAN. It also solves the problem of overlapping definitions discussed in the section on encapsulation; only those subprograms that need access to `Buffer` will have `with Communication; use Communication`.

Packages Can Be Data Structure Managers

When we first saw the idea of information hiding, we said that each module should encapsulate one difficult design decision; we also said that the choice of the representation for a data structure was often such a difficult decision. Therefore, a common use of Ada packages is to encapsulate a data structure and provide a representation independent interface for accessing it.

We can take a stack data structure as a common case. When we design a data structure, one of the first questions we must ask is: "What operations are to be available on this data structure?" We will immediately come up with `Push` and `Pop`; there are others, however. What will happen if we try to `Pop` an element from an empty stack? Surely we will generate an error, but it is preferable for the users to have an `Empty` test so that, if they are unsure about whether the stack is empty, they can test it before they do a `Pop`. We may also want a `Full` test to determine if there is any room in the stack before we do a `Push`. Finally, we will need an error signal or *exception*, `Stack_Error`, which is raised if someone does a `Pop` from an empty stack, and so forth. We also need to know the sort of things that the stack can hold; we will assume they are integers. We now have the information necessary to specify the `Stack1` package:

```
package Stack1 is
    procedure (Push (X : in Integer);
    procedure Pop (X : out Integer);
    function Empty return Boolean;
    function Full return Boolean;
    Stack_Error : exception;
end Stack1;
```

The difficult design decision—whether to represent the stack as an array, linked list, or something else—is hidden in the package.

Once the package has been implemented, and made accessible if necessary by `with`, it can be used as before. For example, if we intend to use the stack over a large part of the program, we can make its names available with a `use`:

```
declare
   use Stack1;
   I, N : Integer;
begin
       :
   Push(I);
   Pop(N);
       :
   if Empty then Push(N); end if;
       :
end;
```

We can also use the "dot" notation to select a public attribute from `Stack1` without using the `use`, for example,

```
Stack1.Push(I);
Stack1.Pop(N);
if Stack1.Empty then Stack1.Push(N); end if;
```

This allows users of the package to be as selective as necessary about the names imported from `Stack1`; again, name access is by mutual consent.

How would we go about implementing the stack package? For the sake of this example, we will assume that we have decided on an array representation for the stack. The implementation of the stack package is shown in Figure 7.8. We can see that this implementation of `Stack1` has two private names: `ST`, the array that holds the stack elements, and `Top`, the pointer to the top of the stack. These are completely invisible and inaccessible to users of the stack; any attempt to access them, for example, by `Stack.ST` or `Stack.Top`, will be diagnosed by the compiler as a program error. The `raise Stack_Error` statement is an example of an *exception*. Exceptions are discussed later (Chapter 8).

▩ ***Exercise 7-14*:*** Write a package body that implements `Stack1` using linked lists. To do this you will need to know a few details about Ada: (1) If `T` is a type, then `access T` is the type of pointers to things of type `T`. (2) If `T` is a record type, then `new T(X1, ..., Xn)` allocates an instance of that record type and returns a pointer to that record. What is the meaning of the `Full` function in a linked implementation of *stacks*?

```
package body Stack1 is
  ST : array (1..100) of Integer;
  Top : Integer range 0..100 := 0;

  procedure Push (X : in Integer) is
  begin
    if Full then raise Stack_Error;
    else
      Top := Top + 1; ST(Top) := X;
    end if;
  end Push;

  procedure Pop (X : out Integer) is
  begin
    if Empty then raise Stack_Error;
    else
      X := ST(Top);
      Top := Top - 1;
    end if;
  end Pop;

  function Empty return Boolean is
  begin return Top = 0; end;

  function Full return Boolean is
  begin return Top = 100; end;

end Stack1;
```

Figure 7.8 Body of Simple Stack Package

Generic Packages Allow Multiple Instantiation

Suppose that the program we are writing requires two stacks. To get a second stack, we will have to repeat the entire definition of `Stack1` with only its name changed:

```
package Stack2 is
  procedure Push (X : in Integer);
  procedure Pop (X : out Integer);
  function Full return Boolean;
  function Empty return Boolean;
  Stack_Error : exception;
end Stack2;

package body Stack2 is
  ... all of the definitions exactly as they
  ... appeared in Stack1.
end Stack2;
```

It is clearly a waste of time to have to copy the entire definition of Stack1 verbatim. Even if this copying is done automatically, say with an editor, it will still create a maintenance problem. Whenever a bug is corrected or the implementation of the package is changed, the modification will have to be repeated for each copy; there is a much greater chance of error. This approach is also inferior from the standpoint of readability since it is not obvious to someone trying to understand the program that the two stacks are really the same; they will have to compare the definitions line by line to determine this. Clearly, what we have here is a failure to modularize; the separate copies of the package should be abstracted out so that they have to be written and maintained only once, which is an example of the Abstraction Principle. This ability is provided by Ada's *generic* facility.

We can see the motivation for this facility by looking at the way that programming languages have solved similar abstraction problems. When we need to repeat the same control sequence several times with different data, we define a procedure that implements the control sequence and then call it with the required data. When we need to repeat the same data structure several times in different storage areas, we define a data type that specifies a *template*, or pattern, for the data structure, and then we use that type in variable declarations so as to create multiple *instances* of the data structure. This is exactly the approach taken with packages. A template for packages, called a *generic package*, is defined. The template can be used to repeat the package by *generic instantiations*. Let's see how this works. A template for a generic stack package would be written

```
generic
package Stack is
  procedure Push (X : in Integer);
  procedure Pop (X : out Integer);
  function Empty return Boolean;
  function Full return Boolean;
  Stack_Error : exception;
end Stack;
```

We can see that this looks exactly like our previous specification of the stack package except that the word generic has been appended to its front. This is what informs us that we are defining a template for stacks and not a particular stack. The body for the generic stack package is exactly like that in Figure 7.8 so we will not repeat it.

We have seen how to write a template for a generic package. Next we must investigate the instantiation of these templates. Suppose we want two stacks called Stack1 and Stack2. We can request the creation of two instances, or copies, of the template Stack with the generic instantiations:

```
package Stack1 is new Stack;
package Stack2 is new Stack;
```

These create two copies of the data areas defined by Stack, which are associated with the names Stack1 and Stack2; the procedural code (for Push, Pop, etc.) can be shared by the instances. The two instances of `Stack` can be used with the dot notation, for example,

```
Stack1.Push(I);
Stack2.Pop(N);
if Stack1.Empty then Stack1.Push(N); end if;
```

Notice that it is not possible to use the `use` construct to enter both stacks into the same scope since procedure calls like `Push(I)` would be ambiguous; there would be no way to tell to which stack the `Push` referred.

You have probably already noticed that package instantiation is analogous to the instantiation of procedures, which we discussed in Chapter 3 on Algol-60. In that case, we create a new activation record, or instance, for a procedure, which contains all of its local variable storage but shares the executable code with other instances. When we study object-oriented languages in a later chapter, we will see that these ideas are very closely related. One difference that must be pointed out here is that while procedures may be *dynamically instantiated*, Ada allows only the *static instantiation* of packages; that is, each instance is associated with a declaration and the number of declarations is determined by the structure of the program. (Note, however, that a package declaration may be a local to a procedure that is dynamically instantiated; thus, there can be one instance of the package for each instance of the procedure. This is the only sense in which Ada packages can be dynamically instantiated.) Some other languages, for example, Simula-67 and Smalltalk (Chapter 12), allow the dynamic instantiation of packages in much the same way that records can be dynamically allocated.

■ ***Exercise 7-15*:*** Discuss the dynamic instantiation of packages. Is there any need for this facility? Why do you suppose it was left out of Ada? Discuss how such a facility might be included in Ada, and any other mechanisms that would have to be included to support it. Are there any efficiency consequences to dynamic instantiation? How about simplicity or readability consequences?

Instances May Be Parametrically Related

The generic facility has more capabilities than the simple copying of templates. In the generic stack package we have seen defined, the stack was limited to a size of 100. Now suppose that instead of two equal-size stacks we needed `Stack1` to be of size 100 and `Stack2` to be of size 64. How would we accomplish this? It may seem that we would have to recopy the definition of `Stack` with all of the occurrences of `100` replaced by `64`. Again, this would be a very inefficient thing to do; it would hurt writability, readability, and maintainability. The analogous problem with procedures is solved by parameters; parameters allow the data to vary from one procedure call to another. Ada adapts this approach to packages by allowing parameters on a generic specification. For example, to allow the length of stack to vary from instance to instance, the package is specified.

```
generic
  Length : Natural := 100;
```

```
package Stack is
   procedure Push (X : in Integer);
   procedure Pop (X : out Integer);
   function Empty return Boolean;
   function Full return Boolean;
   Stack_Error : exception;
end Stack;
```

Notice that the `Length` parameter has been given a default value of 100; this is the value it will have if it is not specified. The body of the package is altered by replacing each occurrence of 100 by `Length` as shown in part here:

```
package body Stack is
   ST : array (1..Length) of Integer;
   Top : Integer range 0..Length := 0;
   . . . the rest of the definitions,
   . . . but with 100 replaced by Length
end Stack;
```

When a stack is instantiated, this parameter must (if not omitted) be bound to a natural number. We can get the 100- and 64-element stacks by

```
package Stack1 is new Stack(100);
package Stack2 is new Stack(64);
```

Since the default stack length is 100, the first instantiation could have been written

```
package Stack1 is new Stack;
```

Generic packages can have any number of parameters of any types, just as procedures do. They can also have several types of parameters that procedures cannot have. Suppose that we needed to use a stack, `Stack3`, that contains only characters. Again it would seem that we must copy the entire definition of `Stack` with every occurrence of `Integer` replaced by `Character`. This is undesirable for all of the reasons we have already discussed; instead it is handled by generics. A specification of type-independent stacks is

```
generic
   Length : Natural := 100;
   type Element is private;
package Stack is
   procedure Push (X : in Element);
   procedure Pop (X : out Element);
   function Empty return Boolean;
   function Full return Boolean;
   Stack_Error : exception;
end Stack;
```

The type parameter is defined to be *private* because it acts like a private type: The only operations available on it (within the package) are assignment and equality comparisons. There are other forms of type parameters in generic packages that allow more operations but on a restricted class of objects; this is too detailed to warrant our attention here. The implementation of `Stack` is shown in Figure 7.9. Given these definitions, the instantiation of general stacks is accomplished as before, for example,

```
package Stack1 is new Stack (100, Integer);
package Stack3 is new Stack (256, Character);
```

These stacks can be used with the dot notation as before, for example, `Stack1.Pop(N)` or `Stack3.Push('A')`. They can also be referred to without the dot notation through the `use` construct:

```
declare
   use Stack1;
   use Stack3;
   I, N : Integer;
   C, D : Character;
begin
   Push(I);
   Push(C);
   Pop(N);
   Pop(D);
   if Stack1.Empty then ...
   if Stack3.Full then ...
end;
```

"Using" both stacks is permitted because context determines which procedure is intended. For example, `Push(I)` is unambiguous because `I` is an `Integer` and there is only one `Integer Push` procedure visible (the other `Push` procedure works on `Characters`). In such a situation, the `Push` procedure is said to be *overloaded* since it bears several meanings at once. This is analogous to the overloaded enumeration-type elements previously discussed and to the built-in overloaded operators (e.g., '+' works on `Integers`, `Floats`, and `Complexes`). Notice that context cannot be used for the `Empty` and `Full` functions because they do not have any arguments (or return values) that depend on the element type; these functions must still be accessed with the dot notation.

Generic Packages Are Difficult to Compile

All of these convenient facilities are not without cost; efficient generation of code for generic packages can be very complicated. We consider a few of the issues in this section. We saw earlier that generic packages without parameters could be instantiated much as procedures are instantiated: A new data area is created for each instance and the executable code is shared by all of the instances. This case is illustrated by Figure 7.10. This same kind of sharing is possible with most simple kinds of parameterization; for example, the generic `Stack` package parameterized just by `Length` can make use of shared code if the differing information, the array length, is stored with the instance. It is also necessary to use the most gen-

```
package body Stack is
   ST : array (1..Length) of Element;
   Top : Integer range 0..Length := 0;

   procedure Push (X : in Element) is
   begin
      .... as before ....
   end Push;

   procedure Pop (X : out Element) is
   begin
      .... as before ....
   end Pop;

   function Empty return Boolean is
   begin return N = 0; end;

   function Full return Boolean is
   begin return N = Length; end;

end Stack;
```

Figure 7.9 Implementation of a General Stack Package

eral representation for Top (i.e., the largest Natural range) since the actual subrange varies from instance to instance. The layout is shown in Figure 7.11. Notice that the variable-length items, ST in this example, have been moved to the end of the package; otherwise it would be necessary to use variable offsets to get to Top. We can see that it is still possible to share code among instances, although there may be more run-time range checking (e.g., to ensure that assignments to Top are legal).

Let's consider a case of more general parameterization: type parameters such as we saw in the general stack package. It is possible to instantiate this template with any type for which assignment and equality are defined, which is almost all types. In various instances Element could be Character, or Integer, or Complex, or personnel records, and so on. We know that it is necessary to know the amount of storage occupied by an array element in order to find the location of any particular element. Therefore, the code to access an element of an array of Characters differs from the code to access an element of an array of Integers. The apparent consequence of this is that each instance of a type parameterized package must have its own procedures, compiled for the types that appear in that generic instantiation. This is very inefficient since there is no sharing at all, even if the code bodies turn out to be the same. There are several ways to improve on this. For example, the com-

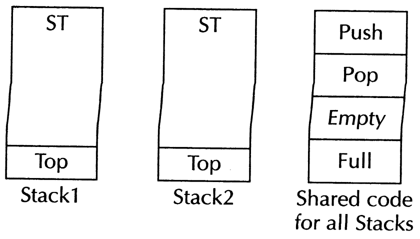

Figure 7.10 Layout of Unparameterized Package Instances

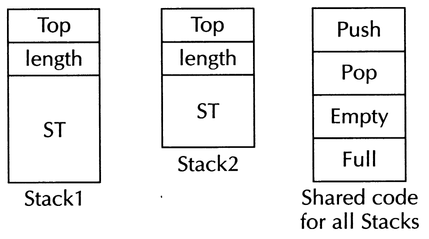

Figure 7.11 Simple Parameterization

piler can keep a record of every set of generic parameters for which it has generated code, that is, it can remember that it has already encountered a `Stack(..., Integer)`, and therefore that it has already generated code for the case `Element = Integer`. Later, if it encounters another `Integer` stack, it can share this code. This approach can be improved further by sharing code whenever the types are represented in the same amount of storage. For example, if in a particular implementation `Integers` and `Floats` both occupy one word, then the code bodies for `Stack(..., Integer)` and `Stack(..., Float)` can be shared. This is possible because the only operations allowed on `Elements` are assignment and equality comparison, both of which are usually independent of everything except the size of the value. Since there may be several type parameters to a generic package and there are other more complicated types of parameters, this attempt to find sharable code bodies can be fairly expensive. There is another approach to generic package implementation that simplifies some of the checking at the cost of decreased execution efficiency. This is to record in the instance the length of all parametrically typed objects in the package, much as was done for the length of the array `ST`. Then this length can be used for computing the position of array elements and similar purposes.

■ ***Exercise 7-16**:*** Read the discussion of generic packages in the Ada 95 Reference Manual. Criticize this mechanism with regard to its complexity, efficiency, and generality. Suggest improvements to this mechanism or show that most of the apparent improvements are less desirable.

Internal and External Representations

We have handled the representation of data structures in two distinct ways. For example, in our `Stack` example the procedures for manipulating stacks were "part" of each stack, so we wrote `Stack1.Pop(N)`, and so on. This is sometimes called an *internal* representation because the operators on a data structure are conceptually inside each instance of that data structure. The other approach is the one we used with the `Complex` package: The operators were in one package that managed all complex objects, so we wrote `Re(Z)`, and so on. For this reason, this arrangement is sometimes called an *external* representation. Frequently, a particular abstract type can be represented either externally or internally. For example, we can use an external representation for stacks by

```
package Stack_Type is
   type Stack is private;
   procedure New_Stack (S: out Stack);
   procedure Push (S: in out Stack; X: in Integer);
   procedure Pop (S: in out Stack; X: out Integer);
   function Empty (S: Stack) return Boolean;
   function Full (S: Stack) return Boolean;
private
   type Stack is record
      ST : array (1..100) of Integer;
      Top : Integer range 0..100 := 0;
      end record;
end Stack_Type;
```

Then stack objects can be declared and initialized:

```
declare
   use Stack_Type;
   Stack1 : Stack := new Stack;
   Stack2 : Stack := new Stack;
begin
    .
    .
    .
   Push (Stack1, I);
   Pop (Stack2, N);
   if Empty (Stack1) then Push (Stack1, N); end if;
end;
```

■ ***Exercise 7-17:*** The `Stack_Type` package implements only integer stacks of length 100. Show how generic packages can be used with an external representation of stacks to provide manipulation of general stacks.

There are several differences between internal and external representations. In Ada, external representations are more general. For example, by using an external representation, it is possible to treat `Stacks` as bona fide data values; they can be assigned, passed as parameters, and made elements of other data structures. For example, if we do not know exactly how many stacks we will need, we can declare an array or linked list of `Stacks` and initialize just as many elements as are required. Thus, for an external representation, the number of instances can be determined dynamically; whereas for an internal representation, the number of instances is limited by the number of generic instantiations the programmer writes. This is not an inherent characteristic of internal representations; for example, Simula and Smalltalk (Chapter 12) use an internal representation for all data abstractions (called *classes*), but allow them to be dynamically instantiated. We will see in Chapter 12 that this is a more *object-oriented* approach to data abstraction.

Ada 95 has been extended to include features for object-oriented programming, but they will be discussed in Chapter 12, since they are fifth-generation features, and our topic now is the fourth generation. Suffice it to say, they add to the overall complexity of language.

▨ ***Exercise 7-18*:*** Compare and contrast Simula's or Smalltalk's *class* facility and Ada 95's *package* facility. Discuss ease of use, efficiency, security, power, and so forth.

Overloaded Procedures Complicate Operator Identification

We have seen that the members of enumeration types can be overloaded. We have also seen (in `Complex_Type`) that the built-in operators can be overloaded; that is, a particular operator symbol (e.g., '+') can stand for several different procedures, which in turn are selected by the context of the symbol's use. For example, `Z := X + Y` may invoke the built-in '+' procedures (for `Integers` and floating-point types) or any user-defined '+' procedure (e.g., one for adding a real number to a complex number), depending on the types of `Z`, `X`, and `Y`. This process is called *operator identification.* In Ada procedures can also be overloaded; this most commonly occurs with generic packages. Suppose `Int_Stack_Type` and `Char_Stack_Type` are packages that implement externally represented stacks of integers and characters, respectively. An example of their use is

```
declare
   S1 : Int_Stack_Type.Stack;
   S2 : Char_Stack_Type.Stack;
begin
      .
      .
      .
   Int_Stack_Type.Push (S1,5);
   Char_Stack_Type.Push (S2,'A');
end;
```

It is very inconvenient to have to prefix every call of `Push` or `Pop` with `Int_Stack_Type` or `Char_Stack_Type`, so we can employ `use` to avoid this:

```
declare
   S1 : Int_Stack_Type.Stack;
   S2 : Char_Stack_Type.Stack;
   use Int_Stack_Type;
   use Char_Stack_Type;
begin
   Push (S1,5);
   Push (S2,'A');
end
```

After the two `use` declarations have been elaborated, there are two definitions of each of the stack procedures (`Push`, `Pop`, `Empty`, `Full`) available, one for integer stacks and one for character stacks. These are distinguished by context, just as for overloaded operators. `Push (S1,5)` must be the `Push` from `Int_Stack_Type` because `S1` is an integer stack and `5` is an integer. Because procedures can be overloaded, we can see that an Ada compiler must go through a process of operator identification for procedure names. This process depends on both the arguments of the subprogram and, if it is a function, its context of use. For example, in the expression

```
Z := F (G (X, Y));
```

the `F` procedure to be used depends both on the type of `Z` (i.e., `F`'s context) and on the type returned by `G` (i.e., `F`'s argument). On the other hand, `G` itself may be an overloaded procedure so that its meaning depends on its arguments (`X` and `Y`) and its context of use (the argument required by `F`). For the Ada program to be correct, there must be a unique selection for each of `F` and `G` that satisfies the above constraints. If there is none, the program is meaningless; if there is more than one, it is ambiguous. We can see that if overloading is used extensively, it may become very difficult for both the human reader and the compiler to determine what an Ada expression means. The operator identification process is further complicated by optional and position-independent parameters, which are discussed in Chapter 8, Section 8.1. Operator identification is usually accomplished by propagating type information up and down an expression tree in several passes, the exact number of passes required being a subject of ongoing research. You may think that overloaded procedures are uncommon, but this is not the case. For example, the Ada 83 input-output package defines over 14 meanings for `Get`, one for each built-in type (in fact, there are more, since there is one for each enumeration type). Further, programmers are encouraged to overload procedure names by allowing them to declare new meanings for procedures directly.

■ ***Exercise 7-19**:*** Here are two conflicting goals: (1) Overloaded operators are very convenient for groups of related operations, for example, addition on various kinds of numbers and matrices and `Push` on various kinds of stacks. (2) This extensible overloading introduces complexity into the language for both the reader and the compiler. Discuss various ways of resolving these conflicting goals, and propose and defend a good solution.

EXERCISES*

1. Defend or attack this statement: Strong typing has gotten out of hand; it now gets in the way of programming rather than simplifying it.

2. An Ada constant declaration can bind names to the values of expressions. These expressions are evaluated when the scope of the declaration is entered. Describe in detail the implementation of Ada's constant declarations.

3. Find examples of the interface specification versus implementation distinction in other engineering disciplines, such as electrical engineering, architecture, automobile construction, and stereo systems.

4. Read and critique the discussion of name and structural type equivalence in J. Welsh, M. J. Sneeringer, and C. A. R. Hoare, "Ambiguities and Insecurities in Pascal." *Software—Practice Exper. 7*, 6 (November 1977).

5. Read about at least one other abstract type language (e.g., Alphard, CLU, Euclid, Modula, Tartan) and write a detailed comparison with Ada.

6. Ada provides only a limited amount of control over access to data structures. In "The Narrowing Gap Between Language Systems and Operating Systems" (*Information Processing 77*, North-Holland, 1977), Anita Jones argues that language designers can learn a lot from operating system mechanisms for access control. Read and critique this paper.

7. In "Information Distribution Aspects of Design Methodology" (*Information Processing 71*, North-Holland, 1971), Parnas does not propose any linguistic mechanisms for supporting information hiding. Attack or defend the position that linguistic mechanisms are needed to enforce information hiding.

8. Evaluate the alternatives to global variables discussed in J. E. George and G. R. Sager's "Variables—Binding and Protection" (*SIGPLAN Notices 8*, 12, December 1973).

9.** We have discussed the similarities between generic packages and procedures; they are both parameterized abstraction mechanisms. If procedures were allowed to return packages, then the two would be the same. Develop this idea in detail. What extensions would have to be made to procedures to capture all of the power of generic packages? What implications would this have for the rest of the language? Would implementation of package returning procedures be more or less difficult than implementation of generic packages?

10.** Packages are second-class citizens in Ada. For example, packages cannot be passed as parameters, stored in variables, or made elements of arrays. Investigate in detail the design and implementation issues that would result from making packages first-class citizens.