Chapter 14

The Representation of Types

Overview

We began the discussion of types in Chapter 5 by the primitive types supported by languages and their connection with the hardware types supported by typical computers. In this chapter we consider the implementation of both primitive and user-defined types. Finally, we continue in Chapter 15 with a discussion of the semantics of types.

To augment the primitive types, most languages permit types to be defined by the programmer. A type definition enables the programmer to define the physical properties of a new type and to name it. A type object represents this type information inside the translator. It is composed of three parts: a name, a type, and a body of information. Various types of types include: primitive, array, record, and enumerated. Each kind of type has its own declaration syntax and corresponds to a distinct type of type object within the translator.

We consider simple types (enumerated types, constrained types, pointer types) compound types (arrays, strings, sets, records), and union types (free and discriminated).

Operations which can be performed on compound objects include value construction—the combination of a set of components into a single compound object, selection—which enables the programmer to reference a part of the compound object, and dereferencing—which maps a reference into a program object or value. Modern programming languages implement some or all of these to various degrees.

14.1 Programmer-Defined Types

A data type is a set of objects with an associated set of functions that permit us to manipulate the objects. To use a type we must be able to represent those objects and functions in our source code and in the computer. A type definition gives us ways to do both. It supplies all information about the physical properties of the new type, and gives names to the type itself, and, if it is a structured type, to its parts. From this information, a translator can build constructors, selectors, and predicates that let us create and manipulate objects of the new type.

In this section, we examine the ways in which types can be represented in the source code and within a translator.[1] A new type representation may be defined by:

- Listing all its members.
- Placing restrictions on an existing type.
- Combining elements from existing types into an aggregate.

We will look at typical forms for these kinds of type declarations and see how a translator uses the information they provide.

14.1.1 Representing Types within a Translator

Whether a type is primitive or defined by the programmer, the properties of the type are stored by the translator when the type is defined. For the time being, let us imagine that all this information is stored in one place, and let us use the term *type-object* to refer to the collection of information that the translator stores about a type. In reality, depending on the language being implemented, the type information could be stored coherently in a type-object, or it could be scattered throughout various translator tables. Moreover, the amount and nature of the information stored varies from language to language, even for similar data types.

There are various types of types: primitive types, array types, record types, enumerated types, and others. For each type of type, a different collection of information must be supplied by the programmer and stored by the translator. Thus a type-object can be seen as a three-part entity, having a name (usually[2]), a type, and a body of information, which we will refer to as the *body*. The *type of a type* tells us how to interpret the body of the type, and the body of the type tells us how to interpret objects of that type. Every kind of type declaration supported by a language corresponds to a distinct type of type within the translator.

A data object also consists of a name, a type, and a body of information. The name is kept in the symbol table, and the type, represented by a pointer to a type-object, may be attached either to the name or to the body, depending on the language. Let us diagram objects as shown in Exhibit

[1]Chapter 15 covers the semantics of types and the ways in which distinct meanings, or semantics, can be given to types with the same representation. In this section we are concerned primarily with the representation of types, not the semantics of similarly represented types.

[2]In some languages, it is possible to create a type with no name.

Exhibit 14.1. Object diagrams.

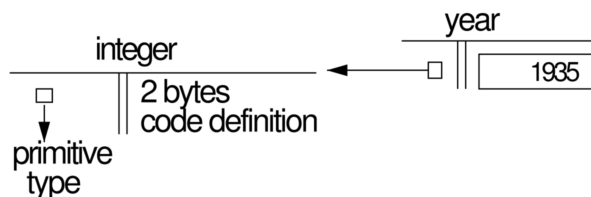
14.1. The object is represented by a "*T*" shaped figure, with the object's name (if there is one) written on top, the type on the left, and the body on the right. The type of an object is always a pointer to some type-object. The body of a variable is a storage object and will be diagrammed as a box. The body of a constant or a type is a series of data values.

The body of a type must contain enough information to support whatever semantics are defined for the type. The nature and amount of this information varies among the different types of types, and from language to language. Examples are: the base type of a pointer or array type, the dimension of an array, and the order, types, and names of the fields of a record type. In addition, a type body might contain extra or redundant information, designed to make use of the type more efficient. One such redundant fact might be the total size, in bytes, of an object of this type. While this might be calculated from other information, it is used frequently and should be kept "handy".

We will define (somewhat arbitrarily) general and useful type-objects for each type of type as we consider it. Although examples of type declarations are drawn from several languages, we should emphasize that the type-objects diagrammed do not necessarily reflect existing translator mechanisms for those languages. However, they would be appropriate as part of the semantic basis of a new object-oriented language with similar features.

Type-objects for the primitive types are defined by the compiler itself. We will diagram a primitive type body by listing only the size, in bytes, and denote the rest of the type's definition as a "code definition". An integer variable named "`year`" is diagrammed, with its type, in Exhibit 14.2.

Exhibit 14.2. An integer object and its type.



14.1.2 Finite Types

When a type is semantically unrelated to existing types and has a small number of members, it is practical to define it by simply listing, or *enumerating*, identifiers for the members of the type. These are called the *type constants*.

If a language supports enumerated types, it must also provide some functions that are automatically defined for every enumerated type. Comparison for equality must be supported. Other useful and common functions include comparison for inequality, successor, and predecessor. Input and output routines are sometimes supplied.

The programmer supplies only identifiers for the type constants; the translator must create an encoding for them. The obvious encoding is to use the integers, in order, starting with zero, to represent the type constants. This encoding makes it easy to implement comparison and successor functions; the corresponding integer functions are simply carried over to the new type. The standards for both Pascal and ANSI C explicitly state that the enumerated type will be represented by integers.[3]

As is common when languages are compared, type declarations that are syntactically very similar can create types with widely varying semantics. C is a language at one extreme. The enumerated type declaration is no more and no less than a convenient way to define identifiers for integer constants. An enumerated type, in C, is implemented by type `int` and has semantics identical to `int`. This is consistent with the use of types in C primarily to allocate and access data objects, rather than as a vehicle for semantics. Enumerated type constants are represented as integers and are considered to be integers. Characters and truth values, the two primitive enumerated types, are integers in C.

The treatment of enumerated types in Pascal is like C, with one very important exception: the enumerated type is semantically distinct from the type integer, which is used to represent it. Type integer is incompatible with enumerated types in Pascal. This means that you cannot, for example, multiply two values of an enumerated type in Pascal, as you could in C! Nor can you mix values of two different enumerated types in an expression.

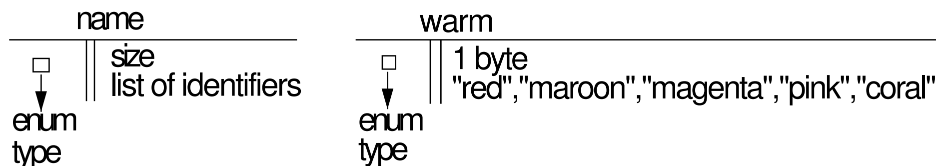
There are some functions that are predefined for any enumerated type in Pascal. These are: `Succ` (successor), `Pred` (predecessor), `Ord` (conversion to type integer), assignment, and all the comparison operators. An enumeration constant or variable may also be used as a subscript. These functions are enough to make enumerated types useful, but not enough to make their use convenient. Unfortunately, Pascal lacks convenient means to read in and print out enumerated constants. Thus every program that uses an enumerated type must contain code to perform input and output conversions. Often this takes the form of two rather tedious `CASE` statements.

We can only speculate why enumerated I/O is not supported in Pascal. One possibility is that the idea was so new when Pascal was designed that Wirth did not realize that omitting I/O would limit the uses of enumerated types. A more likely explanation is the quest for simplicity. Pascal I/O is very simple and less flexible than the formatted I/O functions provided by many languages. Pascal provides less control over detail than FORTRAN, C, or APL. Adding enumerated I/O would

[3] C permits programmers to use the default codes or assign their own integer codes.

Exhibit 14.3. A type-object for an enumerated type.

```
TYPE warm = (red, maroon, magenta, pink, coral);
```



have complicated a simple and elegant design, and may have been considered too unimportant to justify this cost.

To provide I/O for enumerated types, a language definition would have to include format descriptors for type constants. A translator would need the constant identifiers available at run time, so that inputs could be encoded and outputs decoded, automatically. Thus a type-object for an enumerated type would need to contain a list of pointers to the names (represented as character strings) [Exhibit 14.3].

Sending type constants to the output stream is easy enough, but some uniform way would be needed to recognize them in the input stream. All of these things are easy enough to define and implement, and support for input and output would make enumerated types considerably more useful.

An implementation of enumerated type I/O would increase the complexity of the compiler's I/O system and increase the amount of system code that would be included with a program at run time. In a program that used enumerated types, there would be a corresponding decrease in programmer-generated code, because programs would not need to manually encode and decode the enumerated constants. Overall, the costs seem modest and the benefits real.

14.1.3 Constrained Types

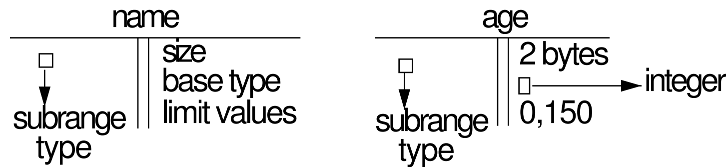
Some languages permit a programmer to define a new type by applying constraints to an existing type. This is a powerful tool for expressing semantic intent. If a language supports constrained types, the run-time system for that language must check each value of the type to ensure that it obeys the constraint. The type-object must therefore include the base type and the limiting values.

Pascal and Ada provide subrange type declarations which define a new type whose members are a consecutive subset of an existing simple type. The programmer specifies the initial and final values that will belong to the new type. (The base type of these values is implicit.) Exhibit 14.4 shows a type-object for a Pascal subrange type.

Values of the constrained type and the underlying base type are compatible and may be combined in operations. Functions for either type may be applied to the other. However, a computed value is checked (at run time) before assigning it to a constrained variable; any violation of the

Exhibit 14.4. A constrained type in Pascal.

```
TYPE age = 0..150;
```



constraints causes a run-time error.

C does not support subrange types. This is consistent with the general philosophy that C types are used to define storage configurations, not semantics. A constraint has no effect on the size or the encoding of the representation; it is relevant only to the semantics of values of that type. (A value may be acceptable or not in the constrained type, while all representable values are legal for the base type.)

14.1.4 Pointer Types

All pointers represent machine addresses, so the semantics of a pointer type do not depend on the semantics of its base type. (The *base type of a pointer* is the type of the object to which the pointer points.) Moreover, the size of a pointer is quite independent of the base type, so no storage layout information is needed in the body of a pointer type except the size of pointers on the target machine. Any pointer value will physically fit into any pointer variable.

This innate relationship among pointer types is exploited in FORTH. It supports pointers in the sense that the address of any object can be obtained and stored in an integer variable. Pointer arithmetic (implemented in terms of integer arithmetic) is possible, and integer variables may be declared and used for pointer values. However, no distinction is made between pointers with different base types.[4]

Most languages, however, require the programmer to declare the base type of every pointer. Therefore, we will diagram a pointer type as shown in Exhibit 14.5. Information about the base type of a pointer is not needed for allocation or for access. It is required only by languages whose compilers use that type information to determine the semantics of the program. Even among languages that require declaration of a pointer's base type, that type information may be used in the following different ways:

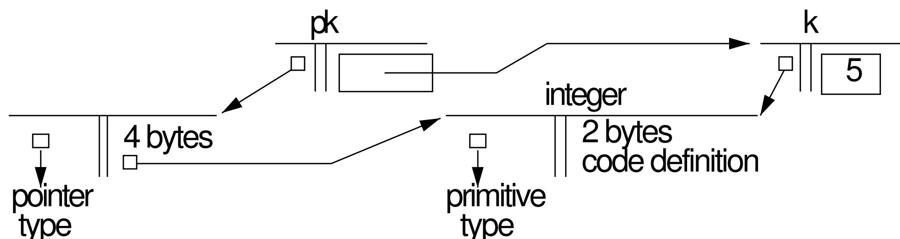
1. To compile the correct meaning of ambiguous operators, such as "+".
2. To determine the number of bytes to fetch when a pointer is dereferenced and used in an expression.

[4]Indeed, no distinction is made between pointers and integers!

Exhibit 14.5. A C pointer and its type-object.

We declare and initialize an integer pointer, `pk`, to point at an integer, `k`, which is initialized to the value 5.

```
int k=5, *pk=&k;
```



3. To determine the type of the result of a dereference, so that it may check for type errors.
4. To dynamically allocate space for an object of the base type.

Similar declarations can have very different meanings in different languages. For example, in C, the base type is used only for purposes 1 and 2. In the following list, assume `p` is a pointer whose base type is BT:

1. Arithmetic on pointers is defined in terms of the base type. The expression `p+1` will cause `p` to be incremented by the length of an object of type BT.
2. The meaning of dereference and fetch depends on the base type. The expression `*p +1` will cause a fetch operation from the memory location stored in p and increment that value by 1. The number of bytes fetched will depend on BT, which must be a simple, primitive type.
3. In general, C does not perform type checking.
4. The type of the pointer is not used for allocating base-type objects. To allocate a base-type object dynamically, one refers directly to the name of the base type, not to the pointer.

In contrast, Pascal uses the same information for the last three purposes. In the following list, assume `p` is a pointer whose base type is BT, and `b` is an object of type BT:

1. There are no operations on pointers whose meaning depends on the base type. Dereferencing is the only pointer operation that is supported, and its meaning is the same for all pointers.
2. Any type of pointer may be dereferenced and used in an assignment. The statement "`b := p↑;`" copies n bytes, where n is the size of BT.

3. Pascal is a fully type-checked language. In the expression "`b + p↑ * 2`", the base type of `p` is checked to ensure that it is appropriate for the surrounding expression.

4. The argument to a call on `NEW` (which performs dynamic allocation in Pascal) must be a pointer, `p`. An object is allocated of `p`'s base type, and the resulting reference is stored in `p`.

14.2 Compound Types

14.2.1 Arrays

Modern languages permit simple types to be combined to form compound, or structured, types. These combinations can be positional (arrays, strings) or not (sets), and homogeneous (arrays, strings, sets) or heterogeneous (records). They can have fixed size (arrays, records) or variable size (strings, sets). Let us look at typical type declarations, type-objects, and accessing methods for these compound types.

An array is a fixed-length sequence of elements of a single type, called the *base type*. The length of the sequence is called its *dimension*. Elements of an array are accessed by position, and the position values of the first and last elements are called the *array bounds*. In theory, array positions and bounds could be values of any simple, discrete type, called the *index type*. Possible index types include the integers and any type that is implemented by mapping onto the integers. In practice, various languages are more or less restrictive about the index types that can be used. For simplicity, in this discussion, let us presume that integers are used for the index type.

To access an individual element of an array, we append a *subscript* to the array name. This is an expression (usually enclosed in parentheses or square brackets) whose value falls within the bounds of the array. The subscript, base address, and base type of the array are used to compute the *effective address*, or address of the desired component. This computation is simplest and most efficient if the language requires *zero-based subscripting*. In this system, if an array has dimension D, its bounds are $0 \ldots D-1$. The effective address formulas for zero-based and arbitrary-based subscripting are shown in Exhibit 14.6.

In the old days, when all computer memory was a series of "words" (not bytes) and all numbers were one word long, arrays of numbers were implemented very simply and efficiently. The array's base address was loaded into the computer's memory address register and the subscript was loaded into an index register. Then the computer's indexed-fetch or indexed-store instruction dynamically computed the desired effective address.

Multidimensional Arrays

Some older languages, such as MAD, FORTRAN, and APL, specifically supported arrays of two or more dimensions. (Following APL terminology, let us use the word *rank* to mean the number of dimensions of an array.) The rank was limited, in FORTRAN, to the number of index registers available on the host computer, since each subscript was kept in a register for efficiency's sake.

Exhibit 14.6. Formulas for the effective address computation.

Let ba be the base address of the array A, and let $size$ be the number of addressable units (bytes) required to store a value of the base type of A. Then the effective address corresponding to $A[s1]$ is:

Zero-based subscript: $ba + size * s1$

Other-based subscript: $ba + size * (s1 - lower\ bound\ of\ A)$

For a multidimensional array, with zero-based subscripting and declared dimensions $d1, d2, \ldots, dn$, the effective address corresponding to $A[s1, s2, \ldots sn]$ is:

$ba + size * ((((s1 * d2 + s2) * d3 + s3) \ldots) * dn + sn)$

However, computing an effective address for an array of rank n cannot be done simply by adding together all the subscripts; it requires $n - 1$ multiplications and additions, plus one index operation [Exhibit 14.6]. An example of a three-dimensional subscript computation for Pascal arrays is shown in Exhibit 14.7.

Support for multidimensional arrays is important in languages that are intended for use by scientists and engineers who work regularly with matrices. In these older languages, special syntax

Exhibit 14.7. Effective address computation for Pascal arrays.

Here we give declarations for two arrays of rank 3. Assume that storage for `Matrix` starts at location 1000 and storage for `Box` starts at location 2000. In this implementation, the size of a real is 4 bytes and the size of an integer is two bytes.

```
VAR Matrix:  array[0..1] of array [0..2] of array [0..4] of real;
       Box:  array[1..2] of array [5..7] of array [-2..2] of integer;
```

`Matrix` has zero-based subscripts, so we use the simpler form of the formula to compute the effective address corresponding to `Matrix[1][2][3]`.

```
1000 + 4 * (((1*3 +3)*5 + 3)) = 1000 + 4 * 33 = 1132
```

`Box` does not have zero-based subscripts, so the lower bound for each dimension must be subtracted from the corresponding subscript. Here we compute the effective address for `Box[2][7][-1]`:

```
2000 + 2 *(( (2-1)*3 + (7-5) )*5 + (-1--2) ) =
2000 + 2 *( 5*5 + 1 )    =    2000 + 2*26    =    2052
```

Exhibit 14.8. A three-dimensional array in FORTRAN.

```
REAL AR3
DIMENSION AR3[5, 2, 4]

* Store a number in one element of the array.
AR3[2,2,1] = 17.0
```

and semantics were provided to support higher-rank arrays [Exhibit 14.8].

Type declarations were invented in the late 1960s, and this changed the way that higher-rank arrays were implemented. Languages designed since then have array type declarations that define arrays in terms of an *arbitrary* base type. Thus a two-dimensional array became, simply, an array of arrays. Special syntax and special semantic rules were no longer needed to handle higher-rank arrays; they could be handled by iterating the type definitions and type accessing syntax for rank one arrays, as shown in Exhibit 14.9, item B.

This is a real simplification in the language. The syntax is simpler because the brackets written to declare and use subscripts need to delimit only a single expression, not a list of expressions. The semantics is simpler because the complex formula for computing a higher-rank effective address is replaced by iterating the simple formula for computing a one-dimensional effective address. The size of the various base types replaces the declared dimensions in the formula [Exhibit 14.10]. C, was designed with streamlined simplicity in mind. It supports only zero-based subscripting and only one-dimensional arrays. A higher-rank array is defined and referenced as an array of arrays.

Pascal, on the other hand, made an interesting concession to custom. Pascal has type declarations and supports arrays with an arbitrary base type, so it does not need to have FORTRAN-like array notation. However, programmers were used to the multisubscript FORTRAN notation shown in Exhibit 14.8. So Wirth compromised. The semantic basis of Pascal is like C: it supports only

Exhibit 14.9. Basic and sugared notation for arrays in Pascal.

A. Sugared notation:

```
VAR Sugared:  array[1..3, 1..10, 0..3] of char;
c := Sugared[1, 5, 2];
```

B. Equivalent array declared using basic notation:

```
VAR Plain:  array[1..3] of array [1..10] of array [0..3] of char;
c := Plain[1][5][2];
```

Exhibit 14.10. Effective address computation for a four-level type.

Assume that storage for the variable named `Plain` from Exhibit 14.9 starts at address 1000. Let us compute the effective address for `Plain[1][5][2]`.

	Type	Dimension	Total Size
1.	char		1 byte
2.	array [0..3] of (1)	4	4 bytes
3.	array [1..10] of (2)	10	40 bytes
4.	array [1..3] of (3)	3	120 bytes

```
Effective address = base address of Plain +
    (1-1)*40 bytes + (5-1)* 4 bytes + (2-0)*1 byte =
    1000 + 0 + 16 + 2 = 1018
```

one-dimensional arrays. But the syntax of Pascal was extended to include FORTRAN-like subscript notation, with all subscripts (optionally) included between a single pair of brackets. This "extra" declaration form is called *syntactic sugar* because it is unnecessary but makes the language sweeter and more attractive for many users. The multidimensional notation that the programmer writes is converted by the parser to the basic notation [Exhibit 14.9]. Of course, the programmer may always choose to write directly in the basic notation. A Pascal code generator contains semantic interpretation routines for the one-dimensional notation only.

Ada Arrays and Slices. In the treatment of arrays, as in many other ways, Ada is like a greatly complicated (and more capable) version of Pascal. Like Pascal, Ada provides syntactic forms for declaring both arrays of arrays and multidimensional arrays. Unlike Pascal, though, the two forms are not equivalent! Ada provides an operation called "slicing"[5] for arrays of arrays that cannot be applied to multidimensional arrays.

A *slice* of an array is the contents of a consecutive series of locations, denoted by the array name with a range of subscripts. If `Ar` is an array with bounds 1...10, then the slice from positions 3 through 5 is denoted thus:

```
Ar(3..5)
```

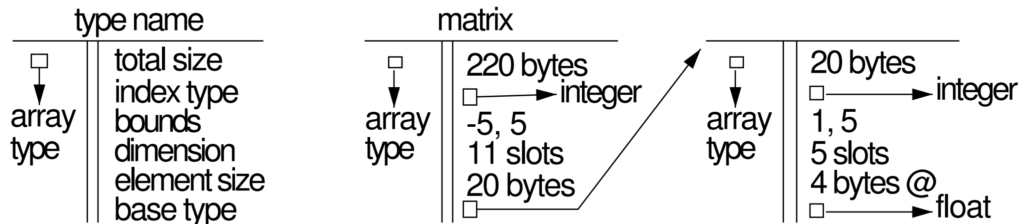
A slice can be used in an assignment or a function call. Slicing provides a nice way to do some array operations coherently that would need to be written with a loop in Pascal or C. However, Ada slices are not as flexible or as powerful a tool as the APL subscripting facility, described in Section 14.3.2.

[5] This term was introduced by ALGOL-68.

Exhibit 14.11. Diagrams of array type-objects.

The type declaration is given using Ada syntax.

```
type Matrix is array(-5..5, 1..5) of float;
```



Type-Objects for Arrays

We see that the type-object for an array type in a modern language only needs to have information about one dimension. For that dimension it must store the total size, the index type, the bounds, and a pointer to the base type. For efficient use in various computations, it might also contain the dimension and the size of the base type. Exhibit 14.11 shows diagrams of two array type-objects, representing an array of arrays in a language with non-zero-based subscripts. Exhibit 14.12 shows the simpler type representation possible with zero-based subscripts. One number, the dimension, takes the place of three: dimension, lower bound, and upper bound.

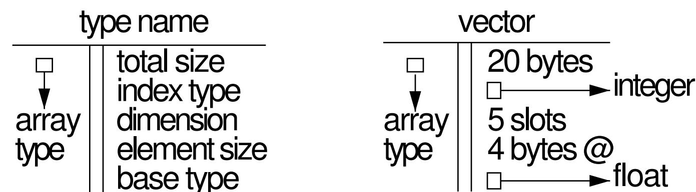
Arrays with Undeclared Bounds. In certain situations it makes sense to declare an array but not specify the array bounds. This is permitted in some languages if either:

1. The omitted information can be deduced from the context, or
2. The omitted information is not needed by the compiler.

Exhibit 14.12. Diagram of a zero-based array type-object.

The type declaration is given using C syntax.

```
typedef float [5] vector;
```



Where structured initializers may be given for arrays, and a particular lower bound is either required or is the default, as in C and Ada, the upper bound of an array can be deduced from an initializer if one is given as part of the declaration. In this situation, omitting the bounds from the declaration reduces the redundancy and eliminates the chance that the programmer might miscalculate the dimension needed. However, it also eliminates the chance that the compiler can detect a faulty initializer. (This is another example of the general rule that redundancy is a pain to write into a program, but can serve as a double check on the typist's accuracy.)

An example of the second principle is seen in C. An array parameter may be declared with an unspecified dimension because C does not use the dimension information for parameters. All C arrays are passed by reference, so only a pointer is allocated for the array in the function's stack frame. Further, C does not perform bounds checks of any sort. Thus the dimension of an array parameter is immaterial within the function. Within the C subroutine, only the base type of an array is ever needed.

Contrast this to Pascal which does pass arrays by value, if desired, and does check array bounds. Both of these facilities can be used to make programs more reliable and easier to debug; however, this security comes at a high price. The bounds of a Pascal array parameter must be declared and fixed at compile time. Therefore each function can accept, as arguments, only one fixed size of array. In C, any length array of the appropriate base type can be passed to a function. Thus the Pascal programmer must edit and recompile subroutines if the length of the data array changes, whereas the C programmer only needs to change the declaration of the data array in one routine.

Semantic Protection with Arrays. We have seen how array type-objects are used to allocate array storage objects and access individual elements. Some languages, such as C, use the type information for these purposes only. This same information can also be used to identify run-time errors, as it is in Pascal and Ada. These languages compare the value of the subscript expression to the declared array bounds, and halt with an error comment if the bounds are violated. Of course, this slows down execution. However, this cost is probably justified. Semidebugged C programs often "run wild" and erase the contents of memory, forcing the user to reboot the workstation. Pascal programs seldom do.

Bounds checking is most useful during program development and debugging. Omitting the checks during this phase would be penny wise and pound foolish. Although the checking code consumes execution time and memory space, a single error caught pays for many tests. However, one might assert that finished production programs should not incur the run-time overhead of bounds checking because fully debugged programs just don't run wild. (Of course, one can debate whether any program is ever fully debugged.) Most programs perform well with these checks included. Occasionally, a bounds check in an inner loop can cause a significant slowdown and is therefore undesirable.

The Ada language includes a way to turn off unwanted checking, called a `suppress PRAGMA`. If a compiler honors a suppression request (it is not required to do so), type checking is suppressed throughout the block that contains the pragma declaration. The design of Ada encourages very

limited and selected use of suppression pragmas. A separate declaration is needed for each kind of check that is to be suppressed. The programmer is urged to use these only in fully debugged code that is unacceptably slow, and then to place the suppression declaration in the smallest block that contains the slow code section. A `PRAGMA` declaration to suppress bounds checking has this format:

```
PRAGMA Suppress (Index_Check);
```

14.2.2 Strings

A string is a variable-length array of characters. In many languages (for example, Pascal and Ada) the type string is treated as a special case of an array, with an integer lower bound of 1 and a base type of character. This treatment does not really capture the essence of strings. Strings are different from general arrays because:

- The length of a string is, in general, unknown at compile time.
- Strings of many different lengths are commonly used together.
- A string variable should be able to "contain" any string.
- A string function should be able to work on any string.
- Strings have special semantics, such as the rule for alphabetic-order comparison of strings with unlike lengths.

The "strings" supported by Ada and Pascal are not variable-length objects. A string variable has a declared length and can only contain string values that are short enough to fit within this maximum. Strings shorter than that must be manually padded with blanks. Using fixed-length arrays to represent variable-length strings doesn't really work.

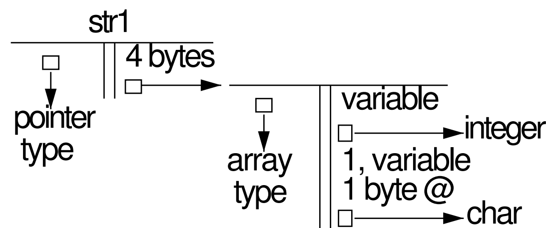
A good representation for strings must embody their essential variable-length nature, use storage efficiently, provide for efficient processing, and have reasonable error-recovery properties. There are two representations for string values that meet these criteria: the *counted string* and the *terminated string*.

In a *counted string* the first byte (in subscript position 0) is an unsigned integer which gives the length of the string. String functions must calculate the length of a newly constructed string and store it in the first byte. With this representation, strings can be processed easily using `for` loops. The error recovery potential is not ideal, but it is adequate. If a program error happens and the count byte is changed to a garbage value, any loop that uses the garbage will still terminate after no more than 255 iterations. No runaway loop in a string function will be infinite and force the user to reboot a machine.

A *terminated string* contains only characters, but following the last data character there is a terminator, probably the null character (ASCII code 0) (strings cannot contain the terminator). Functions must be written to add a terminator to the end of any newly constructed string. Strings are normally processed by looping while the data is nonnull. This representation for strings works

Exhibit 14.13. A type-object for a null terminated string.

```
char *str1 = "This is a string literal.";
```



well for error recovery. If a program error happens and a null terminator gets "wiped out", it is highly likely that another zero byte is somewhere nearby in memory and will terminate the runaway operation soon.

Both of these representations make efficient use of storage and support efficient string processing. The null terminated string, used in C, seems to have slightly better error recovery properties. Exhibit 14.13 shows appropriate type-objects for strings.

Once the representation of a string value is decided, we can deal with the question of string variables. In a compiled language, storage for variables in run-time stack frames is laid out at compile time. The size of each allocation is fixed then and is not variable. This conflicts with the variable-length nature of strings. One good solution is to use a dynamically allocated string storage area, and let a string variable be a pointer into this area. This is how BASIC and SNOBOL IV are implemented. All the string functions take pointer arguments and return pointer results. Storage for newly created strings is taken automatically from the system-managed string store.

C implements a version of strings that is half way between the fully functioning string store and the fixed-length strings of Pascal. Like BASIC, a string in C is a pointer to an array of characters. Unlike BASIC, the semantic basis of C does not include a string store, and the string functions do not allocate new storage. C functions such as "`strcat"`, or string concatenate, require that one argument be a pointer to a preallocated storage object long enough to store the result. The effect of this mixed approach is that C's extensive and powerful string library is less easy to use than the string facilities in SNOBOL.

There is a good reason why C does not supply automatically managed dynamic storage: simplicity and execution efficiency were both design goals in C. Managing dynamic storage is not simple, and it requires some sort of string compaction or garbage collection facility. These facilities are all slow and costly. Using a compaction algorithm becomes necessary when available storage has all been used and is now occupied largely by dead strings. (A dead string is one that is no longer bound to any live name or pointer.)

Rather than include a complex dynamic storage management system, the designers of C chose to let programmers implement whatever portion of such a system might really be needed. One

standard technique used in C string programs is to implement a dynamic string store similar to that which is built into BASIC.

14.2.3 Sets

The "set" types supported by Pascal are variants of Boolean arrays. A set value is like a packed array of Booleans, where each index position, or slot, represents one constant of a subrange type or enumerated type, called the *base type of the set*.

The number of constants in the base type determines the number of Booleans in the set value; the first Boolean in the set value corresponds to the first constant in the base type, and so on. A set with one member is represented by a Boolean array with one element which is `TRUE` and all the others `FALSE`. A set with several members is represented by an array with several `TRUE` elements [Exhibit 14.14].

The "`in`" operation in Pascal is the selector function for a set. The operands of `in` are a value of the base type, v, and a set value, S. The result of the expression v *in* S is the Boolean value stored in the position of S that corresponds to v. For example, in Exhibit 14.14, "`common`" is a set variable of type "`combo`". We would write "`IF green IN common`" to find out whether "`green`" is a "`common`" color.

The standard mathematical set operations union, intersection, and difference are denoted by the Pascal operators "`+`", "`*`", and "`-`", respectively. These take operands of a specific set type and return a result of the same set type. These operators are implemented by applying Boolean operations to corresponding elements of the two operand arrays. Union is implemented by `or`, and intersection by `and`. Set difference ("`-`") is implemented by the composition of "`complement`" and "`and`".

A Boolean value can be represented by a single bit. In an efficient representation of a Boolean array, we *pack* the bits into a series of bytes, so that every bit is used. By packing the Boolean array, we are able to use the efficient bitwise logical operations to implement set operations. These operations let you "turn on" and "turn off" bits singly, or several at a time. Turning a bit on corresponds to adding a member to a set. The diagram on the left in Exhibit 14.15 shows an appropriate type-object for a Pascal set type.

Although sets are included in Pascal, most languages do not have a corresponding set type constructor. This is not surprising, since both the semantics and the representation for sets are somewhat complex. One of the design goals for Pascal was simplicity, and it is interesting to ask why Wirth included this nonsimple type of type in his design. The reason was a combination of three factors: completeness, validity, and simplicity.

Completeness. A powerful language should contain constructs that support standard mathematical notation. It should also reflect as many as possible of the capabilities of the typical modern computer. Including a set type in Pascal, with the representation described, "completes" the language in two ways.

Exhibit 14.14. A set type in Pascal.

We declare an enumerated type, `color`, and a set type, `combo`, whose base type is `color`. Some variables of type `combo` are declared and initialized. The Boolean array representations of these variables are diagrammed below.

```
TYPE  color = (red, pink, orange, yellow, green, blue, violet,
          magenta, brown, black, gray, white);
      combo = set of color;
VAR   tree, daffodil, iris:  combo;
      palette, common:  combo;

daffodil := [white, yellow, orange, green];
iris := [white, yellow, blue, violet, green];
tree := [white, gray, brown, black, green];
palette := iris + daffodil;        { '+' means set union. }
common := iris * tree * daffodil;   { '*' means set intersection. }
```

In the following representations, a "0" represents a `FALSE` value, and a "1" represents a `TRUE` value.

- Cardinality of base type (color): 12
- Number of elements in a `combo` value: 12
- Size of each element: 1 bit
- Representation of `daffodil`: 001110000001
- Representation of `iris`: 000111100001
- Representation of `palette`: 001111100001
- Representation of `common`: 000010000001

The bitwise operations on a computer are vital for applications such as number conversion, using hardware switches, and packing and unpacking data values. With sets, Pascal provides "access" to the bitwise hardware instructions; that is, it provides a type constructor and some operators that translate into the bitwise operations.

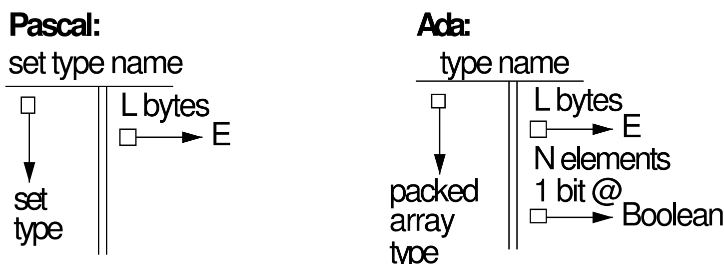
Sets are a standard mathematical notion that can be used to model quite a variety applications. By including an efficient implementation of sets and set operations, Pascal supports an important mathematical notation.

Why, then, does C not support sets? The design goals for C were significantly different from those for Pascal. The fact that mathematicians use sets was of little importance in designing a language for systems programming. The need for access to the bitwise operations was compelling, but it was met in a different, and simpler, way. C includes the bitwise operators "`&`" (and), "`|`"

Exhibit 14.15. Type-objects for set types.

The diagram on the left is a type-object for a Pascal set type. On the right is a type-object appropriate for an Ada-like implementation of sets as Boolean arrays. We use the type-object defined, above, for zero-based arrays. The symbols used in these type-objects are defined thus:

- E, an enumerated type, is the base type of this set type.
- N is the number of enumeration constants in E.
- L is the smallest integer greater than or equal to (N/8).



(or), "^" (exor), and "~" (complement), which can be applied to any type of value.

If a C programmer wanted to implement set semantics he or she would define a set value as an array of unsigned integers, where each integer represents 16 or 32 index positions in the set. To implement the set operations, the programmer would simply iterate the primitive bitwise operators as many times as necessary to process each byte of the array. This is a simple and straightforward implementation. There is no particular need to make sets a primitive type constructor.

Validity. Support for semantic validity was one of Wirth's most important design goals. All primitive operations that he included were semantically valid for some primitive type. Strong type checking ensured that validity was maintained by every operation.

Wirth would have considered the semantics for the bitwise operators in C to be unacceptable. An unsigned integer type should be used to store unsigned integers, not bit strings. The bitwise logical operators have no semantic validity when applied to integers. Basically, they can only be meaningfully applied to packed arrays of Booleans, and that is what Pascal supports.

But semantic validity and completeness were also of primary importance in the design of Ada, and Ada does not have a set type constructor. It does provide an easy way to implement the set semantics, though. In Ada, the type one-dimensional array of Booleans is given special semantics. The logical operators `and, or, xor`, and `not` may be applied to one-dimensional Boolean arrays. The relational operators `=, <=`, and the like may be applied to any one-dimensional array. Thus sets can be easily implemented as Boolean arrays. The diagram on the right in Exhibit 14.15 shows an appropriate type-object for this set implementation. Exhibit 14.16 shows a set of Ada

Exhibit 14.16. Implementation of sets in Ada.

We declare an enumerated type, `color`, and a set type, `combo`, whose base type is `color`. Some variables of type `combo` are declared and initialized. The Boolean-array representations of these variables are diagrammed below.

```
type  color is (red, pink, orange, yellow, green, blue, violet,
          magenta, brown, black, gray, white);
type  combo is array(red..white) of boolean;

T: constant boolean := TRUE;
F: constant boolean := FALSE;

tree:  constant combo := (F,F,F,F,T,F,F,F,T,T,T,T);
daffodil:  constant combo := (F,F,T,T,T,F,F,F,F,F,F,T);
iris:  constant combo := (F,F,F,T,T,T,T,F,F,F,F,T);
palette, common:  combo;

palette := iris or daffodil;
common := iris and tree and daffodil;
```

type declarations to implement types analogous to the Pascal `color` example from Exhibit 14.14. Exhibit 14.17 shows how corresponding set selection operations would be written in Pascal and Ada.

Simplicity. Wirth intended Pascal to be a powerful but simple and minimal language. Highly complex semantic mechanisms did not fit this purpose. He also wished to have a clean, general design with few special cases and few restricted, special-purpose capabilities.[6]

A language with suitably powerful array operations would not need a special type of type to

[6]Note that the special semantics for strings in Pascal is an example of what Wirth wished to avoid. String types break many of the type rules that govern the rest of the language.

Exhibit 14.17. Selecting a member of a set.

In Pascal we select a member of a set using "`in`":

```
if green in common then ...
```

In Ada we use a subscript to select a set member:

```
if common(green) then ...
```

implement the semantics of Pascal sets, as shown by the array-of-Booleans implementation in Ada. APL also supports coherent array operations which make the Boolean-array implementation of sets easy and straightforward.

But the powerful, general array operations in APL do not meet the criteria of simplicity and minimality. The solution in Ada (including primitive bitwise operators that are defined only for one-dimensional Boolean arrays, and coherent array operations that apply only to one-dimensional scalar arrays) is too nongeneral and special-purpose to meet Pascal's design goals. While Pascal sets are not a simple type of type, they are less complex than APL array operations, more elegant than the Ada solution, and more valid than the C approach.

14.2.4 Records

A record is a compound object consisting of an ordered series of components of heterogeneous types. These objects are usually implemented by contiguous blocks of memory, in which the fields of a record object are stored in the order specified by the programmer.[7] A modern language permits the programmer to manipulate a record object coherently or to access its components and process them individually.

Type-Objects for Records

The type-object for a record will contain all the information needed to allocate and access the record. For allocation, only the total size of a record storage object is needed. Intuitively, a record object is no more and no less than the sum of its parts. However, in real implementations, the storage object for a record may contain more bytes than are needed to store the components, and the extra bytes will generally contain garbage. This happens because many computers require all numbers to be *aligned on word or long-word boundaries*. With word alignment, all objects start at even byte addresses; with long-word alignment, all addresses are evenly divisible by 4. Padding is inserted into a record type if the size of some field in a record declaration would cause the following field to break alignment rules. The amount of padding included (if any) depends on the hardware. Thus all we can say about the size of a record is that it is at least as great as the sum of the sizes of its parts.

To access a record component, a compiler needs to know its *offset*, that is, the number of bytes between the beginning of the record object and the beginning of the desired field. Each offset is the sum of the sizes of all preceding fields, plus any preceding padding bytes. When the compiler processes the list of component types in a record declaration it calculates these offsets and binds each one to the corresponding record part name. We will represent all this information (offsets, part names, component types) as part of the type-object for a record [Exhibit 14.18].

[7] This order is by no means necessary unless a language permits the programmer to circumvent the usual accessing methods and access the record's representation directly.

Exhibit 14.18. A type-object for a record.

```
struct personcell{char* name; char sex; float salary; personcell* next};
```

We would diagram the type `personcell` as follows:

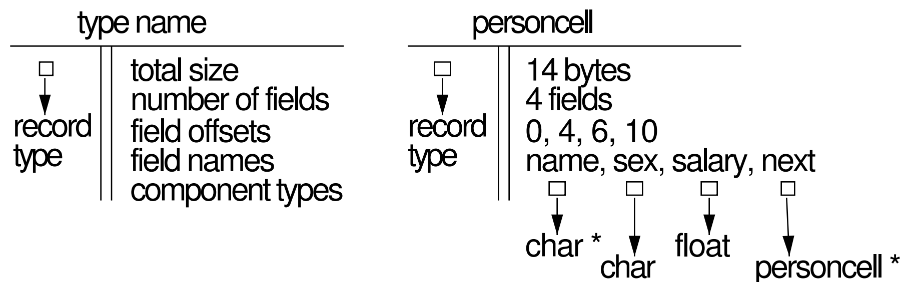


Exhibit 14.19. Defining a record type in FORTH.

We define functions to allocate and access a record type named `personcell`. Objects of this type will be allocated dynamically and attached to linked lists.

```
:  new_personcell
    here 10 allot;    ( Allocate 10 bytes; leave address on stack.     )
( Selector functions for the record type personcell.  All expect a     )
( single argument which is a pointer to a record.  All leave on        )
( the stack a reference to one component.                              )
:  .name ;            ( Zero offset for first field.                   )
:  .sex    2 + ;      ( Assume string pointers take 2 bytes.           )
:  .salary 4 +;       ( One byte for sex, one for padding.             )
:  .next   8 +;       ( Leave four bytes for double-length integer.    )
```

Assume that the variable `employee` is a pointer to a `personcell` record. If we wish to store a number in the `salary` field of this employee's record, we write the following expression. (The operator `D!` performs a double-length assignment.)

```
89700 employee @ .salary D!
```

Part Selectors

The part names for a record type are selector functions. Like functions, each one takes an argument (a reference to a record), performs an action (increments that reference by an offset amount), and returns a reference to another object (one component). The nature of part names can be seen most clearly in FORTH. FORTH has no special provision for records, but it permits the programmer to allocate storage manually, manipulate references to objects, and do address arithmetic. When a FORTH programmer wants to use records he or she defines an allocation function and a set of selector functions [Exhibit 14.19]. These selectors are used very much like Pascal part names; compare the FORTH expression from the last line of the exhibit to the equivalent expressions in Pascal and C:

```
FORTH:    89700 employee @ .salary D!
Pascal:   employee↑.salary := 89700
C:        employee->salary = 89700
```

So we see that part names can be defined as functions in a language that gives access to low-level information. Why, then, do most languages provide special declarations? There are several reasons: convenience, economy, portability, and semantic safety.

Convenience. The part name function definitions in FORTH are simple and brief, but even so, they are somewhat of a nuisance to write out. The record type declaration is clearly a more convenient way to convey this information. A type declaration conveys all the relevant information concisely.

Economy. Implementing selectors as ordinary functions is overkill. Associating a selector name with an offset takes very little storage, and putting these pairs in the type-object gives an extremely brief way to represent the information. Moreover, we do not need the full generality of functions for this purpose. Selectors are constant functions that are applied and expanded by the compiler. Compiled code contains address arithmetic, not function calls.

Most languages permit the same part name to be used in multiple record definitions. Storing the part name/offset mappings in the type-object is an easy way to implement the required kind of ambiguity; it allows the same part name to be bound to different offsets in different types. It is interesting to note that very old C compilers did not work like this. They probably stored the part names in the symbol table with the function names and other identifiers. The result was that each part name could only be used to mean one offset. A part name could be used in more than one `struct` declaration, but each use had to correspond to the same offset amount! This has been modernized by the ANSI C standard.

Portability. To define the FORTH selectors we had to know exactly how many bytes would be required to implement each primitive type. But this varies from machine to machine. Whatever constants we use for the offsets, they will be wrong for some FORTH implementations.

C provides the same access to low-level information as does FORTH, so one could define record selectors the same way in C. However, this would be foolish, because then the user would have to worry about inserting padding bytes and accommodating varying sizes for the primitive types. If the user writes a `struct` declaration, the compiler takes care of all this.

Semantic Safety. The part selectors defined in FORTH are accompanied by absolutely no checking. If an offset was one too large, the boundary between successive parts would be violated. A program that tried to access that component would get the last byte(s) of it and the first byte of the next component! These bytes do not form a valid object of any sort. Moreover, a FORTH function is not restricted to use with the correct type of argument; any selector could be used on an entirely inappropriate type of record, producing a garbage answer.

Pathnames

Let us define the term *pathname* to mean the sequence of identifiers, starting with the name of an object and continuing with a series of selectors (part names or subscripts). A pathname designates a particular field, fn, of a particular object, Ob.

When a compiler translates a pathname, it uses the series of selectors to compute the address of the specified component, or the *effective address*. Initially, the effective address is set to the *base address*, that is, the address of the first location in Ob. The current type-object is set to the type-object of Ob, and the current selector is set to the first selector in the pathname.

The compiler looks for the current selector in the current type-object. This field has an associated type $t1$, and an offset amount, $n1$. For records, $n1$ is listed explicitly in the type-object. For arrays, $n1$ is found by multiplying the component size (from the type-object) by the array index minus the array's lower bound. The compiler then adds $n1$ to the effective address, sets the current type-object to $t1$, and goes on to the next selector in the pathname. This process is iterated until all selectors are used, and a final offset amount is calculated. A pathname that contains no variable subscripts can be processed entirely at compile time. Otherwise, the constant portions of the computation are done at compile time, and the rest must be deferred until run time.

Partial Pathnames. When part names are simply entered into the symbol table, as they are in COBOL, PL/1, and old versions of C, the programmer must be careful about using the same part name in two different record types. We eliminate this problem by storing the record part names in the type-objects. This makes them into local names, that is, names whose meaning within the type is quite independent of any other meanings in other contexts. Localizing names makes it easier to write correct code. However, one "feature" present in older languages was lost by this change.

In PL/1, the programmer could refer to a sub-subfield without specifying a full pathname; the programmer wrote only the name of the object itself and the name of the sub-subfield. This kind of short-cut naming can shorten and simplify code, especially where a record type contains a structured component, and so on, for several levels.

Pascal has one statement type that partially compensates for this loss of convenience. A "with" statement allows the programmer to establish a local context, within which the initial portion of a pathname can be omitted. The form of a with statement is as follows:

with ⟨partial pathname⟩ do ⟨scope⟩

When the compiler begins to translate a with statement, it evaluates the effective address, *Ea_pp*, for the partial pathname. (If there are variable fields in the pathname, code is generated to complete this process at run time.) During compilation of the with scope, *Ea_pp* is the starting point for computing effective addresses of components, and the current type-object is set to the type of the last field in the partial pathname. Within the scope, all references to field names defined for this type are legal and will be interpreted as offsets added to *Ea_pp*.

At run time, any variable fields in *Ea_pp* are evaluated once, when control enters the with block. Within the block, only the "tail" section of each pathname must be specified and evaluated, saving both execution time and space [Exhibit 14.20].

The principle here is valid and important: when doing several operations with one part of a large compound object, it is more efficient (in several ways) to mark the beginning of that component and make local references relative to that mark. Looking at other programming languages, we see more general ways to solve the same problem.

A C programmer does not need a special statement type to accomplish this; she or he simply sets a pointer to the desired component (at entry to the block) and makes references within the block relative to that pointer [Exhibit 14.21]. The with statement is included in Pascal because C's simple solution is not available. (Recall that, in Pascal, pointers to stack-allocated variables are prohibited.) Moreover, the Pascal solution is semantically cleaner because, in C, the pointer is not constrained to be constant within the local scope.

LISP incorporates what might be called the "right" way to solve the problem. Using "let", the programmer can create a new block with a local symbol and bind that symbol to any object. (The programmer would bind the new symbol to a pointer to the *Ea_pp*.) C-style references relative to this pointer could then be used within the local scope.

Records Are Much Like Arrays

There are many similarities between records and arrays; both are compound types with a series of parts, accessed by selector functions. Their type-objects contain similar information, except that an array type-object is simpler because all fields of an array are the same size and type. Where an array type-object contains one piece of information about a component, a record type-object contains a list.

However, the familiar syntax for subscript (with parentheses or brackets) is markedly different from the syntax for record part selection (with a dot). This is partly an accident of history; neither subscript notation nor dot notation is engraved in stone. A language designer could choose to use either syntax, as shown in Exhibit 14.22.

There is a more important difference between arrays and records than the syntax used for part selection. That is the fact that the traditional array selectors are numeric and record selectors are

Exhibit 14.20. Partial pathnames using `with` in Pascal.

`With` is used here to simplify the source code and reduce execution time for access to one part of a complex data structure, a stack of student records. When the `with` block is entered, the effective address of the top student on the stack is calculated. All references to the field names defined for `StudentType` will be interpreted relative to this address.

```
Type NameType =    packed array[1..20] of char;
     StudentType = record LastName, FirstName:  NameType;
                       Sex:  char;
                       Id:  integer
                   end;
     ClassType =   record { A class is a stack of students.  }
                       Top:  integer;
                       Member:  array[1..50] of StudentType;
                   end;

Var  N: Integer;
     CS101:  ClassType;
     Stu:  StudentType;
...
With CS101.Member[ CS101.Top ] Do Begin
     Writeln( Id, FirstName, LastName );
     If Sex='F' Then FemaleTotal := FemaleTotal + 1
                   Else MaleTotal := MaleTotal + 1
End;
```

not. Let us consider the twin questions:

1. Why don't we use symbolic names to select array components?
2. Why don't we use integers to select record components?

The essence of an array is that its components are semantically uniform. They represent objects from the same domain, and we expect to apply the same operations to each, in turn. The number of array elements and position of a particular element in the array is of secondary importance, at most. Because these elements have uniform meaning to the programmer, a uniform way to name them is needed Giving them numbers is a good solution. Giving individual names to a series of similar objects would be silly.

In contrast, the components of a record are not semantically uniform. They represent different aspects of the same object, not separate objects. Even when components are the same type, they

Exhibit 14.21. Using a pointer in C to emulate "with".

```
typedef struct  {
                 char last_name[20], first_name[20];
                 char sex;
                 int id;
        } student_type;
typedef struct  {
                 int top;
                 student_type member[50];
        } class_type; /* A class is a stack of students.  */
...
{   int female_total, male_total;
    class_type cs101;
    student_type *stu;
...
    stu = &cs101.member[ cs101.top ];
         printf("%d %s %s\n", stu->id, stu->first_name, stu->last_name );
         if(stu->sex=='F') ++female_total; else ++male_total;
...
}
```

Exhibit 14.22. Alternative syntax for part selection.

Let `Student` be a `StudentType`, as declared in Exhibit 14.20, and let `Class` be an array of `StudentType`. There could be many clear and unambiguous ways to denote part selection. A few possibilities are listed here.

	Array Selection	Record Selection
Traditional syntax	Class[5]	Student.FirstName
Possible syntax	Class.5	Student[FirstName]
Uniform syntax	Class@5	Student@FirstName
Functional syntax	Subscript(Class,5)	Select (Student, 'FirstName')

represent different concepts and can easily be given different names. Defining symbolic names for record components is, therefore, natural and functional.

However, constant numeric selectors could be used for records. Allowing numeric (as well as symbolic) selection could have some advantages, especially for writing library or utility routines. Variable numeric selectors cannot be used in a strongly typed compiled language because the type of the result of every expression must be determined at compile time. Suppose R is a record variable with five components. Then a compiler can determine the type of $R[3]$ by looking at the third entry in the list of component types in the type-object for R. But a compiler cannot know the type of $R[i]$, where i is a variable; it could be any one of the component types of the record. Thus a variable can't be used to select a record part in a statically type-checked language.

Some languages are interpreted, not compiled. These languages eliminate the restriction that the type of the result of an expression must be known ahead of time. If such a language also permitted the programmer to use the information in a record's type-object, some very useful and powerful routines could be written. For example, we could write a general debugging package with a routine that could print out and label the components of any record. Such a routine would use the type-object to find out how many components were in the record, then execute a loop that used the field name and field type information in the type-object to print out the record's value.

This kind of code is *polymorphic*, that is, the type of a function argument is tested at run time in order to know how to execute the function.[8]

14.2.5 Union Types

Two kinds of types are called *unions* because the semantics of a given storage object can vary. There are actually two kinds of unions, with very different semantic properties, known as *free* unions and *discriminated* unions.

A *free union* is a semantically unsound type. An object of a free union type has two or more possible semantic interpretations, and there is no field, either in the object itself or in its associated type-object, that defines which set of semantics is currently valid. A free union type declaration is not a way to build a compound type out of simpler types. Rather, it is a way to create objects with ambiguous semantics. A free union type-object is shown in Exhibit 14.23. Its form is like the type-object for a record. Its semantics differ from a record in that only enough space for the longest field is allocated, and all fields have an offset of 0 bytes (the offsets, therefore, do not need to be part of the type-object.) Many languages, including Pascal and C, support free union types. These will be dealt with fully in Chapter 15, Section 15.7, after the discussion of type checking.

A *discriminated union* is, in theory[9], a semantically sound type, because the alternative semantic interpretations are controlled by a field that contains a *case specifier*. The general form of a discriminated union is: Common part—key—set of variant parts. The Pascal *variant record* declaration is an example of a discriminated union type constructor. It defines a record type with three sections: an initial section with fields that are common to all variants, a tag field that is a

[8]We will cover polymorphic code in Chapter 17.

[9]We say "in theory" because the implementation in Pascal is faulty and semantically unsafe.

Exhibit 14.23. A type-object for a free union type.

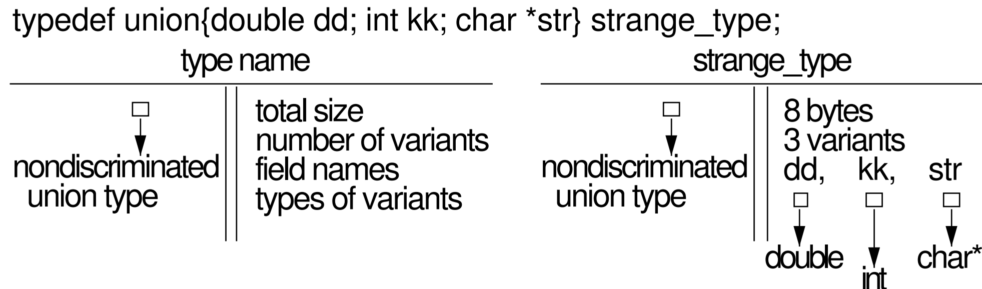


Exhibit 14.24. A type-object for a discriminated union type.

A discriminated union type has two parts. The first part is essentially an ordinary record ending with the discriminant field; the second part is a free union. This is an appropriate type-object for the type declared in Exhibit 14.25.

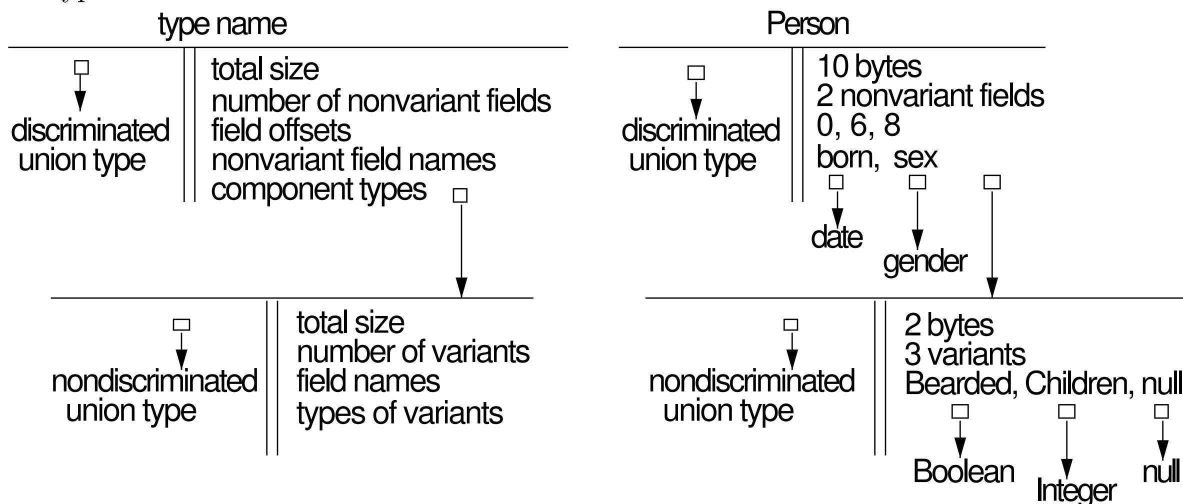


Exhibit 14.25. The discriminated variant record in Ada.

```
type Gender is (Male, Female, Unknown);
type Person(Sex:Gender:=Unknown) is
    record
        Born:  Date;
        case Sex is
            when Male => Bearded:  Boolean;
            when Female => Children:  Integer;
            when Unknown => null;
        end case;
    end record;

Sam:  Person;   -- A variable of type Person.
```

code for the variant currently stored in the variable, and a set of variant parts, each having any number of unrelated parts [Exhibit 14.24]. The clear intent of variant types is to conserve storage by permitting mutually exclusive information fields to occupy the same positions in the record. This reduces the total amount of storage needed for variant objects to the amount needed for the longest variant. We can say that the variant fields *share storage*.

If storage is plentiful, there is no need for this type of type. The same semantics can be achieved by using an ordinary record which contains all possible fields and a tag field to say which subset of the fields is currently meaningful. Storage for all fields would then exist all the time, and some subset would contain meaningless garbage.

There are some curious, perhaps faulty, aspects of Pascal's variant records. The tag field controls which set of field names is defined at any given time and should always correspond to the information actually stored in the record. To assign a value to a variant record, the variant tag must be stored first. This causes the corresponding field names to "become defined". However, nothing forces a programmer to finish the job. A value of one variant type, with its tag, could be stored in the variable. Then the tag could be changed. At this point, the tag label does not match the contents of the variable, and if the variable is used, the bits will be given an invalid interpretation! The rules of Pascal allow this to happen, and thus the variant record is a gaping hole in Pascal's type system.

Ada also supports discriminated variant records that are similar to Pascal's, but with an important difference that corrects the loophole in Pascal's semantic rules. Assignment is restricted so that a variant storage object can never contain a case specifier for one variant and a value of another.

In Exhibit 14.25, we declare a discriminated union type with one common field (`Born`), a discriminant named "`Sex`" whose default value is "`Unknown`", and three sets of variant fields, labeled

Exhibit 14.26. Legal and illegal uses of a discriminated variant record.

```
Sam := (Male, (1970,Jan,3), FALSE);     -- This is ok.
Sam.Sex := Female;                      -- Prohibited.
Sam.Children := 2;                      -- Undefined, Sam is Male.
```

by the elements of type `Gender`. We may construct a value of type `Person` by supplying a list of relevant data items, thus:

```
(Male, (1970,Jan,3), FALSE)
```

We may assign such a value to a `Person` variable. But we may not assign values to the tag field or the variant field independently, These rules forcibly maintain the consistency between the tag field and the information stored. Let `Sam` be a variable of type `Person`. Then the whole-record assignment on the first line of Exhibit 14.26 is permitted, but the partial-record assignment on the second line is not. The third assignment is not permitted because the field name `Children` is not defined for objects with the tag `Male`. These constraints in Ada close the loopholes present in Pascal.

14.3 Operations on Compound Objects

14.3.1 Creating Program Objects: Value Constructors

A program object of a primitive type is created initially by executing a function or by writing a literal in a program. For example, the result of performing an addition operation is a program object. Arithmetic expressions and most functions create program objects as their results. (Sometimes these program objects are references). These values are kept on a run-time stack.

Programming language conventions have developed to allow the writing of literal values of all the primitive types. Most languages have distinguishable syntactic forms for types real, integer, and character, and sometimes for Boolean. Additional primitive types, such as short integer, packed decimal, and byte, are sometimes supported. Sometimes the same program object may be denoted by more than one literal expression. For example, C lets the programmer write an octal, hexadecimal, or decimal literal to denote an integer value.

A language that permits a programmer to define new types should provide some way to write literals for these types. Pascal does not do this; the programmer cannot, therefore, define a constant of a user-defined type. C permits such a literal value to be written as an initializer, in a declaration, but not in any other context. This is a wholly unnecessary restriction.

A language could include a constructor function which we will call "`MAKE`" to solve this problem. The first argument of `MAKE` is a type. Following this is a variable number of literals or expressions appropriate in number and composition for program objects of the specified type [Exhibit 14.27]. `MAKE` takes the separate pure values in this list, bundles them into an object of the given type, and

Exhibit 14.27. Making a record object from its components.

Assume that the type `complex-pair` has been defined as a record containing real and imaginary components, and that imaginary numbers are made out of reals. Then a `complex-pair` would be constructed by executing the following:

```
MAKE( complex-pair, 0.0, MAKE( imag, 5.6) );
```

returns this value as its result. It is important to note that this result should be a coherent object, temporarily residing on the run-time stack. It can, therefore, be dealt with or manipulated as a whole even if it is a record or an array.

Another possible approach with the same semantics but slightly different syntax was used in Ada. Each defined type name becomes a constructor function automatically. To construct a program object of the new type, the new type name is written preceding the appropriate series of components, listed in parentheses [Exhibit 14.28].

14.3.2 The Interaction of Dereferencing, Constructors, and Selectors

Dereferencing maps a reference into a program object, or value. This reference/value relationship is complicated by the introduction of compound data-objects (arrays and records), selection functions, and constructors. A *selection function* takes, as parameters, a compound object and a part specification. It returns a reference to the specified part of the compound [Exhibit 14.29]. A *constructor* combines a set of components into a single compound object.

The selection functions in many familiar languages (e.g., FORTRAN and Pascal) take references to compound objects as parameters and return references to simple objects as results. The simple reference is then dereferenced if the context requires. Simple selection functions are combined to build the essential accessing functions for abstract data types such as stack in Exhibit 14.30.

A constructor creates a compound program object from a set of pure values on the run-time

Exhibit 14.28. Making a record object from its components in Ada.

```
type imag is new float;      -- Floats will be used to represent type imag.
type complex_pair is         -- A complex pair is a float and an imag.
    record rp:  float;
           ip:  imag
    end record;

complex_pair(0.0, imag(5.6)) -- Make an float into an imag, then
                             -- use it with another float to make
                             -- a complex pair.
```

Exhibit 14.29. Pascal **selection functions.**

In Pascal, the part names associated with record types are selection functions. We define these functions when we make a type declaration. The following declaration creates two new selection functions, named `top` (line a) and `store` (line b).

Subscript is a selection function that is predefined for all arrays, but depends on the declared array bounds. Line b defines the legal range of subscripts for the `store` component of a `stack`, and fully determines the meaning of subscript on this kind of object.

```
TYPE stack = RECORD
                 top:  integer;                        {a}
                 store:  array [1..100] of StackItem   {b}
             END;
```

Exhibit 14.30. Using selection functions in Pascal**.**

We combine the selection functions `subscript`, `top`, and `store` to build a "pop" function for a stack.

a. The parameter to the `pop` function is the address of the beginning of a stack storage object. (The keyword `VAR` indicates that a Pascal parameter is a reference.)

b. All three selection functions are composed, to arrive at the address of the element at the "top" of the stack. This address is dereferenced because it appears on the right side of an assignment. This yields a program object which is later returned by the function.

c. A different stack component, the top-of-stack index, is then selected twice. The right-hand occurrence is dereferenced and decremented. The resulting value is stored in the address given on the left (the same location), modifying the value of the compound object.

```
FUNCTION pop(VAR S:stack):StackItem;    {a}
BEGIN
    pop := S.store[ S.top ];            {b}
    S.top := S.top - 1                  {c}
END;
```

Exhibit 14.31. Value constructors in APL.

- Using a literal or a variable name causes the corresponding value to be placed on the stack.
- Assume N is bound to a numeric value; it will be automatically dereferenced and placed on the stack between the other two numbers.
- We use the concatenate operator, comma, to construct a one-dimensional numeric array, or vector, from the three simple values on the stack.

```
2, N, 5.1
```

stack. It can leave the resulting compound object on the stack or allocate a variable of the correct shape and initialize it to the given set of pure values. We call the former a *value constructor* and the latter a *reference constructor*.

Several languages give partial support to value constructors; for example, C permits them in initializing expressions and Ada supports a similar notation for compound literal expressions. APL, however, is one of the few languages that supports explicit, run-time value construction with no restrictions. Use of APL's value constructor, comma, is shown in Exhibit 14.31. The reshape function, ρ, is a combined type cast and constructor; its use is illustrated in Exhibit 14.35.

Reference construction implies dynamic allocation and the use of pointers. It is not supported at all in many common languages, but it has a basic and central importance in LISP. The LISP "`cons`" function is a reference constructor which allocates a new cell and initializes it to the two arguments on the stack.

The possible relationships among references, program objects, compound objects, and their components are captured in Exhibit 14.32. Each arrow represents a type of function, pointing from

Exhibit 14.32. Selection and dereferencing.

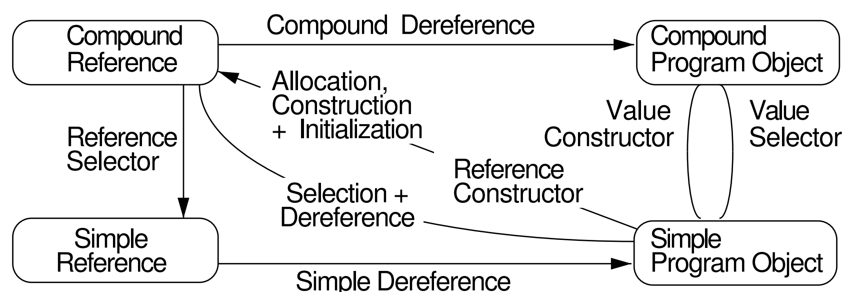
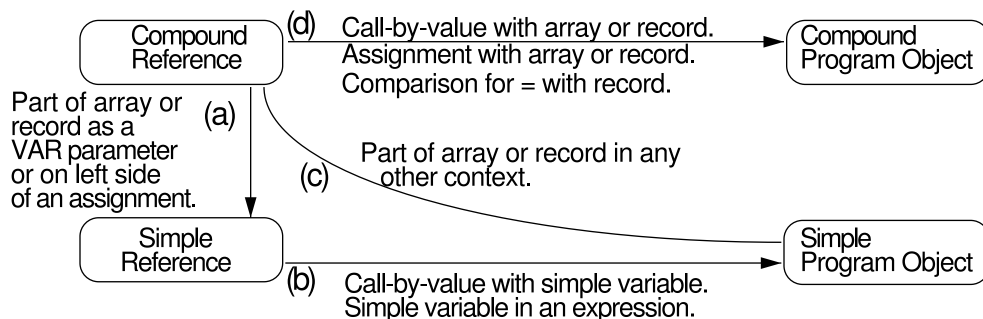


Exhibit 14.33. Pascal **selection and dereferencing.**



the type of its argument to the type of its result. Thus the arrow for "value selector" starts at "compound program object" and ends at "simple program object", and the arrow for "reference constructor" starts at "simple program object" and goes to "compound reference". No arrow in the graph leads from a collection of simple references to a compound reference. This is because programs deal with real storage. Whereas compound storage objects must occupy contiguous blocks of memory, a set of individual storage locations normally are not allocated contiguously and, therefore, cannot be combined directly into a coherent compound object.

Selection and Compound Objects in Pascal. Most languages do not support all of these kinds of selection and dereferencing functions. Pascal incorporates a limited subset of these possible function types as diagrammed in Exhibit 14.33. Moreover, Pascal only supports some of these function types in a highly restricted fashion. (The letters in the diagram key it to the following explanation of these restrictions.)

a: Reference selectors. Like most common languages, Pascal provides selection functions that return references when used on the left side of an assignment statement. This permits a single part of a compound storage object to be changed without manipulating the entire compound. Subscripting an array (or selection of one field of a record) returns a reference to the selected part.

b: Simple dereference. Dereferencing is automatic when a reference appears to the right of an ":=", or in an expression, or as an actual argument corresponding to a value parameter.

c: Selection with dereference. The result of a subscript or selection operation is a simple reference. Like any reference, this is automatically dereferenced unless it is on the left side of an assignment operator or is used as an argument to a function with a `VAR` parameter. Thus a reference to part of an object cannot be obtained and stored or manipulated by the programmer.

d: Compound dereference. Any Pascal variable, even a compound variable, may be passed as a value parameter. This causes the variable to be dereferenced. Otherwise, Pascal's support for compound dereference is severely restricted. Record variables will be dereferenced when they are compared for equality, or when the value of one record variable is copied into another. Array variables can be copied but not compared.

Pascal does not support the other three kinds of functions at all. It provides dynamic storage allocation and pointers, but not reference constructors. A series of separate operations is required to allocate a new compound storage object and initialize its fields. Nothing like the LISP `cons` is supported.

Value constructors are not supported at all in Pascal, not even those which construct compound literal values as in Ada and C. A compound value can only be constructed by storing its components, one at a time, into a compound storage object. Compound values may not even be returned from functions in the standard language.

With no value constructors, value selectors are not needed. All compound values are created, piece by piece, by storing components in a compound variable. Reference selection may then be used to decompose the value, if needed.

Selection and Compound Objects in APL. The power of APL comes from its generalized ability to manipulate compound program objects. Here we will look at the selectors and constructors in APL and note how they differ from those in most languages.

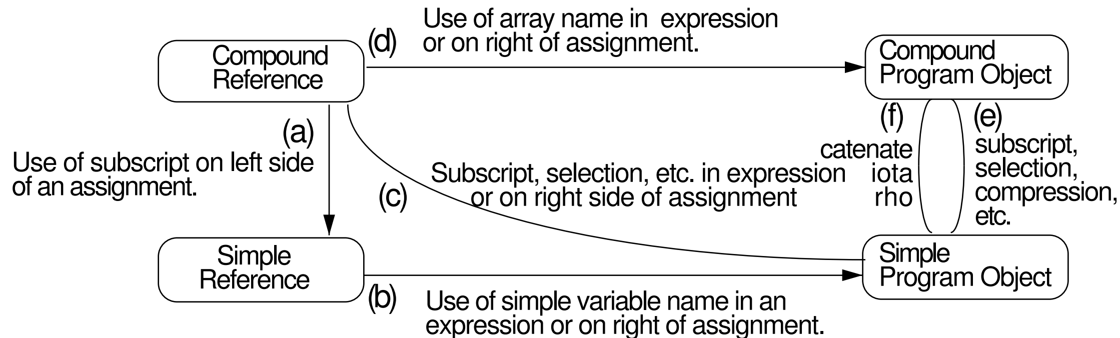
Many APL operations, including the selection functions, can result in compound pure values being left on the stack for further processing by other operators. All the selection operators can be used as value selectors, to select some portion of a compound value that was just computed. A complex expression might perform several computation steps, each time creating, on the stack, an answer of a new size or new number of dimensions. This is in stark contrast to Pascal, where every compound value must be created by a series of assignments, and the size and shape of every result must be known at compile time.

APL Selectors. In FORTRAN, C, and Pascal, selection functions operate only on variables. The question of selecting a part of a compound program object never occurs because these languages do not support either literals of compound types or functions that return objects of compound types. In contrast, APL supports compound program objects and selection functions on both variables and program objects, and APL incorporates a much more extensive and less restricted subset of the possible function types from Exhibit 14.32.

APL compounds are limited to arrays (records do not exist in APL), but the programmer can do many more things with arrays than in Pascal [Exhibit 14.34]. APL gives full support to compound dereferencing (arrow d), value selectors (arrow e), and value constructors (arrow f).

Simple dereferencing (arrow b) works similarly in Pascal and APL; anything on the right side of an assignment arrow, and any variable name in any other kind of expression, is automatically dereferenced. Both languages prohibit storing and calculating with a reference to part of an object.

Exhibit 14.34. APL **selection and dereferencing.**



Compound dereferencing (arrow d), though, is much less restricted in APL than in Pascal. Compound variables, like simple variables, can be used in most expressions. Using a compound variable name as an operand causes compound dereferencing and leaves a compound object on the stack.

APL is like Pascal where reference selection (arrow a) is concerned. Selection results in a reference in APL only when subscripts are used to the left of an assignment arrow; selection is followed by dereferencing (arrow c) for all other uses of subscripts. Unlike Pascal, though, APL lets the programmer write a subscript expression that denotes any subset of the rows and columns of a matrix. All of these locations will be selected and receive assignments or be dereferenced [Exhibit 14.36].

Unlike Pascal, subscript is not the only selection function that operates on arrays; there are several selection operators that take an array or matrix operand and an operand that specifies which part of the array is to be selected, and compute a new array of a different shape. These are listed in Exhibit 14.37.

Exhibit 14.35. Constructing matrices in APL**.**

These lines create two literal matrices and bind them to the identifiers `M1` and `M2`. Diagrams of these matrices are shown below.

```
M1 ← 3 5 ρ 11 12 13 14 15 21 22 23 24 25 31 32 33 34 35
M2 ← 4 4 ρ 0
```

M1:

11	12	13	14	15
21	22	23	24	25
31	32	33	34	35

M2:

0	0	0	0
0	0	0	0
0	0	0	0
0	0	0	0

Exhibit 14.36. Selecting a submatrix in APL.

Reference selection and two operators that perform value selection are illustrated below. These operations use the matrix values `M1` and `M2` created in Exhibit 14.35. The first line selects two characters, `"no"`, from the string `"random"` and binds the resulting value to `M3`. The second line selects six values from `M1` (rows 1 and 3, columns 2, 4, and 5) and assigns them to six selected locations in `M2` (rows 1 and 4, columns 1, 2, and 3). The result is diagrammed.

```
M3 ← 0 0 1 0 1 0 / "random"
M2[1 4; 1 2 3] ← M1[ 1 3; 2 4 5]
```

Result of value selection:	12	14	15	
	32	34	35	
Result of reference selection and assignment of a compound value to parts of M2:	12	14	15	0
	0	0	0	0
	0	0	0	0
	32	34	35	0

Exhibit 14.37. APL selection functions.

We give the most basic meaning of APL's selector functions. The "/" has additional meanings when used in other contexts.

Symbol	Name	Semantics
N ↑ V	take	Select N values from V and include them in the answer. If N is positive, take the first N values; if N is negative, take the last N.
N ↓ V	drop	Eliminate N values from V and include the rest of V in the answer. If N is positive, drop the first N values; if N is negative, drop the last N.
V1 / V2	select	The two array operands must have the same length, and the value in each position of V1 should be a positive integer, N. Then N copies of the value in the corresponding position of V2 will be selected and included in the result.
[...]	subscript	Works for matrices of any number of dimensions. An array element is selected if all of its indices (row, column, etc.) are included in the subscript list.

Exhibit 14.38. Using value constructors in APL.

We give the most basic meaning of APL's three constructors. All three symbols have additional meanings when used in other contexts.

Symbol	Name	Semantics
ι N	iota	Create an array value consisting of the numbers 1 through N.
A1 , A2	catenate	If A1 and A2 are arrays, form a new array value by concatenating the elements of A1 and A2.
D ρ A	reshape	D is an array of dimensions, and A is an array of values. Form a new matrix, whose shape is specified by D, containing the values in A. If there are too few values in A, use them cyclically until the new matrix is filled up.

APL Constructors. A value constructor takes two or more values, on the stack, and combines them into a compound value, which it leaves on the stack. APL relies everywhere on value constructors. All APL operators, when given compound arguments, can construct compound values as their results. An array literal is denoted simply by writing a series of simple literal numbers (without any delimiters or punctuation) and an array may be input by typing a similar series of numbers. In addition, there are two operators whose sole purpose is to construct array values [Exhibit 14.38], and one operator that can be used to "cast" a one-dimensional array into any multidimensional shape.

Because APL supports compound selectors, compound dereferencing, and value constructors in very general ways, compound objects can be processed as easily as simple objects. APL combines these unusual and powerful data-handling facilities with implicit iteration[10]. The result is that many algorithms can be written succinctly or as "one liners" in APL that would require several lines of code and explicit loops in most other languages.

14.4 Operations on Types

Often a systems programmer wants to design and write library functions or general implementations of common, useful algorithms. He or she is faced with a type-declaration dilemma:

- The actual size and structure of an argument must generally be known to process that structure correctly.
- But a generally applicable program should be usable for many variations of a data structure. For example, a package of stack functions should work on stacks implemented by arrays of any length or base type.

[10]Chapter 10.4

Exhibit 14.39. The lone type operation in C.

Assume that T1 is a defined type name. Line (a) declares two pointers of type T1*, or pointer-to-T1. Line (b) allocates one storage object large enough to hold a value of type T1. The result of malloc must be explicitly cast to type T1* before it stored in the pointer variable, pt1. Line (c) allocates an array of n such objects and stores the reference in pt2.

```
T1 *pt1, *pt2;                          /* a */
pt1 = (T1*)(malloc(sizeof T1));         /* b */
pt2 = (T1*)(calloc(n, sizeof T1));      /* c */
```

Binding time becomes a problem in a compiled language. Data types for all objects must be *bound* (fully specified and fixed) before code can be compiled; allocation and selection functions depend on this information. In Ada the binding time problem is solved by putting the library code in a *generic package*, which is a code schema with one or more type-parameters. To use such a package you first *instantiate* the package with particular type-arguments. This binds the types and creates fully specified code which can then be compiled normally.

But there is still a problem. Code often depends on some particular property of a type, such as the size of objects of that type or the number of elements in an array. The code in the body of a generic package needs to have access to this kind of information about the type-arguments used in the instantiation process. Even when we write library functions for an interpreted language, we need *type predicates* that can test the types of arguments and conditionally execute the appropriate code.

Type-objects provide a clean solution to these problems. They are objects and can be passed as arguments, just like data objects. A pointer to a type-object serves as a unique identifier for the type, providing an easy implementation for a type-predicate. A type-object has a body that stores specific information about the properties of objects of that type. To find the *attributes* of a type, we need the ability to access the information stored in its type-object. Thus we need selectors *for the type-objects.*

The actual selectors that are meaningful for a type-object depend, obviously, on the type of the type. The type pointer in a type-object provides a way to find out what selectors are appropriate. Finally, the language definition must list the selector functions that maybe used for each type of type.

C provides us with one example of a type operation. The sizeof operator in C may be applied to any type or any object. It accesses the type-object and returns the value of the "size" field. The sizeof operator is necessary in C to provide program portability. It is most commonly used in conjunction with the dynamic allocation functions, malloc and calloc, which require the programmer to tell the allocator how many bytes to allocate [Exhibit 14.39]. Since the size of any structure can vary from machine to machine, an operator was provided that would return the size of a type in the current implementation.

Exhibit 14.40. Some of the type operations supported by Ada.

Attribute	Meaning
For any type:	
'storage_size	Total number of storage units needed
'size	Number of bits needed for type or allocated for object
For integer types, subranges, and enumerated types:	
'first	Minimum value in the type
'last	Maximum value in the type
For floating point types:	
'digits	Number of decimals in mantissa of representation
'emax	Largest exponent value
'epsilon	Difference between two successive representable values
'large	Largest positive value
For array types:	
'first(N)	Lower bound for Nth index position
'last(N)	Upper bound for Nth index position

Ada supports generic packages and, therefore, must provide a variety of type-selectors. These are referred to as *attributes*, and they give a package access to many kinds of important information. Some, but by no means all, of these attributes are listed in Exhibit 14.40. The first one is the analog of C's `sizeof`. The last selectors listed apply to any array type and return the bounds of the Nth dimension. These type-selectors let the library programmer write code that can process any type array.

Many of these attribute selectors define the limits inherent in an implementation of the basic types. Note the similarity between the third selector, `'last`, and Pascal's implementation-dependent constant `maxint`. The type-objects for the basic types in Ada are more complex than those we diagrammed, since they must also contain the limit information. The ability to get this kind of information about an implementation can be an important ingredient in writing reliable, portable code.

Exercises

1. What is the difference between a primitive and a programmer-defined type?
2. What is a type object? What are its components?
3. The process of type-checking in Pascal is very simple and fast, even if the types being compared are complex and nested. Explain. How is this related to type objects?

4. What is a type constant? Why is it used?

5. What is a pointer type? What information is necessary to represent a pointer type? Explain two ways that this information might be used by the compiler.

6. What is a compound type?

7. What information is necessary to represent an array type?

8. What is an index type? What is its role in an array?

9. How is an effective address computed for a one-dimensional array?

10. How was the implementation of multidimensional arrays simplified in the late 1960's?

11. What is a slice of an array?

12. Compare the highly flexible arrays in APL and the much more restricted ones in Pascal. Comment on: selection functions, coherent operations, use as parameters, and returning arrays as function results.

13. What is bounds checking? What is the advantage of having it in a language? The disadvantage?

14. What is a string? A counted string? A terminated string?

15. What is the difference in representation between a C string and a Pascal string?

16. Why was the set type included in Pascal?

17. What are the similarities between an array and a record?

18. What are the differences between an array and a record? Explain why a representation of a record type is more complex than a representation of an array type.

19. Explain why the fields of a record cannot be accessed by subscript (like elements of an array) in a strongly typed language.

20. How can a record component be accessed?

21. What is a pathname?

22. What is the difference between a free union and a discriminated union?

23. How has Ada closed the loophole found in Pascal's variant record?

24. How are literal objects of primitive types created in a program?

25. How are literal objects of user-defined types created in Pascal? C?
26. What is a selection function? What does it return?
27. How does a value constructor create a compound object?
28. Why is binding time a problem in compiled languages?
29. How can we find the attributes of a type?
30. Give an example of a type operation in C and Ada.