5 Composite Data Types

From the beginning, programming languages have provided support for composite data types. Arrays are used to model vectors and matrices in mathematical models of the real world. Records are used in business data processing to model the forms and files that collect a variety of data into a single unit.

As with any type, we must describe the set of values of composite types and the operations on these values. In the case of composite types: how are they constructed from elementary values, and what operations can be used to access components of the composite value? The predefined operations on composite types are relatively limited, and most operations must be explicitly programmed from operations on the components of the composite type.

Since arrays are a specialization of records, we begin the discussion with records (called *structures* in C).

5.1 Records

A value of a record type is composed of a set of values of other types called *components* (Ada), *members* (C) or *fields* (Pascal). Within the type declaration, each field has a name and a type. The following declaration in C shows a structure with four components: one of type string, one of a user-defined enumeration type and two of integer type:

C

```
typedef enum {Black, Blue, Green, Red, White} Colors;

typedef struct {
    char        model[20];
    Colors      color;
    int         speed;
    int         fuel;
} Car_Data;
```

The equivalent declaration in Ada is:

Ada

```
type Colors is (Black, Blue, Green, Red, White);

type Car_Data is
```

```
record
    Model:      String(1..20);
    Color:      Colors:
    Speed:      Integer;
    Fuel:       Integer;
end record;
```

Once a record type has been defined, objects (variables and constants) of the type can be declared. Assignment is possible between variables of the same record type:

C

```
Car_Data c1, c2;
c1 = c2;
```

and in Ada (but not in C) it is also possible to check equality of values of the type:

Ada

```
C1, C2, C3: Car_Data;
if C1 = C2 then
    C1 = C3;
end if;
```

Since a type is a set of values, you would think that it is always possible to denote a *value* of a record type. Surprisingly, this is not possible in general; for example, C permits record values only in initializations. In Ada, however, it is possible to construct a value of a record type, called an *aggregate*, simply by supplying a value of the correct type for each field. The association of a value with a field may be by position within the record, or by name of the field:

Ada

```
if C1 = ("Peugeot", Blue, 98, 23) then ...
C1 := ("Peugeot", Red, C2.Speed, C3.Fuel);
C2 := (Model => "Peugeot", Speed => 76,
          Fuel => 46, Color => White);
```

This is extremely important because the compiler will report an error if you forget to include a value for a field; if you use individual assignments, it is easy to forget one of the fields:

Ada

```
C1.Model := "Peugeot";
                              -- Forgot C1.Color
C1.Speed := C2.Speed;
C1.Fuel := C3.Fuel;
```

Individual fields of a record can be *selected* using a period and the field name:

C

```
c1.speed = c1.fuel * x;
```

Once selected, a field of a record is a normal variable or value of the field type, and all operations appropriate to that type are applicable.

The names of record fields are local to the type definition and can be reused in other definitions:

C

```
typedef struct {
        float speed;          /* Reuse field name */
} Performance;

Performance p;
Car_Data c;
p.speed = (float) c.speed;  /* Same name, different field */
```

Single records as such are not very useful; the importance of records is apparent only when they form part of more sophisticated data structures such as arrays of records, or dynamic structures that are created with pointers (Section 8.2).

Implementation

A record value is represented by a sufficient number of memory words to include all of its fields. The layout of the record Car_Data is shown in Figure 5.1. The fields are normally laid out in order of their appearance in the record type definition.

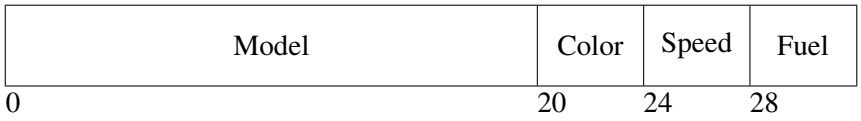

Figure 5.1: Implementation of a record

Accessing a specific field is very efficient because each field is placed at an offset from the beginning of the record which is constant and known at compile-time. Most computers have addressing modes that enable a constant to be added to an address register while the instruction is decoded. Once the address of the beginning of the record has been loaded into a register, subsequent accesses of fields do not require additional instructions:

```
load    R1,&C1      Address of record
load    R2,20(R1)   Load second field
load    R3,24(R1)   Load third field
```

Since a field may need an amount of memory that is not a multiple of the word size, the compiler may *pad* the record to ensure that each field is on a word boundary, because accesses not on a word boundary are much less efficient. On a 16-bit computer, the following type definition:

C

```
typedef struct {
        char  f1;           /* 1 byte, skip 1 byte */
        int   f2;           /* 2 bytes */
        char  f3;           /* 1 byte, skip 1 byte */
        int   f4;           /* 2 bytes */
};
```

may cause four words to be allocated to each record variable so that fields of type int are word-aligned, whereas:

C

```
typedef struct {
        int    f2;          /* 2 bytes */
        int    f4;          /* 2 bytes */
        char   f1;          /* 1 byte */
        char   f3;          /* 1 byte */
};
```

would require only three words. When using a compiler that packs fields tightly, you may be able to improve efficiency by adding dummy fields to pad out fields to word boundaries. See also Section 5.9 for ways of explicitly mapping fields. In any case, *never* assume a specific record layout, as that will make your program totally non-portable.

5.2 Arrays

An array is a record all of whose fields are of the same type. Furthermore, these fields (called *elements* or *components*) are not named by identifiers but by position within the array. The power of the array data type comes from the ability to efficiently access an element by using an *index*. Since all the elements are of the same type, it is possible to compute the location of a specific element by multiplying the index by the size of an element. Using indices it is easy to search through an array for a specific element, or to sort or otherwise rearrange the elements.

The index type in Ada may be any discrete type, that is any type on which "counting" is possible; these are integer types and enumeration types (including Character and Boolean):

Ada

```
type Heat is (Off, Low, Medium, High);
type Temperatures is array(Heat) of Float;
Temp: Temperatures;
```

C restricts the index type to integers; you specify *how many* components you want:

C

```
#define Max 4
float temp[Max];
```

and the indices are implicitly from 0 to one less than the number of components, in this case from 0 to 3. C++ allows any constant expression to be used for the array count, which improves readability:

C++

```
const int last = 3;
float temp[last+1];
```

The array component may be of any type:

C

```
typedef struct {...} Car_Data;
Car_Data database[100];
```

In Ada (but not in C) the operations of assignment and equality checking may be done on arrays:

Ada

```
type A_Type is array(0..9) of Integer;
A, B, C: A_Type;

if A = B then A := C; end if;
```

As with records, array values called aggregates exist in Ada with an extensive range of syntactic options:

Ada

```
A := (1,2,3,4,5,6,7,8,9,10);
A := (0..4 => 1, 5..9 => 2);        -- Half one's, half two's
A := (others => 0);                 -- All zeros
```

In C, array aggregates are limited to initial values.

The most important array operation is *indexing*, which selects an element of an array. The index value, which may be an arbitrary expression of the array index type, is written following the array name:

Ada

```
type Char_Array is array(Character range 'a' .. 'z') of Boolean;
A: Char_Array := (others => False);
C: Character := 'z';

A(C) := A('a') and A('b');
```

Another way of understanding arrays is to consider them to be a function from the index type to the element type. Ada (like Fortran but unlike Pascal and C) encourages this view by using the same syntax for function calls and for array indexing. That is, without looking at the declaration you cannot know whether A(I) is a call to a function or an indexing operation on an array. The advantage to this common syntax is that a data structure may be initially implemented as an array, and later on if a more complex data structure is needed, the array can be replaced by a function without modifying the array accesses. The use of brackets rather than parentheses in Pascal and C is mostly for the convenience of the compiler.

Records and arrays may be nested arbitrarily in order to construct complex data structures. To access an elementary component of such a structure, field selection and element indexing must be done one after the other until the component is reached:

C

```
typedef int A[10];          /* Array type */
typedef struct {            /* Record type */
   A      a;                /* Array within record */
   char   b;
   }  Rec;

Rec    r[10];               /* Array of records of arrays of int */
int    i,j,k;

k = r[i+1].a[j-1];          /* Index, then select, then index */
                            /* Final result is an integer value */
```

Note that partial selecting and indexing in a complex data structure yields values that are themselves arrays or records:

C

```
r                           Array of record of array of integer
r[i]                        Record of array of integer
r[i].a                      Array of integer
r[i].a[j]                   Integer
```

and these values can be used in assignment statements, etc.

5.3 Reference semantics in Java

Perhaps the worst aspect of C (and C++) is the unrestricted and excessive use of pointers. Not only are pointer manipulations difficult to understand, they are extremely error-prone as described in Chapter 8. The situation in Ada is much better because strong type-checking and access levels ensure that pointers cannot be used to break the type system, but data structures still must be built using pointers.

Java (like Eiffel and Smalltalk) uses *reference semantics* instead of *value semantics*.[1] Whenever a variable of non-primitive type[2] is declared, you do not get a block of memory allocated; instead, an implicit pointer is allocated. A second step is needed to actually allocate memory for the variable. We now demonstrate how reference semantics works in Java.

Arrays

If you declare an array in C, you are allocated the memory you requested (Figure 5.2(a)):

[1] Do not confuse these terms with the similar terms used to discuss parameter passing in Section 7.2.

[2] Java's primitive types are: float, double, int, long, short, byte, char, boolean. Do not confuse this term with *primitive operations* in Ada (Section 14.5).

C

```
int a_c[10];
```

while in Java, you only get a pointer that *can be* used to store an array (Figure 5.2(b)):

Java

```
int[] a_java;
```

Allocating the array takes an additional instruction (Figure 5.2(c)):

Java

```
a_java = new int[10];
```

though it is possible to combine the declaration with the allocation:

Java

```
int[] a_java = new int[10];
```

If you compare Figure 5.2 with Figure 8.4, you will see that Java arrays are similar to structures defined in C++ as `int *a` rather than as `int a[]`. The difference is that the pointer is implicit so you do not need to concern yourself with pointer manipulation or memory allocation. In fact, in the

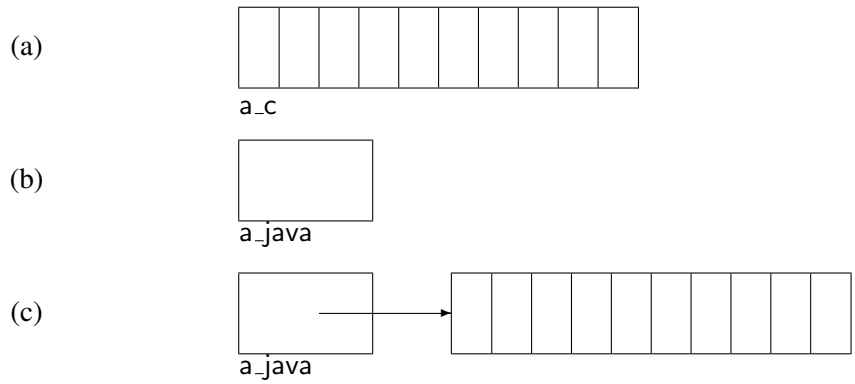

Figure 5.2: Value versus reference semantics

case of an array, the variable will be a dope vector (Figure 5.5), enabling bounds checking to be done when accessing the array.

Note that the Java syntax is rather easier to read than the C++ syntax: `a_java` is of *type* `int[]`, that is "integer array"; in C++, the component type `int` and the array indication `[10]` appear on opposite sides of the variable name.[3]

Under reference semantics, dereferencing of the pointer is implicit, so that once the array is created you access the array as usual:

[3]To ease the transition to Java, the language also accepts the C++ syntax.

Java

```
for (i = 1; i < 10; i++)
    a_java[i] = i;
```

Of course, this indirect access can be much less efficient than direct access unless optimized by the compiler.

Note that allocating an object and assigning it to a variable can be done in any statement, resulting in the following possibility:

Java

```
int[] a_java = new int[10];
...
a_java = new int[20];
```

The variable a_java, which pointed to a ten-element array, now points to a twenty-element array, and the original array is now garbage (Figure 8.9). The Java model specifies that a garbage collector must exist within the JVM.

5.4 Arrays and type checking

Probably the most common cause of difficult bugs is indexing that exceeds array limits:

C

```
int a[10];
for (i = 0; i ¡= 10; i++)
    a[i] = 2*i;
```

The loop will also be executed for i=10 but the last element of the array is a[9].

The reason that this type of bug is so common is that index expressions can be arbitrary expressions, but valid indices fall within the range defined in the array declaration. The slightest mistake can cause the index to receive a value that is outside this narrow range. The reason that the resulting bug is so disastrous is that an assignment to a[i] if i is out of range causes some *unrelated* memory location to be modified, perhaps even in the operating system area. Even if hardware protection restricts the modification to data within your own program, the bug will be difficult to find because the *symptoms* will be in the wrong place, namely in unrelated statements that use the modified memory rather than in the statement that caused the modification.

To demonstrate: if a numerical mistake causes a variable speed to receive the value 20 instead of 30:

C

```
int x = 10, y = 50;
speed = (x + y) / 3;        /* Compute average ! */
```

the symptom is the incorrect value of speed, and the cause (division by 3 instead of 2) is in a statement that calculates speed. The symptom is directly related to the mistake, and by using breakpoints or watchpoints you can quickly locate the problem. In the following example:

C

```
int a[10];
int speed;

for (i = 0; i ¡= 10; i++)
    a[i] = 2*i;
```

speed is a victim of the fact that it was arbitrarily declared just after a and thus modified in a totally unrelated statement. You can trace the computation of speed for days without finding the bug.

The solution to these problems is to check indexing operation on arrays to ensure that the bounds are respected. Any attempt to exceed the array bounds is considered to be a violation of type checking. The first language to propose index checking was Pascal:

Pascal

```
type A_Type = array[0..9] of Integer;
A: A_Type;
A[10] := 20;                (* Error *)
```

With array type checking, the bug is caught *immediately* where it occurs and not after it has "smeared" some unrelated memory location; an entire class of difficult bugs is removed from programs. More precisely: such errors become compile-time errors instead of run-time errors.

Of course you don't get something for nothing, and there are two problems with array type checking. The first is that there is a run-time cost to the checks which we will discuss in a later section. The second problem has to do with an inconsistency between the way we work with arrays and the way that type checking works. Consider the following example:

Pascal

```
type A_Type = array[0..9] of Real;   (* Array types *)
type B_Type = array[0..8] of Real;
A: A_Type;                           (* Array variables *)
B: B_Type;

procedure Sort(var P: A_Type);       (* Array parameter *)

sort(A);                             (* OK *)
sort(B);                             (* Error! *)
```

The two array type declarations define distinct types. The type of the actual parameter to a procedure must match the type of the formal parameter, so it seems that two different Sort procedures are needed, one for each type. This is inconsistent with our intuitive concept of arrays and array operations, because the detailed programming of procedures like Sort does not depend on the number of elements there are in an array; the array bounds should simply be additional parameters. Note that this problem does not occur in Fortran or C simply because those languages do not have array parameters! They simply pass the address of the start of the array and it is the responsibility of the programmer to correctly define and use the array bounds.

The Ada language has an elegant solution to this problem.[4] The type of an array in Ada is determined solely by its *signature*, that is the type of the index and the type of the element. Such a type is called an *unconstrained array type*. To actually declare an array, an *index constraint* must be appended to the type:[5]

Ada

```
type A_Type is array(Integer range <>) of Float;
                              -- Unconstrained array type declaration

A: A_Type(0..9);              -- Array with index constraint
B: A_Type(0..8);              -- Array with index constraint
```

The signature of A_Type is a one-dimensional array with Integer indices and Float components; the index bounds are *not* part of the signature.

As in Pascal, indexing operations are fully checked:

Ada

```
A(9) := 20.5;                 -- OK, index range is 0..9
B(9) := 20.5;                 -- Error, index range is 0..8
```

The importance of unconstrained arrays is apparent when we look at procedure parameters. Since the type of (unconstrained) array parameters is determined by the signature only, we can call a procedure with any actual parameter of that type regardless of its index constraint:

Ada

```
procedure Sort(P: in out A_Type);
                              -- Unconstrained array type parameter

Sort(A);                      -- Type of A is A_Type
Sort(B);                      -- Type of B is also A_Type
```

The question now arises: how can the Sort procedure access the array bounds? In Pascal, the bounds were part of the type and so were also known within the procedure. In Ada, the constraint of the actual array parameter is automatically passed to the procedure at run-time, and can be accessed using functions called *attributes*. Given any array A:

- A'First is the index of the first element in A.
- A'Last is the index of the last element in A.
- A'Length is the number of elements in A.
- A'Range is equivalent to A'First..A'Last.

For example:

[4] The Pascal standard defines *conformant array parameters* whose bounds are implicitly passed to a procedure, but we will focus on the Ada solution which introduces a new generalized concept not limited to array parameters.

[5] The symbol "<>" is read "box".

Ada

```
procedure Sort(P: in out A_Type) is
begin
   for I in P'Range loop
      for J in I+1 .. P'Last loop
   ...
end Sort;
```

The use of array attributes enables the programmer to write software that is extremely robust to modifications: any change in the array bounds is automatically reflected in the attributes.

To summarize: array type checking is a powerful tool for improving the reliability of programs; however, the definition of the array bounds should not be part of the static definition of a type.

5.5 * Array subtypes in Ada

The subtypes that we discussed in Section 4.5 were defined by appending a *range constraint* to a discrete type (integer or enumeration). Similarly, an array subtype may be declared by appending an *index constraint* to an unconstrained array type:

```
type A_Type is array(Integer range <>) of Float;
subtype Line is A_Type(1..80);
L, L1, L2: Line;
```

A value of this named subtype may be used as an actual parameter corresponding to a formal parameter of the underlying unconstrained type:

```
Sort(L);
```

In any case, the unconstrained formal parameter of Sort is dynamically constrained by the actual parameter each time the procedure is called.

The discussion of subtypes in Section 4.5 is applicable here. Arrays of different subtypes of the same type can be assigned to each other (provided that they have the same number of elements), but arrays of different types cannot be assigned to each other without an explicit type conversion. The definition of a named subtype is just a convenience.

Ada has powerful constructs called *slices* and *sliding*, which enable partial arrays to be assigned:

```
L1(10..15) := L2(20..25);
```

which assigns a slice of one array to another, sliding the indices until they match. The type signatures are checked at compile-time, while the constraints are checked at run-time and can be dynamic:

```
L1(I..J) := L2(I*K..M+2);
```

The difficulties with array type definitions in Pascal led the designers of Ada to generalize the solution for arrays to the elegant concept of subtypes: separate the static type specification from the constraint which can be dynamic.

5.6 String type

Basically strings are simply arrays of characters, but additional language support is required for programming convenience. The first requirement is to have a special syntax for strings, otherwise working with arrays of characters would be too tedious. Both of the following declarations are valid but of course the first form is much more convenient:

C

```
char s[] = "Hello world";
char s[] = {'H','e','l','l','o',' ','w','o','r','l','d', '\0'};
```

Next we have to find some way of dealing with string length. The above example already shows that the compiler can determine the size of a string without having the programmer specify it explicitly. C uses a convention on the representation of strings, namely that the first zero byte encountered terminates the string. String processing in C typically contains a while-loop of the form:

C

```
while (s[i++] != '\0') ...
```

The main drawback to this method is that if the terminating zero is somehow missing, memory can be smeared just like any statement that causes array bounds to be exceeded:[6]

C

```
char s[11] = "Hello world";/* No room for zero byte */
char t[11];
strcpy(t, s);                    /* Copy s to t. How long is s? */
```

Other disadvantages of this method are:

- String operations require dynamic allocation and deallocation of memory which is relatively inefficient.
- Calls to string library functions require that the string length be recalculated.
- The zero byte cannot be part of the string.

An alternative solution used by some dialects of Pascal is to include an explicit length byte as the implicit zero'th character of a string whose maximum length is specified when it is declared:

[6] This particular error would be caught in C++ which does not ignore extra members in an initialization.

Pascal

```
S: String[10];
S := 'Hello world';          (* Needs 11 bytes *)
writeln(S);
S := 'Hello';
writeln(S);
```

The first output of the program will be "Hello worl" because the string will be truncated to fit the declared length. The second output will be "Hello" because the writeln statement takes the implicit length into account. Unfortunately, this solution is also flawed because it is possible to directly access the hidden length byte and smear memory:

Pascal

```
s[0] := 15;
```

In Ada there is a predefined unconstrained array type called String whose definition is:[7]

Ada

```
type String is array(Positive range <>) of Character;
```

Each string must be of fixed length and declared with a constraint:

Ada

```
S: String(1..80);
```

Unlike C where all string processing is done using library procedures like strcpy, Ada includes operators on strings such as concatenation "&", equality and relational operators like "¡". Type checking is strictly enforced, so a certain amount of gymnastics with attributes is needed to make everything work out:

Ada

```
S1: constant String := "Hello";
S2: constant String := "world";
T: String(1 .. S1'Length + 1 + S2'Length) := S1 & ' ' & S2;
Put(T);                          -- Will print Hello world
```

The exact length of T must be computed before the assignment is done! Fortunately, Ada supports array attributes and a construct for creating subarrays (called slices) that make it possible to do such computations in a portable manner.

Ada 83 provides the framework for defining strings whose length is not fixed, but not the necessary library subprograms for string processing. To improve portability, Ada 95 defines standard libraries for all three categories of strings: fixed, varying (as in Pascal) and dynamic (as in C).

[7] Using predefined subtype Positive is Integer range 1..Integer'Last.

5.7 Multi-dimensional arrays

Multi-dimensional matrices are extensively used in mathematical models of the physical world, and multi-dimensional arrays have appeared in programming languages since Fortran. There are actually two ways of defining multi-dimensional arrays: directly and as complex structures. We will limit the discussion to two-dimensional arrays; the generalization to higher dimensions is immediate.

A two-dimensional array can be directly defined in Ada by giving two index types separated by a comma:

Ada

```
type Two is
   array(Character range <>, Integer range <>) of Integer;
T: Two('A'..'Z', 1..10);
I: Integer;
C: Character;

T('X', I*3) := T(C, 6);
```

As the example shows, the two dimensions need not be of the same type. Selecting an array element is done by giving both indices.

The second method of defining a two-dimensional array is to define a type that is an array of arrays:

Ada

```
type I_Array is array(1..10) of Integer;
type Array_of_Array is array (Character range <>) of I_Array;
T: Array_of_Array('A'..'Z');
I: Integer;
C: Character;

T('X')(I*3) := T(C)(6);
```

The advantage of this method is that by using one index operation, the elements of the second dimension (which are themselves arrays) can be accessed:

Ada

```
T('X') := T('Y');          -- Assign 10-element arrays
```

The disadvantage is that the elements of the second dimension must be constrained before they can be used to define the first dimension.

In C, only the second method is available and of course only for integer indices:

C

```
int a[10][20];
a[1] = a[2];               /* Assign 20-element arrays */
```

Pascal does not distinguish between a two-dimensional array and an array of arrays; since the bounds are considered to be part of the array type this causes no difficulties.

5.8 Array implementation

Arrays are represented by placing the array elements in sequence in memory. Given an array A the address of A(I) is (Figure 5.3):

$$addr\,(A) + size\,(element) * (i - A'First)$$

For example: the address of A(4) is $20 + 4 * (4 - 1) = 32$.

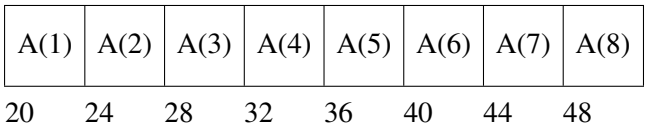

Figure 5.3: Implementation of array indexing

The generated machine code is:

```
load    R1,I          Get index
sub     R1,A'First    Subtract lower bound
multi   R1,size       Multiply by size -> offset
add     R1,&A         Add array addr. -> element addr.
load    R2,(R1)       Load contents
```

It may surprise you to learn how many instructions are needed for each array access!

There are many optimizations that can be done to improve this code. First note that if A'First is zero, we save the subtraction of the index of the first element; this explains why the designers of C specified that indices always start at zero. Even if A'First is not zero but is known at compile time, the address calculation can be rearranged as follows:

$$(addr\,(A) - size\,(element) * A'First) + (size\,(element) * i)$$

The first expression in parentheses can be computed at compile-time saving the run-time subtraction. This expression will be known at compile-time in a direct use of an array:

Ada

```
A: A_Type(1..10);
A(I) := A(J);
```

but not if the array is a parameter:

Ada

```
procedure Sort(A: A_Type) is
begin
   ...
   A(A'First+I) := A(J);
   ...
end Sort;
```

The main obstacle to efficient array operations is the multiplication by the size of the array element. Fortunately, most arrays are of simple data types like characters or integers whose size is a power of two. In this case, the costly multiplication operation can be replaced by an efficient shift, since shift left by n is equivalent to multiplication by 2^n. If you have an array of records, you can squeeze out more efficiency (at the cost of extra memory) by padding the records to a power of two. Note that the portability of the program is not affected, but the *improvement* in efficiency is not portable: another compiler might lay out the record in a different manner.

C programmers can sometimes improve the efficiency of a program with arrays by explicitly coding access to array elements using pointers rather than indices. Given definitions such as:

C

```
typedef struct {
    ...
    int field;
} Rec;
Rec a[100];
```

it may be more efficient (depending on the quality of the compiler's optimizer) to access the array using a pointer:

C

```
Rec* ptr;

for (ptr = &a; ptr < &a+100*sizeof(Rec); ptr += sizeof(Rec))
    ... ptr->field ...;
```

than using an indexing operation:

C

```
for (i = 0; i ¡ 100; i++)
    ... a[i].field ...;
```

However, this style of coding is extremely error-prone, as well as hard to read, and should be avoided except when absolutely necessary.

A possible way of copying strings in C is to use:

C

```
while (*s1++ = *s2++)
    ;
```

where there is a null statement before the semicolon. If the computer has block-copy instructions which move the contents of a block of memory cells to another address, a language like Ada which allows array assignment would be more efficient. In general, C programmers should use library functions which are likely to be implemented more efficiently than the naive implementation shown above.

Multi-dimensional arrays can be very inefficient because each additional dimension requires an extra multiplication to compute the index. When programming with multi-dimensional arrays, you must also be aware of the layout of the data. Except for Fortran, all languages store a two-dimensional array as a sequence of rows. The layout of:

Ada

```
type T is array(1..3, 1..5) of Integer;
```

is shown in Figure 5.4. This layout is consistent with the similarity between a two-dimensional

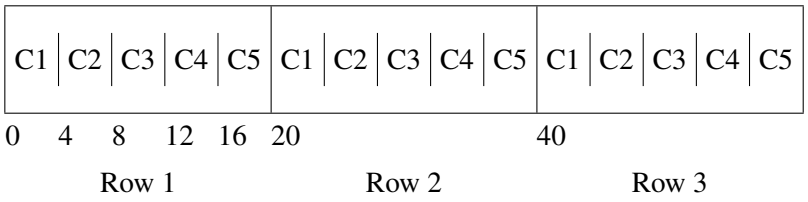

Figure 5.4: Implementation of a multi-dimensional array

array and an array of arrays. If you wish to compute a value for every element of a two-dimensional array, make sure that the last index is in the inner loop:

C

```
int matrix[100][200];

for (i = 0; i ¡ 100; i++)
    for (j = 0; j ¡ 200; j++)
        m[i][j] = . . . ;
```

The reason is that operating systems that use paging are vastly more efficient if successive memory accesses are close together.

If you wish to squeeze the best performance out of a program in C, you can ignore the two-dimensional structure of the array and pretend that the array is one-dimensional:

C

```
for (i = 0; i ¡ 100*200; i++)
        m[0][i] = . . . ;
```

Needless to say, this is not recommended and should be carefully documented if used.

Array type checking requires that the index be checked against the bounds before each array access. The overhead to this check is high: two comparisons and jumps. Compilers for languages like Ada have to invest significant effort to optimize the instructions for array processing. The main technique is to utilize available information. In the following example:

Ada

```
for I in A'Range loop
    if A(I) = Key then . . .
```

the index I will take on exactly the permitted values of the array so no checking is required. In general, the optimizer will perform best if all variables are declared with the tightest possible constraints.

When arrays are passed as parameters in a type-checking language:

Ada

```
type A_Type is array(Integer range <>) of Integer;
procedure Sort(A: A_Type) is ...
```

the bounds must also be implicitly passed in a data structure called a *dope vector* (Figure 5.5). The dope vector contains the upper and lower bounds, the size of an element and the address of

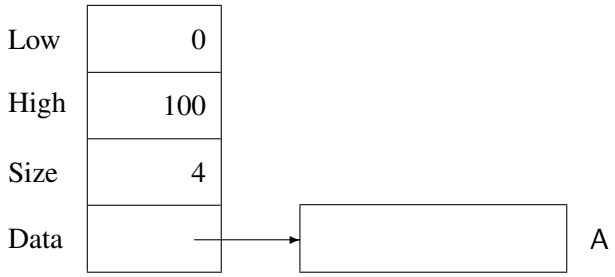

Figure 5.5: Dope vector

the start of the array. As we have seen, this is exactly the information needed to compute an array indexing operation.

5.9 * Representation specification

This book repeatedly stresses the importance of viewing a program as an abstract model of the real world. Nevertheless, many programs such as operating systems, communications packages and embedded software require the ability to manipulate data at the physical level of their representation in memory.

Bitwise computation

C contains Boolean operators which compute on individual bits of *integer* values: "&" (and), "—" (or), "^" (xor), "~" (not).

The Boolean operators in Ada: and, or, xor, not can also be applied to arrays of Boolean:

Ada

```
type Bool_Array is array(0..31) of Boolean;
B1: Bool_Array := (0..15 => True, 16..31 => False);
B2: Bool_Array := (0..15 => False, 16..31 => True);

B1 := B1 or B2;
```

However, just declaring a Boolean array does not ensure that it is represented as a bit string; in fact, a Boolean value is usually represented as a full integer. The addition of the directive:

Ada

```
pragma Pack(Bool_Array);
```

requests the compiler to pack the values of the array as densely as possible. Since only one bit is needed for a Boolean value, the 32 elements of the array can be stored in one 32-bit word. While this does provide the required functionality, it does lack some of the flexibility of C, in particular the ability to use octal or hexadecimal constants like 0xf00f0ff0 in Boolean computations. Ada does provide a notation for such constants, but they are integer values and not arrays of Boolean and so cannot be used in bitwise computation.

These problems are solved in Ada 95: modular types (Section 4.1) can be used for bitwise computation:

Ada

```
type Unsigned_Byte is mod 256;
U1, U2: Unsigned_Byte;

U1 := U1 and U2;
```

Subword fields

Hardware registers are usually composed of several subword fields. Classically, the way to access such fields is to use shift and mask; the statement:

C

```
field = (i >> 4) & 0x7;
```

extracts the three-bit field located four bits from the right of the word i. This programming style is dangerous because it is very easy to make mistakes in specifying the shift counts and masks. In addition, the slightest change in the layout of a word can require massive modification of the program.

An elegant solution to this problem was first introduced by Pascal: use ordinary records, but pack the fields into a single word.[8] An ordinary field access Rec.Field is automatically translated by the compiler into the correct shift and mask.

In Pascal, the mapping was implicit; other languages have a notation for explicit mapping. C allows bit-field specifiers on a structure field (provided that the fields are of integer type):

C

```
typedef struct {
    int     : 3;      /* Padding */
    int f1  : 1;
```

[8] The motivation for this came from the CDC 6000 series computers that were used in the first Pascal implementation. These computers, intended for numerical calculations, used a 60 bit word and had no instructions for subword access except shift and mask.

```
    int f2   : 2;
    int      : 3;      /* Padding */
    int f3   : 2;
    int      : 4;      /* Padding */
    int f4   : 1;
} reg;
```

and then ordinary assignment statements can be used, even though the fields are part of a word (Figure 5.6) and the assignment is implemented by the compiler using shift and mask:

C

```
reg r;
int i;

i = r.f2;
r.f3 = i;
```

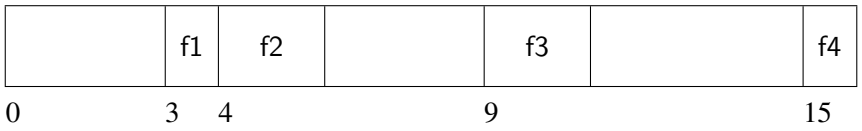

Figure 5.6: Subword record fields

Ada insists that type declarations be abstract; *representation specifications* use a different notation and are written separately from the type declaration. Given the record type declaration:

Ada

```
type Heat is (Off, Low, Medium, High);

type Reg is
    record
        F1: Boolean;
        F2: Heat;
        F3: Heat;
        F4: Boolean;
    end record;
```

a record specification can be appended:

Ada

```
for Reg use
    record
        F1 at 0 range 3..3;
        F2 at 0 range 4..5;
        F3 at 1 range 1..2;
        F4 at 1 range 7..7;
    end record;
```

The at clause specifies the byte within the record and range specifies the bit range assigned to the field, where we know that one bit is sufficient for a Boolean value and two bits are sufficient for a Heat value. Note that padding is not necessary because exact positions are specified.

Once the bit specifiers in C and representation specifications in Ada are correctly programmed, all subsequent accesses are ensured to be free of bugs.

Big endians and little endians

Conventionally, memory addresses start at zero and increase. Unfortunately, computer architectures differ in the way multibyte values are stored in memory. Let us assume that each byte can be separately addressed and that each word takes four bytes. How is the integer 0x04030201 to be stored? The word can be stored so that the most significant byte is first, an order called *big endian*, or so that the least significant byte has the lowest address, called *little endian*. Figure 5.7 shows the numbering of the bytes for the two options.

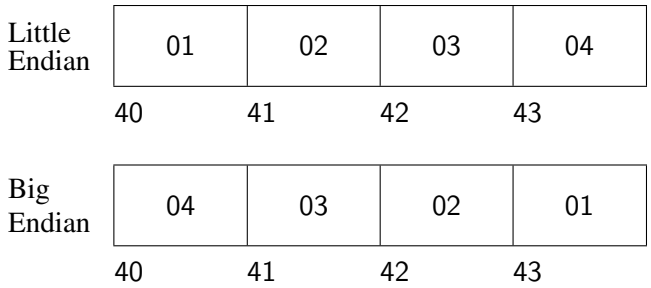

Figure 5.7: Big endian and little endian

This aspect of the computer architecture is handled by the compiler and is totally transparent to the programmer who writes abstractly using records. However, when representation specifications are used, the differences between the two conventions can cause a program to be non-portable. In Ada 95, the bit ordering of a word can be specified by the programmer, so a program using representation specifications can be ported by changing only one statement.

Derived types and representation specifications in Ada

A derived type in Ada (Section 4.6) is defined as a new type whose values and operators are a copy of those of the parent type. A derived type may have a representation that is different from the parent type. For example, once an ordinary type Unpacked_Register is defined:

Ada

```
type Unpacked_Register is
   record
      ...
   end record;
```

a new type may be derived and a representation specification associated with the derived type:

Ada

```
type Packed_Register is new Unpacked_Register;

for Packed_Register use
    record
        ...
    end record;
```

Type conversion (which is allowed between any types that are derived from each other) causes a change of representation, namely, packing and unpacking the subword fields of the record to ordinary variables:

Ada

```
U: Unpacked_Register;
P: Packed_Register;

U := Unpacked_Register(P);
P := Packed_Register(U);
```

This facility can contribute to the reliability of programs, because once the representation specification is correctly written, the rest of the program can be completely abstract.

5.10 Exercises

1. Does your compiler pack record fields or align them on word boundaries?

2. Does your computer have a block-copy instruction, and does your compiler use it for array and record assignments?

3. Pascal contains the construct with which opens the scope of a record so that the field names can be used directly:

   Pascal

   ```
   type Rec =
       record
           Field1: Integer;
           Field2: Integer;
       end;
   R: Rec;

   with R do Field1 := Field2;(* OK, direct visibility *)
   ```

   What are the advantages and disadvantages of the construct? Study the Ada renames construct and show how some of the same functionality can be obtained. Compare the two constructs.

4. Explain the error message that you get if you try to assign one array to another in C:

C

```
int a1[10], a2[10];
a1 = a2;
```

5. Write sort procedures in Ada and C, and compare them. Make sure that you use attributes in the Ada procedure so that the procedure will work on arrays with arbitrary indices.

6. Which optimizations does your compiler do on array indexing operations?

7. Icon has associative arrays called tables, where a string can be used as an array index:

   ```
   count["begin"] = 8;
   ```

   Implement associative arrays in Ada or C.

8. Are the following two types the same?

   Ada

   ```
   type Array_Type_1 is array(1..100) of Float;
   type Array_Type_2 is array(1..100) of Float;
   ```

   Ada and C++ use *name equivalence*: every type declaration declares a new type, so two types are declared. Under *structural equivalence* (used in Algol 68), type declarations that look alike define the same type. What are the advantages and disadvantages of the two approaches?

9. An array object of anonymous type (without a named type) can be defined in Ada. In the following example, is the assignment legal? Why?

   Ada

   ```
   A1, A2: array(1..10) of Integer;
   A1 := A2;
   ```

10. Compare the string processing capabilities of Ada 95, C++ and Java.