Array Abstract Data Type

Table of Contents

1 What this module is about

This module contains a description of the array data type and some discussion as to how we allocate memory space to arrays.

The value of defining arrays as an abstract data type is primarily for systems programmers, who work behind the scenes and bring you all the wonderful software that comes with an operating system, such as compilers, linkers, file managers, text editors, etc. For most of us mortal people, we simply use arrays in our programming languages without thought of the more abstract nature of arrays.

We do not go into detailed program implementations.

2 Introduction

The array data type is the simplest structured data type. It is such a useful data type because it gives you, as a programmer, the ability to assign a single name to a homogeneous collection of instances of one abstract data type and provide integer names for the individual elements of the collection. Figure 1 shows an array[1..7] of COLOUR. You, the programmer, can then compute the name of a specific element and avoid the need for explicitly declaring a unique name for each array element. Consider how difficult it would be to program if you had to declare explicitly a separate name to each array element and use such names in your programs.

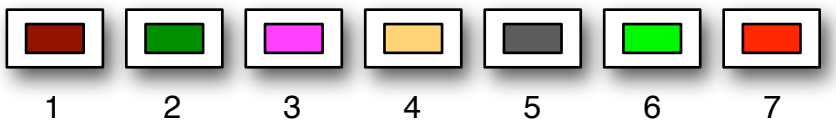


Figure 1: A seven element array of COLOUR.

Because the names of array elements are computed, the computer requires a means of finding – at execution time – the physical storage location corresponding to an array element from the

index, or name, of the array element. As a consequence, not only do we require space for the array elements themselves, but we also require space for an **array descriptor** which is used at execution time to map array indices to memory addresses. The array descriptor contains the size of the array elements, the number of dimensions and bounds of each dimension.

> NOTE: In fact, for all data types beyond the primitive data types we require data descriptors. One could even define the primitive data types as those data types which do not require data descriptors. If you reflect on the declaration section of any programming language, you will find that the non primitive data types have complex definitional parameters. These definitional parameters correspond to the information required in the data descriptors.

To make it easy to compute the memory address of an array element and to make memory management simple, we have the restriction that each element of the array must be of the same type, so that each array element can be allocated a pre−defined and equal amount of memory space.

Multi−dimensional arrays are defined as multiple levels of one dimensional arrays. For example, the following definition of the three−dimensional array of colours

```
array[1 .. 5 ; -2 .. +6 ; -10 .. -1] of COLOUR
```

actually means

```
array[1 .. 5] of
   array[-2 .. +6] of
      array[-10 .. -1] of COLOUR
```

3 Data type objects

The entire array is a single entity, represented by its name. For example in the following, the name chess_board refers to the entire array.

```
chess_board : array[1 .. 8 , 1 .. 8] of SQUARE
```

3.1 Array elements

Each array element is a separate object. Array elements are represented by the array name plus a set of subscripts. Each subscript is in turn represented by an arithmetic expression. For example, we could have a two−dimensional array called chess_board and we could refer to one of its elements – one of the squares on the chess board – using the following notation.

```
chess_board[row-1, column+2]
```

Another example is the the postion of an airplane over time. This requires three space coordinates, latitude, longitude and height, and one time coordinate.

```
plane_position[its_latitude, its_longtidue, its_height, universal_time]
```

3.2 Sub -arrays

If a subscript expression represents a range of subscript values then we consider the corresponding sub−array as individual object. The structure, or shape, of the sub−array depends upon what ranges we permit subscript ranges to have. The following shows some examples.

1. `chess_board[*, 3]` – refers to the sub−array of all squares in column 3.
2. `chess_board[2 .. 6 , 3]` – refers to the sub−array of squares in rows 2, 3, 4, 5 and 6 and in column 3.

3. chess_board[2 .. 6 , 3 .. 5] – refers to the sub–array of squares in rows 2, 3, 4, 5 and 6, and in columns 3, 4 and 5.

3.3 Array descriptor

An array also has associated with it an array descriptor. The array descriptor consists of the following objects – see Figure 2.

1. The address, A_0 , of the array element which represents the location of the array element with all subscripts equal to zero. This may be a physically non–existent location in main memory because arrays may be defined with arbitrary upper and lower bounds. Users of arrays do not see this object. We include it because it is required for the complete understanding of how array element addresses are computed.
2. The number of dimensions, dim_count, of the array.
3. A descriptor triple for each dimension of the array.

   LB_i – the lower bound of dimension i
   UB_i – the upper bound of dimension i
   SZ_i – the size (space requirement) for dimension i

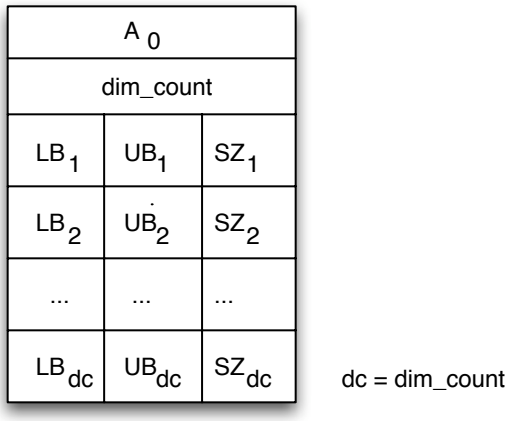


Figure 2: A prototypical array descriptor.

4 Data type operations

The operations on arrays are described in this module as procedure and function calls. However, because the array data type is built into most programming languages, a different special purpose syntax is used when arrays and array elements are referenced – see the sections following the section *Data type objects*.

4.1 Enquiry operations

These operations retrieve information from the array descriptor.

4.1.1 How many dimensions does an array have?

```
dimensions ( an_array : ARRAY ) : INTEGER
```

require an_array ≠ void.
ensure Result = dim_count.
Program text is not referenced

4.1.2 What is the lower bound of a given dimension of an array?

```
lower_bound ( an_array : ARRAY ; dimension : INTEGER ) : INTEGER
```

require an_array ≠ void **and** 1 ≤ dimension ≤ dim_count
ensure Result = $LB_{dimension}$.
Program text is not referenced

4.1.3 What is the upper bound of a given dimension of an array?

```
upper_bound ( an_array : ARRAY ; dimension : INTEGER ) : INTEGER
```

require an_array ≠ void **and** 1 ≤ dimension ≤ dim_count
ensure Result = $UB_{dimension}$.
Program text is not referenced

4.1.4 What is the amount of space used by an array element of a given dimension?

```
size_of_element ( an_array : ARRAY ; dimension : INTEGER ) : INTEGER
```

require an_array ≠ void **and** 1 ≤ dimension ≤ dim_count
ensure Result = $SZ_{dimension}$.
Program text is not referenced

4.2 Read operations

Only one function is necessary.

4.2.1 What is the address of a specifically indexed array element?

index (an_array : ARRAY ; $index_1$, … , $index_{dim\ count}$: INTEGER) : REFERENCE

require an_array ≠ void
∀ i : 1 .. dim_count • LB_i ≤ $index_i$ **and** $index_i$ ≤ UB_i
ensure Result = location_of(an_array[$index_1$, … , $index_{dim\_count}$]).

The function returns the address of the array element with the specified index set. We represent the address as a reference to a memory location.
Program text is not referenced

4.3 Write operations

The only operations are to be able to create a new array and dispose of an existing array.

4.3.1 Create a new array

create_array (an_array : NAME ; dim_count , basic_size
, LB_1 , … , LB_{dim_count} , UB_1 , … , UB_{dim_count} : INTEGER) : REFERENCE

require an_array ≠ void
ensure Result = address of an_array that is an array with the specified number of dimensions and with the specified lower and upoer bounds for each dimension.

Program text is not referenced

4.3.2 Dispose of an existing array

```
delete_array ( an_array : ARRAY )
```

require an_array ≠ void
ensure an_array is no longer a valid name as the array is removed from the environment.

Program text is not referenced

4.4 Sparse array operations

Sparse arrays are arrays which have mostly undefined or zero elements. Special storage techniques are used to save space by not reserving space for the unused array elements. Sparse arrays require two special operations.

4.4.1 Add an element to a sparse array

```
add_array_element ( array_name : ARRAY , value : VALUE
                  , index1, … , indexdim_count : INTEGER )
```

require an_array ≠ void
∀ i : 1 .. dim_count • $LB_i \le index_i$ **and** $index_i \le UB_i$ value is a valid array value
ensure The value is inserted into the array at the specified index position. If the element is already in the array the value will be changed.

Program text is not referenced

4.4.2 Delete an element from a sparse array

```
delete_array_element ( array_name : ARRAY , index1, … , indexdim_count : INTEGER )
```

require an_array ≠ void
∀ i : 1 .. dim_count • $LB_i \le index_i$ **and** $index_i \le UB_i$
ensure The appropriate array element is removed from the table.

Program text is not referenced

5 Array memory allocation methods

In this module, we only describe, in more detail, row and column order memory allocation and sparse matrix memory allocation. We do not describe in detail memory allocation methods for other special cases such as **triangluar arrays** (values above or below the main diagonal are all zero) and **symmetric arrays** (where the relationship A[i,j] = A[j,i] holds). In the last two cases, it is wasteful to store all those zero or repetative array values.

5.1 Row and column storage

This is the most common method of allocating memory space to an array. With this method we assume that all of the array elements are to be equally accessible at all times.

The first step is to compute the total space required by all the array elements and allocate one contiguous chunk of memory big enough to contain all the array elements. The starting address for the allocated memory is A_S.

The second step is to assign array elements to a sequence of memory locations. The method chosen must have the property that an efficient index function can be created to compute quickly the memory location of any array element from a given set of indices.

The assignment of array elements to memory locations is done by systematically varying the index values. We treat each index position as a digit in a mixed radix numbering scheme where the digit values in each position vary between the lower and upper bounds associated with the index position. The systematic variation is to "add one" to a given index set to obtain the next array element in sequence.

NOTE: In decimal notation, each digit position in a number is assigned the same range of digit values, '0' to '9'. In array storage allocation each digit position has its own range of digit values.

We now have two choices as to how to "add one". We can add at the right most index position and propagate carries to the left. Or, we can add at the left most index position and propagate carries to the right. If the right most subscript changes faster, we have **row order** storage – used in C and Java. If the left most subscript changes faster, we have **column order** storage – used in Fortran. The terminology arises from the pictorial representation of storing a two dimensional array.

1,1	1,2	1,3
2,1	2,2	2,3
3.1	3,2	3,3
4,1	4,2	4,3

Figure 3: A 4x3 two−dimensional array.

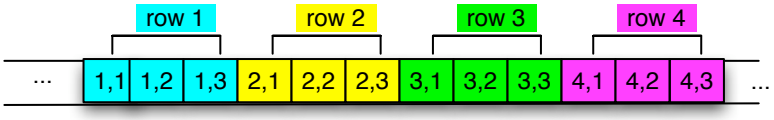


Figure 4: Row order storage for the array in Figure 3.

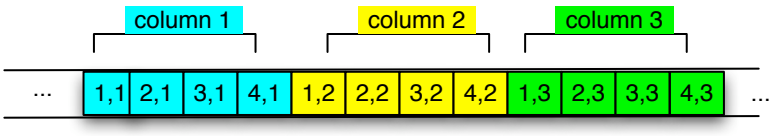


Figure 5: Column order storage for the array in Figure 3.

5.2 General formula to compute the address of an array element

The formula, F1, is used to compute the address of an array element.

$$(F1) \qquad A_S + \sum_{k=1}^{dc} (I_k - LB_k) \times SZ_k \qquad \text{where } dc = dim\_count$$

A_S = address of the first array element

The calculation of the size, SZ_k , of the elements in each dimension of the array is done once at the time the array is created and is stored in the array descriptor. The recurrence relation (R1) is used to compute the values of SZ_k .

$$(R1) \qquad SZ_{dim\_count} = \text{the size of the base type for the array}$$

$$SZ_k = SZ_{k+1} \times (UB_{k+1} - LB_{k+1} + 1) \qquad \forall k : dim\_count - 1 \ldots 1$$

NOTE: If, for the computation, we number the dimensions in A[*, … , *] from left to right we have row order storage. If, for the computation, we number the dimensions from right to left we have column order storage.

The following arithmetic relationship, R2, is true.

$$(R2) \qquad \sum_{k=1}^{dc} (I_k - LB_k) \times SZ_k = \sum_{k=1}^{dc} I_k \times SZ_k - \sum_{k=1}^{dc} LB_k \times SZ_k$$

We substitute the right hand side of R2 into the formula F1 to obtain the formula F2.

$$(F2) \qquad A_S + \sum_{k=1}^{dc} I_k \times SZ_k - \sum_{k=1}^{dc} LB_k \times SZ_k$$

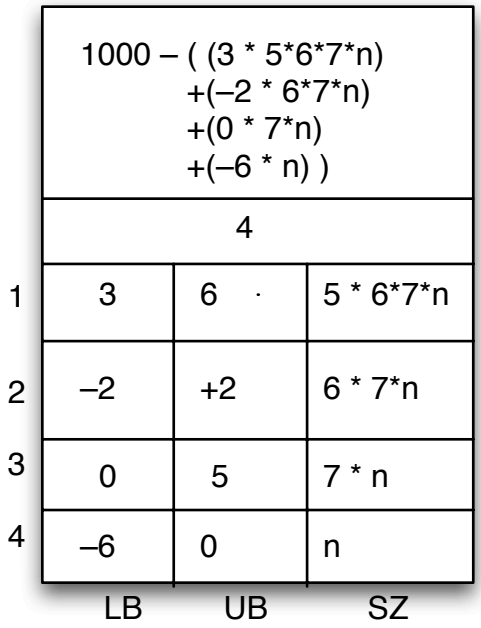
Note that A_S and $\sum_{k=1}^{dc} LB_k \times SZ_k$ are constants, so we can define the following relationship R3. A_0 is the memory address where the array element with all zero indicies would be stored.

$$(R3) \qquad A_0 = A_S - \sum_{k=1}^{dc} LB_k \times SZ_k$$

A_0 is a constant that only needs to be computed once at the time the array is created. As a consequence, the value of A_0 is stored in the array descriptor and we can compute the address of an array element using the formula F3.

$$(F3) \qquad A_0 + \sum_{k=1}^{dc} I_k \times SZ_k$$

Example 1: Figure 6 shows an example definition for a four−dimensional array and its corresponding descriptor that corresponds to row order storage. The arithmetic terms are left unsimplifed to show how the formulas are applied.



array[3..6 , -2..+2 , 0..5 , -6..0] of T
---->
array [3..6] of
array [-2..+2] of
array [0..5] of
array [-6..0] of T

T, the base type, is of size n bytes.

A_s is 1000

Figure 6: An example array descriptor for a 4-dimensional array

5.3 Sparse arrays

A special case occurs when most of the array elements are zero or undefined. In such cases, we do not want to store all the zero elements. There are various schemes used to store such arrays which make efficient use of memory. What all the methods have in common is that extra information must be stored that maps array indices to memory locations.

A table, called the index–value table, is formed which stores the index values for the non–zero array elements together with the array element value corresponding to the index set. When a reference is made to an array element, the indices are compared with those in the index–value table until a match is found and the array element value is returned, or, if there is no match, and a zero value is returned.

The differences among ways of storing sparse arrays occur in the method used to store the index–value table. The table could be implemented using arrays or, as is also common, using a multi–list structure.