UNIT 1 INTRODUCTION TO DATA STRUCTURES; ARRAY

Structure

1.0 INTRODUCTION

This unit is an introductory unit and gives you an understanding of what a data structure is. Knowledge of data structures is required of people who design and develop computer programs of any kind : systems software or applications software. As you have learnt in earlier blocks, data are represented by data values held temporarily within program's data area or recorded permanently on a file. Often the different data values are related to each other. To enable programs to make use of these relationships, these data values must be in an organised form. The organised collection of data is called a data structure. The programs have to follow certain rules to access and process the structured data. We may, therefore, say data are represented that

Data Structure = Organised Data + Allowed Operations.

If you recall, this is an extension of the concept of data type. We had defined adata type as

Data Type = Permitted Data Values + Operations

Further, we had seen that simple data type can be used to built new scalar data types, for example subrange and enumerated type in Pascal. Similarly there are standard data structures which are often used in their own right and can form the basis for complex data structures. One such basic data structure called Array is also discussed in this unit Arrays are basic building block for more complex data structures. Designing and using data structures is an important programming skill. In this and in subsequent units, we are going to discuss various data structures. We may classify these data structures as linear and non-linear data structures. However, this is not the only way to classify data structures. In linear data structure the data items are arranged in a linear sequence like in an array. In a non-linear, the data items are not in sequence. An example of a non linear data structure is a tree. Data structures may also be classified as homogenous and non- homogenous data structures. An Array is a homogenous structure in which all elements are of same type. In non-homogenous structures the elements may or may not be of the same type. Records are common example of non-homogenoes data structures. Another way of classifying data structures is as static or dynamic data structures. Static structures are ones whose sizes and structures associated memory location are fixed at compile time. Dynamic structure are ones which expand or shrink as required during the program execution and their associated memory locations change. Records are a common example of non-homogenous data structures.

In this unit first we have discussed about the performance of an algorithm, You may find it very relevant as You read on the subsequent blocks and develop programs. Then we introduce the array data structure. Then the array declarations in Pascal and C are reviewed. A section is devoted to discussion on how single and multi-dimensional arrays are mapped to storage. Finally a discussion on sparse arrays closes the unit .

1.1 OBJECTIVES

At the end of the Unit, you would be able to

- define a data structure
- differentiate between data type and data structure
- analyze the trade offs of the data handling needs of a particular problem situation
- store and retrieve data from an array

1.2 PROGRAM ANALYSIS

Now, since you have learnt about developing programs in Pascal and C, let us understand what happens when a program is executed - the sequence of actions executed and the changes in the program state that occur during a run.

There are many ways of analysing a program, for instance:

(i) verifying that it satisfies the requirements.

(ii) proving that it runs correctly without any logic errors.

(ii) determining if it is readable.

(iii) checking that modifications can be made easily, without introducing new errors.

(iv) we may also analyze program execution time and the storage complexity associated with it i.e. how fast does the program run and how much storage it requires.

Another related question can be : how big must its data structure be and how many steps will be required to execute its algorithm?

Since this course concerns data representation and writing programs, we shall analyse programs in terms of storage and time complexity.

Performance Issues

In considering the performance of a program, we are primarily interested in

(i) how fast does it run?

(ii) how much storage does it use?

Generally we need to analyze efficiencies, when we need to compare alternative algorithms and data representations for the same problem or when we deal with very large programs.

We often find that we can trade time efficiency for space efficiency, or vice-versa. For finding any of these i.e. time or space efficiency, we need to have some estimate of the problem size. Let's assume that some number N represents the size of the problem. The size of the problem or N can reflect one or more features of the problem, for instance N might be the number of input data values, or it is the number of elements of an array etc.

Let us consider an example here.

Suppose that we are given two algorithms for finding the largest value in a fist of N numbers. It is also given that second algorithm executes twice the number of instructions executed by the first algorithm for each N value.

Let first algorithm executes S number of instructions. Then second algorithm would execute 2S instructions. If each instruction takes 1 unit of time, say 1 millisecond then for N= 10, 100, 1000 and 10000 we shall have the number of operations and estimated execution time may be as given below:

	Algorithm 1		**Algorithm II**	
N	**Number of instructions**	**Estimated Execution Time**	**Number of instructions**	**Estimated Execution Time**
10	10S	10 msec	20S	20 msec
100	100S	100 msec	200S	200 msec
1000	1000S	1000 msec	2000S	2000 msec
10000	10000S	10000 msec	20000 S	20000 msec

1 millisec = 1 msec = 1/1000 sec

You may notice that for larger values of N, the difference between execution time of the two algorithm is appreciable, and one may clearly say that Algorithm II is slower than Algorithm I. Also the difference of algorithm I is comparatively better as the problem size get larger. This kind of performance improvement is termed as order of improvements. Two algorithm may compare with each other by a constant factor, i.e. improvement from one to another does not change as the problem size gets larger. For example, one of them can be two times faster than the other and it always remain two times better regardless of the problem size.

Associated to order of improvements are the several order of classification of algorithum as given below. We shall be discussing those again in Block 6, when we shall introduce you to various solving and searching algorithms.

- If the problem size doubles and the algorithm takes one more step, we relate the number of steps to the number of steps to the problem size by

  $O(\log_2 N)$

  It is read as order of $\log_2 N$.

- If the problem size doubles and the algorithm takes twice as many steps, the number of steps is related to problem size by

  $O(N)$

  i.e. order of N, i.e.number of steps is directly proportional to N.

- If the problem size doubles and the algorithm takes more than twice as many steps, i.e. the number of steps required grow faster than the problem size, we use the expression

  $O(N \log_2 N)$

  You may notice that the growth rate complexities is more than the double of the growth rate of problem size, but it is not a lot fasten

- If the number of steps used is proportional to the square of problem size ,we say the com- plexity is of the order of N^2 or $O(N^2)$.

- If the algorithm is independent of problem size, the complexity is constant in time and space, i.e. O(1).

The notation being used, i.e. a capital O() is called Big- Oh notation.

Average case and worst case analysis

An algorithm chooses an execution path depending on the set of data values (input). Therefore an algorithm may perform differently for two different sets of data values. If we take a set of data values for which the algorithm takes the longest possible execution time, it leads us to the worst case execution time. On the other hand, an average case execution time is the execution time takes by algorithm for an expected range of data values. For analysis of an algorithm to predict its average or worst case execution time, we need to make certain assumption such as assuming that all operations take about the same amount of time. We shall be seeing the worst case and average case behaviour of sorting and searching algorithms in Block 6.

1.3 ARRAYS

In applications where we have a small number of items to handle, we tend to specify separate variables names for each item. When we have to keep track of more pieces of data, we need to organise data. Such that we can use one name to refer to several items. Let us see this through a simple example. Consider the following problem:

Read 25 numbers and print them in reverse order.

The problem requires all the numbers as they are read. Further we cannot print anything until all 25 numbers are read; therefore, we need to store all the twenty five numbers. Reading 25 numbers in 25 different variables will be quite cumbersome and so would be writing these numbers in reverse order. It is much simpler to call the numbers NUM_1 NUM_2, NUM_3 .... NUM_{25}.

Each number is a NUM, and numbers are distinguish by subscripts. Also they are read in succession. Thus we can abbreviate this sequence as NUM_i for i = 0,1, 2. ..... 24. Such a subscripted variable is called an Array. More formally an array is a Finite ordered set of homogenous elements which are stored in adjacent cells in memory. Arrays are usually used when a program include a list of recovering elements.

In C subscripts are Placed in square brackets [ ]. Repetition over a sequence Of values of i may also be implemented using a loop construct. For example, the following statement reads all 25 values:

```
for ( i = 0; i<25;++1) { scanf ("%d",NUM[i]);
```

A similar approach works out for Printing the values.

The simplest form of an array is a one-dimensional array or vector. As stated earlier, the various elements of an array are distinguished by giving each piece of data separate index or subscript. The subscript of an

element designates its position in array's ordering. An array named A which consists of N elements can be depicted as shown in figure 1.

A(0) .A(1) .A(2).....A(N-1)

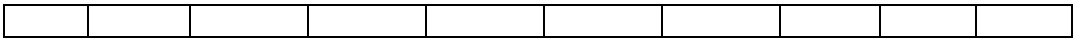

Figure 1 : One Dimensional Array

Arrays can be multi-dimensional. Any array defined to have more than one dimension is considered to be multi-dimensional array. An array can be 2-dimensional, 3-dimensional, 4- dimensional, or N-dimensional although they rarely exceed three dimensions. Two-dimensional arrays, sometimes called matrices, are quite common. The best way to think about a two-dimensional array is to visualize a table of columns and rows: the first dimension in. the array refers to the rows, and the second dimension refers to the columns.

Let us see an example of a 2-dimensional array.

A collection of data about the grades of students in a class in the four different exams can be represented using a 2-dimensional arrays. If we have 10 students and each given grades in 4 exams, we can depict it as in the table (Figure 2).

		Grade			
		1	2	3	4
	1				
	2				
	3				
	4				
Student	5				
Number	6				
	7				
	8				
	9				
	10				

Figure 2

Each cell in this table contains a grade value for the student Number (given by the corresponding row number) and exam number (given by the corresponding column no.). We may map it on to an array A of order 10x4. A [I] [J] represents an element of A, where I runs from 0 to 9 and J runs from 0 to 3. A [3] [4] will have the grade value of 4th student in fifth exam, A [8] [1] will have the grade value of 9th student in second exam, and so on.

By convention the first subscript of a 2-dimensional array refers to a row of the array, while the second subscript refers to a column of the array. These rows and columns are one more than what is represented as subscripts.

In general an array of the order M X N (read as M by N) consists of M rows, N columns and MN elements. It may be depicted as

0 1 .....N-1

0					
1					
.					

.					
M-1					

Figure 3

To assign a value in a multi-dimensional array, specify all the dimensions, as shown in following statement:

GRADE [3] [4]: ='A'; CUBE [1] [3] [2]: = 0;

Let us now discuss the syntax and semantics of an array. We can divide our discussion in three parts:

- Array declaration
- Storage of Arrays in Main Memory
- Use of Arrays in Programs
-

1.4 ARRAY DECLARATION

Three things need to be specified to declare an array in most of the programming languages:

- the array name
- the type of data to be stored in array elements
- the subscript range

In C language the array declaration is as follows:

```
Int A[24];
Int B [100] [25];
```

In first declaration A is the array name; the elements of A can hold integer data and the number of elements is 24 i.e. subscripts range from 0 to 23.

In the next declaration B is the array name; the data type of its elements is real and it is a 2-dimensional array with subscripts ranging from 0 to 99 and 0 to 24.. It makes more sense to start the array at a value that corresponds to the context of your data. You may notice that the subscript is may not always be positive. It can be negative or zero or negative subscripts.

Failing to remember that the zero element is the first item in the array - and therefore, the element at index 5 is the sixth, not the fifth - is a frequent cause of programming bugs.

The reason for this is that some languages - C and C++ for example - require arrays to begin with zero indexes.

An array declaration tells the computer two major pieces of information about an array. First, the range of subscripts allow the computer to determine how many memory locations must be allocated. Second the array type tells the computer how much space is required to hold each value. Let us consider the following declarations:

```
Int A[10];
Float B[10];
```

The first declaration tells the computer to allocate enough space for the variable A to store 10 integers. The second declaration tells the computer to allocate enough space for the variable B to store 10 rears. Since a real number takes more space than an integer the storage allocated would not be same. Array declarations have already been discussed in Block 1, Unit 3 and Block 3, Unit 1 respectively.

Operations on Arrays

The array is a homogenous structure, i.e. the elements of an array are of the same type. Following set of operations are defined for this structure.

(i) creating an array

(ii) initialising an array

(iii) storing an element

(iv) retrieving an element

(v) inserting an element

(vi) deleting an element

(vii) searching for an element

(viii) sorting elements

(ix) printing an array

These operations apply to array of any dimension. Some of these functions have already been discussed in unit on arrays in C, i.e. Block 1, Unit 3 and Block 2, Unit ... The operation of searching for value in an array looks for a mark of value with the elements in the array. If found it returns the index of such an element that matches the value. Otherwise it returns an appropriate message. Sorting means rearranging the array elements in a particular order.

Searching and sorting operations are discussed in detail in Block 6.

1.5 STORAGE OF ARRAYS IN MAIN MEMORY

Let us now see how the data represented in an array is actually stored in the memory cells of the machine. Because computer memory is linear, a one-dimensional array can be mapped on to the memory cells in a rather straight forward manner. Storage for element A[I+ 1] will be adjacent to storage for element A [I] for I = 0, 1, 2, ... N-1. To find the actual address of an element one merely needs to subtract one from the position of the desired entry and then add the result to the address of the first cell in the sequence. Let us see it through an example. Consider an array A of 26 elements. We require to find the address of A[4]. If the first cell in the sequence A[0] A[1], A[2], ... A[25]. was at address 16, then A[4] would be located at 16 + 4 = 20, as shown in Figure 4. We assume that the size of each element stored is one unit.

	16	17	18	19	20	21	22	
Memory Cells								

	A[0]	A[1]	A[2]	A[3]	A[4]	..	..	..

Figure 4

Therefore, it is necessary to know the starting address of the space allocated to the array and the size of the each element which is same for all the elements of an array. We may call the starting address as a base address and denote it by B. Then the location of I_{th} element would be

B + I*S (1)

where S is the size of each element of array.

Let us now consider storage mappings for multi-dimensional arrays. As we had seen in previous section that in a 2-dimensional array we think of data being arranged in rows and columns. However Machine's memory is arranged as a row of memory cells. Thus the rectangular structure of a 2-dimensional array must be simulated. We first calculate the amount of storage area needed and allocate a block of contiguous memory cells of that size. One way to store the data in the cells is row by row. That is, we store first the first row of the array, then the second row of the array and then the next and so on. For example the array defined by A which logically appears as given in Figure 5; appears physically as given in Figure 6. Such a storage scheme is called Row Major Order.

The order alternative is to store the array column by column. It is called Column Major Order. The array of Figure 7.

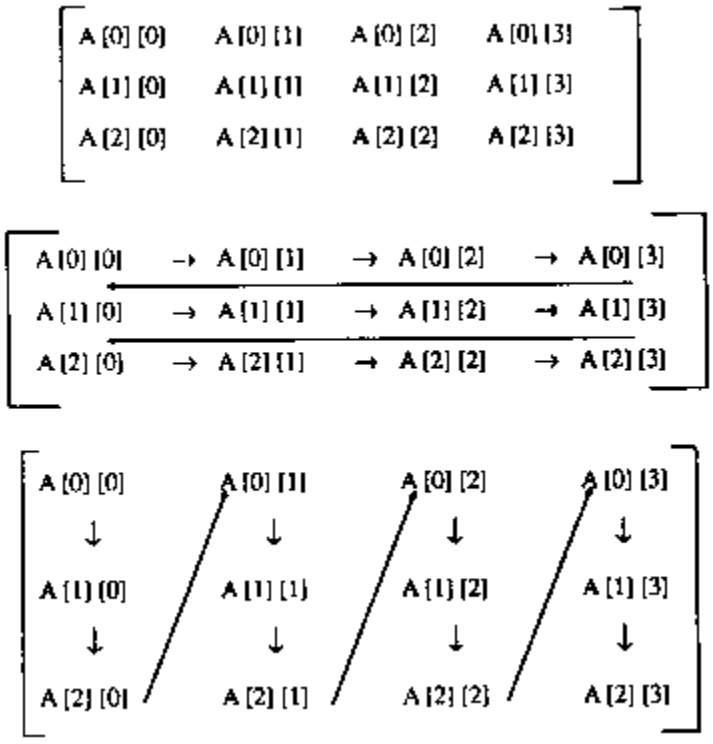

FIGURE 7 Column Major Representation

Check Your Progress 1

Question 1: Create a two-dimensional array whose number of rows are 10 and columns are 26 and the component type is character.

Question 2: Show how the array

$$\begin{matrix} 1 & 3 & 7 \\ 5 & 2 & 8 \\ 9 & 7 & 1 \end{matrix}$$

would appear in memory when stored in
(i) row major order
(ii) column major order

In all implementations of C the storage allocation scheme used is the Row Major Order.

Let us now see how do we calculate the address of an element of a 2-dimensional array, which is mapped in Row Major Order. Consider a 4x6 array A[4] [6]. Take B as the array's base address and S as the size of element of the array.

To locate element A [I] [J] we must skip (I-1) rows; each having 6 elements, each element of S length and (J- 1) elements of Ith row, each of length S. Therefore, the address of element A[I] [J] would be

B+I*6*S + J*S (4)

We may now generalize this expression for a 2-dimensional array

$A[U_0], [U_1]$

where $(U_0\text{-}1)$ and $(U_1\text{-}1)$ are the upper bounds of the two subscript ranges.

The location of an element A[I],[J] for such an array would be

$B+I * (U_1*S + J*S$

The row major order varies the subscripts in right to left order. For example the elements of a 2-dimensional array $A[U_1] [U_2]$ would be stored in following order:

A[0] [0]
A[0] [1]
.
.
.
$A[0] [U_2\text{-}1]$
A[1] [0]
A[1] [1]
A[1] [2]
.
.
.
$A[1] [U_2\text{-}1]$
.
.
.
.
.
.

.
$A[U_1-1]\ [U_2-1]$

We may generalize it for an N-dimensional array $A[U_0]\ [U_1]\ ....\ [U_{n-1}]$. The elements would be stored in following order:

$A[0]\ [0]\ ....\ [0]$
$A[0]\ [0]\ ....\ [1]$
.
.
.
$A[0]\ [0]\ ....\ [U_{n-1}{}^{-1}]$
$A[0]\ [0]\ ....,[1]\ [0]$
.
.
.
$A[0]\ [0]\ .....[1]\ [U_{n-1}{}^{-1}]$
.
.
.
$A[U_0-1]\ [U_1-1]\ .....\ U_{n-1}\ ]$

Let us see how the above expressions work out for a column major order.

We once again consider a 4x6 array A[4] [6]. Also take B as base address and S as size of each element. Then the address of A[I,J] would be

$B+J*4*S+ I*S$

To reach A[I] [J] we shall skip J-1 columns, each of length 4* S and I-1 elements each of lengths. We further generalize it for an array $A[U_1]\ [U_2]$.

Following the same logic, the address of A[I] [J] would be given as

$B+(J*U_1*S)+(I*S)$

The column major order varies the subscripts in left to right order. For example the elements of a 2-dimensional array A[4] [3] would be stored in the sequence as given below:

$A[0]\ [0]$
$A[1]\ [0]$
$A[2]\ [0]$
$A[3]\ [0]$
.
.
.
.
$A[0]\ [2]$
$A[1]\ [2]$
$A[2]\ [2]$
$A[3]\ [2]$

Compare this sequence with the one you obtain in check your progress.

Check Your Progress 2

Question 1: How would a 4x-3 array A[4] [3] stored in Row Major Order?

Question2: How would a m x n array A [m] [n] stored in Column Major Order?

As we had done for Row Major Order, we may generate the sequence of N-dimensional array

$A[U_1] [U_2] \ldots\ldots [U_n]$

as stored in Column Major Order. It would be as given below:

A[0] [0] ... [0]
A[1] [0]... [0]
$A[U_1-1]$ [0] ....[0]
.
.
.
.

A[0] [1]... [0]
A[0] [2]... [0]
.
.
.
A[0] [U2-1... [0]
.
.
.
$A[U_2-1] [U_2-1] \ldots [U_n-1]$

1.6 SPARSE ARRAYS

Sparse arrays are special arrays which arise commonly in applications. It is difficult to draw the dividing line between the sparse and non-sparse array. Loosely an array is called sparse if it has a relatively number of zero elements. For example in Figure 8, out of 49 elements, only 6 are non-zero This is a sparse array.

0	0	0	0	0	1	0
0	0	0	0	1	0	0
0	0	0	0	0	0	0
0	0	0	0	3	0	0
0	2	0	0	0	0	0
0	0	0	0	4	0	0
0	0	0	0	2	0	0

Figure 8: A Sparse Array

If we store those array through the techniques presented in previous section, there would be much wasted space.

Let us consider 2 alternative representations that will store explicitly only the non-zero elements.

1. Vector representation
2. Linked List representation

We shall discuss only the first representation here in this Unit. The Linked List representation shall be discussed in a later Unit.

Each element of a 2-dimensional array is uniquely characterized by its row and column position we may, therefore, store a sparse array in another array of the form.

A[n+1] [3]

where n is number of non-zero elements.

The sparse array given in Figure 8 may be stored in the array A[7] [3] as shown in Figure 9.

		1	2	3
A	0	7	7	6
	1	1	6	1
	2	2	5	1
	3	4	5	3
	4	5	2	2
	5	6	5	4
	6	7	5	2

Figure 9: Sparse Array Representation

The Elements A[O] [0] and A[0] [1] contain the number of rows and columns of the sparse array. A[0] [2] contains the number of non- zero elements of sparse array. The first and second element of each of the rows store the number of row and column of the non- zero term and the third element stores the value of non-zero term. In other words, each non-zero element in a 2-dimensional sparse array is represented as a triplet with the format (row subscript, column subscript, value).

If the sparse array was one-dimensional, each non-zero element would be represented by a pair. In general for an N-dimensional sparse array, non-zero elements are represented by an entry with N+1 values.

1.7 SUMMARY

In this unit, we formally introduced the concept of data structure followed by the most common and simple data structure Array.

Arrays help to solve problems that require to keep track of many pieces of data. To use array structure, the name of the array, the type of its elements and the type of its subscripts must be allowed. The declaration tells the computer the allocate the appropriate memory space.

We have also discussed the storage of arrays in the main memory in row major order and in column major order. In the last section, we learnt about a special kind of arrays called sparse arrays. In a sparse array, a large proportion of the elements are zero, but those which are non-zero randomly distributed.

Many application involving sparse arrays use example which are large to be represented in the standard way. Some way of compacting the information contained in an array is needed. One of the methods to store non zero elements is presented in this unit.

Arrays are used in programming languages for holding group of elements all of the same kind. Vectors, matrices, chess boards, networks, polynomials, etc. can be represented as arrays. The space requirement of

array can be large. Good programming practice suggests that arrays should not be used unless there is a good reason for their use.

MODEL ANSWERS

Check Your Progress 1

1. chan ARRAY_TYPE [10] [26]

2. No model answer is given.

Check Your Progress 2

1. No model answer is given.

2. No model answer is given.