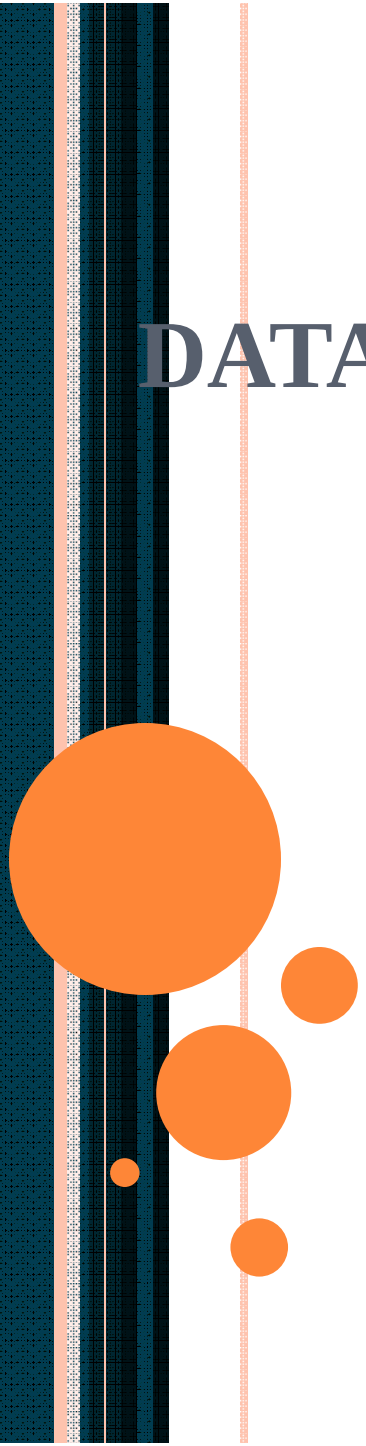
DATA STRUCTURES USING ‘C’



Lecture-7

Data Structures

Lecture Objectives

After studying this chapter, the student should be able to:

- **Define a data structure.**
- **Define an array as a data structure and how it is used to store a list of data items.**
- **Distinguish between the name of an array and the names of the elements in an array.**
- **Describe operations defined for an array.**
- **Define a record as a data structure and how it is used to store attributes belonging to a single data element.**

11-1 ARRAYS

Imagine that we have 100 scores. We need to read them, process them and print them. We must also keep these 100 scores in memory for the duration of the program. We can define a hundred variables, each with a different name, as shown in Figure 11.1.

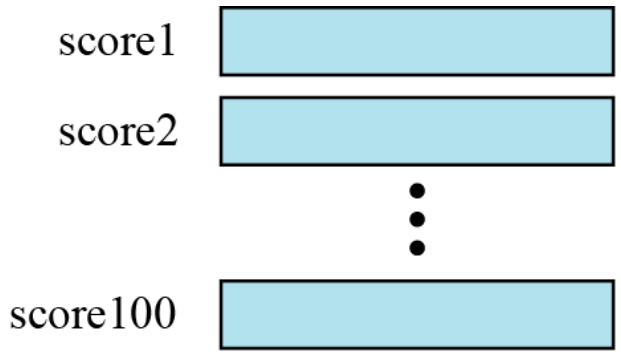


Figure 11.1 **A hundred individual variables**

But having 100 different names creates other problems. We need 100 references to read them, 100 references to process them and 100 references to write them. Figure 11.2 shows a diagram that illustrates this problem.

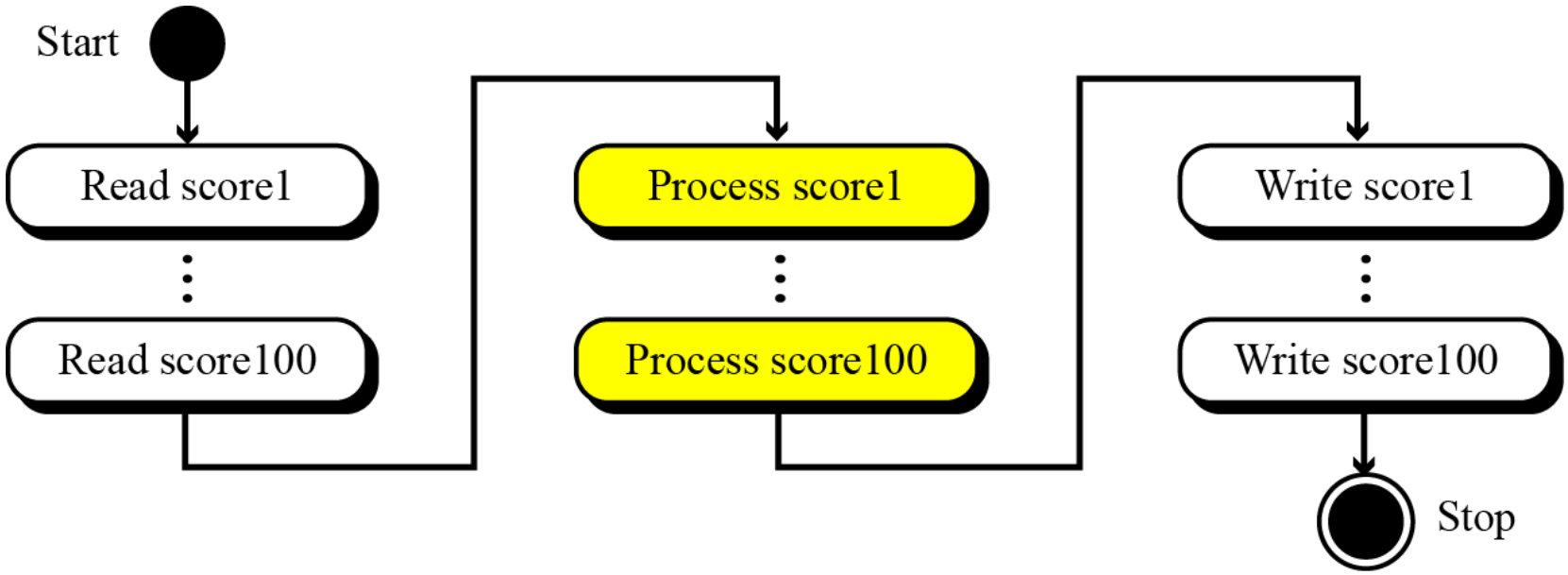


Figure 11.2 **Processing individual variables**

An array is a sequenced collection of elements, normally of the same data type, although some programming languages accept arrays in which elements are of different types. We can refer to the elements in the array as the first element, the second element and so forth, until we get to the last element.

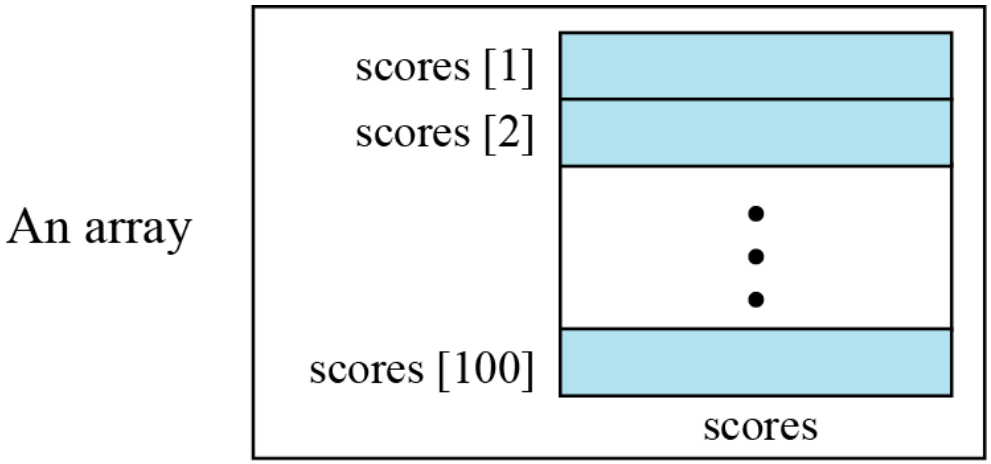


Figure 11.3 **Arrays with indexes**

We can use loops to read and write the elements in an array. We can also use loops to process elements. Now it does not matter if there are 100, 1000 or 10,000 elements to be processed—loops make it easy to handle them all. We can use an integer variable to control the loop and remain in the loop as long as the value of this variable is less than the total number of elements in the array (Figure 11.4).

We have used indexes that start from 1; some modern languages such as C, C++ and Java start indexes from 0.

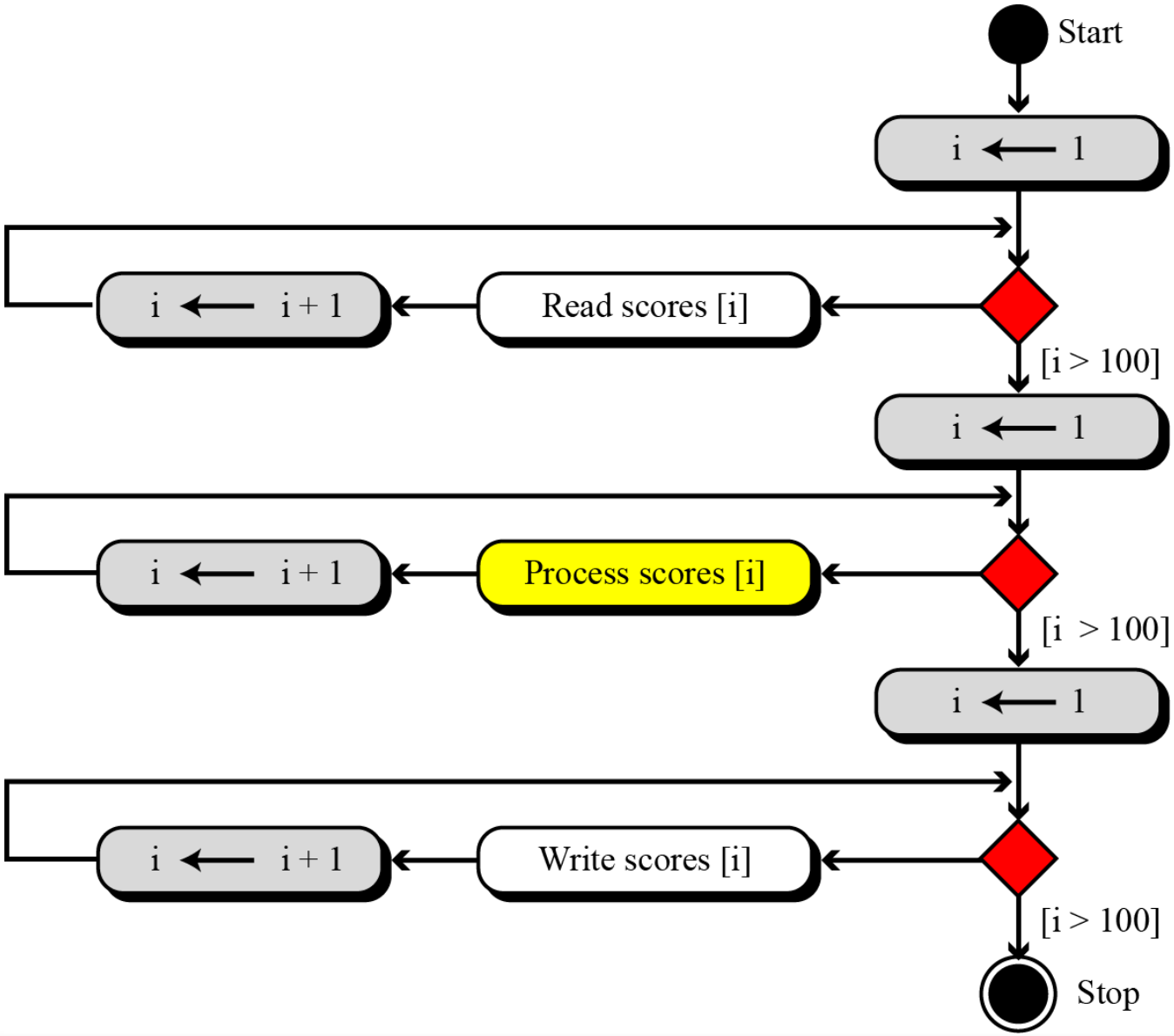


Figure 11.4 **Processing an array**

Example 11.1

Compare the number of instructions needed to handle 100 individual elements in Figure 11.2 and the array with 100 in Figure 11.4. Assume that processing each score needs only one instruction.

Solution

- In the first case, we need 100 instructions to read, 100 instructions to write and 100 instructions to process. The total is 300 instructions.
- In the second case, we have three loops. In each loop we have two instructions, for a total of six instructions. However, we also need three instructions for initializing the index and three instructions to check the value of the index. In total, we have twelve instructions.

Example 11.2

The number of cycles (fetch, decode, and execute phases) the computer needs to perform is not reduced if we use an array. The number of cycles is actually increased, because we have the extra overhead of initializing, incrementing and testing the value of the index. But our concern is not the number of cycles: it is the number of lines we need to write the program.

Example 11.3

In computer science, one of the big issues is the reusability of programs—for example, how much needs to be changed if the number of data items is changed. Assume we have written two programs to process the scores as shown in Figure 11.2 and Figure 11.4. If the number of scores changes from 100 to 1000, how many changes do we need to make in each program? In the first program we need to add $3 \times 900 = 2700$ instructions. In the second program, we only need to change three conditions ($I > 100$ to $I > 1000$). We can actually modify the diagram in Figure 11.4 to reduce the number of changes to one.

Array name versus element name

In an array we have two types of identifiers: the name of the array and the name of each individual element. The name of the array is the name of the whole structure, while the name of an element allows us to refer to that element. In the array of Figure 11.3, the name of the array is ***scores*** and name of each element is the name of the array followed by the index, for example, ***scores*****[1]**, ***scores*****[2]**, and so on. In this chapter, we mostly need the names of the elements, but in some languages, such as C, we also need to use the name of the array.

Multi-dimensional arrays

The arrays discussed so far are known as **one-dimensional** arrays because the data is organized linearly in only one direction. Many applications require that data be stored in more than one dimension. Figure 11.5 shows a table, which is commonly called a **two-dimensional** array.

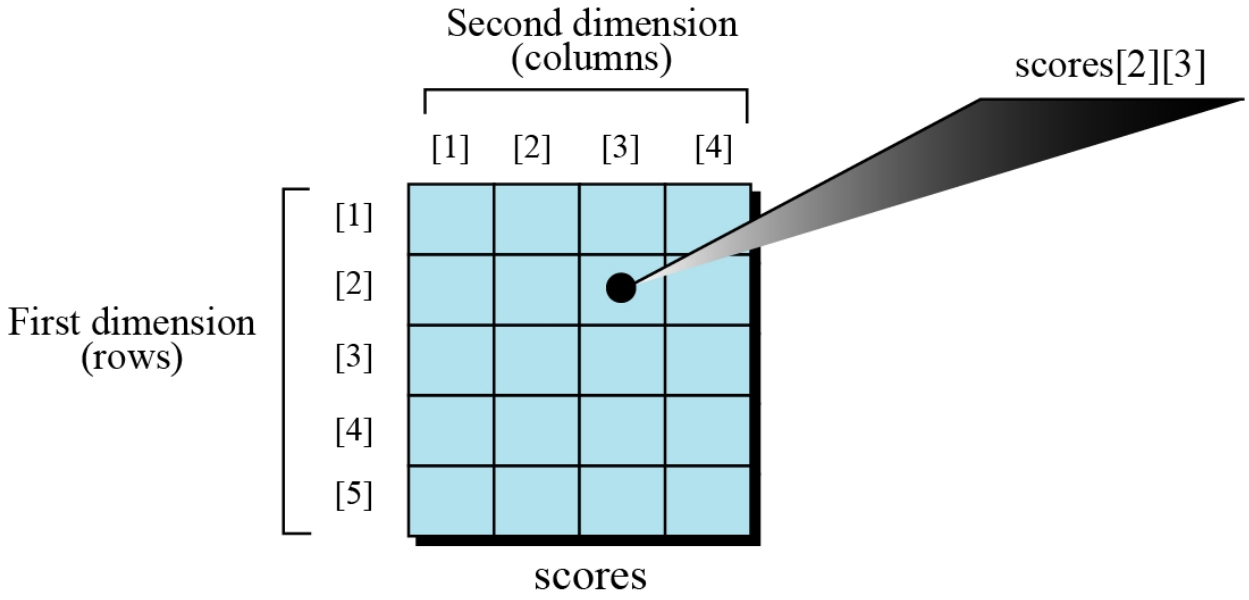


Figure 11.5 **A two-dimensional array**

Memory layout

The indexes in a one-dimensional array directly define the relative positions of the element in actual memory. Figure 11.6 shows a two-dimensional array and how it is stored in memory using row-major or column-major storage. Row-major storage is more common.

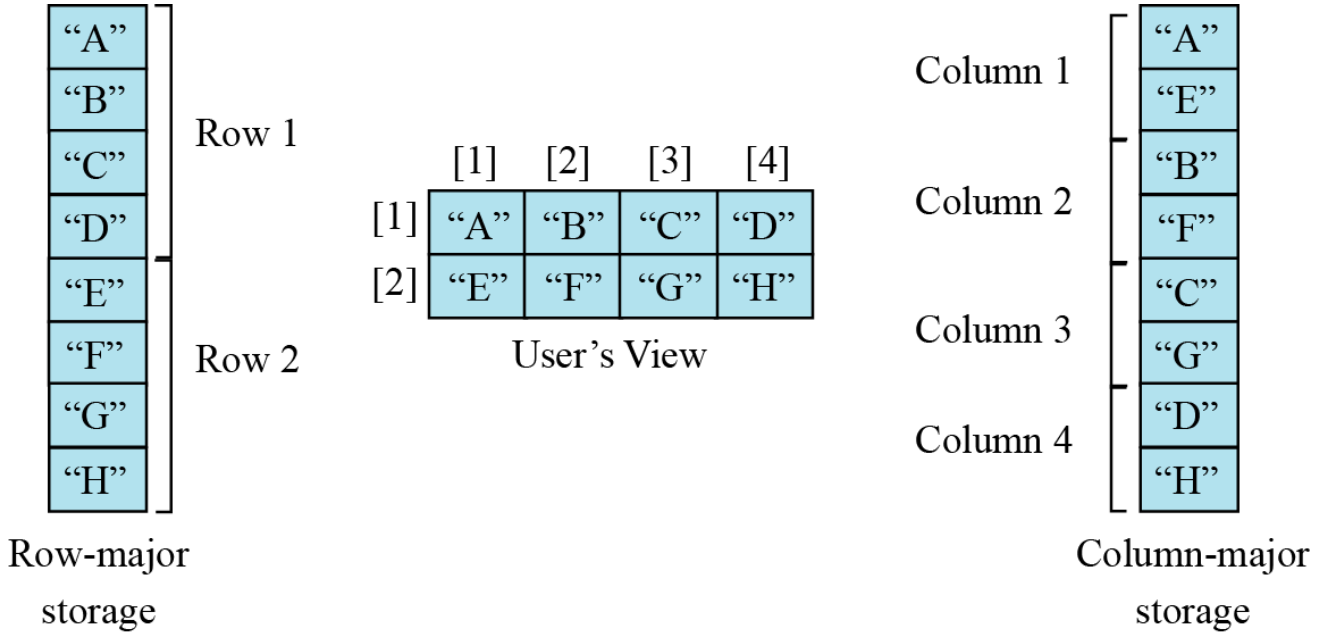


Figure 11.6 **Memory layout of arrays**

Example 11.4

We have stored the two-dimensional array students in memory. The array is 100 × 4 (100 rows and 4 columns). Show the address of the element students[5][3] assuming that the element student[1][1] is stored in the memory location with address 1000 and each element occupies only one memory location. The computer uses row-major storage.

Solution

We can use the following formula to find the location of an element, assuming each element occupies one memory location.

$$y = x + \text{Cols} \times (i - 1) + (j - 1)$$

If the first element occupies the location 1000, the target element occupies the location 1018.