

10.2 Arrays

Name: _____ Class: _____ Date: _____

Total: 35 marks

KEY WORDS bounds • linear search • index • element • arrays

Objective

An **array** lets a weather station store, search and process many readings using loops.

Learn: declarations, bounds, tracing, totals, counts, searching and array processing. Continue with **10.2.1** for bubble sort and insertion/deletion, then **10.2.2** for combined exam practice.

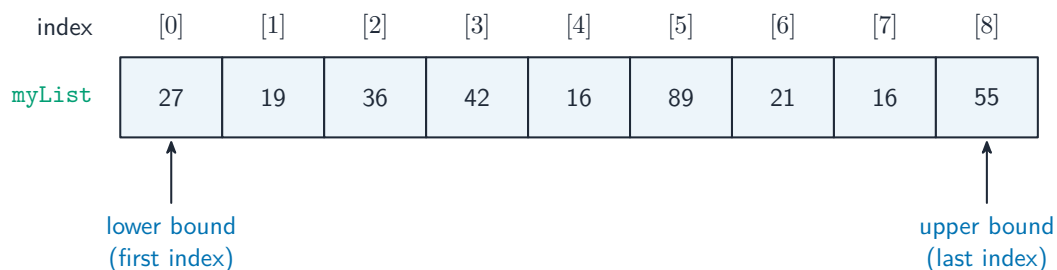
1 Key knowledge and worked examples

■ Elements, indices and bounds

An array holds a fixed number of same-type **elements**, each reached by an integer **index**. **Lower bound** and **upper bound** are the first and last valid indices, both included. A **dimension** is one indexing direction.

```
DECLARE Scores : ARRAY[0:4] OF INTEGER
Scores[3] ← 75
```

`Scores[3]` is the fourth element; `Scores[5]` is invalid. Number of elements = upper bound – lower bound + 1, so this array holds $4 - 0 + 1 = 5$ elements. Indices need not start at 1.



Use **1-D** for a sequence; **2-D** for two directions, such as month and shop. Declare all bounds and the type. Declaration does not initialise values.

■ Initialise, input, total and count

Worked example: enter five marks, output the mean and count marks at least 50. Replace INPUT Mark[Index] with Mark[Index] ← 0 to initialise elements.

```
DECLARE Mark : ARRAY[1:5] OF INTEGER
DECLARE Index, Total, PassCount : INTEGER
Total ← 0
PassCount ← 0
FOR Index ← 1 TO 5
    INPUT Mark[Index]
    Total ← Total + Mark[Index]
    IF Mark[Index] >= 50 THEN
        PassCount ← PassCount + 1
    ENDIF
NEXT Index
OUTPUT "Mean: ", Total / 5, ", passes: ", PassCount
```

For 40, 50, 70, 20, 80: total 260, mean 52, count 3. A total adds **values**; a count adds **one**. Initialise before the loop; output afterwards. Trace updates in statement order.

■ Largest value and its position

For non-empty integer array A[1:n], start with its first value. Starting at zero fails for all-negative data.

```
DECLARE Index, Largest, Position : INTEGER
Largest ← A[1]
Position ← 1
FOR Index ← 2 TO n
    IF A[Index] > Largest THEN
        Largest ← A[Index]
        Position ← Index
    ENDIF
NEXT Index
OUTPUT "Largest: ", Largest, ", index: ", Position
```

For -8, -2, -2, -5, the maximum is -2 at index 2. > keeps the first tie; >= keeps the last. Reverse the comparison for a minimum.

■ Two dimensions: rows, columns and totals

		<i>column index</i>			
		1	2	3	4
<i>row index</i>	1				
	2			99	
	3				

← Grid[2,3] ← 99

```
DECLARE Grid : ARRAY[1:3, 1:4] OF INTEGER
Grid[2, 3] ← 99
```

There are $3 \times 4 = 12$ elements. Here the first index is row, the second column; follow the question's convention.

Worked example: Grid is filled. Output each row total and the whole-grid total.

```
DECLARE Row, Col, RowTotal, GrandTotal : INTEGER
GrandTotal ← 0
FOR Row ← 1 TO 3
  RowTotal ← 0
  FOR Col ← 1 TO 4
    RowTotal ← RowTotal + Grid[Row, Col]
  NEXT Col
  OUTPUT "Row ", Row, ": ", RowTotal
  GrandTotal ← GrandTotal + RowTotal
NEXT Row
OUTPUT GrandTotal
```

Reset RowTotal per row. For column totals, put columns in the outer loop. For one row, fix the row index and loop over columns.

■ Linear search: return the first match

A **linear search** checks in order. This function uses global integer array `A[1:n]`, with $n \geq 1$. A function returns a value; a procedure has no return value.

```
FUNCTION FindFirst(Target : INTEGER) RETURNS INTEGER
  DECLARE Index, FoundAt : INTEGER
  Index ← 1
  FoundAt ← -1
  WHILE Index <= n AND FoundAt = -1
    IF A[Index] = Target THEN
      FoundAt ← Index
    ENDIF
    Index ← Index + 1
  ENDWHILE
  RETURN FoundAt
ENDFUNCTION
```

For 4, 7, 7, 2, searching for 7 returns 2; 9 returns -1. Stop at the first match. Both 0 and -1 could mean failure here: choose a value outside the **declared** bounds.

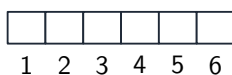
■ Capacity and used elements

Capacity is the number allocated; **Count** is the number used. In `Data[1:20]` with `Count = 3`, process only 1:3. Before adding, check `Count < 20`, increment it, then assign `Data[Count]`.

2 Practice

Now apply the methods above.

2.1 Explain the array terms **element** and **bounds**. [2]

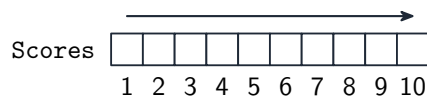


2.2 Declare a 1-D array **Scores** of **10** integers, then set the **4th** element to 75. [2]



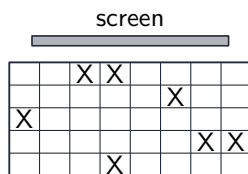
2.3 Write a FOR loop to output **all 10** elements of **Scores**.

[2]



2.4 A cinema has **5** rows and **8** seats per row, each marked 'F' (free) or 'X' (taken). Declare a suitable **2-D** array **Seats**.

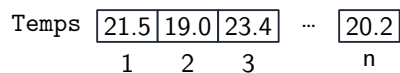
[2]



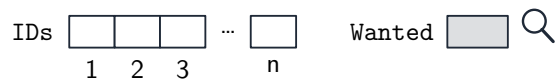
3 Exam-style questions

3.1 Array **Temps[1:n]** holds **n** real temperatures, where **n** ≥ 1 . Write pseudocode to find and output the **largest** value.

[4]



3.2 Write pseudocode for a **linear search** of `IDs[1:n]` for a value `Wanted`. Output the position of the **first** match, or "Not found" if it is absent. Assume `n >= 1`; stop searching once a match is found. [4]



3.3 Global array `Sales[1:12, 1:4]` contains real monthly sales. Row = month; column = shop. Write pseudocode to output the annual total for **each** shop and the number of months in which that shop's sales were above 10000. Declare local variables. [6]



3.4 A weather station stores the last 24 hourly temperature readings in a 1D array `Reading` of type `REAL`, indexed from 1 to 24.

(a) **Complete** procedure `Store()`, which inputs 24 real readings and stores them in consecutive array elements. [5]

```
PROCEDURE Store()
    DECLARE Index : .....
    FOR Index ← 1 TO .....
        ..... Reading[.....]
        ..... Index
    ENDPROCEDURE
```

(b) Explain why a running maximum should start at `Reading[1]` rather than 0. [1]



Reading

--	--	--

 ...

--

1 2 3 24

(c) A function `FindFirstAbove(Limit)` returns the **index** of the first reading above `Limit`, or -1 if there is none. Explain why -1 is suitable for “not found”, and whether 0 would also be suitable for this declaration. [2]

(d) **Write** pseudocode for the function `FindFirstAbove()`, which takes the real parameter `Limit`. Stop searching as soon as a reading above `Limit` is found. [5]