Arrays

Ref As Pseudocode

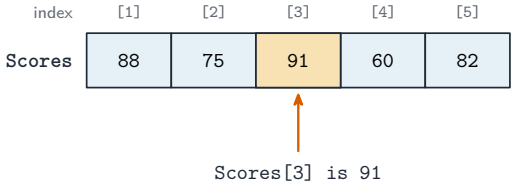
One-dimensional arrays

An **array** stores many values of the **same data type** under one name. Each value is an **element**, and you reach it by its **index**: its position number. An array has a fixed size that does not change.

```
DECLARE <identifier> : ARRAY[<lower>:<upper>] OF <data type>
```

```
DECLARE Scores : ARRAY[1:5] OF INTEGER
Scores[1] ← 88
Scores[2] ← 75
Scores[3] ← 91
Scores[4] ← 60
Scores[5] ← 82
OUTPUT "The third score is ", Scores[3]
```

- `[1:5]` gives the **lower bound** 1 and the **upper bound** 5. So this array has 5 elements.
- Cambridge usually starts at **1**. Always write the lower bound, because some languages start at 0.
- The index goes in **square brackets**: `Scores[3]`.
- The index can be any expression that gives a whole number: `Scores[i]`, `Scores[n + 1]`.



Scores[1:5]: five boxes, one name, each reached by its index

Use a loop to visit every element

A `FOR` loop with the counter as the index visits each element in turn. This is the most common pattern in Paper 2.

```
DECLARE Names : ARRAY[1:4] OF STRING
DECLARE i : INTEGER
Names[1] ← "Ali"
Names[2] ← "Bo"
Names[3] ← "Chen"
Names[4] ← "Dara"
FOR i ← 1 TO 4
   OUTPUT i, ": ", Names[i]
NEXT i
```

You **cannot** fill a group of elements in one statement. `Names[1 TO 4] ← ""` is not allowed. Use a loop:

```
DECLARE Names : ARRAY[1:30] OF STRING
DECLARE Index : INTEGER
FOR Index ← 1 TO 30
   Names[Index] ← ""
NEXT Index
OUTPUT "All 30 names cleared"
```

You **can** copy a whole array to another array of the same size and type in one line: `SavedGame ← Board`.

Common mistakes

- An index outside the bounds, such as `Scores[6]` or `Scores[0]` for `ARRAY[1:5]`. This is a run-time error.
- Using round brackets: `Scores(3)`. Arrays use square brackets.
- Forgetting the data type: every array needs `OF <data type>`.

Two-dimensional arrays

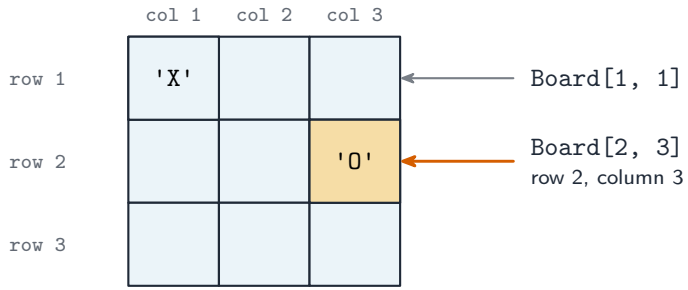
A **two-dimensional** array is like a table, with **rows** and **columns**. You need two indices to reach one element.

```
DECLARE <identifier> : ARRAY[<lower1>:<upper1>, <lower2>:<upper2>] OF
↪  <data type>
```

```
DECLARE Board : ARRAY[1:3, 1:3] OF CHAR
Board[1, 1] ← 'X'
Board[2, 3] ← 'O'
OUTPUT "Row 2, column 3 holds ", Board[2, 3]
```


exercises.lol exercises.com.cn

- By convention the first index is the **row** and the second is the **column**.
- Both indices go inside **one** pair of square brackets, separated by a comma: `Board[2, 3]`, not `Board[2][3]`.



Board[1:3, 1:3]: the first index picks the row, the second the column

Nested loops for a 2D array

To visit every element, use one loop for the rows and another loop inside it for the columns.

```
DECLARE Amount : ARRAY[1:2, 1:3] OF INTEGER
DECLARE Row : INTEGER
DECLARE Column : INTEGER
DECLARE RowTotal : INTEGER
Amount[1, 1] ← 4
Amount[1, 2] ← 7
Amount[1, 3] ← 1
Amount[2, 1] ← 9
Amount[2, 2] ← 2
Amount[2, 3] ← 5
FOR Row ← 1 TO 2
   RowTotal ← 0
   FOR Column ← 1 TO 3
      RowTotal ← RowTotal + Amount[Row, Column]
   NEXT Column
   OUTPUT "Total for row ", Row, " is ", RowTotal
NEXT Row
```

`RowTotal ← 0` is **inside** the outer loop, so each row starts again from zero.

Common mistakes

- Swapping the indices: `Board[Column, Row]` when the array was set up as rows first.

exercises.lol exercises.com.cn

- Setting `RowTotal ← 0` **before** the outer loop. Then the second row's total also contains the first row.

Totals, counts, largest and smallest

Most array questions ask for one of a few patterns. Learn them as shapes.

Total and average

```
DECLARE Marks : ARRAY[1:5] OF INTEGER
DECLARE Total : INTEGER
DECLARE i : INTEGER
Marks[1] ← 56
Marks[2] ← 71
Marks[3] ← 48
Marks[4] ← 90
Marks[5] ← 65
Total ← 0
FOR i ← 1 TO 5
   Total ← Total + Marks[i]
NEXT i
OUTPUT "Total: ", Total
OUTPUT "Average: ", Total / 5
```

Set the **total** to 0 **before** the loop, add inside it, and divide **after** it.

Count the elements that pass a test

```
DECLARE Marks : ARRAY[1:5] OF INTEGER
DECLARE PassCount : INTEGER
DECLARE i : INTEGER
Marks[1] ← 56
Marks[2] ← 71
Marks[3] ← 48
Marks[4] ← 90
Marks[5] ← 65
PassCount ← 0
FOR i ← 1 TO 5
   IF Marks[i] >= 60 THEN
      PassCount ← PassCount + 1
   ENDIF
NEXT i
OUTPUT PassCount, " marks are 60 or more"
```

Largest and smallest

Start with the **first element**, not with 0. Then compare every other element with it.

```
DECLARE Marks : ARRAY[1:5] OF INTEGER
DECLARE Highest : INTEGER
DECLARE Lowest : INTEGER
DECLARE i : INTEGER
Marks[1] ← 56
Marks[2] ← 71
Marks[3] ← 48
Marks[4] ← 90
Marks[5] ← 65
Highest ← Marks[1]
Lowest ← Marks[1]
FOR i ← 2 TO 5
   IF Marks[i] > Highest THEN
      Highest ← Marks[i]
   ENDIF
   IF Marks[i] < Lowest THEN
      Lowest ← Marks[i]
   ENDIF
NEXT i
OUTPUT "Highest ", Highest, ", lowest ", Lowest
```

Why not start `Lowest` at 0? Because no mark is below 0, so `Lowest` would stay 0 and be wrong. Starting with `Marks[1]` always works, even with negative numbers.

Practice

1. An array `Temps[1:7]` holds a week of temperatures. Output the number of the day with the highest temperature (not the temperature itself).

<details> <summary>Model answer</summary>

exercises.lol exercises.com.cn

```
DECLARE Temps : ARRAY[1:7] OF INTEGER
DECLARE HottestDay : INTEGER
DECLARE Day : INTEGER
Temps[1] ← 21
Temps[2] ← 25
Temps[3] ← 19
Temps[4] ← 28
Temps[5] ← 24
Temps[6] ← 28
Temps[7] ← 22
HottestDay ← 1
FOR Day ← 2 TO 7
   IF Temps[Day] > Temps[HottestDay] THEN
      HottestDay ← Day
   ENDIF
NEXT Day
OUTPUT "Hottest day: ", HottestDay
```

It prints 4. Store the **index** of the best element so far, not its value. Day 6 is also 28, but `>` does not replace an equal value, so the first hottest day is kept.

</details>

Common mistakes

- Setting `Total ← 0` inside the loop. Then it restarts on every pass.
- Dividing by the wrong number for the average. Divide by how many values were added.
- Using `>=` when the question wants the **first** best value. On a tie `>=` replaces the stored value again, so you end up with the **last** best one.