Teaching guide: Data structures (Two-dimensional data structures)

This resource will help with understanding data structures and the use of two-dimensional data structures. It supports Section 3.2.6 of our current GCSE Computer Science specification (8520). The guide is designed to address the following learning outcomes:

- Appreciate that an array is a linear structure and an array of arrays can be seen as a two-dimensional array.
- Know how to create, retrieve and update data in two-dimensional arrays.
- Combine records and arrays to create two-dimensional complex data types (not on the specification).

Arrays

As we have already seen arrays are a reference to the start of a block of memory where an integer is then used as an index to count along the memory slots. By convention only homogeneous data types are stored in those memory slots so we will have arrays of integers, or arrays of Booleans and so on but not an array that consists of, for example, some characters and some reals.

An array itself can be thought of as a data type (a number that points to a part of memory), lots of arrays all have the same type as they are all pointers to memory and so it is entirely within the rules to have an array of arrays.

A simple way to think of this is like the folders used to keep files organised on your computer. You can click on a folder and you can see all of the files in that folder, but if you had a lot of files, or alternatively you're just very organised, you might click on a folder that is in turn just made up of more folders, each one of these holds different files. Each folder points the way to another place where things or stored, either files or more folders. An array points the way to more data so an array of arrays is a series of pointers where each one leads to a different series of data.

Array of arrays

We can use an array of arrays to represent two-dimensional spaces such as a computer screen made up of different pixels (a single point of colour in an image) or a chess board made up of different squares. To simplify the process, start using a 3x3 grid used to play noughts and crosses.

Firstly create an array to represent the top row (rows will have the characters `'X'` and `'O'` to show if a player has been in that square or the space character `' '` if the square is empty):

```
row_1 ← [' ', ' ', ' ']
```

The second and third rows are created in the same way:

```
row_2 ← [' ', ' ', ' ']
row_3 ← [' ', ' ', ' ']
```

Place these rows on top of each other by creating a new array that has `row_1` as its first element, `row_2` as its next element and `row_3` as its last:

```
grid ← [row_1, row_2, row3]
```

It may not be immediately obvious why this represents a 3x3 grid so instead of creating the grid out of three existing arrays we'll create them all at the same time and arrange them on different lines:

```
grid ← [ [' ', ' ', ' '],
         [' ', ' ', ' '],
         [' ', ' ', ' '] ]
```

How can the individual cells in this array of arrays be accessed? For example, how could the `'X'` character be assigned to the cell shown below:

```
[ [' ', ' ', ' '],
  [' ', ' ', ' '],
  [' ', ' ', ' '] ]
```

The character is in the third array of grid, so the first part of the expression will be:

```
grid[2]
```

This narrows it down to this array:

```
[' ', ' ', ' ']
```

And it is obviously the first element of this array, so the two are put together to give:

```
grid[2][0]
```

So in answer to the question, the `'X'` character is assigned to that cell using the statement:

```
grid[2][0] ← 'X'
```

This is a possible configuration of the grid after two goes by each player:

```
[ [' ', 'X', ' '],
  [' ', 'O', ' '],
  ['X', 'O', ' '] ]
```

And this is one possible sequence of instructions that could have achieved this:

```
grid[2][0] ← 'X'
grid[1][1] ← 'O'
grid[0][1] ← 'X'
grid[2][1] ← 'O'
```

Two-dimensional arrays do not need to be squares, as long as every element in the array is itself an array there is no limitation on the length that they can be.

The table below shows the distances travelled, in metres, by four different paper planes (not all four planes had four flights):

	plane 1	plane 2	plane 3	plane 4
flight 1	3.2	1.2	7.4	3.7
flight 2	4.1	2.4	9.1	6.2
flight 3	1.6	1.7	-	5.7
flight 4	-	1.9	-	4.2

Each plane's distance could be an element in an array (for example plane 2's array would be `[1.2, 2.4, 1.7, 1.9]`) but they would not all be the same size. If we were to put each of the four plane's distance arrays into one array then we would have a jagged array (an arrays of different sized arrays):

```
flights ← [ [3.2, 4.1, 1.6],
            [1.2, 2.4, 1.7, 1.9],
            [7.4, 9.1],
            [3.7, 6.2, 5.7, 4.2] ]
```

To calculate the mean average flight distance of all the flights then we might (incorrectly) write:

```
total_distance ← 0
FOR i ← 0 TO LEN(flights)-1
   total_distance ← total_distance + flights[i]
ENDFOR
mean_flight ← total_distance / LEN(flights)
```

This would calculate the mean average correctly if all of the flight distances were in one longer array but in this data structure the expression `flights[i]` is another array, not a real value.

To carry out the calculation correctly in this instance we need to further iterate over each element in the array given by `flights[i]` by nesting another `FOR` loop inside our existing loop:

```
total_distance ← 0
FOR i ← 0 TO LEN(flights)-1
```

```
   FOR j ← 0 TO LEN(flights[i])-1
      total_distance ← total_distance + flights[i][j]
   ENDFOR
ENDFOR
mean_flight ← total_distance / LEN(flights)
```

The final error in the program is in the last line as although `total_distance` is now the correct sum of all the flights, the program still divides by length of flights which is `4` (because it contains four arrays) and we should be dividing it by the total number of flights. To solve this we could introduce a new counter that goes up by one every time a new flight is encountered:

```
total_distance ← 0
total_flights ← 0
FOR i ← 0 TO LEN(flights)-1
   FOR j ← 0 TO LEN(flights[i])-1
      total_distance ← total_distance + flights[i][j]
      total_flights ← total_flights + 1
   ENDFOR
ENDFOR
mean_flight ← total_distance / total_flights
```

If you need to iterate over every element in a multi-dimensional array you will almost certainly need to use nested loops – one loop for every dimension - so an array of arrays of arrays would need a loop within a loop within a loop! The syntax used to reference, say, the third element in the second array of the sixth array of an array called `A` would be `A[5][1][2]`. For the purposes of the specification though only knowledge of two-dimensional arrays is required.

Array of records

Finally let's return to the problem of representing the swimmers' times in the first Teaching guide section on arrays. This is where the solution was left:

```
times_filho ← [21.80, 21.54, 21.59]
times_fratus ← [21.82, 21.63, 21.61]
times_jones ← [21.95, 21.54, 21.54]
times_manaudou ← [22.09, 21.80, 21.34]
```

The main problems are that we have four different arrays which makes comparison between them difficult, also we have to infer what the numbers represent which makes reading and understanding the code hard and lastly – because arrays are homogeneous data structures – they cannot include the names of the swimmers.

Each collection of data includes a swimmer's name (a string) and three times (reals) – we need a heterogeneous data structure to include all of this so we'll use a record:

```
RECORD Freestyle_50
   name
   heat
   semi
   final
ENDRECORD
```

We could capture all of the information about C. Jones like so:

```
c_jones ← Freestyle_50('C. Jones', 21.95, 21.54, 21.54)
```

Now if we wanted to know the difference between Jones' final time and his heat time we could write this statement which is very easy to decipher:

```
time_difference ← c_jones.final - c.jones.heat
```

Continuing to assign three new records to completely reflect the swimmers data we have:

```
c_filho ← Freestyle_50('C. Cielo Filho', 21.80, 21.54, 21.59)
b_fratus ← Freestyle_50('B. Fratus', 21.82, 21.63, 21.61)
f_manaudou ← Freestyle_50('F. Manaudou', 22.09, 21.80, 21.34)
```

Our data is certainly easier to access now and we have recaptured the information on the swimmer's names but comparison between the records is still awkward as we have to write out each identifier in our code. The solution? Each of these records is the same type (they are all `Freestyle_50` records) so we can legitimately put them into an array like so:

```
swimmers ← [c_jones, c_filho, b_fratus, f_manaudou]
```

In fact, we could completely miss out the part of the code where each of the four records are assigned to a different variable and write the data directly into the array swimmers like so:

```
swimmers ← [Freestyle_50('C. Jones', 21.95, 21.54, 21.54),
            Freestyle_50('C. Cielo Filho', 21.80, 21.54, 21.59),
            Freestyle_50('B. Fratus', 21.82, 21.63, 21.61),
            Freestyle_50('F. Manaudou', 22.09, 21.80, 21.34)]
```

If we wanted to calculate the time difference in Jones' final and heat – just like earlier – but accessing the values through the array swimmers we would have to alter our code to:

```
time_diff ← swimmers[0].final - swimmers[0].heat
```

At first glance this looks like we've just added a layer of complexity to the data, but by keeping all of the data together in a complex, 2-dimensional structure we have gained the ability to answer the questions asked when this data was introduced without having to know anything about the data itself.

For example, one of the questions was, 'was the fastest swimmer in the heats also the fastest in the final?' Our answer in structured English could be:

1. Find the position in the array swimmers of the fastest in the heats
2. Find the position in the array swimmers of the fastest in the final
3. Compare the two positions:
    a. if they are the same then output yes with some relevant details
    b. else output no with some relevant details

Step 3 looks fairly simple to do but steps 1 and 2 both need more thought. Fortunately, the answer to both steps is going to be almost exactly the same apart from step 1 is referencing the heat field of swimmers and step 2 is referencing the final field. The solution to step 1 could be:

1. assume the first swimmer's heat time is the fastest and remember the position of this swimmer
2. compare with the next swimmer's heat time in the array
    a. if the next swimmer's heat time is less then set the position of the fastest heat time to be this position
3. repeat step 2 for all the remaining swimmers in the array

(The solution to all of this problem is very similar to one answered earlier about finding out the name of the oldest student – take a look again at this solution to see the similarities.)

Assuming that the array swimmers has already been declared and assigned values then this algorithm could be written:

```
# find the position of the fastest heat
fastest_heat ← swimmers[0].heat
fastest_heat_position ← 0
FOR i ← 1 TO LEN(swimmers)-1
   IF swimmers[i].heat < fastest_heat THEN
      fastest_heat_position ← i
      fastest_heat ← swimmers[i].heat
   ENDIF
ENDFOR

# find the position of the fastest final
fastest_final ← swimmers[0].final
fastest_final_position ← 0
FOR i ← 1 TO LEN(swimmers)-1
   IF swimmers[i].final < fastest_final THEN
      fastest_final_position ← i
      fastest_final ← swimmers[i].final
   ENDIF
```

```
ENDFOR[1]

# compare the two positions and output relevant details
IF fastest_heat_position = fastest_final_position THEN
   OUTPUT 'Yes'
   OUTPUT swimmers[fastest_heat_position].name
   OUTPUT 'was fastest in both the heat and final'
ELSE
   OUTPUT 'No'
   OUTPUT swimmers[fastest_heat_position].name
   OUTPUT 'was fastest in the heats and'
   OUTPUT swimmers[fastest_final_position].name
   OUTPUT 'was fastest in the final'
ENDIF
```

This is the most complex algorithm we have created so far but it is only built out of a small handful of things: variables, data structures, output, loops and iteration.

[1] As both of the blocks of code that find the fastest heat and the fastest final both loop over the same data structure the same number of times then this code could have been combined into one loop (it is shown separately to make it more obvious how the code works).