Teaching guide: Data structures (Arrays)

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

- recognise the need for data structures in code
- understand how arrays can be implemented in memory
- know how to create, retrieve and update data in arrays.

Data structures

Data structures allow programmers to store together large amounts of related data – the only alternative is to use a large number of variables. The following examples show how arrays can be used to store separate data items using one identifier and an index.

Example 1: Flipping a coin

When you flip a coin, it can either land as heads or tails. If you flip it often enough, the number of times you get a head and the number of times you get a tail will probably be similar. If we wanted to check this with a coin we might flip it 6 times and record the result each time (although instead of using 'heads' and 'tails' we can use a Boolean interpretation: `True` for heads and `False` for tails). A program could look like this:

```
flip1 ← True
flip2 ← False
flip3 ← False
flip4 ← True
flip5 ← True
flip6 ← True
```

If we wanted to use these variables to find the number of times a coin was a head, a program something like this could help:

```
number_of_heads ← 0
IF flip1 THEN  # flip1 is equivalent to flip1 = True
  number_of_heads ← number_of_heads + 1
ENDIF
IF flip2 THEN
  number_of_heads ← number_of_heads + 1
ENDIF
IF flip3 THEN
```

```
  number_of_heads ⟵ number_of_heads + 1
ENDIF
# and so on until…
IF flip6 THEN
   number_of_heads ← number_of_heads + 1
ENDIF
```

(You might expect the first Boolean expression to be flip1 = True but if flip1 is False then the statement flip1 = True also evaluates to False, whereas if flip1 is True then the statement flip1 = True also evaluates to True (because True is equal to True).

There are a number of problems with this approach:

- the large amount of code needed for something quite simple
- the need for the programmer to keep track of a large number of very similar variables when they are developing their program
- the fact that if the conditions for the program change (for instance running 30 tests instead of 6) the program will have to be substantially rewritten.

To overcome this, programmers use data structures which are similar to variables but have important differences, an array is a type of data structure. A variable is a name that points to a location in memory; an array is similar but instead of pointing to just one memory slot it points to the start of a block of memory, this is not true for all languages, but it helps to think of them like this regardless.
We can create an array in a very similar way to declaring a variable but the assignment of values uses new syntax. There is some controversy surrounding the decision to say the first element in the list is accessed using the number 0 – some programming languages start counting the index from 1 instead of 0 so for example, using this logic, the third element would have index 3.

This is our 6 coin flips rewritten in an array, called `coin_flips`:

```
coin_flips ← [True, False, False, True, True, True]
```

Getting to the values in the array is straightforward, we use the name of the array followed by the location of the value in the array. For example, the first element in the array (the `True` just after the `[` symbol (because array elements are read from left to right) is accessed using this syntax.

```
coin_flips[0]
```

The second element is accessed in the same way:

```
coin_flips[1]
```

And so on until the last element is accessed using:

```
coin_flips[5]
```

Array syntax varies across languages as does the name of the structure (for instance Python's lists, although not identical to arrays, can be used for our purposes in very similar ways).

Another advantage of using arrays is that we can rely on the program to know how many elements are in the array (its length). To find this out we can use the length subroutine which could be written as:

```
LEN(coin_flips)
```

And can be read as, 'what is the length of the array called coin_flips'. This is an example of a subroutine, at the moment all we need to know is that this expression evaluates to the integer `6`, and so another way to access the last element in the list to replace the index of the value `5` with the index of the expressions `LEN(coin_flips)-1`:

```
coin_flips[LEN(coin_flips)-1]
```

If we wanted to change the first element from `True` to `False` then we use the assignment arrow familiar from assigning values to variables.

```
coin_flips[0] ← False
```

And the array would now start:

```
[False, False, False, True, ...]
```

Each element in the array can be thought of as an individual variable, it can be assigned a value, accessed and updated in the same way, instead of having its own individual identifier it instead has a combination of the array identifier and an integer location.

Example 2: Swimmers

The following example uses the race times of the top four finalists in the London 2012 Olympics Men's 50m Freestyle swimming event. Again it shows the limitations of using excessive variables instead of data structures for data that naturally belongs in a group:

	Heats	Semi-Finals	Final
C. Cielo Filho	21.80	21.54	21.59
B. Fratus	21.82	21.63	21.61
C. Jones	21.95	21.54	21.54
F. Manaudou	22.09	21.80	21.34

There are twelve distinct real values in the table that could be recorded in a program using variables:

```
filho_heats ← 21.80
fratus_heats ← 21.82
```

```
jones_heats ← 21.95
manaudou_heats ← 22.09
filho_semis ← 21.54
fratus_semis ← 21.63
jones_semis ← 21.54
manaudou_semis ← 21.80
filho_final ← 21.59
fratus_final ← 21.61
jones_final ← 21.54
manaudou_final ← 21.34
```

This data can be used to answer the following questions:

- Was the fastest swimmer in the heats also the fastest in the final?
- What was B. Fratus' slowest time?
- Did anyone get slower between the heats, the semi-finals and the final?

For all of these questions, it is easier to find the answer using the data presented in the table rather than as a list of variables because it is easier to find or interpret data when it's presented in an ordered way. In addition to being able to look at the data more easily, it is also easier to add another swimmer to the table than it is to write out three more variables and assign each with the respective value. It is also easier to program with ordered data because instead of treating every item of data individually we can refer to the group to which it belongs.

We can view the table another way that leads itself to how this could be represented in code. We can make an assumption that the first column in the table after the swimmers name will be the heats time, the second column will be the semi-final time and the third column will be the final time. The revised table will look like:

C. Cielo Filho	21.80	21.54	21.59
B. Fratus	21.82	21.63	21.61
C. Jones	21.95	21.54	21.54
F. Manaudou	22.09	21.80	21.34

Finally, instead of the swimmers' names we will use a descriptive variable identifier instead. The table can now be rewritten as:

`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 12 separate variables have now been compacted into four different structures that each hold three different values. The pseudo-code we will use to create this in code is:

```
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]
```

If we wanted to write an expression to find out the final time of F. Manaudou from our data structures we would use the name of the data structure and the index of the value within it; the final time is the third element of data within the structure and so the expression would be:

```
times_manaudou[2]
```

This is still not an ideal way to represent this information, it requires the extra knowledge that the first element represents the heats, the second element represents the semi-finals and the third is the final. This knowledge may not be obvious to everyone reading the program. Also, the data is still spread out over four separate arrays which makes comparison between the swimmers awkward as the array identifiers all need to be known. A further solution will be given at the end of the section in the 2-dimensional arrays resource on our website.

Arrays and FOR-loops

We now have the syntax for creating and assigning values in an array, combining this with loops, we can write expressive programs that allow us to deal with the array as a single entity instead of all the elements separately. To see how this works take a look at this code to work out the sum total of ages:

```
ages ← [14, 16, 12, 17, 14]
total_ages ← ages[0] + ages[1] + ages[2] + ages[3] + ages[4]
```

This long expression involves a repeating pattern, we can use a loop to simplify this code. The second line of code above is logically identical to initiating `total_ages` to the value `0` and then adding to this all the values in the array ages in turn. That is:

```
ages ← [14, 16, 12, 17, 14]
total_ages ← 0
total_ages ← total_ages + ages[0]
```

```
total_ages ← total_ages + ages[1]
total_ages ← total_ages + ages[2]
total_ages ← total_ages + ages[3]
total_ages ← total_ages + ages[4]
```

Displaying it in this way makes it easy to see the repeating pattern in the code where the difference in the five statements is the array index which is an integer that starts at `0` and increases by `1` until it reaches `4`.

Recall that the loops we have covered are condition-controlled (`WHILE` and `REPEAT-UNTIL`), which repeat for an unknown number of times until a Boolean condition is met, and `FOR` which repeats for a specified number of times. We know how long our array is so a `FOR` loop is the obvious choice in this situation. The code can be rewritten as:

```
ages ← [14, 16, 12, 17, 14]
total_ages ← 0
FOR i ← 0 TO 4
   total_ages ← total_ages + ages[i]
ENDFOR
```

There is a final amendment we can make to this code to make it more general, currently this works because there are exactly 5 elements in the array, but if we change the array to include another element then we will also have to change this:

```
FOR i ← 0 TO 4
```

to this:

```
FOR i ← 0 TO 5
```

Previously we mentioned the length subroutine that evaluates to the number of elements in an array. One less than this number, because our arrays start indexing at `0`, is the number that goes after the `TO` in our `FOR` loop, so this final rewriting of the program now works regardless of how many values there are in ages:

```
ages ← [14, 16, 12, 17, 14]
total_ages ← 0
FOR i ← 0 TO LEN(ages)-1
   total_ages ← total_ages + ages[i]
ENDFOR
```

We can write similar programs for different purposes. To calculate the mean average of the ages then we extend the last program by one line (note that `total_ages` is divided by `LEN(ages)` instead of the value `5` to ensure that this program still works if the size of the array changes:

```
ages ← [14, 16, 12, 17, 14]
total_ages ← 0
FOR i ← 0 TO LEN(ages)-1
```

```
    total_ages ← total_ages + ages[i]
ENDFOR
mean_average ← total_ages / LEN(ages)
```

To modify the program to find the oldest age in our array then we need to think more carefully about the solution. In structured English we could:

1. assume that the first age in the array is the oldest
2. compare the oldest with the next element in the array
    - if the next element is greater then set oldest to be this element
3. repeat step 2 for every element in the array

Step 3 involves iterating over every element in our array so we should be thinking about a `FOR` loop. However, step 1 happens outside of the loop and so our `FOR` loop should start at the next element's index which will be `1`. The program could look like this:

```
oldest ← ages[0]
FOR i ← 1 TO LEN(ages)-1
    IF ages[i] > oldest THEN
        oldest ← ages[i]
    ENDIF
ENDFOR
```

Using a `FOR` loop to iterate over an array is a very common programming technique. Some languages also have a looping structure generally called a `FOREACH` that instead of iterating over integers which can be used as the indices of an array it instead iterates over the actual elements themselves.

When to use an array

Arrays are by no means the only data structure available to programmers, they are considered a foundational type of data structure that can be used as a basis for other structures. Arrays appear all the time in programs. When a programmer wants to store related values together especially when their order is important. Some general guidelines may help:

- use a data structure to store similar items of data
- use an array if it is important to keep this data in a linear sequence (ie the data should be ordered in a straight line)
- if you need to iterate over a whole array your first thought should be a `FOR` loop.