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 GCSE Computer Science specification (8525).

Data structures

Data structures allow programmers to store together 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 normal 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 will use a Boolean interpretation: `True` for heads and `False` for tails). A program might 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 several problems with this approach:

- the large amount of code needed for something quite simple
- the need for the programmer to keep track of many 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 array data structures. A simple variable is a name that refers to a location in memory where its value is stored; while an array variable instead refers to the start of a block of memory locations.

We can create an array in a very similar way to declaring a variable but the retrieval, assignment and updating of values uses new syntax.

This is our six 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 straight forward, 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]
```

Note that the first element is at position `0` in the array. Some languages start at position `1` but unless stated otherwise in an exam paper you may assume that the first element of the array is 'at' location `0`.

The second element is accessed in the same way:

```
coin_flips[1]
```

and so on until the last (sixth) element is accessed using:

```
coin_flips[5]
```

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

To find out the number of elements in the array we use the **`LEN`** subroutine:

```
LEN(coin_flips)
```

which returns the length of the array called `coin_flips` in this case the value `6`, and so another way to access the last element in the array is to replace the value `5` with the value of the expression **`LEN`**`(coin_flips) - 1` (remember the `- 1` is because the first element is at location `0`):

```
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 be:

```
[False, False, False, True, True, 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, but instead of having its own individual identifier it instead has a combination of the array identifier and an integer location (known as an *index*).

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 and again shows the limitations of using individual variables instead of an array for data that naturally belongs in a group:

Finalists	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 number 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 questions such as:

- was the fastest swimmer in the heats also the fastest in the final?
- what was B. Fratus's 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 give each one 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 to how this is represented in code. Rather than include the column headings we use the fact that the first column in the table after the swimmer's 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 looks like this:

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 now looks like this:

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 replaced with four different arrays that each hold three different values. The pseudo-code to create this 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 array and the index of the required value within it; the final time is the third element of data within the structure and so the expression would be (remember the first item is at index `0`, the second at index `1` and the third at index `2`):

```
times_manaudou[2]
```

This is still not an ideal way to represent this information since:

- it needs the knowledge that the first element represents the heats, the second element the semi-finals and the third the final, knowledge that won't be obvious to everyone reading the program
- 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
- adding another swimmer is difficult since a new array variable must be created.

A further improved solution will be given at the end of the section in the 2-dimensional arrays resource on our website.

Arrays and FOR-loops

We can now create arrays, assign values to their elements and access those elements. By combining these techniques with loops we can write 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 total of the ages stored in an array called `ages`:

```
ages ← [25, 23, 28, 22]
total_age ← ages[0] + ages[1] + ages[2] + ages[3]
```

The long expression (`ages[0] + ages[1] + …`) involves a repetitive pattern and we can use a loop to simplify it. The second line of code above is identical to initialising `total_age` to the value `0` and then adding to this all the values in the array ages in turn. That is:

```
ages ← [25, 23, 28, 22]
total_age ← 0
total_age ← total_age + ages[0]
total_age ← total_age + ages[1]
total_age ← total_age + ages[2]
total_age ← total_age + ages[3]
```

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 `3`.

The types of loop that are covered in the Teaching Guide – programming concepts (iteration) are condition-controlled (**`WHILE`** and **`REPEAT-UNTIL`**), which repeat for an unknown number of times while or until a Boolean condition is met, and count-controlled (**`FOR`**) which repeats for a specified number of times. We know how long the `ages` array (and so how many times we need to fo through the loop) so a **`FOR`** loop is the obvious choice in this situation. The code can be rewritten as:

```
ages ← [25, 23, 28, 22]
total_age ← 0
FOR i ← 0 TO 3
   total_age ← total_age + 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 four elements in the array, but if we change the array to include another element then we will also have to change the

```
FOR i ← 0 TO 3
```

to this

```
FOR i ← 0 TO 4
```

Previously we mentioned the **LEN** subroutine that returns the number of elements in an array. The number that appears after the **TO** in our **FOR** loop is always one less than this value (because our array starts indexing at 0) so a final rewriting of the program now works regardless of how many values there are in ages:

```
ages ← [25, 23, 28, 22]
total_age ← 0
FOR i ← 0 TO LEN(ages) - 1
   total_age ← total_ages + ages[i]
ENDFOR
```

To calculate the mean (average) of the ages then we extend the last program by one line (note that total_age is divided by **LEN**(ages) instead of the value 4 to ensure that this program still works if the size of the array changes:

```
ages ← [25, 23, 28, 22]
total_age ← 0
FOR i ← 0 TO LEN(ages)-1
   total_age ← total_age + ages[i]
ENDFOR
mean ← total_age / 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 (almost) every element in our array so we should be thinking about a **FOR** loop. However, step 1 happens outside the loop and so our **FOR** loop should start at the next element's index which will be 1. The program then looks like this:

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

Alternatively, since every age must be positive, we could initialize oldest to −1, then iterate over **all** the elements in the array (what initial value would you use for a variable youngest if you wanted to find the youngest swimmer?):

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

This second version does the same thing as the first but means that anyone reading the first doesn't think 'Why does it start at the second element in the array?'

In the first version what would happen if the **FOR** loop started at 0 rather than 1?