Teaching guide: Data structures (records)

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

- Construct bespoke data structures for use with specific problems
- Know how to create, retrieve and update data in records

Complex data types

Arrays are normally used to store data of the same type – in the three examples used in the 'Teaching guide – Data Structure (Arrays)' the elements in the arrays were of type Boolean, real and lastly integer. Some languages will allow array-like structures to have mixed types although we will stick to the convention that arrays are homogeneous data structures, which simply means that all the elements in an array have the same type.

What then if you want to store items of data that are still related but have different types (heterogeneous as opposed to homogeneous)? The answer to this is actually very programming language specific but we will introduce the concept of records before covering how these may be implemented in a range of languages.

A record is a complex data type that is built up of other items of data (each of which of course has its own type). For example, the following information could be associated with works of art:

- title (of type string)
- artist (of type string)
- year it was created (of type integer)
- on public display? (of type Boolean)

Having to create four separate variables for every artwork would go back to the problems in the 'Teaching guide – Data Structure (Arrays)' that were overcome with arrays but, because the data types that comprise this record are different, creating arrays is not a good solution. Records are a way to aggregate (put together) different items of data. The syntax we will use is this

```
RECORD Artwork
   title
   artist
   year_created
```

```
   on_public_display
ENDRECORD
```

This data type is created by the programmer as opposed to being given to them by the inbuilt types of the language (such as integer or array) – for this reason it is called a bespoke data type, meaning that it is created by the programmer for a particular purpose. There are analogies here with clothing – a bespoke suit is one created specifically for one person that is made-to-measure whereas an off-the-peg suit is one of many identical suits that you can walk into a shop and buy. A bespoke suit fits the person better but takes more time and individual skill to create whereas an off-the-peg suit might not do the job quite as well but is available to use straight away and, if you have a problem with it, lots of other people own the same suit and can help you.

The lines of code above define a new complex data structure called `Artwork` but it does not contain any actual data itself. The record definition can be used to create three new artworks like so:

```
art1 ← Artwork('The Scream', 'Munch', 1893, True)
art2 ← Artwork('The Thinker', 'Rodin', 1901, True)
art3 ← Artwork('Seascape Folkestone', 'Turner', 1845, False)
```

A new variable of type `Artwork` is created using the name of the record, namely `Artwork`, followed by the opening and closing parentheses, `(` and `)`. Within the parentheses is a list of the data in the same order as the record definition (`title`, `artist`, `year_created` and `on_public_display`), each one separated by a comma.

The way that the data within these records is accessed normally differs from the technique used with arrays because the definition of a record uses names and not indices to identify the data located within it. For example, if The Scream was suddenly taken out of public display then the `on_public_display` field of `art1` should be set to `False`. The individual items of data within a record are commonly called fields. This could be accomplished by doing this:

```
art1.on_public_display ← False
```

Where the full stop between the name of the artwork and the particular field indicates the particular item of data in the record that is being assigned and which can be read, 'the on public display item of data within the Artwork record art1 is set to False.'

If you wanted to write the code to find out if The Thinker was created before Seascape Folkestone then you could use the `year_created` fields of both of these records:

```
OUTPUT art2.title
IF art2.year_created < art3.year_created THEN
   OUTPUT 'is older than'
ELSE
   OUTPUT 'is not older than'
```

```
ENDIF
OUTPUT art3.title
```

It is helpful to think of every item of data within a record as a separate variable – they can all be accessed and updated in the same way but the identifier syntax is different.

You could put all three of these records together in an array (because although these records contain different types of data within them, all works of art have the same overall type: the bespoke type `Artwork`):

```
art_collection ← [art1, art2, art3]
```

Accessing records within an array looks complex until it is broken down into the array index used to access the record and the field name. To amend the title of the painting `'Seascape Folkestone'` to the more correct `'Seascape, Folkestone'` we would need to access the title field of `art3` which is element `2` of the array `art_collection`.

```
art_collection[2].title ← 'Seascape, Folkestone'
```

More on this in the Teaching guide - Data structure (Two Dimensional Data Structures).

Implementation in programming languages

All of the pseudo-code syntax used in these Teaching guides is designed to be easily implemented in real programming languages but unfortunately the use of records is an exception because their use in different languages varies so greatly. Records are also not to be confused with classes which are a way to aggregate not just data but also actions that can be performed with that data (although, confusingly, a class can be used to represent a record in Python).