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 GCSE Computer Science (8525) specification. The resource is designed to address the following learning outcomes:

- construct 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 structures (arrays)' the elements in the arrays were all of the same type: Boolean (coin flips), real (swimming times) and integer (ages). Some languages **will** allow array-like structures to have mixed types although for this specification all the elements in an array will have the same type.

What if you want to store items of data that are related but have different types?

A record is a 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 structures (arrays)' that were overcome with arrays but, because the data types that comprise this information differ, using arrays is not a solution. Records are a way to aggregate (put together) different items of data (called fields). The syntax we will use is this

```
RECORD Record_identifier
    field1 : <data type>
    field2 : <data type>
    …
ENDRECORD
```

So, a record to hold the details of the latest artwork a gallery has acquired could be defined like this:

```
RECORD Artwork
    title : String
    artist : String
    year : Integer
    on_public_display : Boolean
ENDRECORD
```

The way that the data within these records is accessed 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.

To declare a new variable of this new type we use the following syntax:

```
variableName ← Record_identifier(value1, value2, …)
```

So for example

```
TheScream ← Artwork('The Scream', 'Munch', 1893, True)
```

and the name of the artist for The Scream could be output by executing:

```
OUTPUT TheScream.artist
```

Once created a single field of a record can be updated independently of the rest. For example, if *The Scream* (the painting) were suddenly taken off public display then the `on_public_display` field of `TheScream` (the record) should be set to `False` :

```
TheScream.on_public_display ← False
```

Records to hold the details of two more artworks could be defined in exactly the same way:

```
TheThinker ← Artwork('The Thinker, 'Rodin', 1901, True)
```

and:

```
SeascapeFolkestone ← Artwork('Seascape Folkestone', 'Turner',
                                                    1845, True)
```

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

```
OUTPUT TheThinker.title
IF TheThinker.year < SeascapeFolkestone.year THEN
   OUTPUT 'is older than'
ELSE
   OUTPUT 'is not older than'
ENDIF
OUTPUT SeascapeFolkestone.title
```

It is helpful to think of every item of data within a record as a separate variable since they can each be accessed and updated in the same way but the identifier syntax (`Record_identifier.field_identifier`) is different.

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

```
art_collection ← [TheScream, TheThinker, SeascapeFolkestone]

on_display ← 0
FOR i ← 0 TO LEN(art_collection) - 1
   IF art_collection[i].on_public_display THEN
      On_display ← on_display + 1
   ENDIF
ENDFOR
OUTPUT on_display, ' works of art on public display'
```

Accessing records within an array looks complex until it is broken down into the array *index* used to access one record within the array and then the field *name* to access the field within the record.

As another example changing the title of the painting *Seascape Folkestone* to the more correct *Seascape, Folkestone* we would need to access the title field of the `SeascapeFolkestone` record which is the third element of the array `art_collection` (this means its index is `2`).

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

More on this in ‘Teaching guide - data structures (two dimensional data structures)’.

Implementation in programming languages

All the pseudo-code syntax used in these Teaching guides is designed to be easily implemented in real programming languages. Records should not be confused with classes (in Object Oriented Programming which is **not** on the 8525 specification) which are also a way to aggregate not only data but also actions that can be performed with that data (although in Python the simplest way to create a record is to use a class – see the Python section below).

C#

A record type is defined with the keyword `struct` and this type is then used in a variable declaration, eg

```
// This defines a structure type called Car and must be
// defined outside the main function
struct Car
{
  public string make;
  public string model;
  public double price;
  public int doors;

  public Car(string make, string model,
                    double price, int doors)
  {
      this.make = make;
      this.model = model;
      this.price = price;
      this.doors = doors;
  }
}
```

The subroutine `Car` is called a constructor, and its name must be the same as the structure name, ie `Car` in this case. It's also very important that the `this.` comes before the names of the fields, otherwise the code will not work. In the line `this.make = make` the `this.make` on the left hand side of the = refers to the `make` field of the `Car` record being created, and the `make` on the right hand side refers to the parameter being passed to the function when a record is created. You don't need to call the parameters to the `Car` subroutine the same as the corresponding fields but it helps (and applies to all the programming languages).

```
// This code should go within the main function
// This defines a variable of type Car called myCar

Car myCar = new Car("Ford", "Focus", 1399.99, 5);

Console.WriteLine($"My car is a {myCar.make} {myCar.model}");
Console.WriteLine($"Price £{myCar.price} doors {myCar.doors}");

myCar.price = 1000.0;

Console.WriteLine($"My car is a {myCar.make} {myCar.model}");
Console.WriteLine($"Price £{myCar.price} doors {myCar.doors}");
```

Python

Unlike C# and VB.NET Python has no direct support for records. Classes are usually used for this:

```
class Car:
    def __init__(self, make, model, price, doors):
        self.make = make
        self.model = model
        self.price = price
        self.doors = doors
```

In Python the constructor is always called `__init__` and its first parameter should always be called `self`. As with C# it's very important that the `self.` comes before the names of the fields, otherwise the code will not work. In the line `self.make = make` the `self.make` on the left hand side of the = refers to the `make` field of the `Car` record being created, and the `make` on the right hand side refers to the parameter being passed to the function when a record is created.

```
myCar = Car("Ford", "Focus", 1399.99, 5)

print(f"My car is a {myCar.make} {myCar.model}")
print(f"Price £{myCar.price} doors {myCar.doors}")

myCar.price = 1000.0

print(f"My car is a {myCar.make} {myCar.model}")
print(f"Price £{myCar.price} doors {myCar.doors}")
```

From Python 3.7 onwards students may use *namespaces* or *dataclasses* as the following example shows (the following examples are taken from a discussion on the Computing At School Community which unfortunately is no longer available on the CAS site):

```
import types

car = types.SimpleNamespace()

car.make = "Ford"
car.model = "Focus"
car.price = 1000.0
car.doors = 5

print(f"{car.make}, {car.model}, {car.price}, {car.doors}")
```

or

```
from dataclasses import dataclass

@dataclass
class Car:
    make: str = ""       # Initialising variables is not
                         # essential but the type hints are
    model: str = ""
    price: float = 0.0
    doors: int = 0

car = Car()

car.make = "Ford"
car.model = "Galaxy"
car.price = 12000.0
car.no_of_doors = 5

print(f"{car.make}, {car.model}, {car.price}, {car.doors}")
```

VB.Net

In a similar way to C# the structure type `Car` is defined first (outside your `Main` subroutine), and then used in a `Dim` statement to declare a variable:

```
Structure Car
  Dim make As String
  Dim model As String
  Dim price As Single
  Dim doors As Integer

  Sub New(make As String, model As String,
                        price As Double, doors As Integer)
    Me.make = make
    Me.model = model
    Me.price = price
    Me.doors = doors
  End Sub

End Structure
```

For VB.Net the constructor is always called `New`. As with the other two languages it's very important that the `Me.` comes before the names of the fields, otherwise the code will not work. In the line `Me.make = make` the `Me.make` on the left hand side of the = refers to the `make` field of the `Car` record being created, and the `make` on the right hand side refers to the parameter being passed to the function when a record is created.

```
' This defines a variable of type Car called myCar
Dim myCar As Car

myCar = New Car("Ford", "Focus", 1399.99, 5)

Console.WriteLine($"My car is a {myCar.make} {myCar.model}")
Console.WriteLine($"Price £{myCar.price}, doors
{myCar.doors}")
```