Introduction to Containers

- Containers store multiple pieces of data (*elements*) and allow us to access them
- The string type we have seen is a bit like a container for a collection of **char**s
- Consider 3 containers here
  - Array []
  - Sequence @[]
  - Tuple ()

Arrays - summary

- Fixed-length container, known at compile time.
- Each element must have the same type (“homogenous”)
- Arrays are constructed using []
- Elements are accessed by indexing
    - Usually bounds-checked, OOB is an error.
- Arrays provide iterators over their elements

Defining Arrays Recap

- Use []
- If we can do the whole initialisation with a literal assignment type and size can be inferred (feasible for short arrays)
- Syntax for explicit type declaration is array[size,< element type>]
  - Necessary for something more complex - set it up, then fill it on e.g. a by-element basis
  - Size must be known at compile time, if it is a named variable must be **const**

```
1  # defining arrays
2  var arr1 = [1,2,3] # type and size inferred
3  var arr2 : array[3,int] = [1,2,3] # (unneccesary) explicity size and element type
4  var arr3 : array[3,int]  # uninitialised, but know sz and type - will be all 0's initially
5
```

Array Indexing recap

- Zero based
- OOB is an error
- Low and high functions give low and high indexes

```
1   # indexing
2   var arr = [1,2,3]
3
4   echo arr[0] # zero-based
5   echo arr[1]
6   echo arr[2]
7
8   #echo arr[3] # error - 3 is not in 0..2 (out of bounds)
9
10  # can use a loop in conjunction with indexing
11  for i in 0..2:
12    echo arr[i]
13
14  # with functions provided to explicitly get the indexes for us so we dont make a mistake
15  for i in arr.low..arr.high:
16    echo arr[i]
```

Sequences - Summary

- Similar to arrays in that
  - All elements same type (“homogeneous”)
  - Provide access to those elements (e.g. through indexing, through iterators)
  - low, high, len operators provided
- Different in that they are of variable length
  - Can change during run time
  - Can add elements and delete them
- Sequences are constructed using @[]
- (will cover later) - many functions provided for working on sequences, see **sequtils** module

Defining Sequences Recap

- Use @[]
- If we can do the whole initialisation with a literal assignment type and size can be inferred (feasible for short seq)
- Syntax for explicit type declaration is seq[< element type>]

```
1   # sequence definition
2   var seq1 = @[1.0,2.0,3.0] # infers type as float and initial size 3
3   var seq2 : seq[int] # initially empty
4
```

Sequence length, adding, deleting elements recap

- len, add, delete procedure provided by default

- .add(<type>)
- .delete(<index>)

```
1  # sequence length recap
2  var seq1 = @[1.0,2.0,3.0]
3  echo seq1
4  seq1.add(4.0)
5  echo seq1
6  seq1.delete(0) # delete element at index 0
7
8  # len function can tell you current size
9  echo seq1.len
```

Tuples - Summary

- Dissimilar to arrays and sequences - contain dater of different types (*heterogeneous*)
- Fixed size
- Allows access through indexing like arrays and sequences
- Allows named-fields and access with “object notation”
  - E.g. echo tup.field1
- Tuples are constructed with ()

```
1   let tup1 = ("sillyBilly", 97, 1024, 0.4, 0.5)
2   var tup2 = (name: "sillyBilly", hp: 97, xp: 1024, x: 0.4, y:0.5)
```

```
4   # can access through field name
5   tup2.hp = 10 # ouch!
```

```
7   # tuples aren't directly iterable
8   # for elem in tup2:
9   #   echo "iterating thru ", elem
10
11  # a function is provided that gives you the iterator you want
12  for f in tup2.fields:
13    echo "f", f
```