Arrays

This reading demonstrates the array data structure, including how to initialize, manage and join arrays. All programming languages have some form of an array, but how they operate might have slight differences. For instance, some languages specify that an array must hold the same element type, like a `string` array or `int` array.

In contrast, other languages allow mixed elements to be stored in an array. The functionality of arrays might also differ. Sometimes, an array will be a fully-fledged object with complex functionality. At the same time, it can be treated as a storage type for primitives; operations and functionality would then be externally applied.

Initializing arrays

Arrays can be created statically or dynamically. In a static language, the array would be kept on the stack and require that the array type be specified *a priori*. Dynamic languages offer more fluidity, sometimes calling for the size to be set and do not require the type to be specified before use. Such an instance would be stored on a heap. Arrays have indexes, which is a contiguous value starting at 0 and increasing until the end of the array. When an item is required, the array's name followed by square brackets and the index location are given.

`array_name[0]`

This example will retrieve the first item in the array. Any number could be used and the element at that index location will be returned. Specifying a number greater than the size of the array would throw an out-of-bounds error. A typical action you will make with an array is to iterate over the array and investigate the elements.

```
n <- size of array arr
FOR (i <- 0;i<(n-1); i <-(i+1)) DO
    process element arr[i]
END
```

The general shape of a `for loop` is the same in all approaches: you need the size of the array, an integer `i` that increases at every iteration and a general `for loop` for syntax. The appearance may vary slightly, but the underlying mechanism is the same. Starting at 0, go through each item in the array and do something. An array will always have some way of accessing the size. Some examples are `.size()`, `len(array)` and `.length()`.

Managing arrays in memory

One fundamental difference regarding arrays in different programming languages is where they are stored. An item can be stored in heap or stack memory. Stack memory is created when running a function. It is created for that function and discarded after the execution. Items in this memory allocation are only available for that function. In contrast, heap memory is created during the execution of instructions and is available to all.

Care should be taken when altering elements in a static environment. On instantiation of an array, memory space equal to the specified initialization size is created on the call stack. A call stack is created to execute the goal when a function is called. Altering an array and returning it from the stack may lead to corrupted memory as the stack is discarded after the function completes.

Dynamic languages avoid this issue by storing the array in a heap. Thus, the array remains unaffected when the function ends, and the stack is discarded. Stacks, by their nature, hold contiguous memory blocks, making accessing the information more manageable. A heap is less organized and it can take more time to access the elements. Once again, you should consider the trade-off between size and convenience, or accessibility versus speed.

Ordinarily, when a program is finished with an array, the memory is deallocated and becomes available for something

else. How this deallocation and garbage collection is handled is down to the programming language selected and can only improve performance if done effectively. Poor memory management can lead to leaks, which may crash your application upon repeated calls.

Two memory-related concepts you may encounter are shallow copy and deep copy. The first instance does not make a copy of the array but returns an index location. A deep copy will create a new instance of an array. Making a shallow copy optimizes memory usage; however, you must ensure that no unexpected changes are inadvertently made to an array shared by two variables.

Joining arrays

A matrix is a two-dimensional array (or an array composed of arrays) that can act like a table. It can be used to represent rows and columns. It gives an element a 2D (x,y) coordinate. Here **x** would refer to the rows and **y** to the columns. As before, one would use square brackets to access an element in a two-dimensional array. However, two index locations are required for a matrix, as below.

`Matrix[x][y]`

A matrix will exhibit a square shape with the same number of columns and rows. This need not necessarily be the case. But if you choose not to maintain uniformity, ensure that your loop acts accordingly.

```
int[][] matrix = new int[5][7]
for (int i = 0; i < matrix.length; i++) {
    for(int j = 0; j < matrix[i].length; j++)
    {
        doSomethingNeo(matrix[i][j])
    }
}
```

In this example, a 2D array called a `matrix` is initialized to hold integers. The outer collection can contain five elements (integer arrays). The inner arrays all have a capacity of 7. The first array in the outer array is selected with the `i`, the size is determined (7), and each element of that inner array is then passed to the method. Notice how the size of the inner array was accessed dynamically using the `.length` attribute. This ensures that there can be no out-of-bounds errors during the iteration.

Conclusion

Arrays are a fundamental data structure that exists in most programming languages. They help store related data. It is worth noting that their implementation varies depending on the language, so care should be taken when using them. This reading taught you about the array data structure, including how to initialize, manage and join arrays.