Module Five

Arrays and Slices

Go is like a very delicious trifle - the further into it you go, the more delicious things you find. The quality is clear throughout. Go is very unassuming. You start by wondering what the big deal is + getting annoyed with the minor differences from other C-syntax languages before slowly progressing towards quite liking it, then eventually once you grok how simple + elegant and well engineered it is you come to love it.

Lorenzo Stoakes

A fool with a tool is still a fool.

Martin Fowler

5.1 Introduction

Arrays in Go work much like arrays do conceptually in other C-style programming languages and use the same syntax that most C-style programmers will recognize and are comfortable with. However Go arrays are quite different how they are implemented which has the dual effect of making them safer and more efficient to use from an implementation perspective, but at the cost of reduced functionality and convenience from the programming perspective.

Arrays in most C-style programming language are pointers or references to blocks of memory. Arrays can be passed efficiently in function argument lists – which we call pass by reference or pass by pointer – and can be used in other ways that are quite elegant because they are pointers dressed up with array notation.

However this elegance comes with a cost in terms of the behind the scenes overhead like memory management (every programmer has spent at least on all nighter looking at code muttering “What null pointer? Why does the program say there is a null pointer?”), memory leaks, problems with defining equality, shallow versus deep copies, and program safe code issues (C is quite happy to modify the 10th element of a four element array for example).

What Go has done is split the normal C-style concept of an array into two parts. A safe value type (that means it is managed in memory like an int or a float) which Go calls the array type, and a second part called a slice which is a pointer that can reference parts of the array in a way that C-style programmers are used to.

By using both arrays and slices, we get the advantages of arrays that we expect from other C-style languages but we mitigate the risks of using treating an array as a pointer to a chunk of heap memory.

5.2 Arrays

Arrays in Go are *value* data types. This means that they are managed the same way in memory as an int or float. This is different than most of the other C-style languages where an array is usually a *reference* data type, which means that arrays are implemented as pointers or references to chunks of memory that have to be allocated in the code.

An array in Go is a sequence of elements where:

1. Each element is of the same type, including user defined types (which will be covered in module eight).
2. The length of the array is fixed.

The size of an array, which can be referenced with the function len(array) is part of the array type. So and [5]int and [6]int array are of different types because they are of different lengths.

The value that specifies the length of the array must either be a constant or the result of a constant expression. Remember that a constant expression is one that can be evaluated at compile time. Of course it goes without saying that the size must also be an integer and non-negative.

The length of an array is immutable, there is no way to grow or shrink the size of an array once it has been created.

5.2.1. Declaring Arrays

Arrays are declared with the following syntax

```
var name [size]type
var ar   [3]int
```

The above line declares a length 3 array of ints referenced by the variable ar. When arrays are declared without an explicit initialization, the array elements are all initialized to their zero values.

5.2.2 Accessing arrays

Arrays are accessed using the standard array [..] notation common to most C-style languages

Like Java, Go checks for array indexes that are out of bounds. Because this cannot be checked for at compile time, if an array index is out of bounds a runtime panic is generated when the illegal access occurs, just like Java.

```go
// Example 05-01 Basic array syntax

package main

import "fmt"

func main() {
  var a [3]int
  fmt.Println("a =", a)
  a[0] = 1
  a[1] = a[0] + 1
  fmt.Println("a =", a)
}
```

```
[Module05]$ go run ex05-01.go
a = [0 0 0]
a = [1 2 0]
```

5.2.3 Initializing arrays

Arrays can be initialized using what are called array literals. These are of the form

```
var arrayname = [size]type{list of element literals}
arrayname := [size]type{list of element literals}
```

Arrays can declared using the size [...] when initializing with literals. In this case the compiler will count the number of literals and insert the correct size.

It is possible to also initialize some elements of the array and let the others take their zero values. In case below the 3rd and 4th elements are initialized explicitly while the others are initialized to their zero values (remember we count from 0 for indices).

```
var ar = [5]int{3: 3, 4: 4}
```

or

```
var ar = [...]int{3: 3, 4: 4}
```

In the last case, in order to be able to count the right number of literals for initialization, the last element of the array must be explicitly initialized. Alternatively we could say that the last explict intitalization determines the length of the array.

```go
// Example 05-02 Explicit array initialization

package main

import "fmt"

func main() {
  var ar1 = [5]int{0, 1, 2, 3, 4}
  var ar2 = [...]string{"Hello", "World"}
  ar3 := [2]bool{true, false}
  ar4 := [...]int{3: -1, 4: -1}
  fmt.Println("ar1=", ar1, "length=", len(ar1))
  fmt.Println("ar2=", ar2, "length=", len(ar2))
  fmt.Println("ar3=", ar3, "length=", len(ar3))
  fmt.Println("ar4=", ar4, "length=", len(ar4))
}
```

```
[Module05]$ go run ex05-02.go
ar1= [0 1 2 3 4] length= 5
ar2= [Hello World] length= 2
ar3= [true false] length= 2
ar4= [0 0 0 -1 -1] length= 5
```

5.2.4 *Array Operations*

Comparison

Since array are values, they can be compared using the comparison operators == and ! =. However in order to be comparable, they must both have elements of the same type and be the same size.

Equality is defined to mean that the two arrays are equivalent if they are of the same type and that the corresponding elements in the two arrays are equivalent. Other operators like < are not defined because there is no meaningful or useful interpretation of those sorts of comparisons for arrays.

```go
// Example 05-03 Array comparison

package main

import "fmt"

func main() {
  var ar1 = [5]int{0, 1, 2, 3, 4}
  var ar2 = [5]int{0, 1, 2, 3, 4}
  fmt.Println("ar1 == ar2 is ", ar1 == ar2)
  fmt.Println("ar1 != ar2 is ", ar1 != ar2)

}
```

```
[Module05]$ go run ex05-03.go
ar1 == ar2 is  true
ar1 != ar2 is  false
```

Arguments to Functions

Arrays can be passed as arguments to functions. However because functions pass by value, a copy of the array is used in the called function. This means a function cannot change the contents of the original array passed as an argument from the calling function.

This is inconvenient and uses up a lot of memory making copies, especially if the array being passed is a large array. But this also avoids a lot of unwanted side effects because the original array cannot be affected by anything done to the passed copy.

We can go the route of creating a pointer to an array and then passing the pointer instead of the array, but this is considered unseemly in Go – that is a C idiom, not a Go idiom. In the next section, we will see that we can do the same thing with slices that we could with pointers, but using slices is much more the Go way to do this.

```go
// Example 05-04 Array as parameter

package main

import "fmt"

func delta(prm [3]int) {
  prm[0] = -1
  fmt.Println("prm = ", prm)
}

func main() {
  var arg = [3]int{99, 98, 97}
  fmt.Println("arg = ", arg)
  delta(arg)
  fmt.Println("arg = ", arg)
}
```

```
[Module05]$ go run ex05-04.go
arg =  [99 98 97]
prm =  [-1 98 97]
arg =  [99 98 97]
```

Assignment

Since array are values, they can be compared using the comparison operators == and != . However in order to be comparable, they must both have elements of the same type and be the same size.

Equality is defined to mean that the two arrays are of the same type and that the corresponding elements in the two arrays are equivalent. Other operators like < are not defined because there is no meaningful or useful interpretation of those sorts of comparisons for arrays.

```go
// Example 05-05 Array assignment

package main

import "fmt"

func main() {
  var ar1 = [3]int{99, 98, 97}
  var ar2 [3]int
  ar2 = ar1
  ar2[0] = 0
  fmt.Println("ar1 =", ar1)
  fmt.Println("ar2 =", ar2)
}
```

```
[Module05]$ go run ex05-05.go
ar1 = [99 98 97]
ar2 = [0 98 97]
```

Iterating over arrays

The range function was introduced in module two as a way to iterate over sequences. Just as would be expected, we can iterate over an array the same as shown in example 05-06 on the next page.

The range function returns two values for each iteration. The first is the current index of the position being processed. The second is the element in the array at that index. A common idiom is to use the blank variable "_" to receive the index when we are not interested in where in the array we are, we just want to iterate sequentially through the entire array and do something to each value in turn .

```
// Example 05-06 Iteration with range

package main

import "fmt"

func main() {
  words := []string{"the", "best", "of", "times"}
  for index, value := range words {
       fmt.Println(index, " ", value)
  }
}
```

```
[Module05]$ go run ex05-06.go
0   the
1   best
2   of
3   times
```

5.2.5 Multidimensional Arrays

As mentioned before arrays are one dimensional, however they can be "layered" to form multi-dimensional arrays similar to how we build up multidimensional arrays in other C-style languages. The following snippets both define the same multi-dimensional array.

```
var matrix [2][3]int
var matrix [2]([3]int)
```

These can be read to mean that matrix is either a 2x3 array of ints, or it is an array of length 2 where each element of the array is itself an array of length 3 where each element of each of those arrays is an int. These two definitions are just two different ways of describing the same underlying structure. Either one is acceptable in Go and it is often a matter of choosing the one that makes the programmer intent clear in a given context. Higher dimensions work in a similar way.

This means that all multidimensional arrays must be square. In the example, matrix is a two dimensional array of one dimensional arrays of type [3]int. Remember that the size is part of the array type. If we wanted a ragged two dimensional array, we would have to have matrix be an array where the first element is an array of type [2]int and the second an array of say [3]int. But then the rule that all of the elements of an array must be of the same type would be violated, which means there is no way that we can make matrix into a ragged array.

```go
// Example 05-07 Multidimensional array

package main

import "fmt"

func main() {
  var matrix [2][3]int
  value := 10
  for row, col := range matrix {
       for index, _ := range col {
            matrix[row][index] = value
            value++
       }
  }
  fmt.Println("Matrix: ", matrix)
}
```

```
[Module05]$ go run ex05-07.go
Matrix:  [[10 11 12] [13 14 15]]
```

Example 05-07 above demonstrates iterating over a two dimensional array. A simple for loop with defined loop indices could have been used but in this example nested ranges are used to accomplish the same effect. Notice that we are not interested in the value returned by range in the inner for loop so we throw it away by using the blank variable "_"

5.2.6 Multidimensional Initialization

A multi-dimensional array can be initialized using literals in several different formats. In the first form in example 05-08, the syntax emphasizes the layers that the array is built up from.

```go
// Example 05-08 Multidimensional initialization

package main

import "fmt"

func main() {
  var matrix = [4][4]int{
       [4]int{1, 2, 3, 4},
       [4]int{2, 4, 8, 16},
       [4]int{3, 9, 27, 81},
       [4]int{4, 16, 64, 256},
  }
  fmt.Println("Matrix: ", matrix)
}
```

```
[Module05]$ go run ex05-08.go
Matrix:  [[1 2 3 4] [2 4 8 16] [3 9 27 81] [4 16 64 256]]
```

This form helps void possible ambiguity errors. However we can also leave out the extra info when the context is unambiguous, which is which is demonstrated in example 05-09, which is logically equivalent to the 05-08. Both initialize the array and one might be preferred over the other by some because they might feel that it depicts more clearly exactly what the conceptual structure of the array is.

```
  // Example 05-09 Multidimensional initialization

  package main

  import "fmt"

  func main() {
    var matrix = [4][4]int{
         {1, 2, 3, 4},
         {2, 4, 8, 16},
         {3, 9, 27, 81},
         {4, 16, 64, 256},
    }
    fmt.Println("Matrix: ", matrix)
}
```

```
[Module05]$ go run ex05-08.go
Matrix:  [[1 2 3 4] [2 4 8 16] [3 9 27 81] [4 16 64 256]]
```

Because this code gets really dense fast, either of these are probably not a good way to initialize higher dimensional or large multi-dimensional arrays. It would not take long before increases in size or dimension would produce an unreadable wall of text.

5.3 Slices

Slices are references to contiguous segments of an underlying array. A slice can be thought of as consisting of a pointer to a starting position somewhere in the array and a count of the number of elements from the start which make up the rest of the slice. From a programming point of view, slices look and behave syntactically just like arrays.

For C programmers, slices can be thought of as something analogous to how pointers into an array work. Given a chunk of memory, we can define multiple pointers into that chunk of memory where each pointer defines an overlay that slices up the chunk ol memory into logical arrays starting at different points.

Slices can be created from existing arrays by taking a sub-sequence of the array as illustrated in the 05-10. Trying to take sub-sequence where the ranges would be outside the bounds of the underlying array is an error.

```go
// Example 05-10 Basic Slice

package main

import "fmt"

func main() {

  var a = [6]int{0, 1, 2, 3, 4, 5}

  s := a[2:] // s is a slice of the array a
  fmt.Println("s= ", s)

  a[4] = -20 // changing underlying array
  fmt.Println("s= ", s)

  s[0] = 999 // change the array via the slice
  fmt.Println("a= ", a)
}
```

```
[Module05]$ go run ex05-10.go
s=  [2 3 4 5]
s=  [2 3 -20 5]
a=  [0 1 999 3 -20 5]
```

5.3.1 Length and Capacity

Each slice has two values associated with it.

1. *Length*: Just like an array, the length of a slice is the number of elements contained in the slice. This value is dynamic and may change during execution. The minimum length of a slice is 0 and the maximum size of a slice is the length of the underlying array.
2. *Capacity*: The capacity of a slice is how long the slice can become. Capacity is calculated by adding the current length of the slice and the number of elements from the end of the slice to the end of the underlying array.

Trying to make a slice larger than its capacity is an error when the slice is taken from an existing array. For example

```
a := [...]int{1,2,3}
s := a[0:8]
```

will be an error since have exceeded the slice capacity.

5.3.2 Working with elements in a Slice.

Slices do not contain their own data. That means that multiple slices can overlay or reference the same element in the underlying array, and this can lead to problems if we are not careful. Syntactically, we access elements of a slice exactly like we access elements in an array.

5.3.3 Slices of slices

We can also take slices of other slices, not just of underlying arrays. The code in example 05-11 illustrates this and how the slices can overlap. Notice that the third slice s2 seems to be defined with more elements that exist in what it is a slice of, namely s1. But the elements of a1 are not really "in" the s1 slice but are in the underlying array, taking a slice of a slice is just another way of taking a slice of the underlying array.

In example 05-12, two overlapping slices s0 and s1 are defined on the same underlying array. Changing an element in one slice is actually a change to the element in the underlying array which then changes the value of the second slice.

```go
// Example 05-11 Slices

package main

import "fmt"

func main() {

  var a = [...]int{0, 1, 2, 3, 4, 5}

  s0 := a[1:4]
  s1 := s0[1:3]
  s2 := s1[0:4] // ???

  fmt.Println("s0 length=", len(s0), " s1 length=", len(s1),
       " s2 length=", len(s2))
  fmt.Println("s0=", s0, " s1=", s1, " s2=", s2)
}
```

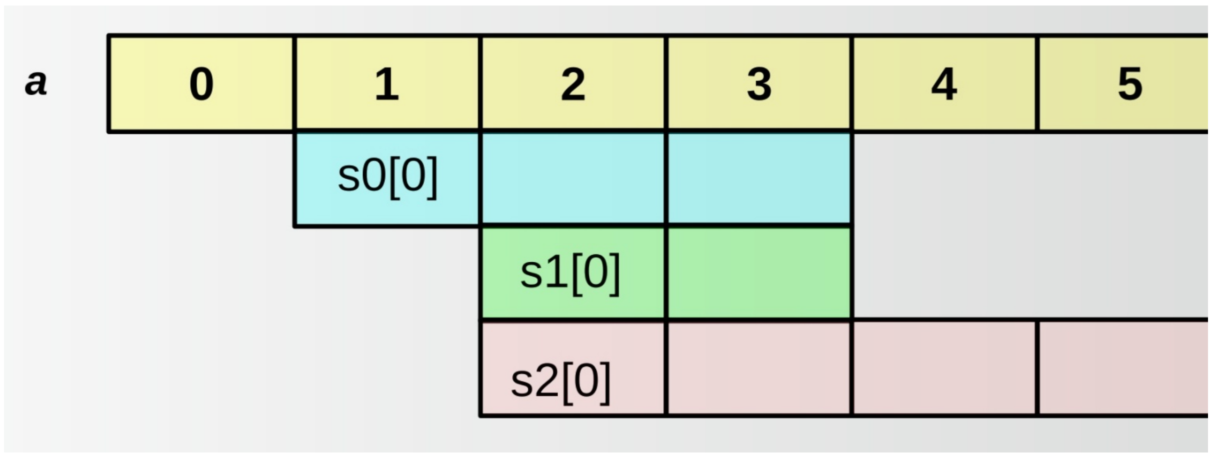
```
[Module05]$ go run ex05-11.go
s0 length= 3  s1 length= 2  s2 length= 4
s0= [1 2 3]  s1= [2 3]  s2= [2 3 4 5]
```

The actual situation in the above code is depicted graphically below.

Remember that we are still working with the underlying array "a". This means that the slices are like sliding windows on top of the array. When we said

```
s2 := s1[0:4]
```

what we were really saying was: "give me four elements of the underlying array starting at the position pointed to by slice s1"



5.4 Creating Slices Directly

So far we have looked at created slices from an existing array. However, since we tend to use slices more in Go than arrays, we can also create a slice directly in one of two ways.

5.4.1 Slice initialization

Slices can be initialized with an array literal using the following syntax

```
s1 := []int{1,2,3}
```

which looks very similar to the array initialization syntax except for the missing ellipsis between the square brackets. Since slices themselves cannot hold data because they are pointers into an underlying array, Go creates an array large enough to hold the initializing literal and then assigns s1 the pointer to that newly created array.

This underlying array is anonymous, there is no way to access it without going through the slice so for all intents and purposes, we can think of the slice as being the actual object itself which, while not technically accurate, is a useful conceptualization.

5.4.2 Using Make

We can also create a slice directly without having to initialize it explicitly by using the make() function like this

```
slice := make([]type,length)
```

The make() function does essentially the same thing as creating a slice using an initialization. An underlying anonymous array is created of [length]type and a reference to this array is returned and assigned to the slice variable.

The underlying array, just like in the case with initialization, cannot be accessed directly but can only be accessed through the slice, and any other slices we define from that original slice. Example 05-12 demonstrates these two way s of creating slices.

```go
// Example 05-12 Slice creation

package main

import "fmt"

func main() {

  s1 := []int{1, 2, 3}

  s2 := make([]int, 3)

  fmt.Println(s1, s2)
}
```

```
[Module05]$ go run ex05-12.go
[1 2 3] [0 0 0]
```

The only sign that we are creating a slice and not an array when we use an array literal is that a slice does not have the ellipsis between the [ ].

This statement creates an array

```go
    a := [...]int{1,2,3}
```

but this statement creates a slice

```go
    s := []int{1,2,3}
```

5.4.3 Appending Elements

The append function allows us to add elements to the end of a slice. However if we don't understand what is going on with slices, then we can get what appear to be counter intuitive results such as the results of example 05-13.

```go
// Example 05-13 Odd behavior

package main

import "fmt"

func main() {

  a := [...]int{100, 200, 300}
  s := a[:2]
  fmt.Println("Initially a=", a, "s= ", s)
  s = append(s, -1) //result is a[2] == -1
  s[0] = 0          // result a[0] == 0
  fmt.Println("After first op a=", a, "s=", s)
  s = append(s, -2) // this would go in a[3]?
  fmt.Println("After second op a=", a, "s=", s)
  s[0] = 999 // a now remains unchanged
  fmt.Println("After third op a=", a, "s=", s)
}
```

```
[Module05]$ go run ex05-13.go
Initially a= [100 200 300] s=  [100 200]
After first op a= [0 200 -1] s= [0 200 -1]
After second op a= [0 200 -1] s= [0 200 -1 -2]
After third op a= [0 200 -1] s= [999 200 -1 -2]
```

In the example, the first two operations work as we expect, we make modifications to the slice “s” and they show up in the underlying array “a”. The length of "s" is 2 but has capacity of 3 since it doesn't extend to the end of the underlying array. When we append -1 to "s", that has the effect of extending "s" to include the last position "a". So far, so good – everything works exactly as we would expect it to.

Then we append one more element to "s" which does not generate an error but now "a" and "s" no longer appear to be linked. Changes to "s" do not affect "a" and vice-versa.

When we create a slice directly, Go creates the underlying array with the length specified either by the length argument in make() function or by the number of items provided in the literal used to initialize the slice. But slices can grow dynamically so when we try to add an element once the array is full, Go creates a new array double the size as the old one then copies the old array contents into the new array. The slice is now reassigned so that it references this new array, and the old array is garbage collected.

The problem is that because there is separate reference to the original array, the variable “a”, Go cannot garbage collect the original array and are still able to access it. If we had defined the slice without using an previously defined array, we would not see this behavior.

5.4.4 Specifying Slice Capacity

As stated before, the capacity of a slice can be thought of as how many times we can append something to that slice before Go has to re-size the underlying array. For slices with anonymous underlying arrays, the initial length of the slice determines the initial size of the underlying array; by default the capacity is the same as the length. We can provide a third argument to make() to specify the initial capacity of the slice.

```
Slice := make([]type,length, capacity)
```

Every time try we append something to the slice that would exceed the capacity, Go re-sizes the array. This is demonstrated in example 05-14.

```go
// Example 05-14 Re-sizing Slices

package main

import "fmt"

func main() {

  s1 := make([]int, 1, 3)
  for i := 0; i < 10; i++ {
       s1 = append(s1, i)
       fmt.Println("s1=", s1, "len=", len(s1),
                                       "Cap=", cap(s1))
  }
}
```

```
[Module05]$ go run ex05-14.go
s1= [0 0] len= 2 Cap= 3
s1= [0 0 1] len= 3 Cap= 3
s1= [0 0 1 2] len= 4 Cap= 6
s1= [0 0 1 2 3] len= 5 Cap= 6
s1= [0 0 1 2 3 4] len= 6 Cap= 6
s1= [0 0 1 2 3 4 5] len= 7 Cap= 12
s1= [0 0 1 2 3 4 5 6] len= 8 Cap= 12
s1= [0 0 1 2 3 4 5 6 7] len= 9 Cap= 12
s1= [0 0 1 2 3 4 5 6 7 8] len= 10 Cap= 12
s1= [0 0 1 2 3 4 5 6 7 8 9] len= 11 Cap= 12
```

In the example, we created a slice with length 1 and capacity 3. When we tried to add a fourth element, then the slice doubled in capacity.

Deciding on an initial capacity is really a matter of what is important to the programmer – whether you want to avoid the costs of constantly re-sizing the array or want to avoid having arrays that are not being re-sized very often but have big chunks of unused space.

5.5 Slices as function arguments

Earlier in this module we looked at the issues of trying to pass an array as an argument to a function as a result of Go passing the array by value. However, since a slice is a reference to an array, passing a slice as a parameter has the effect of passing the array by reference.

Example 5-15 demonstrates this.

```go
// Example 05-15 Slices as parameters

package main

import "fmt"

func f(p []int) {
  p[0] = -1
}

func main() {

  s := []int{1, 2, 3, 4}
  fmt.Println("Before call", s)
  f(s)
  fmt.Println("After call", s)
}
```

```
[Module05]$ go run ex05-15.go
Before call [1 2 3 4]
After call [-1 2 3 4]
```