Introduction to Programming in Go

5. Arrays and Slices

Module Topics

1. Arrays
2. Slices

Arrays

Arrays in Go

1. Go arrays are similar to other C-Style programming languages.
2. Fixed length numbered sequence of elements of a single type.
3. The length is part of the array type eg. [3]int and [5]int are types.
4. Array types are always one-dimensional.
5. Arrays may be combined to form multi-dimensional types.
6. The length of the array can be found using len(array).
7. Array elements are accessed as in Java or C with [...] operator.
8. Arrays are value objects like int and float32, not reference objects.

Basic Array Syntax

1. Arrays are defined using the syntax:

   var name [size] type

   var a [3]int defines an array of type [3]int
2. By default all elements are initialized to their zero values.
3. Access to array elements is the same as in other C-style languages.
4. Attempt to access elements out of range generates a panic.

Basic Array Syntax

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

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]
```

Explicit Array Initialization

1. Arrays can be initialized with array literals:

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

2. Using [...] for makes the compiler count the literal values for the size.

3. Some of the elements can be initialized using the syntax:

   var ar = [size]type{$index_1$: $value_1$, ... $index_n$: $value_n$}

4. All other elements are initialized to their zero value.

5. If [...] is used with the indes:value syntax as in point 3, the size is the last index referenced.

Explicit Array Initialization

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

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
```

Array Operations

1. Arrays of the same type can be compared using == and !=
2. Array assignment copies the array from the RHS to LHS.
3. Arrays are passed by value as function call arguments.
4. Iteration over arrays is done with the range operation.

Array Comparison

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

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
```

Arrays as Function Parameters

```
// Example 05-04 Array as parameter
...
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]
```

Array Assignment

```
// Example 05-05 Array assignment
...

func main() {
    var ar1 = [3]int{99, 98, 97}
    var ar2 [3]int
    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]
```

Array Iteration with range

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

func main() {
    words := [...]string{99, 98, 97}
    for index, value := range words {
        fmt.Println(index, " ", value)
    }
}
```

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

Multidimensional Arrays

1. Multi-dimensional arrays are built up in layers.
2. For example a two dimensional array would be defined as either

   [2][2]int = { {1,2}.{3,4}}

   or

   [2][2]int = {[2]int {1,2},[2]int{3,4}}
3. Ragged arrays (ie. non-rectangular) are not allowed.
4. All of the usual rules for arrays still apply.

Multidimensional Arrays

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

func main() {
    var matrix [2][3]
    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]]
```

Multidimensional Initialization

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

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]]
```

Multidimensional Initialization

```
// Example 05-09 Multidimensional initialization
...

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 ex05-09.go
Matrix:  [[1 2 3 4] [2 4 8 16] [3 9 27 81] [4 16 64 256]]
```

Slices

Slices

1. A slice is a reference to a contiguous segment of an underlying array.
2. A slice has a start position, a length and a capacity.
3. Slices can be thought of as sub-sequences of arrays.
4. Slices look and act syntactically just like arrays.
5. Slices are dynamic – their length can change.
6. Slice capacity is how big the slice can become.
7. Slices are used more in Go code that are arrays.

Slices

```
// Example 05-10 Slices
...

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]
```

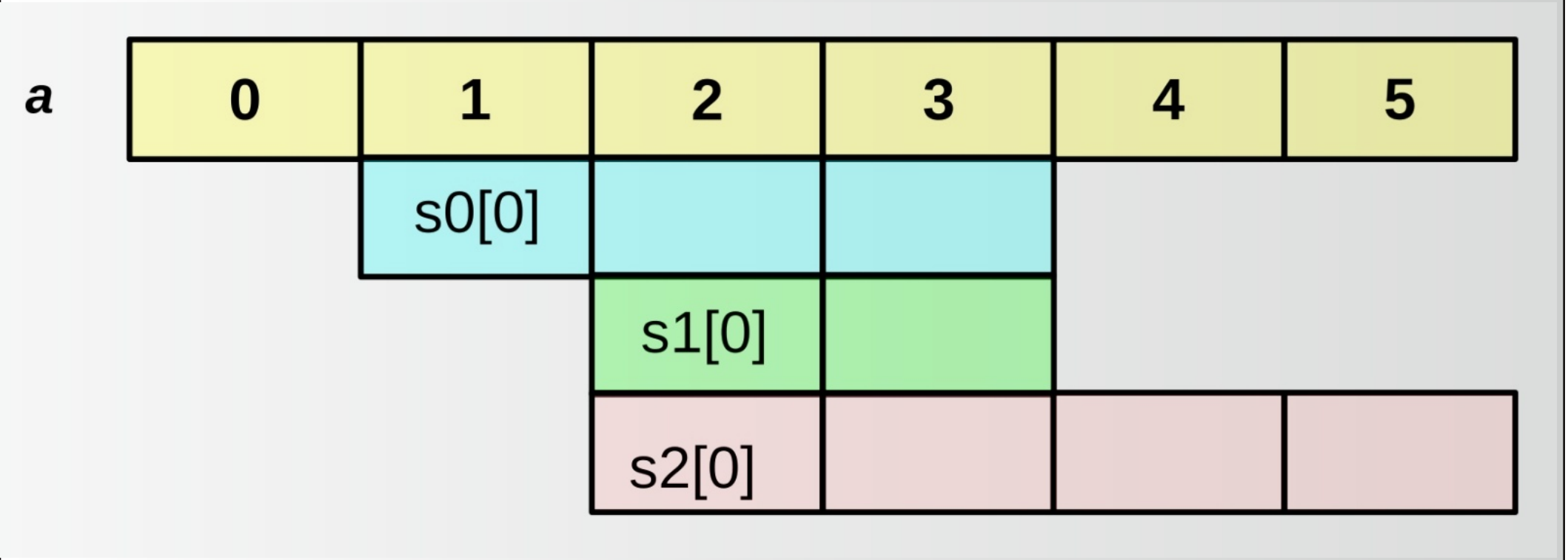
Slices of slices

```
// Example 05-11 Slices
...

func main() {
    var a = [...]int{0, 1, 2, 3, 4, 5}
    s0 := a[1:4]
    s1 := a[1:3]
    s2 := a[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]
```

Slices of slices



Creating Slices Directly

1. We can create slices directly in two ways:
2. i) Using an array literal

   `[]type{list of elements}`
3. Key point [...] defines array, [] defines slice.
4. ii) Using make()

   `s := make([]type,len)` where len is initial length.
5. In both cases, Go makes an anonymous underlying array.
6. The array can only be accessed through the slice.

Creating Slices Directly

```
// Example 05-12 Slices
...

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]
```

Appending Elements

1. append() function adds elements to the end of a slice.

```
slice := append(slice,item)
```

2. If appending exceeds capacity of slice, the slice is re-sized.
3. Re-sizing doubles the capacity of the slice.

Appending Elements

```
// Example 05-13 Appending Slices
...

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]
```

Specifying Initial Capacity

```
// Example 05-14 Initial Capacity
...

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
```

Slices as Function Arguments

```
// Example 05-15 Slice as Function Argument
...
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]
```

Lab 5: Arrays and Slices