**Golang**

1. a Go array is a value type. It’s not a pointer to the first element of a block of memory. So argument to a function is array, entire array is copied.
2. A slice itself is a value type, but it references the array it uses with a pointer. In the diagram below array points to the actual array below.

Table

Description automatically generated

1. S1 := s[2:4] => still points to the same array.

func f1(s []int) {

// slicing the slice creates a new slice but does not copy the array data

s = s[2:4]

// modifying the sub-slice changes the array of slice in main function as well

for i := range s {

s[i] += 10

}

fmt.Println("f1", s, len(s), cap(s))

}

func main() {

s := []int{1, 2, 3, 4, 5}

// **passing a slice as an argument makes a copy of the slice properties (pointer, len and cap)**

**// but the copy shares the same array**

f1(s)

fmt.Println("main", s, len(s), cap(s))

}

**f1 [13 14] 2 3**

**main [1 2 13 14 5] 5 5**

Diagram, shape

Description automatically generated with medium confidence

1. If array cap is reached, a new array is allocated for the slice.

s := make([]int, 3, 4)

s2 := append(s, 4) => only len is changed, but it points to the same array

s3 := append(s2, 5) => Now a new array is allocated for this slice. len=5, ca

1. Nil slice vs empty slice

func main() {

var s []int // this is a nil slice

s2 := []int{} // this is an empty slice

// looks like the same thing here:

fmt.Println(s, len(s), cap(s)) // [] 0 0

fmt.Println(s2, len(s2), cap(s2)) // [] 0 0

// **but s2 is actually allocated somewhere**

fmt.Printf("%p %p", s, s2) // 0x0 0x65ca90(any address)

}

1. Append to a nil slice works as usual.
2. Slice of slices for multidimensional array. For example, Y \* X slice:

s := make([][]int, y)

for i := range s {

s[i] = make([]int, x)

}

1. runtime.GC() => run garbage collector
2. var m runtime.MemStats

runtime.ReadMemStats(&m)

m.Alloc => gives the number of bytes allocated in memory

1. String is a value type. It has a pointer to byte array and a fixed length. No null string. Var s string is equals to “”. Strings are immutable. Usually the data within a string is encoded as UTF-8 but it doesn’t need to be. If a = “Rahul”, then a[1] will print **65** i.e. \x41**. U+0041.** String(a[0]) should print “A” i.e. encoding 41 to it’s corresponding UTF-8
2. Strings are immutable. One way to modify a string is convert it to byte slice and then back to string.

str := "darkercorners"

bytes := []byte(str)

bytes[0] = 'D'

str2 := string(bytes)

1. sort.Slice( intervals, func( i, j int ) bool { return intervals[ i ][ 0 ] < intervals[ j ][ 0 ] } ) => sorting slices
2. var isOverlap = func ( interval1 []int, interval2 []int ) bool { => That is how you declare closures
3. for loop basic => for i := 1; i < len( intervals ); i++ {
4. Another for loop basic => for i, elem := range slice {}
5. Append to the slice => output = append( output, interval )
6. Max of two numbers => int( math.Max( float64( a ), float64( b ) ) ) }
7. Math.Atan2(y, x) => gives theta in float64. i.e. 90 degrees will be 1.57 i.e. pi/2
8. To use a composite key or tuple in map, use struct => m := map[ Key ]bool. Key is a struct
9. strconv.Itoa() to convert integer to string and strconv.Atoi() to convert string to integer
10. var x map[int]int

fmt.Println(x == nil, x[0])

true 0

|  |
| --- |
| x := []int{1, 2, 3, 4, 5, 6, 7}  fmt.Printf("%p %d %d\n", x, len(x), cap(x))  // Same array only length is changed  y := x[0:3]  fmt.Printf("%p %d %d\n", y, len(y), cap(y))  // Pass by value example  rs := randomStruct{}  rs.list = append(rs.list, 1, 2, 3, 4, 5, 6)  list := rs.list //<----------- this is copy by value into variable list. rs.list remains unaffected later  list = rs.list[0:3]  fmt.Println(list, rs.list)  0xc0000b2000 7 7  0xc0000b2000 3 7  [1 2 3] [1 2 3 4 5 6]  Program exited. |

1. Golang custom errors: <https://gobyexample.com/errors>
2. For deque, simply use golang slices. Since they are value objects, they can internally change the references when you pop from left and not incur O(N)