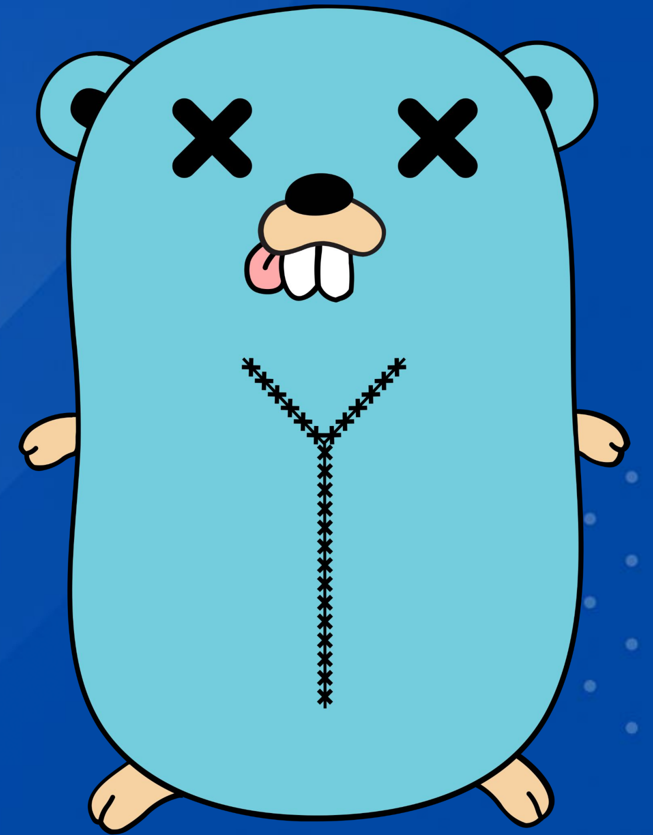
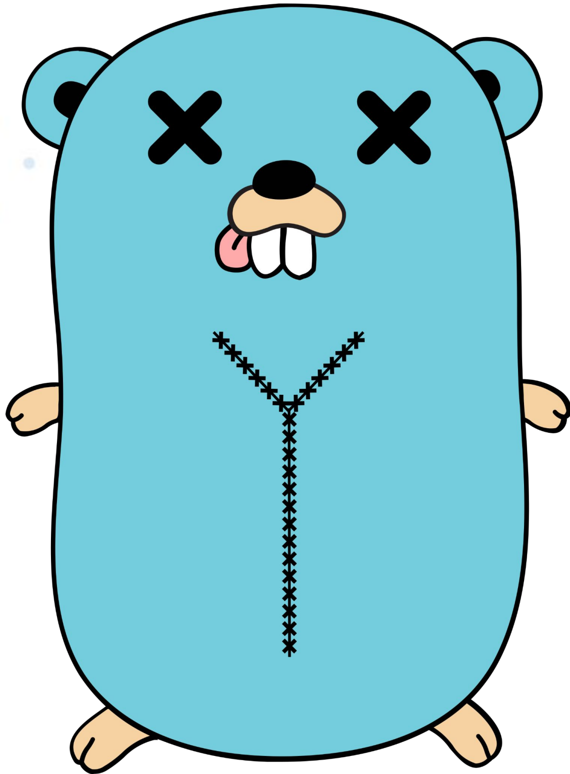
Dissecting Channels, Slices, and Maps in Go

// Jesús Espino - Staff Engineer @ Mattermost



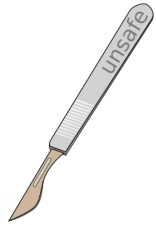
Mattermost

Introduction



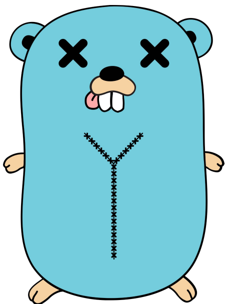
- Slices, Maps, and Channels are the most commonly used built-in structures in go.
- We understand how to use them, but not necessarily how they work.
- We are going to analyze how they work under the hood.
- We are going to do it through an experimental approach.
- After this talk you will understand better how the structures are shaped in memory and what are the implications of that.

Class materials



- The scalpel

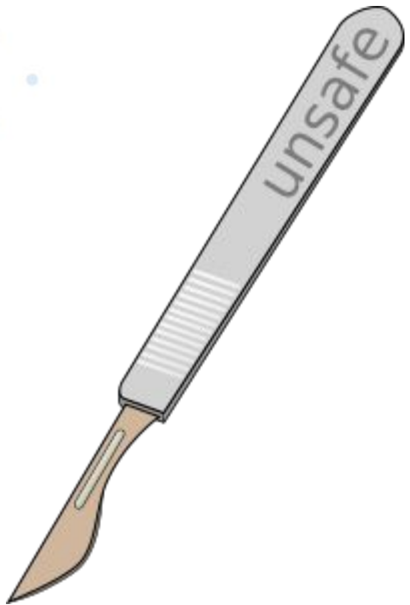
- The microscope



- The subject

Slices

The scalpel




```go
func Scalpel(slice *[]int) *sliceStruct {
    ss := unsafe.Pointer(slice)
    return (*sliceStruct)(ss)
}
```

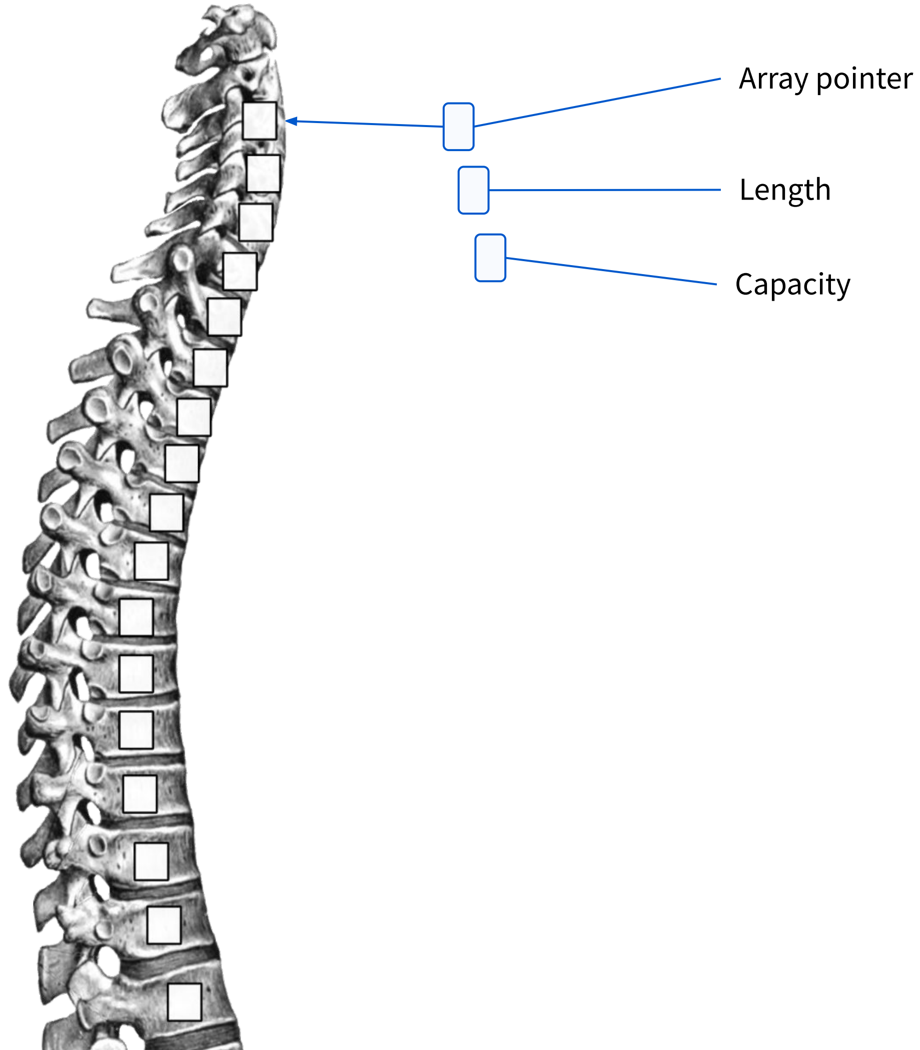
The microscope

```go
func Microscope(ss *sliceStruct) {
    fmt.Printf("Array Memory address: 0x%x\n", ss.array)
    fmt.Printf("Slice length: %s\n", ss.len)
    fmt.Printf("Slice capacity: %s\n", ss.cap)
    fmt.Printf("Stored data: [")
    for x := 0; x < ss.cap; x++ {
        fmt.Printf("%d,",
            *(*int)(unsafe.Pointer(
                uintptr(ss.array) +
                uintptr(x) *
                unsafe.Sizeof(int(0))
            ))
        )
    }
    fmt.Println("]")
}
```

The subject

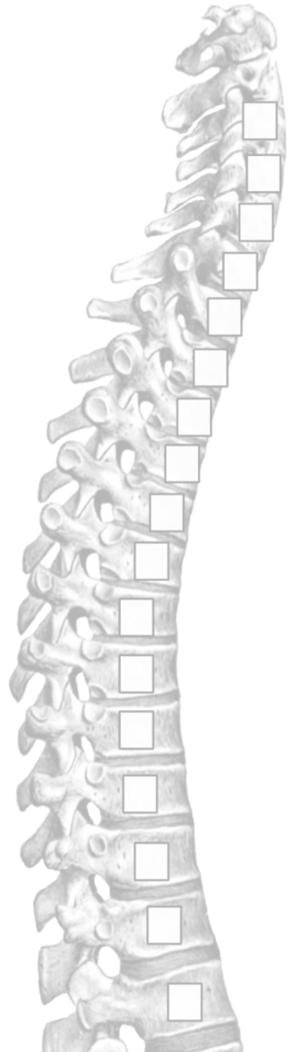
- An array
- One or more slices

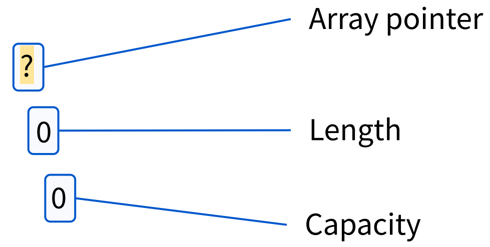
Inside the subject




```
type sliceStruct struct {
    Array unsafe.Pointer ([x]int)
    len int
    cap int
}
```

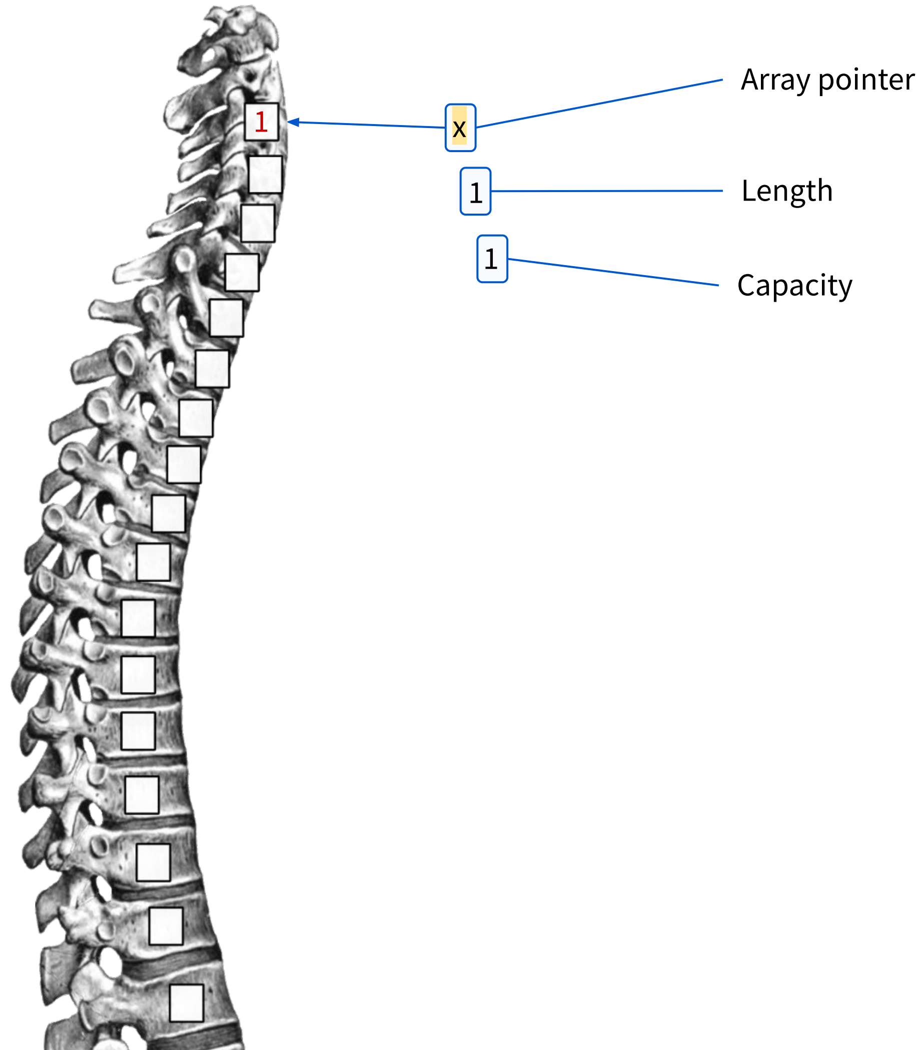
Slice creation





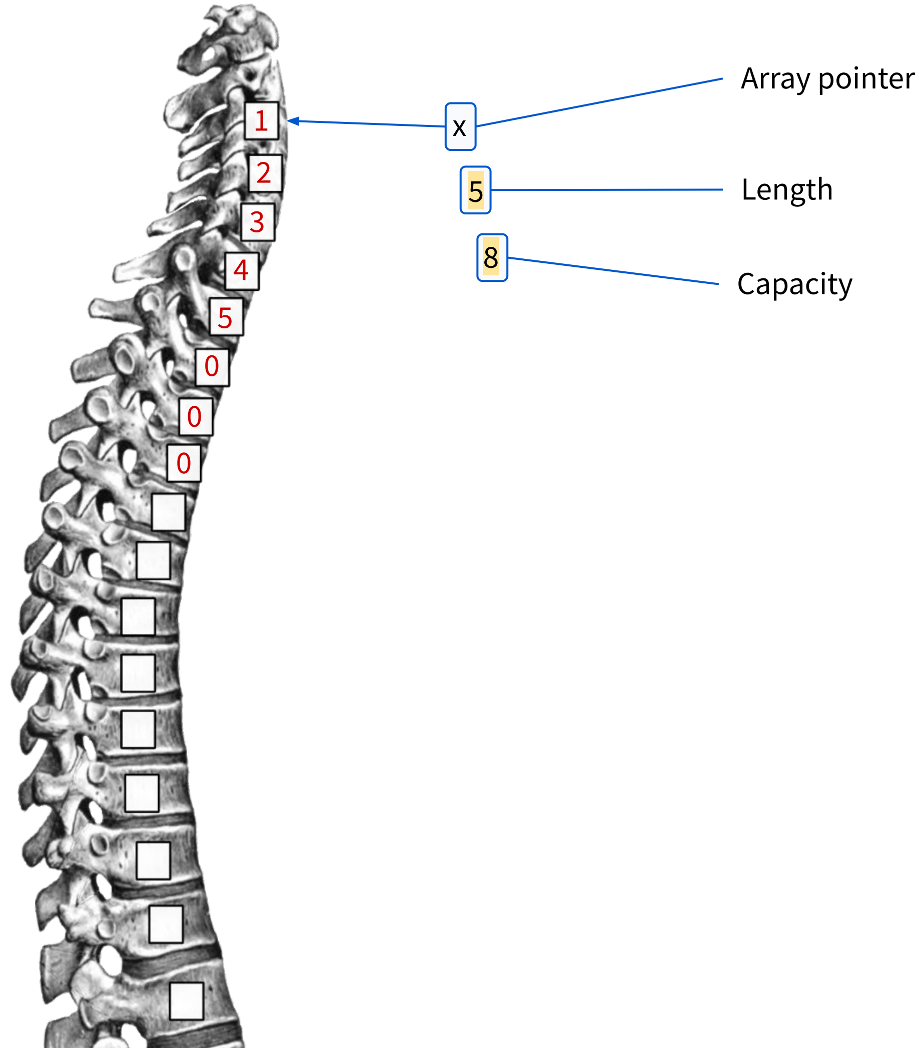

```
s := []int{}
ss = Scalpel(&s)
Microscope(ss)
--------------
Array Memory address: 0x555f30
Slice length: 0
Slice capacity: 0
Stored data: []
```

Insert something into the slice



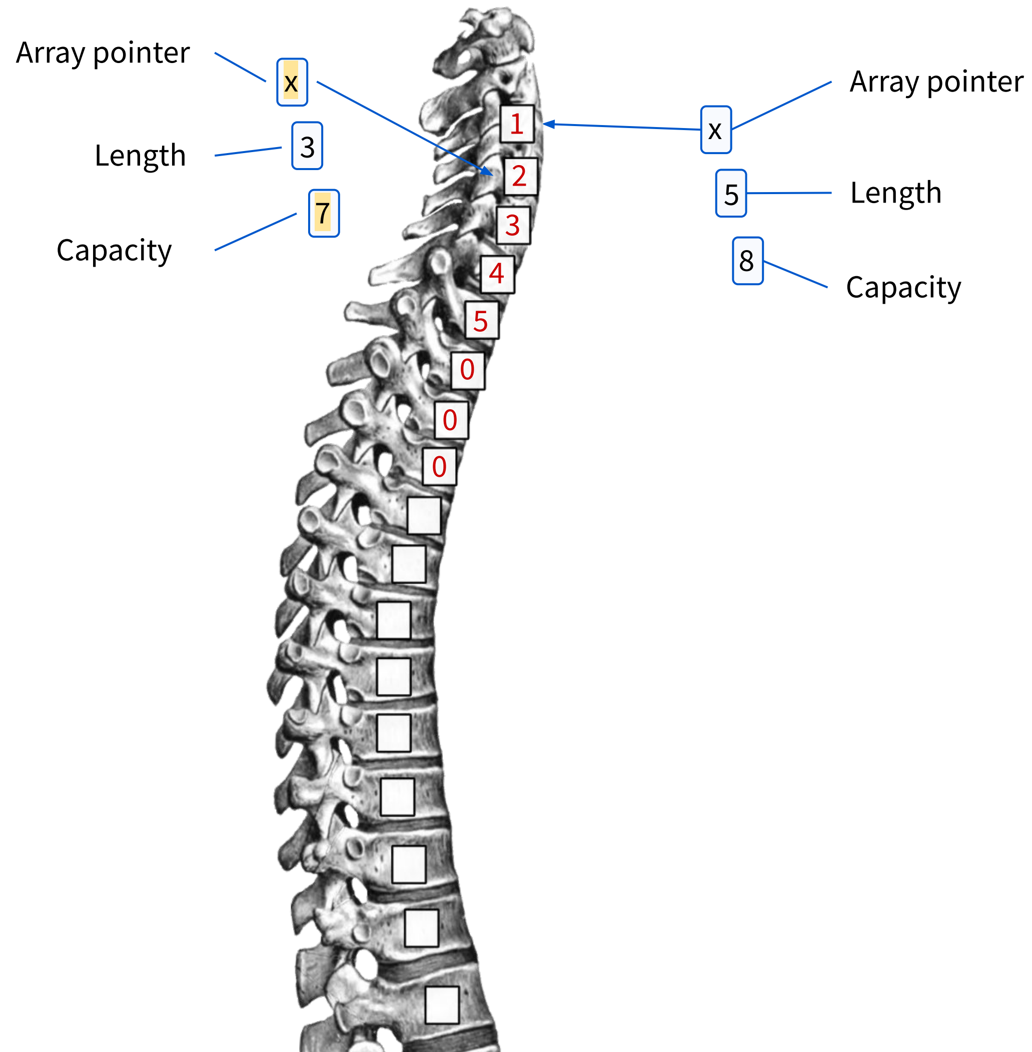

```
s = append(s, 1)
Microscope(ss)
--------------
Array Memory address: 0xc0000be000
Slice length: 1
Slice capacity: 1
Stored data: [1,]
```

Inserting more data into the slice



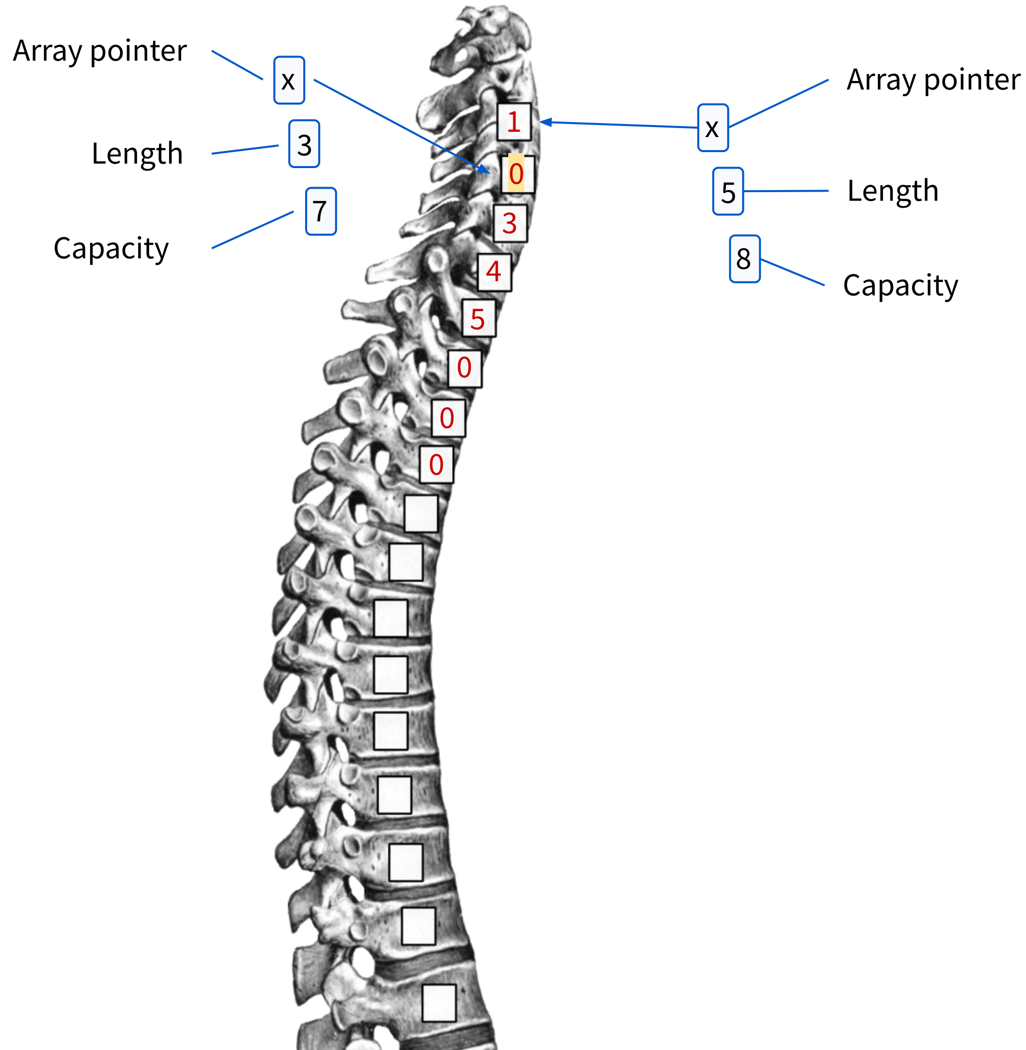

```
s = append(s, 2)
s = append(s, 3)
s = append(s, 4)
s = append(s, 5)
Microscope(ss)
--------------
Array Memory address: 0xc00001a540
Slice length: 5
Slice capacity: 8
Stored data: [1,2,3,4,5,0,0,0,]
```

Creating a “sub” slice



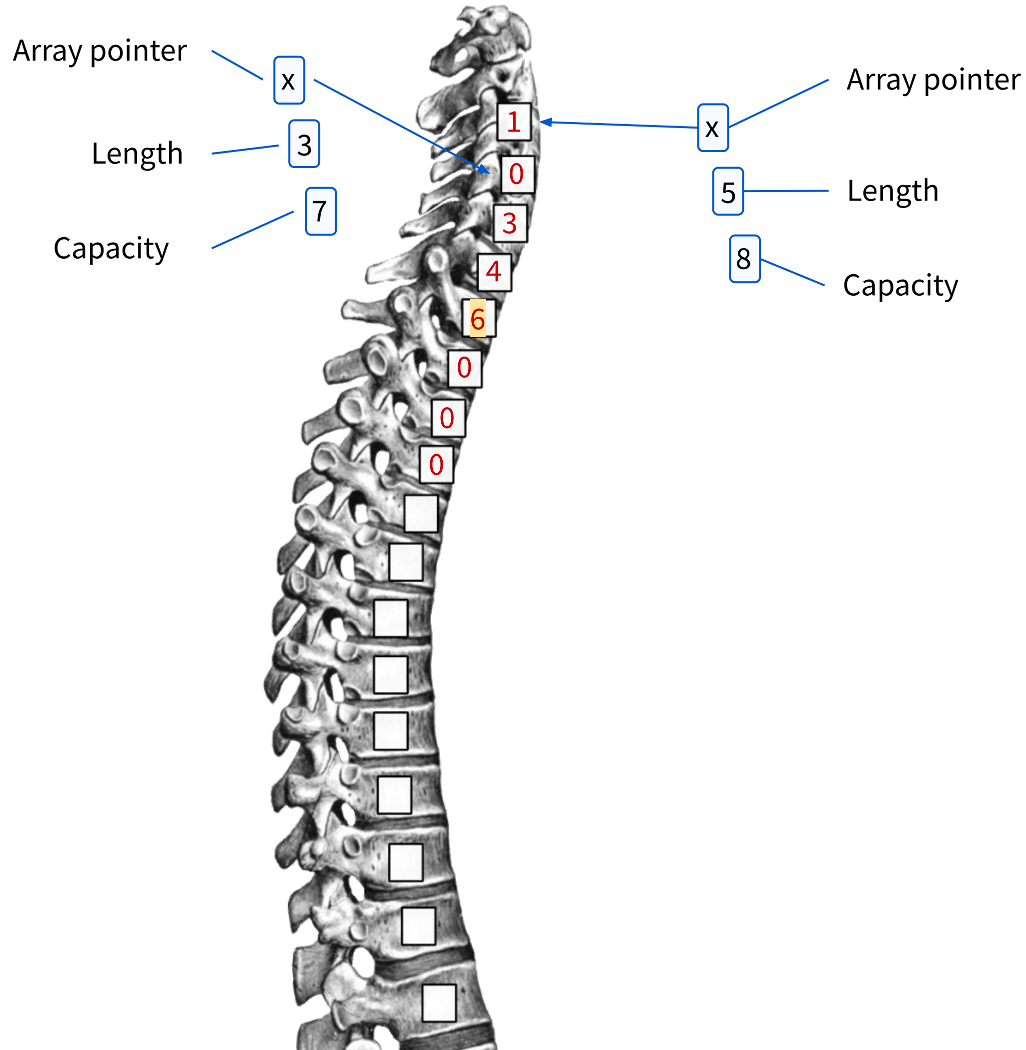

```
subSlice := s[1:4]
Microscope(Scalpel(&subSlice))
--------------
Array Memory address: 0xc00001a548
Slice length: 3
Slice capacity: 7
Stored data: [2,3,4,5,0,0,0,]
```

Setting a value in a subslice



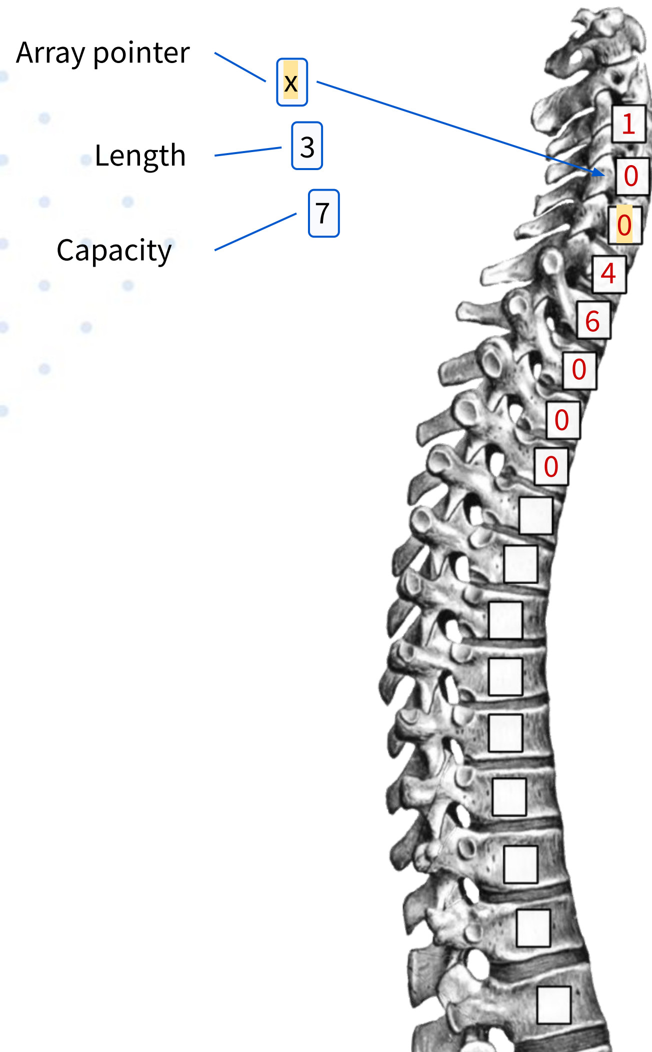

```
subSlice[0] = 0
Microscope(ss)
Microscope(Scalpel(&subSlice))
--------------
Array Memory address: 0xc00001a540
Slice length: 5
Slice capacity: 8
Stored data: [1,0,3,4,5,0,0,0,]
Array Memory address: 0xc00001a548
Slice length: 3
Slice capacity: 7
Stored data: [0,3,4,5,0,0,0,]
```

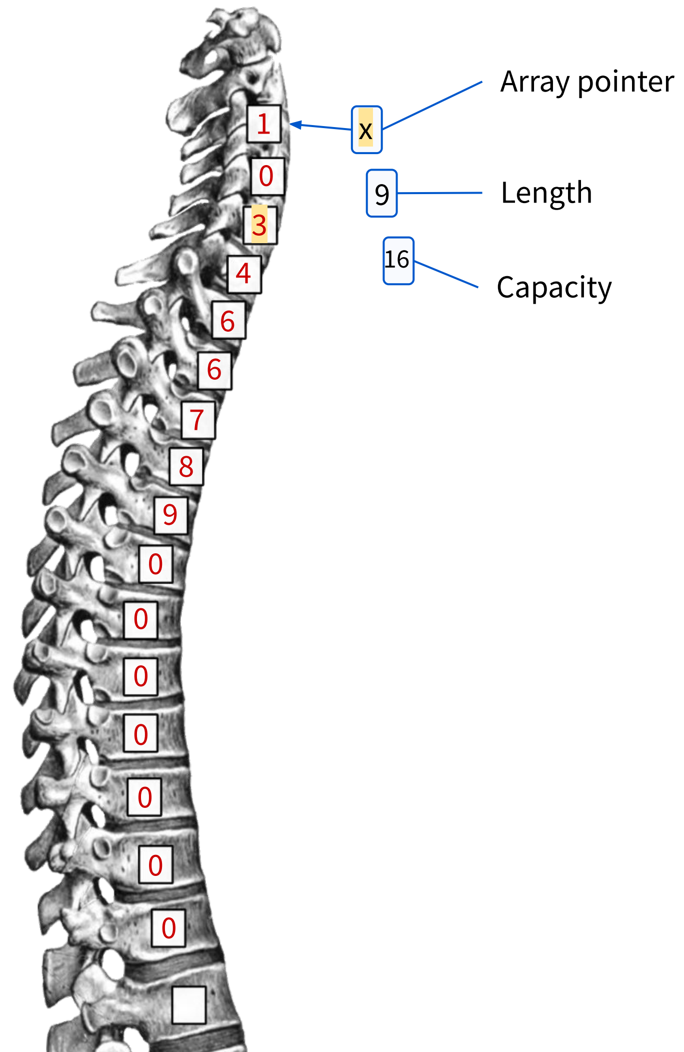
Appending a value in a subslice




```
subSlice = append(subSlice, 6)
Microscope(ss)
Microscope(Scalpel(&subSlice))
--------------
Array Memory address: 0xc00001a540
Slice length: 5
Slice capacity: 8
Stored data: [1,0,3,4,6,0,0,0,]
Array Memory address: 0xc00001a548
Slice length: 4
Slice capacity: 7
Stored data: [0,3,4,6,0,0,0,]
```

Gotcha!






```
s = append(s,6,7,8,9)
subSlice[1] = 0
Microscope(ss)
Microscope(Scalpel(&subSlice))
--------------
Array Memory address: 0xc000092000
Slice length: 9
Slice capacity: 16
Stored data: [1,0,3,4,6,6,7,8,9,0,0,0,0,0,0,0,]
Array Memory address: 0xc00001a548
Slice length: 3
Slice capacity: 7
Stored data: [0,0,4,6,0,0,0,]
```

The code

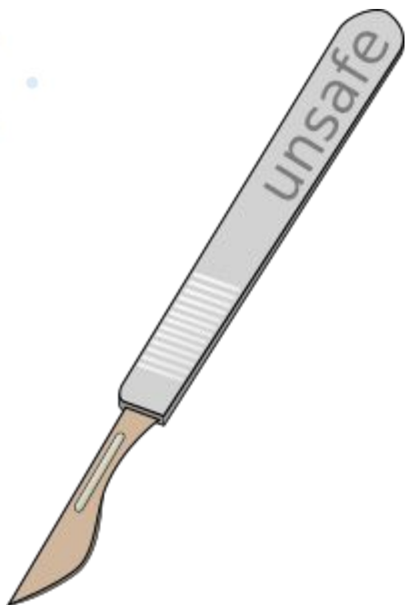
```go
package main

import (
	"fmt"
	"unsafe"
)

type sliceStruct struct {
	array unsafe.Pointer
	len   int
	cap   int
}
```

```go
func Scalpel(slice *[]int) *sliceStruct {
	ss := unsafe.Pointer(slice)
	return (*sliceStruct)(ss)
}

func Microscope(ss *sliceStruct) {
	fmt.Printf("Array Memory address: 0x%x\n",
ss.array)
	fmt.Printf("Slice length: %d\n", ss.len)
	fmt.Printf("Slice capacity: %d\n", ss.cap)
	fmt.Printf("Stored data: [")
	for x := 0; x < ss.cap; x++ {
		fmt.Printf("%d,",
*(*int)(unsafe.Pointer(uintptr(ss.array) +
uintptr(x)*unsafe.Sizeof(int(0)))))
	}
	fmt.Println("]")
}
```

```go
func main() {
	s := []int{}
	ss := Scalpel(&s)
	Microscope(ss)

	s = append(s, 1)
	Microscope(ss)

	s = append(s, 2)
	s = append(s, 3)
	s = append(s, 4)
	s = append(s, 5)
	Microscope(ss)

	subSlice := s[1:4]
	Microscope(Scalpel(&subSlice))

	subSlice[0] = 0
	Microscope(ss)
	Microscope(Scalpel(&subSlice))

	subSlice = append(subSlice, 6)
	Microscope(ss)
	Microscope(Scalpel(&subSlice))

	s = append(s, 6, 7, 8, 9)
	subSlice[1] = 0
	Microscope(ss)
	Microscope(Scalpel(&subSlice))
}
```

Maps

The scalpel

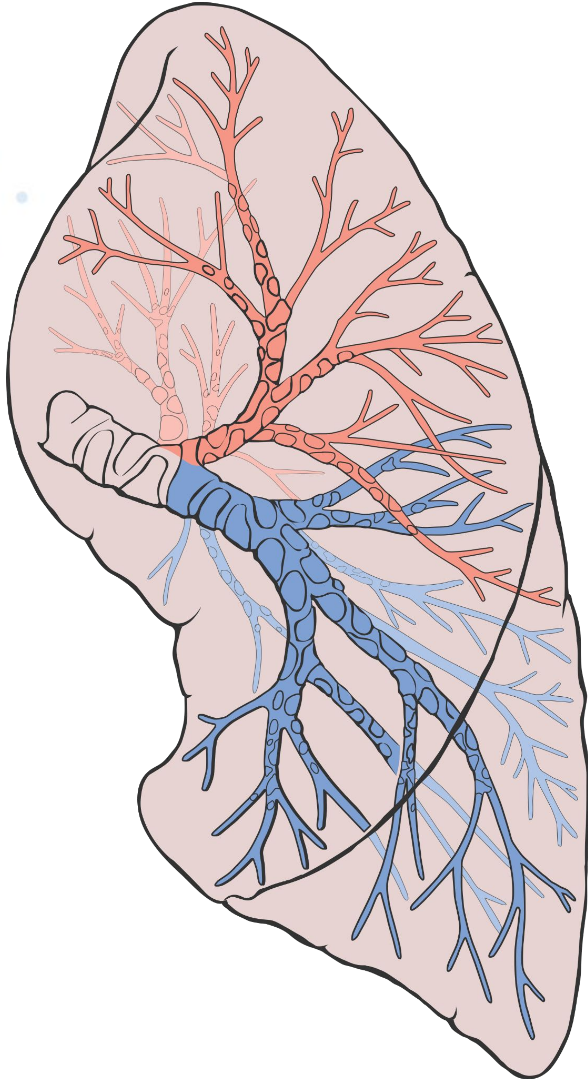



```
func Scalpel(mapValue *map[int]int) *mapStruct {
    ms := unsafe.Pointer(*(*uintptr)(unsafe.Pointer(mapValue)))
    return (*mapStruct)(ms)
}
```

The microscope

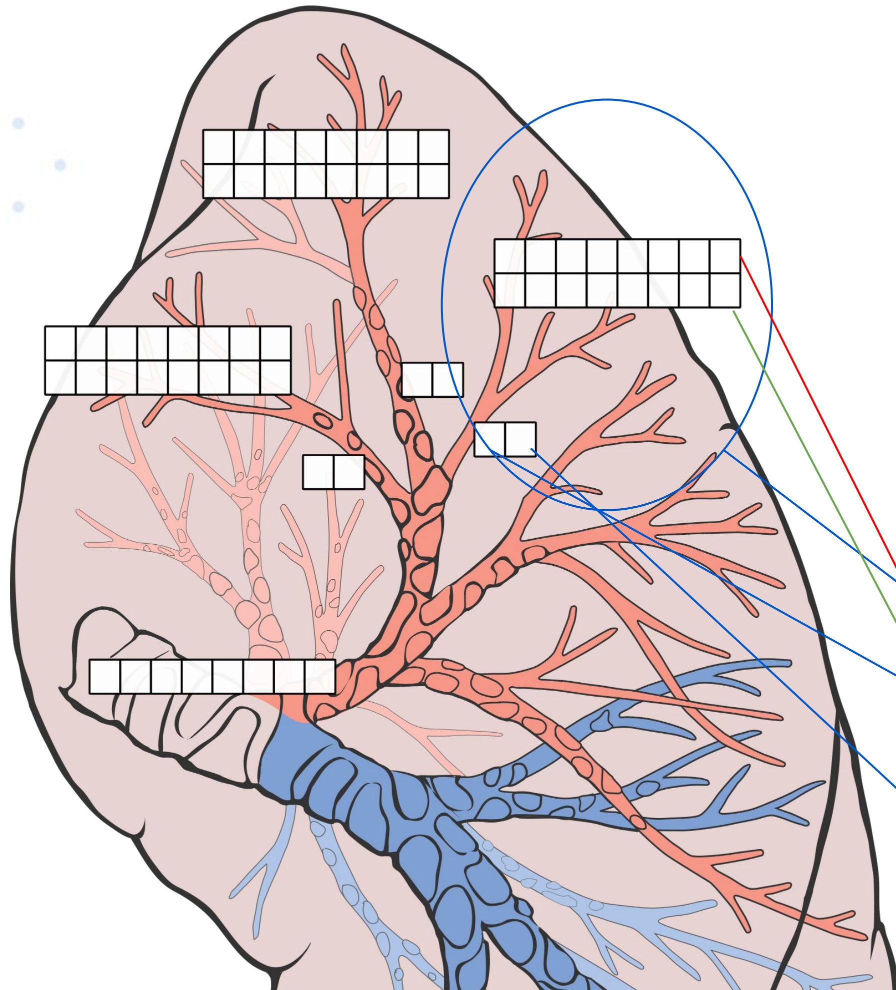
```
func Microscope(ms *mapStruct) {
    totalBuckets :=int(math.Pow(2, float64(ms.B)))
    oldTotalBuckets :=int(math.Pow(2, float64(ms.B-1)))
    fmt.Printf("Map size: %d\n", ms.count)
    fmt.Printf("Map flags: %d\n", ms.flags)
    fmt.Printf("Map B: %d\n", ms.B)
    fmt.Printf("Map number of overflow buckets (aprox): %d\n", ms.noverflow)
    fmt.Printf("Map hash seed: %d\n", ms.hash0)
    fmt.Printf("Map buckets: %v\n", ms.buckets)
    for x := 0; x < totalBuckets; x++ {
        bucket :=uintptr(ms.buckets) + unsafe.Sizeof(bucketStruct{})*uintptr(x)
        data := (*bucketStruct)(unsafe.Pointer(bucket))
        fmt.Printf("  Bucket %d:\n", x)
        fmt.Printf("    Tophash: %v\n", data.topHash)
        fmt.Printf("    Keys: %v\n", data.keys)
        fmt.Printf("    Values: %v\n", data.values)
        fmt.Printf("    OverflowPtr: %v\n", data.overflowPtr)
        if data.overflowPtr !=0 {
            ovfBucket := data.overflowPtr
            ovfData := (*bucketStruct)(unsafe.Pointer(ovfBucket))
            fmt.Printf("      Overflow, Tophash: %v, Keys: %v, Values: %v, OverflowPtr: %v\n", ovfData.topHash, ovfData.keys, ovfData.values, ovfData.overflowPtr)
        }
    }
    fmt.Printf("Map old buckets: %v\n", ms.oldbuckets)
    if ms.oldbuckets !=nil {
        for x := 0; x < oldTotalBuckets; x++ {
            bucket :=uintptr(ms.oldbuckets) + unsafe.Sizeof(bucketStruct{})*uintptr(x)
            data := (*bucketStruct)(unsafe.Pointer(bucket))
            fmt.Printf("  Bucket %d:\n", x)
            fmt.Printf("    Tophash: %v\n", data.topHash)
            fmt.Printf("    Keys: %v\n", data.keys)
            fmt.Printf("    Values: %v\n", data.values)
            fmt.Printf("    OverflowPtr: %v\n", data.overflowPtr)
            if data.overflowPtr !=0 {
                ovfBucket := data.overflowPtr
                ovfData := (*bucketStruct)(unsafe.Pointer(ovfBucket))
                fmt.Printf("      Overflow:\n")
                fmt.Printf("        Tophash: %v\n", ovfData.topHash)
                fmt.Printf("        Keys: %v\n", ovfData.keys)
                fmt.Printf("        Values: %v\n", ovfData.values)
                fmt.Printf("        OverflowPtr: %v\n", ovfData.overflowPtr)
            }
        }
    }
    fmt.Printf("Map number of evacuated buckets: %d\n", ms.nevacuate)
}
```

The subject



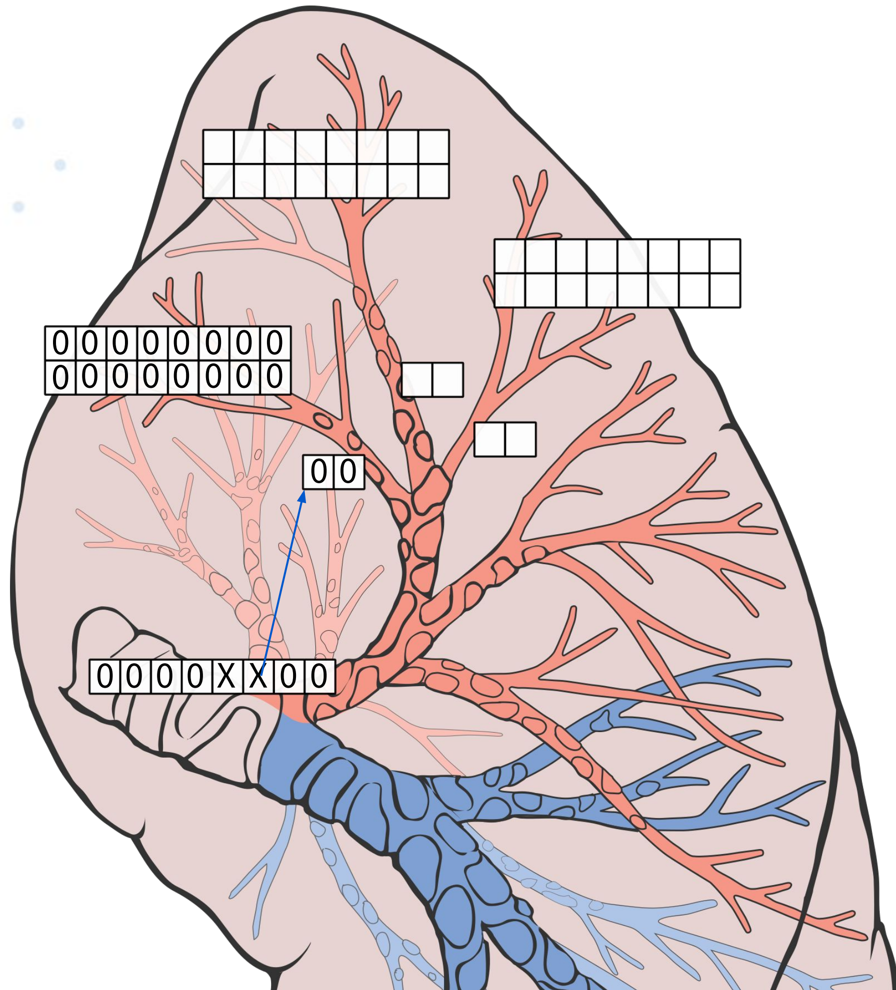
- Map metadata
- Some buckets to store the data

Inside the subject



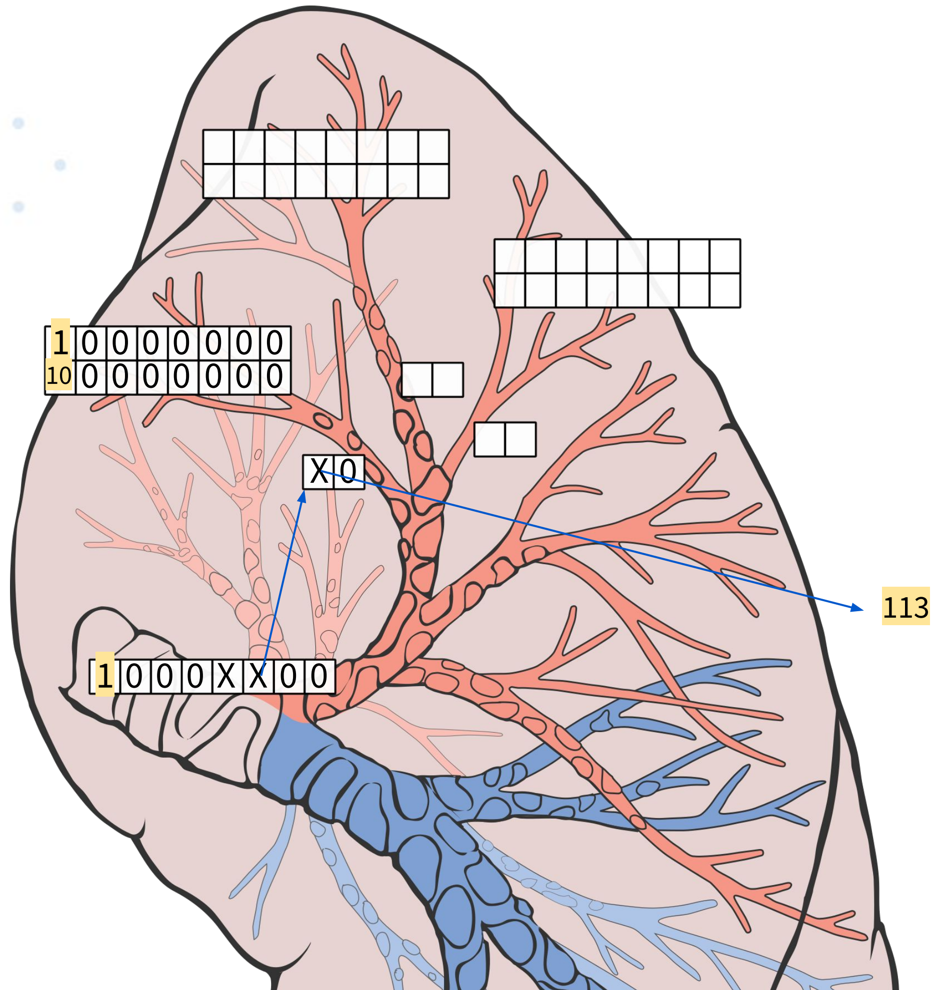
```
type mapStruct struct {
      count     int
      flags     uint8
      B         uint8
      noverflow uint16
      hash0     uint32
      buckets    unsafe.Pointer
      oldbuckets unsafe.Pointer
      nevacuate  uintptr
      extra *struct {
            overflow    []*bucketStruct
            oldoverflow []*bucketStruct
            nextOverflow *bucketStruct
      }
}

type bucketStruct struct {
      topHash     uint64
      keys        [8]int
      values      [8]int
      overflowPtr uintptr
}
```

Map creation



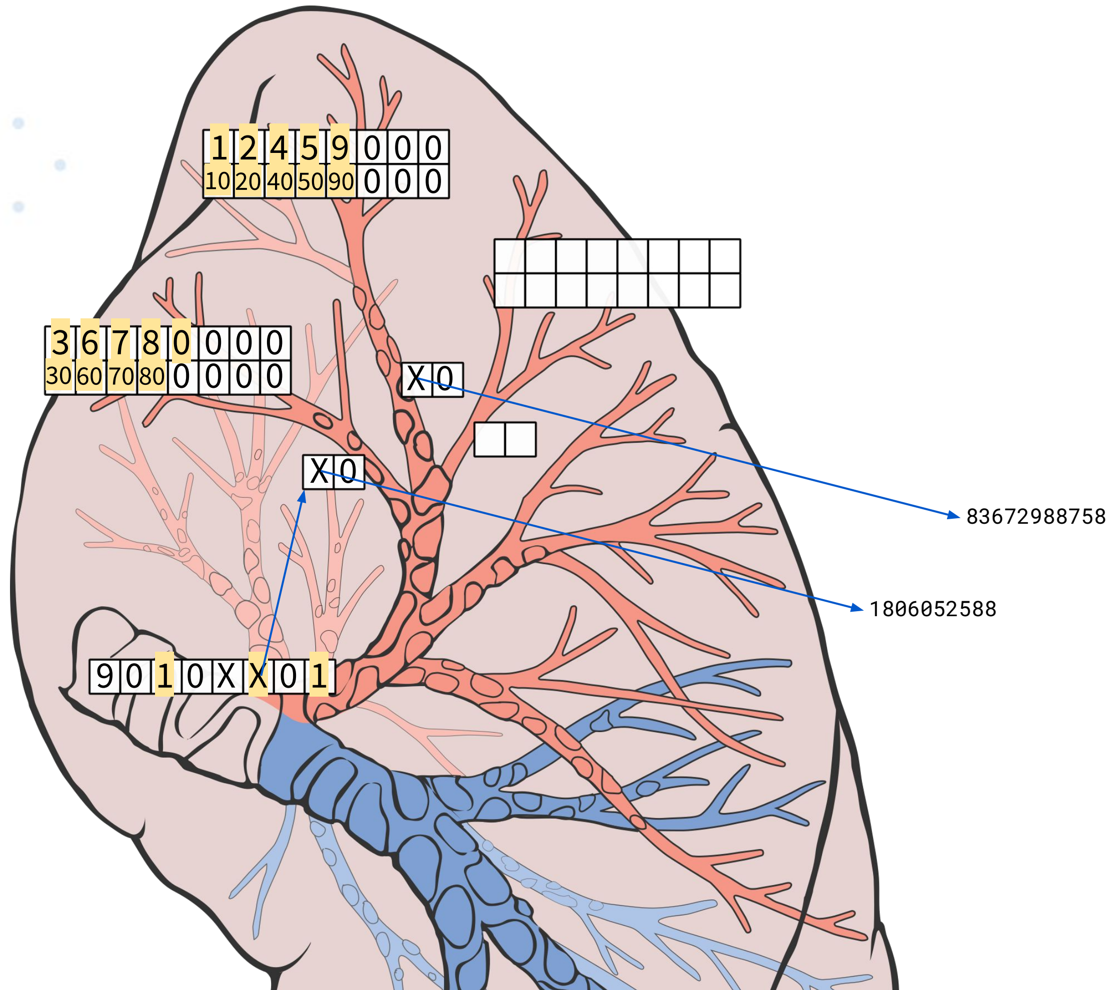

```
m := map[int]int{}
ms = Scalpel(&m)
Microscope(ms)
---------------
Map size: 0
Map flags: 0
Map B: 0
Map number of overflow buckets (aprox): 0
Map hash seed: 3390069684
Map buckets: 0xc000108ea0
  Bucket 0:
    Tophash: 0
    Keys: [0 0 0 0 0 0 0 0]
    Values: [0 0 0 0 0 0 0 0]
    OverflowPtr: 0
Map old buckets: <nil>
Map number of evacuated buckets: 0
```

Insert something into the map



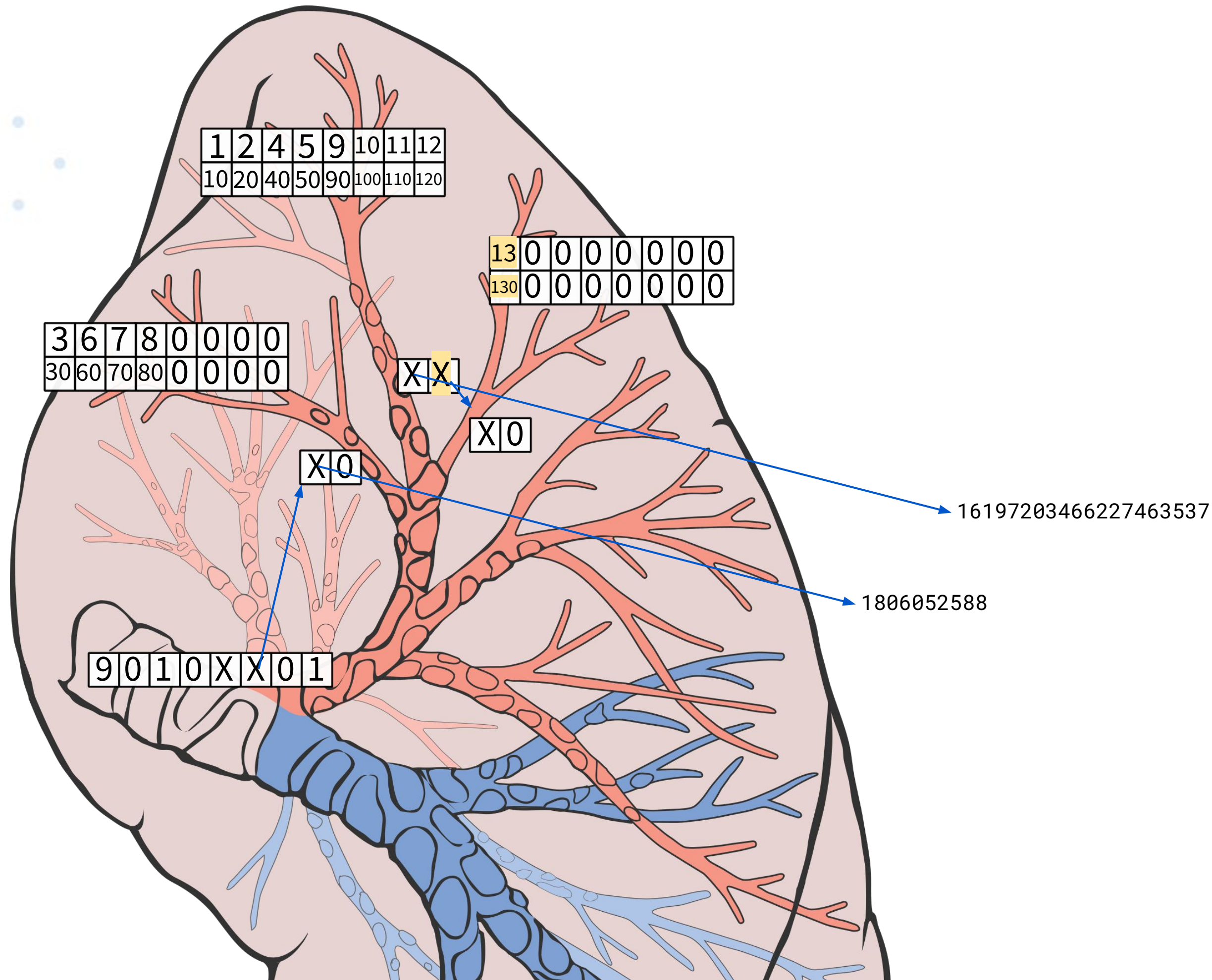

```
m[1] = 10
Microscope(ms)
--------------
Map size: 1
Map flags: 0
Map B: 0
Map number of overflow buckets (aprox): 0
Map hash seed: 3390069684
Map buckets: 0xc000108ea0
  Bucket 0:
    Tophash: 113
    Keys: [1 0 0 0 0 0 0 0]
    Values: [10 0 0 0 0 0 0 0]
    OverflowPtr: 0
Map old buckets: <nil>
Map number of evacuated buckets: 0
```

Insert more data into the map




```
m[2] = 20
m[3] = 30
m[4] = 40
m[5] = 50
m[6] = 60
m[7] = 70
m[8] = 80
m[9] = 90
Microscope(ms)
--------------
Map size: 9
Map flags: 0
Map B: 1
Map number of overflow buckets (aprox): 0
Map hash seed: 3390069684
Map buckets: 0xc0000c0000
  Bucket 0:
    Tophash: 1806052588
    Keys: [3 6 7 8 0 0 0 0]
    Values: [30 60 70 80 0 0 0 0]
    OverflowPtr: 0
  Bucket 1:
    Tophash: 83672988758
    Keys: [1 2 4 5 9 0 0 0]
    Values: [10 20 40 50 90 0 0 0]
    OverflowPtr: 0
Map old buckets: <nil>
Map number of evacuated buckets: 1
```

Overflows




```
m[10] = 100
m[11] = 110
m[12] = 120
m[13] = 130
Microscope(ms)
--------------
Map size: 13
Map flags: 0
Map B: 1
Map number of overflow buckets (aprox): 1
Map hash seed: 3390069684
Map buckets: 0xc0000c0000
  Bucket 0:
    Tophash: 1806052588
    Keys: [3 6 7 8 0 0 0 0]
    Values: [30 60 70 80 0 0 0 0]
    OverflowPtr: 0
  Bucket 1:
    Tophash: 16197203466227463537
    Keys: [1 2 4 5 9 10 11 12]
    Values: [10 20 40 50 90 100 110 120]
    OverflowPtr: 824634851328
      Tophash: 52
      Keys: [13 0 0 0 0 0 0 0]
      Values: [130 0 0 0 0 0 0 0]
      OverflowPtr: 0
Map old buckets: <nil>
Map number of evacuated buckets: 1
```

Big resizes

- New set of buckets get reserved.
- New values are inserted in the new bucket.
- Old buckets are still in use.
- The data gets migrated gradually during subsequent operations.

The code

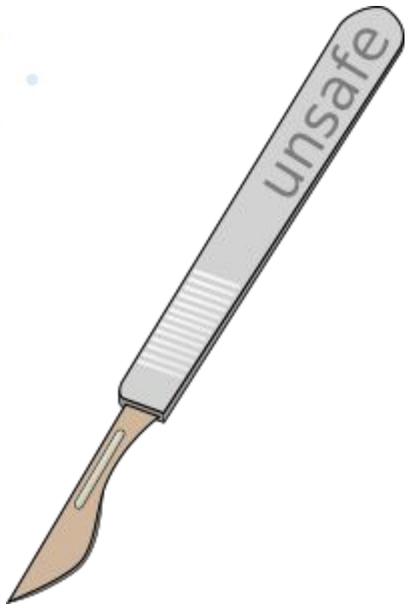
```go
package main

import (
    "fmt"
    "math"
    "unsafe"
)

type mapStruct struct {
    count     int
    flags     uint8
    B         uint8
    noverflow uint16
    hash0     uint32

    buckets    unsafe.Pointer
    oldbuckets unsafe.Pointer
    nevacuate  uintptr

    extra *struct {
        overflow     []*bucketStruct
        oldoverflow  []*bucketStruct
        nextOverflow *bucketStruct
    }
}

type bucketStruct struct {
    topHash     uint64
    keys        [8]int
    values      [8]int
    overflowPtr uintptr
 }
```

```go
func Scalpel(mapValue *map[int]int) *mapStruct {
    ms := unsafe.Pointer(*(*uintptr)(unsafe.Pointer(mapValue)))
    return (*mapStruct)(ms)
}

func Microscope(ms *mapStruct) {
    totalBuckets := int(math.Pow(2, float64(ms.B)))
    oldTotalBuckets := int(math.Pow(2, float64(ms.B-1)))
    fmt.Printf("Map size: %d\n", ms.count)
    fmt.Printf("Map flags: %d\n", ms.flags)
    fmt.Printf("Map B: %d\n", ms.B)
    fmt.Printf("Map number of overflow buckets (aprox): %d\n", ms.noverflow)
    fmt.Printf("Map hash seed: %d\n", ms.hash0)
    fmt.Printf("Map buckets: %v\n", ms.buckets)
    for x := 0; x < totalBuckets; x++ {
        bucket := uintptr(ms.buckets) + unsafe.Sizeof(bucketStruct{})*uintptr(x)
        data := (*bucketStruct)(unsafe.Pointer(bucket))
        fmt.Printf("  Bucket %d:\n", x)
        fmt.Printf("    Tophash: %v\n", data.topHash)
        fmt.Printf("    Keys: %v\n", data.keys)
        fmt.Printf("    Values: %v\n", data.values)
        fmt.Printf("    OverflowPtr: %v\n", data.overflowPtr)
        if data.overflowPtr != 0 {
            ovfBucket := data.overflowPtr
            ovfData := (*bucketStruct)(unsafe.Pointer(ovfBucket))
            fmt.Printf("      Overflow, Tophash: %v, Keys: %v, Values: %v, OverflowPtr: %v\n", ovfData.topHash, ovfData.keys, ovfData.values, ovfData.overflowPtr)
        }
    }
    fmt.Printf("Map old buckets: %v\n", ms.oldbuckets)
    if ms.oldbuckets != nil {
        for x := 0; x < oldTotalBuckets; x++ {
            bucket := uintptr(ms.oldbuckets) + unsafe.Sizeof(bucketStruct{})*uintptr(x)
            data := (*bucketStruct)(unsafe.Pointer(bucket))
            fmt.Printf("  Bucket %d:\n", x)
            fmt.Printf("    Tophash: %v\n", data.topHash)
            fmt.Printf("    Keys: %v\n", data.keys)
            fmt.Printf("    Values: %v\n", data.values)
            fmt.Printf("    OverflowPtr: %v\n", data.overflowPtr)
            if data.overflowPtr != 0 {
                ovfBucket := data.overflowPtr
                ovfData := (*bucketStruct)(unsafe.Pointer(ovfBucket))
                fmt.Printf("      Overflow:\n")
                fmt.Printf("        Tophash: %v\n", ovfData.topHash)
                fmt.Printf("        Keys: %v\n", ovfData.keys)
                fmt.Printf("        Values: %v\n", ovfData.values)
                fmt.Printf("        OverflowPtr: %v\n", ovfData.overflowPtr)
            }
        }
    }
    fmt.Printf("Map number of evacuated buckets: %d\n", ms.nevacuate)
 }
```

```go
func main() {
    m := map[int]int{}
    ms := Scalpel(&m)
    Microscope(ms)

    m[1] = 10
    Microscope(ms)
    m[2] = 20
    m[3] = 30
    m[4] = 40
    m[5] = 50
    m[6] = 60
    m[7] = 70
    m[8] = 80
    m[9] = 90
    Microscope(ms)
    m[10] = 100
    m[11] = 110
    m[12] = 120
    m[13] = 130
    Microscope(ms)
 }
```

Channels

The scalpel

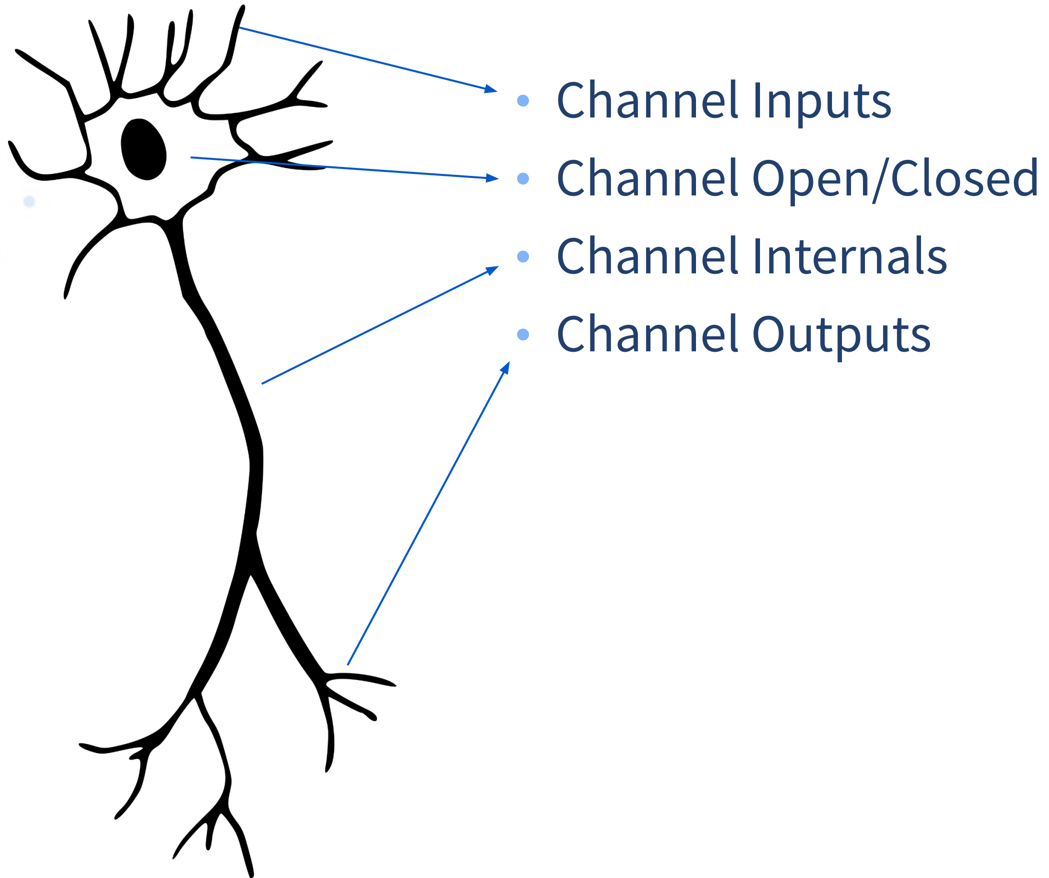



```go
func Scalpel(channel *(chan int32)) *channelStruct {
    cs := unsafe.Pointer(*(*uintptr)(unsafe.Pointer(channel)))
    return (*channelStruct)(cs)
}
```

The microscope

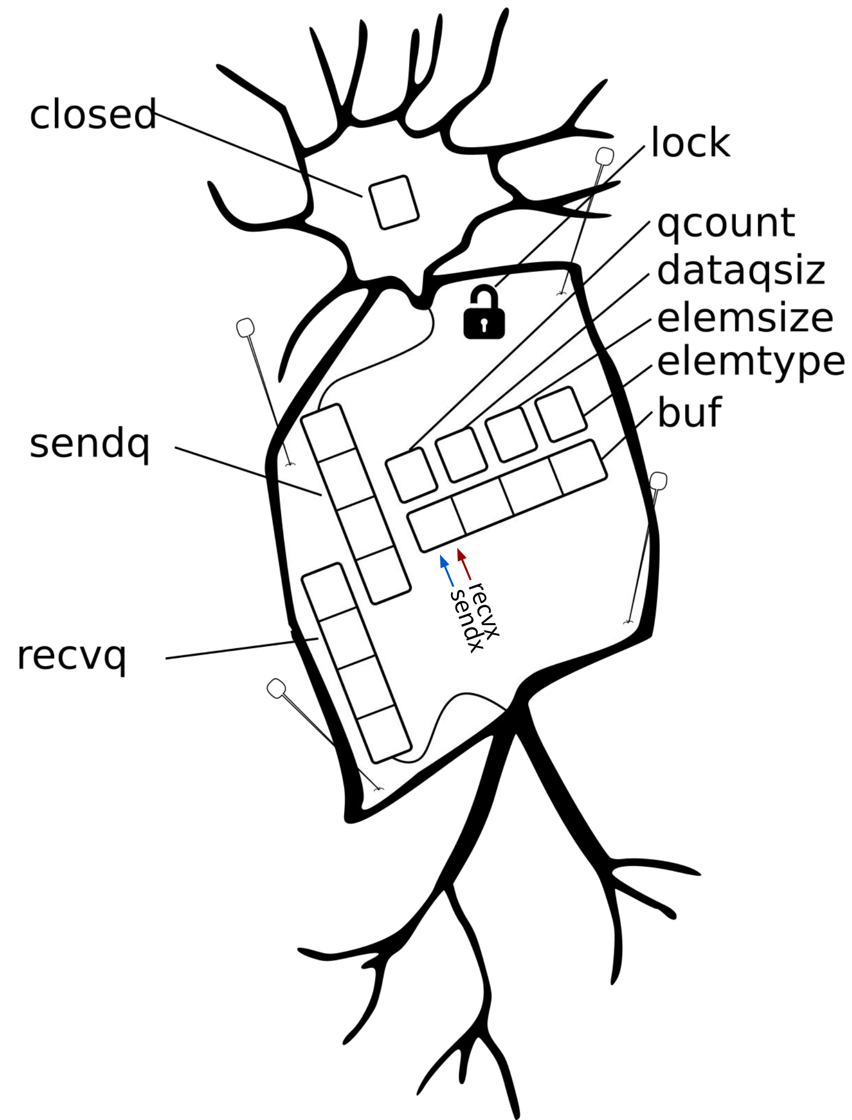
```go
func Microscope(cs *channelStruct) {
    fmt.Printf("Total data in queue: %d\n", cs.qcount)
    fmt.Printf("Size of the queue: %d\n", cs.dataqsiz)
    fmt.Printf("Buffer address: %p\n", cs.buf)
    fmt.Printf("Element size: %d\n", cs.elemsize)
    fmt.Printf("Queued elements: %v\n", *cs.buf)
    fmt.Printf("Closed: %d\n", cs.closed)
    fmt.Printf("Element Type Address: %d\n", cs.elemtype)
    fmt.Printf("Send Index: %d\n", cs.sendx)
    fmt.Printf("Receive Index: %d\n", cs.recvx)
    fmt.Printf("Receive Wait list first address: 0x%x\n", cs.recvq.first)
    fmt.Printf("Receive Wait list last address: 0x%x\n", cs.recvq.last)
    fmt.Printf("Send Wait list first address: 0x%x\n", cs.sendq.first)
    fmt.Printf("Send Wait list last address: 0x%x\n", cs.sendq.last)
    fmt.Println("-------------------------------")
}
```

The subject



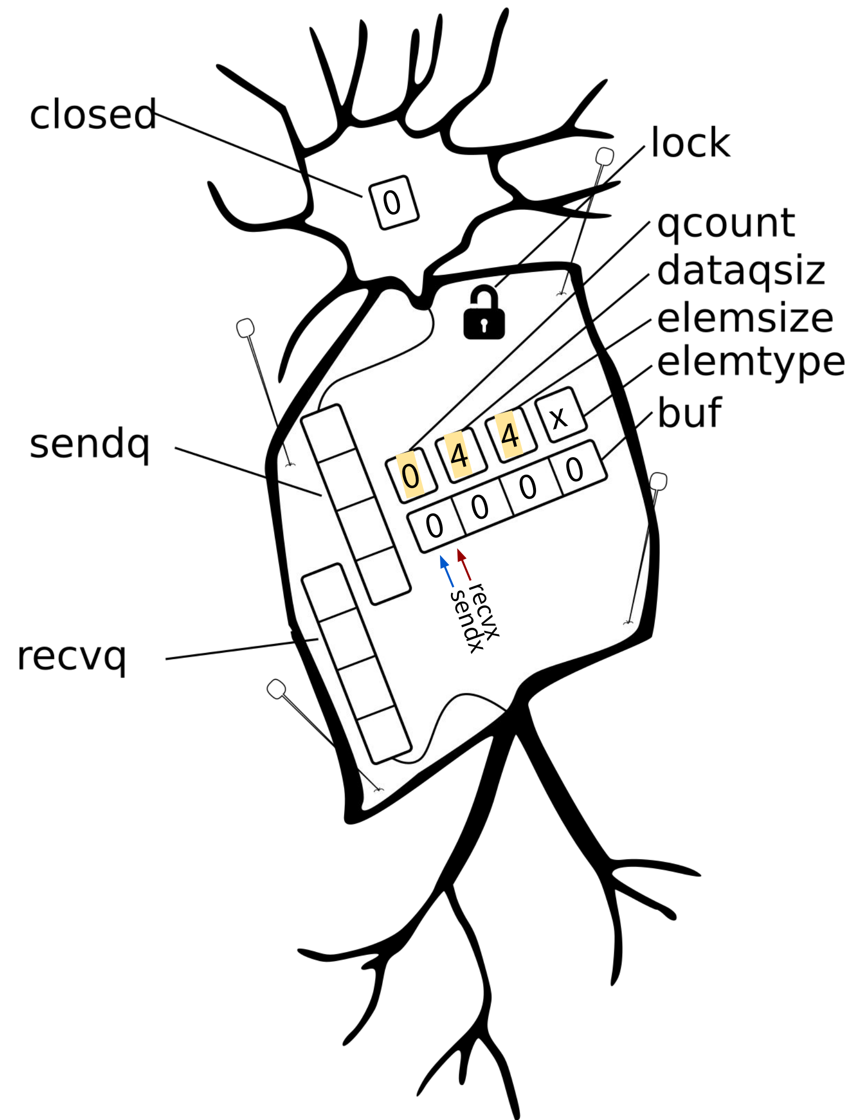
- Channel Inputs
- Channel Open/Closed
- Channel Internals
- Channel Outputs

Inside the subject



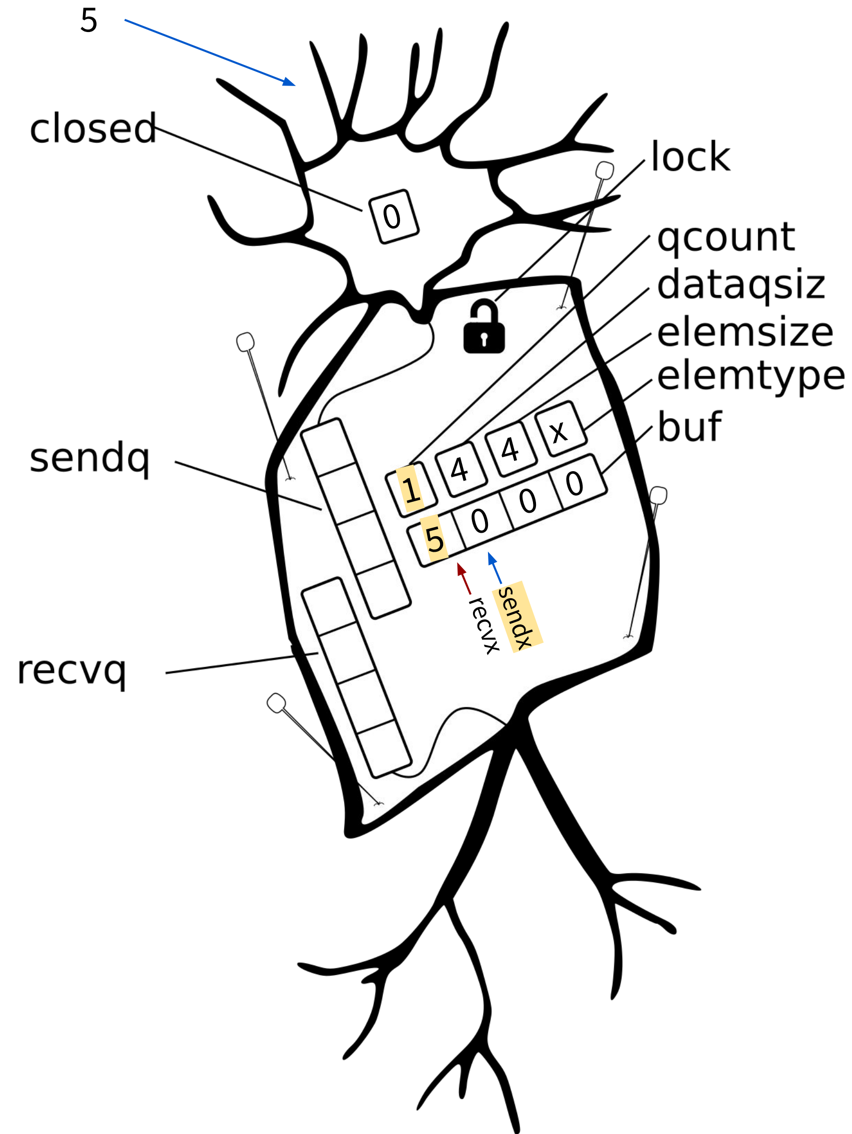

```
type waitq struct {
    first uintptr
    last  uintptr
}

type channelStruct struct {
    qcount   uint
    dataqsiz uint
    buf      *[4]int32
    elemsize uint16
    closed   uint32
    elemtype uintptr
    sendx    uint
    recvx    uint
    recvq    waitq
    sendq    waitq
    lock uintptr
}
```

Channel creation



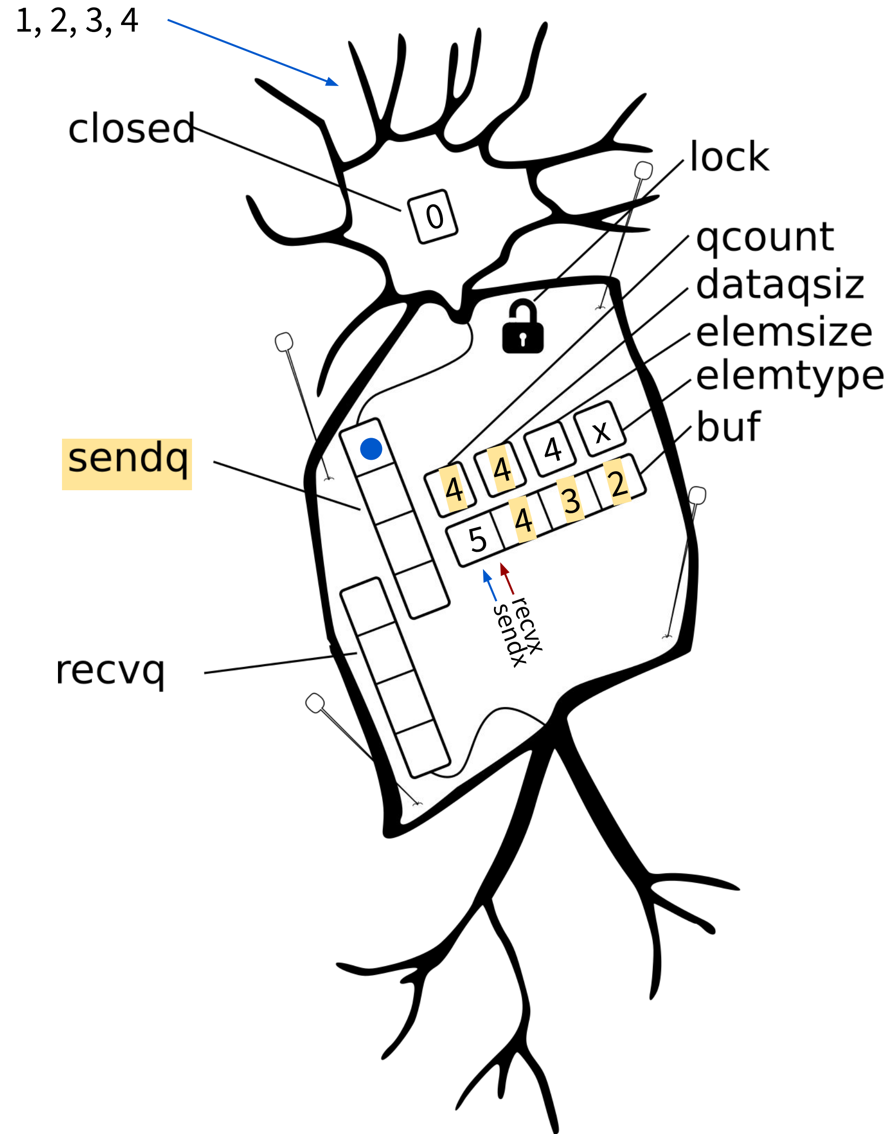

```
c := make(chan int32, 4)
cs = Scalpel(&c)
Microscope(cs)
--------------
Total data in queue: 0
Size of the queue: 4
Buffer address: 0xc000130060
Element size: 4
Queued elements: [0 0 0 0]
Closed: 0
Element Type Address: 4870720
Send Index: 0
Receive Index: 0
Receive Wait list first address: 0x0
Receive Wait list last address: 0x0
Send Wait list first address: 0x0
Send Wait list last address: 0x0
```

Insert something into the channel



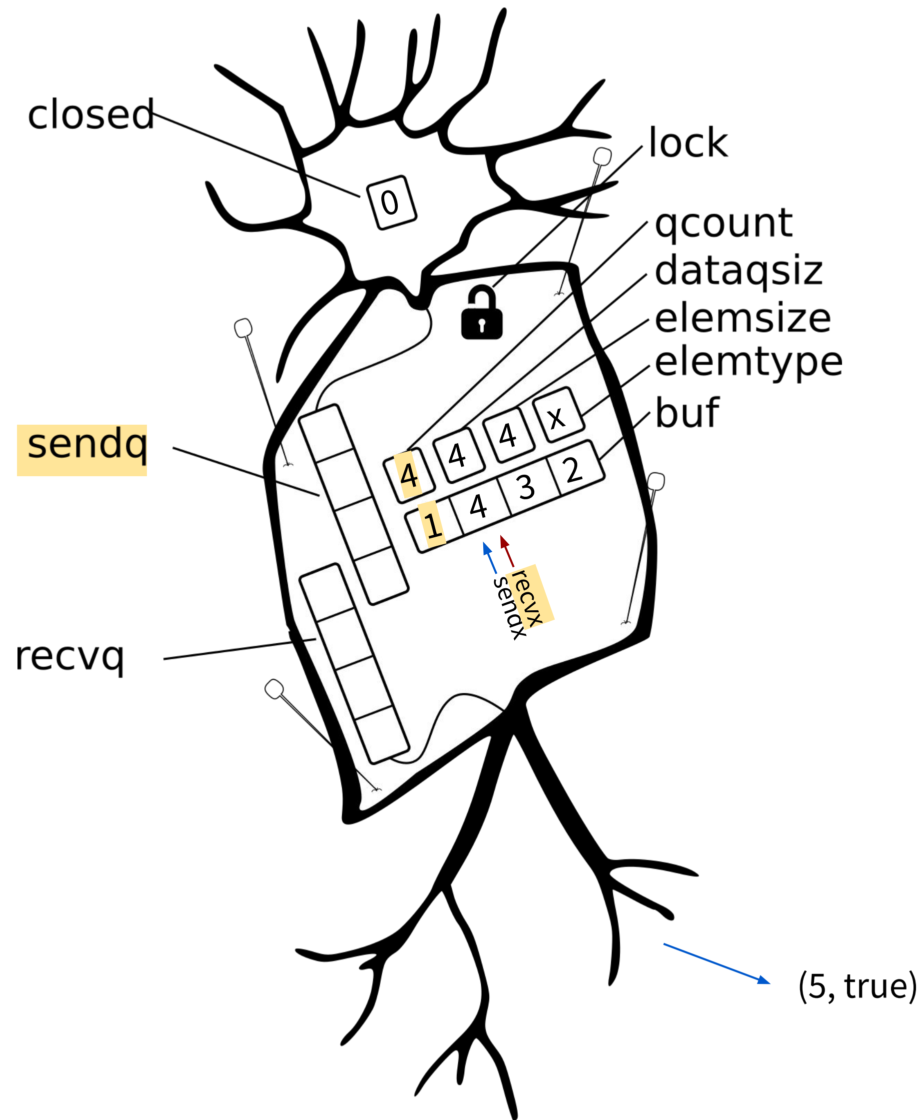

```
c <- 5
Microscope(cs)
--------------
Total data in queue: 1
Size of the queue: 4
Buffer address: 0xc000130060
Element size: 4
Queued elements: [5 0 0 0]
Closed: 0
Element Type Address: 4870720
Send Index: 1
Receive Index: 0
Receive Wait list first address: 0x0
Receive Wait list last address: 0x0
Send Wait list first address: 0x0
Send Wait list last address: 0x0
```

Fill in the channel buffer




```
c <- 4
c <- 3
c <- 2
c <- 1
Microscope(cs)
--------------
Total data in queue: 4
Size of the queue: 4
Buffer address: 0xc000130060
Element size: 4
Queued elements: [5 4 3 2]
Closed: 0
Element Type Address: 4870720
Send Index: 0
Receive Index: 0
Receive Wait list first address: 0x0
Receive Wait list last address: 0x0
Send Wait list first address: 0xc000028060
Send Wait list last address: 0xc000028060
```

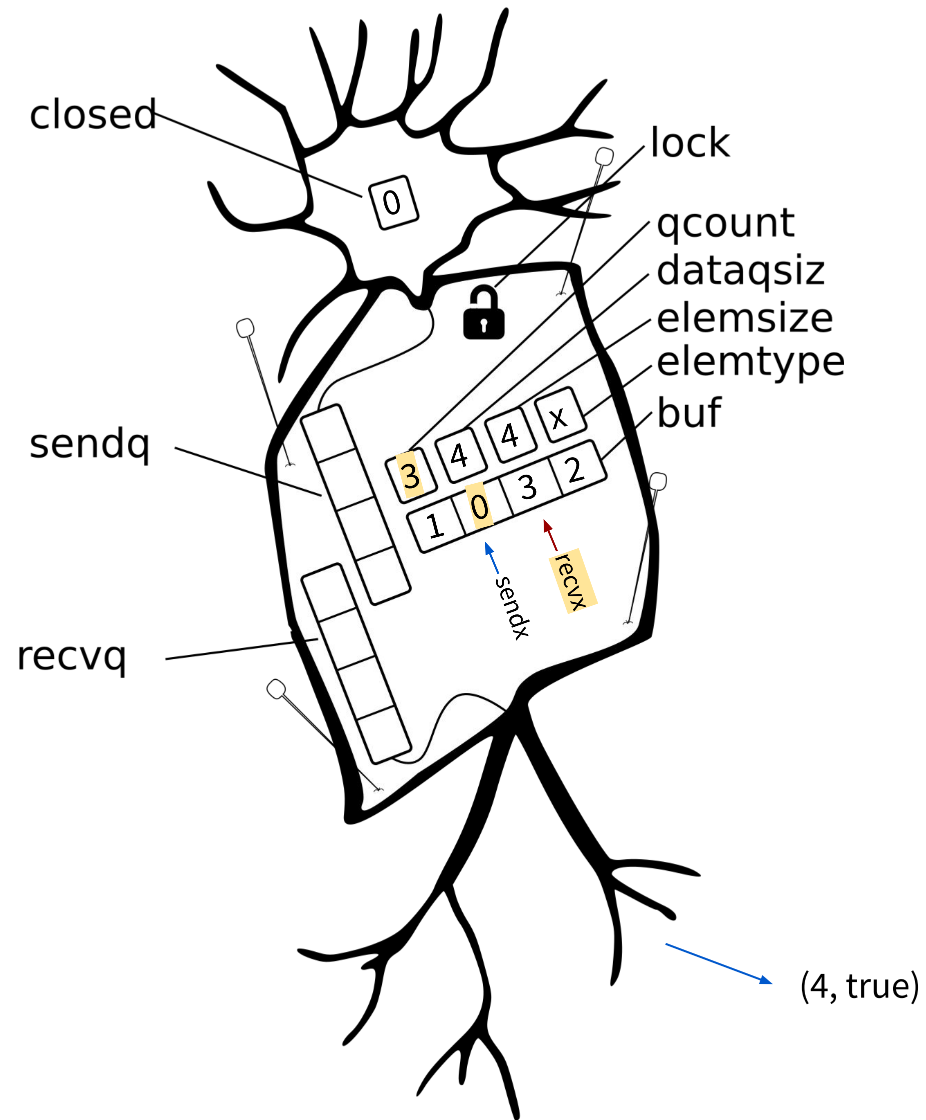
Read from a channel



(5, true)

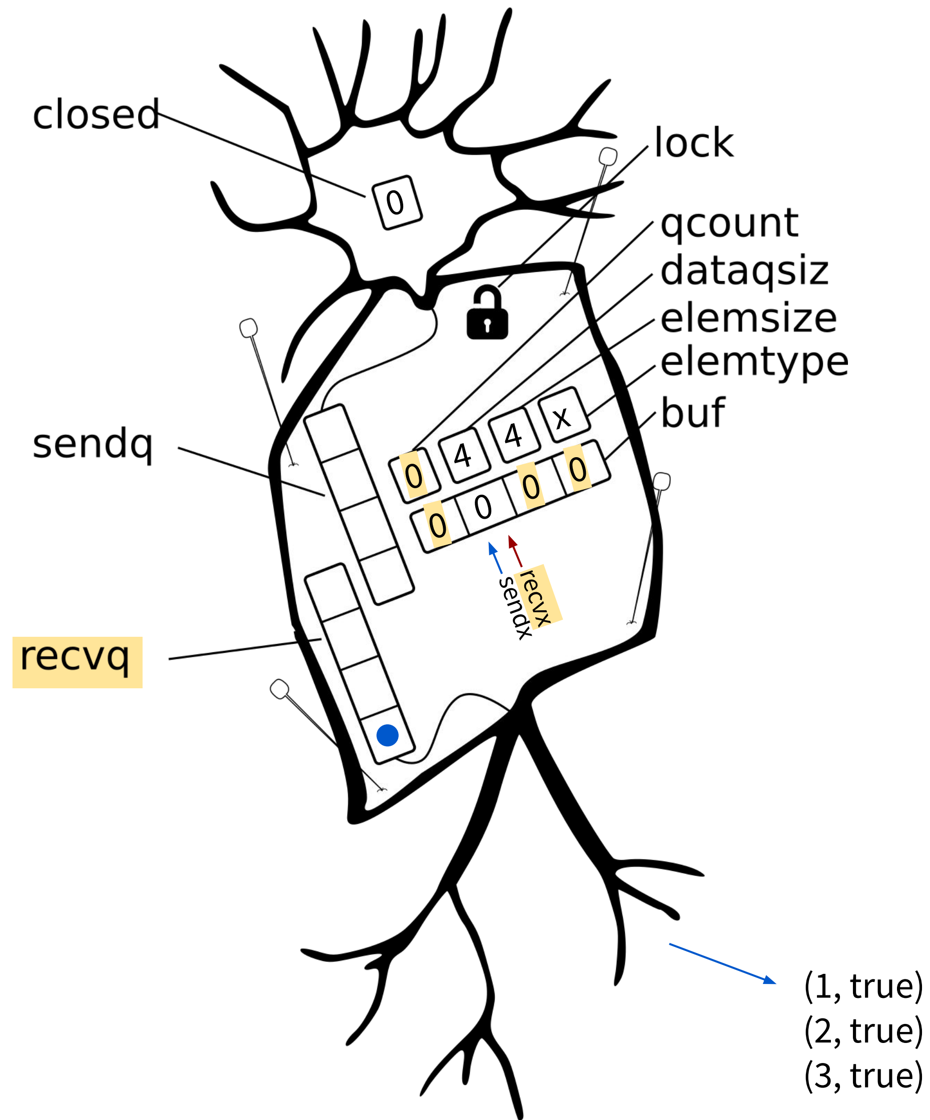
```
<-c
Microscope(cs)
--------------
Total data in queue: 4
Size of the queue: 4
Buffer address: 0xc000130060
Element size: 4
Queued elements: [1 4 3 2]
Closed: 0
Element Type Address: 4870720
Send Index: 1
Receive Index: 1
Receive Wait list first address: 0x0
Receive Wait list last address: 0x0
Send Wait list first address: 0x0
Send Wait list last address: 0x0
```

More reading from a channel



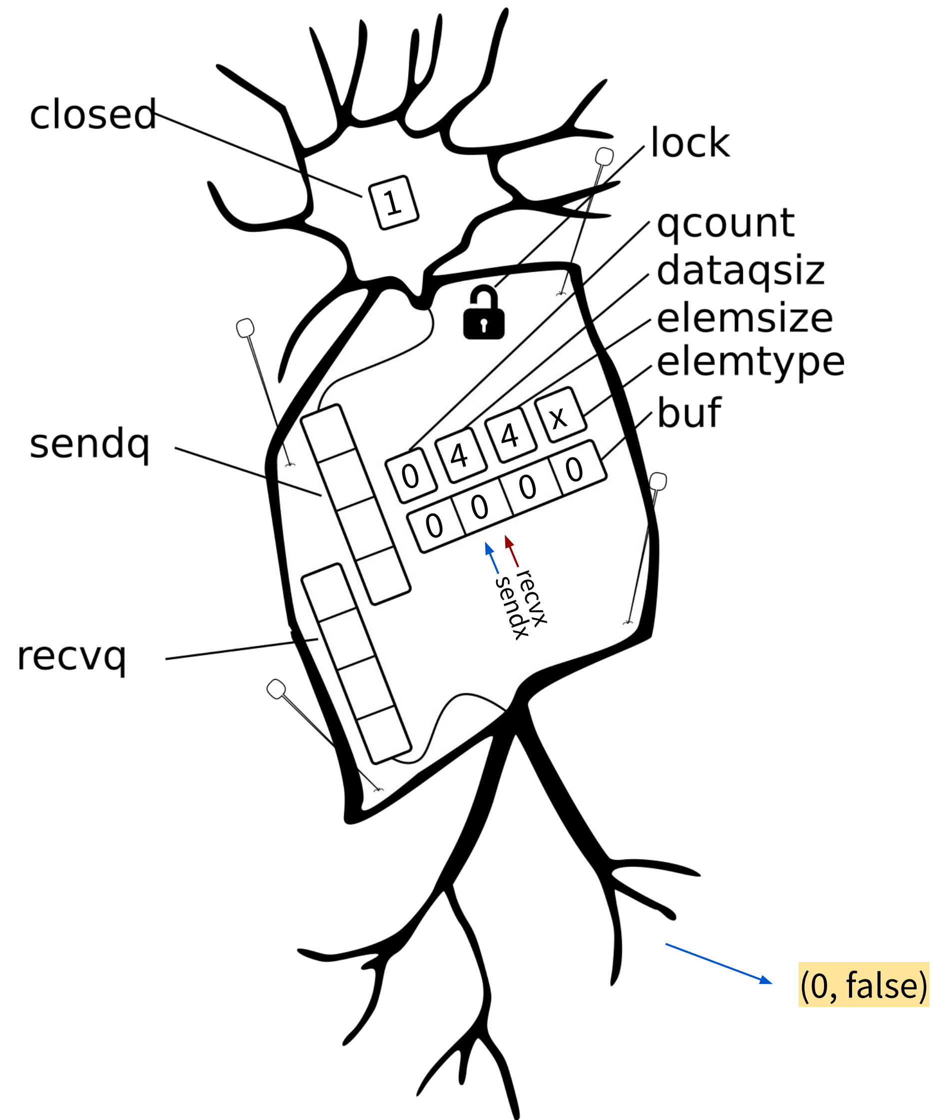

```
<-c
Microscope(cs)
--------------
Total data in queue: 3
Size of the queue: 4
Buffer address: 0xc000130060
Element size: 4
Queued elements: [1 0 3 2]
Closed: 0
Element Type Address: 4870720
Send Index: 1
Receive Index: 2
Receive Wait list first address: 0x0
Receive Wait list last address: 0x0
Send Wait list first address: 0x0
Send Wait list last address: 0x0
```

Wait for channel data




```
<-c
<-c
<-c
<-c
Microscope(cs)
--------------
Total data in queue: 0
Size of the queue: 4
Buffer address: 0xc000130060
Element size: 4
Queued elements: [0 0 0 0]
Closed: 0
Element Type Address: 4870720
Send Index: 1
Receive Index: 1
Receive Wait list first address: 0xc000194000
Receive Wait list last address: 0xc000194000
Send Wait list first address: 0x0
Send Wait list last address: 0x0
```

Close a channel



(0, false)

```
close(c)
Microscope(cs)
--------------
Total data in queue: 0
Size of the queue: 4
Buffer address: 0xc000130060
Element size: 4
Queued elements: [0 0 0 0]
Closed: 1
Element Type Address: 4870720
Send Index: 1
Receive Index: 1
Receive Wait list first address: 0x0
Receive Wait list last address: 0x0
Send Wait list first address: 0x0
Send Wait list last address: 0x0
```

The code

```go
package main

import (
    "fmt"
    "time"
    "unsafe"
)

type waitq struct {
    first uintptr
    last  uintptr
}

type channelStruct struct {
    qcount   uint      // total data in the queue
    dataqsiz uint      // size of the circular queue
    buf      *[4]int32 // points to an array of dataqsiz elements
    elemsize uint16
    closed   uint32
    elemtype uintptr // element type
    sendx    uint    // send index
    recvx    uint    // receive index
    recvq    waitq   // list of recv waiters
    sendq    waitq   // list of send waiters
    lock     uintptr
}
```

```go
func Scalpel(channel *(chan int32)) *channelStruct {
    cs := unsafe.Pointer(*(*uintptr)(unsafe.Pointer(channel)))
    return (*channelStruct)(cs)
}

func Microscope(cs *channelStruct) {
    fmt.Printf("Total data in queue: %d\n", cs.qcount)
    fmt.Printf("Size of the queue: %d\n", cs.dataqsiz)
    fmt.Printf("Buffer address: %p\n", cs.buf)
    fmt.Printf("Element size: %d\n", cs.elemsize)
    fmt.Printf("Queued elements: %v\n", *cs.buf)
    fmt.Printf("Closed: %d\n", cs.closed)
    fmt.Printf("Element Type Address: %d\n", cs.elemtype)
    fmt.Printf("Send Index: %d\n", cs.sendx)
    fmt.Printf("Receive Index: %d\n", cs.recvx)
    fmt.Printf("Receive Wait list first address: 0x%x\n", cs.recvq.first)
    fmt.Printf("Receive Wait list last address: 0x%x\n", cs.recvq.last)
    fmt.Printf("Send Wait list first address: 0x%x\n", cs.sendq.first)
    fmt.Printf("Send Wait list last address: 0x%x\n", cs.sendq.last)
    fmt.Println("------------------------------")
}
```

```go
func main() {
    c := make(chan int32, 4)
    cs := Scalpel(&c)
    Microscope(cs)

    c <- 5
    Microscope(cs)
    go func() {
        c <- 4
        c <- 3
        c <- 2
        c <- 1
    }()
    time.Sleep(2 * time.Millisecond)
    Microscope(cs)

    <-c
    Microscope(cs)

    <-c
    Microscope(cs)

    go func() {
        <-c
        <-c
        <-c
        <-c
    }()
    time.Sleep(2 * time.Millisecond)
    Microscope(cs)

    close(c)

    Microscope(cs)
}
```

References

- The slice go code: [`src/runtime/slice.go`]
- The map go code: [`src/runtime/map.go`]
- The channel go code: [`src/runtime/chan.go`]
- My code: http://github.com/jespino/dissecting-go

Conclusions

- Understanding the basic building blocks of the language helps you understand the implications of its usage.
- There are behaviors that can be unexpected or surprising - be careful.
- The tradeoffs made by the go team can have implications in your software.
- You will not need this knowledge in your day to day work, but it can help you in very specific situations.

Thank you.