Types

slice, map, new, make, struct

Types

- Method sets
- Boolean types
- Numeric types
- String types
- Array types
- Slice types
- Struct types
- Pointer types
- Function types
- Interface types
- Map types
- Channel types

https://golang.org/ref/spec

slice

list

slice vs. slicing vs. index access

main.go

```go
package main

import "fmt"

func main() {

    mySlice := []string{"a", "b", "c", "g", "m", "z",}
    fmt.Println(mySlice)
    fmt.Println(mySlice[2:4])   // slicing a slice
    fmt.Println(mySlice[2])     // index access; accessing by index
    fmt.Println("myString"[2])  // index access; accessing by index
}
```

Terminal

```
00_slice_slicing $ go run main.go
[a b c g m z]
[c g]
c
83
00_slice_slicing $
```

Slice types

A slice is a descriptor for a contiguous segment of an *underlying array* and provides access to a numbered sequence of elements from that array. A slice type denotes the set of all slices of arrays of its element type. The value of an uninitialized slice is `nil`.

```
SliceType = "[" "]" ElementType .
```

```
mySlice := []string{"a", "b", "c", "g", "m", "z",}
```

```
mySlice := []int{1, 3, 5, 7, 9, 11,}
```

Like arrays, slices are indexable and have a length. The length of a slice `s` can be discovered by the built-in function `len`; unlike with arrays it may change during execution. The elements can be addressed by integer indices 0 through `len(s)-1`. The slice index of a given element may be less than the index of the same element in the underlying array.

A slice, once initialized, is always associated with an underlying array that holds its elements. A slice therefore shares storage with its array and with other slices of the same array; by contrast, distinct arrays always represent distinct storage.

The array underlying a slice may extend past the end of the slice. The *capacity* is a measure of that extent: it is the sum of the length of the slice and the length of the array beyond the slice; a slice of length up to that capacity can be created by *slicing* a new one from the original slice. The capacity of a slice `a` can be discovered using the built-in function `cap(a)`.

A new, initialized slice value for a given element type `T` is made using the built-in function `make`, which takes a slice type and parameters specifying the length and optionally the capacity. A slice created with `make` always allocates a new, hidden array to which the returned slice value refers. That is, executing

```
make([]T, length, capacity)
```

produces the same slice as allocating an array and slicing it, so these two expressions are equivalent:

```
make([]int, 50, 100)
new([100]int)[0:50]
```

Like arrays, slices are always one-dimensional but may be composed to construct higher-dimensional objects. With arrays of arrays, the inner arrays are, by construction, always the same length; however with slices of slices (or arrays of slices), the inner lengths may vary dynamically. Moreover, the inner slices must be initialized

main.go

```go
package main

import "fmt"

func main() {

    mySlice := []int{1, 3, 5, 7, 9, 11,}

    for i, value := range mySlice {
        fmt.Println(i, " - ", value)
    }

}
```

Terminal

```
01_int-slice $ go run main.go
0  -  1
1  -  3
2  -  5
3  -  7
4  -  9
5  -  11
01_int-slice $
```

making a slice

making a slice

main.go

```go
package main

import "fmt"

func main() {

    greeting := []string{
        "Good morning!",
        "Bonjour!",
        "dias!",
        "Bongiorno!",
        "Ohayo!",
        "Selamat pagi!",
        "Gutten morgen!",
    }

    for i, currentEntry := range greeting {
        fmt.Println(i, currentEntry)
    }

    for j := 0; j < len(greeting); j++ {
        fmt.Println(greeting[j])
    }

}
```

Terminal

```
02_string-slice $ go run main.go
0 Good morning!
1 Bonjour!
2 dias!
3 Bongiorno!
4 Ohayo!
5 Selamat pagi!
6 Gutten morgen!
Good morning!
Bonjour!
dias!
Bongiorno!
Ohayo!
Selamat pagi!
Gutten morgen!
02_string-slice $
```

slicing a slice

main.go

```go
package main

import "fmt"

func main() {

    greeting := []string{
        "Good morning!",
        "Bonjour!",
        "dias!",
        "Bongiorno!",
        "Ohayo!",
        "Selamat pagi!",
        "Gutten morgen!",
    }

    fmt.Print("[1:2] ")
    fmt.Println(greeting[1:2])
    fmt.Print("[:2] ")
    fmt.Println(greeting[:2])
    fmt.Print("[5:] ")
    fmt.Println(greeting[5:])
    fmt.Print("[:] ")
    fmt.Println(greeting[:])
}
```

Terminal

```
03_slicing-a-slice $ go run main.go
[1:2] [Bonjour!]
[:2] [Good morning! Bonjour!]
[5:] [Selamat pagi! Gutten morgen!]
[:] [Good morning! Bonjour! dias! Bongiorno! Ohayo! Selamat pagi! Gutten morgen!]
03_slicing-a-slice $
```

making a slice

main.go

```go
package main

import "fmt"

func main() {

    customerNumber := make([]int, 3)
    // 3 is length & capacity
    // // length - number of elements referred to by the slice
    // // capacity - number of elements in the underlying array
    customerNumber[0] = 7
    customerNumber[1] = 10
    customerNumber[2] = 15

    fmt.Println(customerNumber[0])
    fmt.Println(customerNumber[1])
    fmt.Println(customerNumber[2])

    greeting := make([]string, 3, 5)
    // 3 is length - number of elements referred to by the slice
    // 5 is capacity - number of elements in the underlying array
    // you could also do it like this

    greeting[0] = "Good morning!"
    greeting[1] = "Bonjour!"
    greeting[2] = "dias!"

    fmt.Println(greeting[2])
}
```

If we think that our slice might grow, we can set a capacity larger than length. **This gives our slice room to grow without golang having to create a new underlying array every time our slice grows.** When the slice exceeds capacity, then a new underlying array will be created. These arrays double in size each time they're created (2, 4, 8, 16) up to a certain point, and then they scale in some smaller proportion.

Terminal

```
04_make $ go run main.go
7
10
15
dias!
04_make $
```

main.go

```go
package main

import "fmt"

func main() {

    greeting := make([]string, 3, 5)
    // 3 is length - number of elements referred to by the slice
    // 5 is capacity - number of elements in the underlying array

    greeting[0] = "Good morning!"
    greeting[1] = "Bonjour!"
    greeting[2] = "buenos dias!"
    greeting[3] = "suprabadham"

    fmt.Println(greeting[2])
}
```

error
index out of range

Terminal

```
05_append-invalid $ go run main.go
panic: runtime error: index out of range

goroutine 1 [running]:
main.main()
        /Users/tm002/Documents/go/src/github.com/goestoeleven/GolangTraining/17_slices/05_append-invalid/main.go:15 +0x1ac

goroutine 2 [runnable]:
runtime.forcegchelper()
```

appending to a slice

main.go

```go
package main

import "fmt"

func main() {

    greeting := make([]string, 3, 5)
    // 3 is length - number of elements referred to by the slice
    // 5 is capacity - number of elements in the underlying array
    // you could also do it like this

    greeting[0] = "Good morning!"
    greeting[1] = "Bonjour!"
    greeting[2] = "buenos dias!"
    greeting = append(greeting, "Suprabadham")

    fmt.Println(greeting[3])
}

```

Terminal

```
06_append $ go run main.go
Suprabadham
06_append $
```

appending beyond capacity to a slice

main.go

```go
package main

import "fmt"

func main() {

    greeting := make([]string, 3, 5)
    // 3 is length - number of elements referred to by the slice
    // 5 is capacity - number of elements in the underlying array

    greeting[0] = "Good morning!"
    greeting[1] = "Bonjour!"
    greeting[2] = "buenos dias!"
    greeting = append(greeting, "Suprabadham")
    greeting = append(greeting, "Zǎo'ān")
    greeting = append(greeting, "Ohayou gozaimasu")
    greeting = append(greeting, "gidday")

    fmt.Println(greeting[6])
    fmt.Println(len(greeting))
    fmt.Println(cap(greeting))
}
```

Terminal

```
07_append-beyond-capacity $ go run main.go
gidday
7
10
07_append-beyond-capacity $
```

If we think that our slice might grow, we can set a capacity larger than length. This gives our slice room to grow without golang having to create a new underlying array every time our slice grows. When the slice exceeds capacity, then a new underlying array will be created. These arrays typically double in size each time they're created (2, 4, 8, 16).

main.go

```go
package main

import "fmt"

func main() {

    mySlice := []int{1, 2, 3, 4, 5}
    myOtherSlice := []int{6, 7, 8, 9}

    mySlice = append(mySlice, myOtherSlice...)

    fmt.Println(mySlice)
}
```

appending a slice to a slice

notice the syntax
args...

Terminal

```
08_append_slice-to-slice $ go run main.go
[1 2 3 4 5 6 7 8 9]
08_append_slice-to-slice $
```

appending a slice to a slice

main.go

```go
package main

import "fmt"

func main() {

    mySlice := []string{"Monday", "Tuesday"}
    myOtherSlice := []string{"Wednesday", "Thursday", "Friday"}

    mySlice = append(mySlice, myOtherSlice...)

    fmt.Println(mySlice)
}

```

notice the syntax

Terminal

```
02_slice-of-strings $ go run main.go
[Monday Tuesday Wednesday Thursday Friday]
02_slice-of-strings $
```

main.go

```go
package main

import "fmt"

func main() {

    mySlice := []string{"Monday", "Tuesday"}
    myOtherSlice := []string{"Wednesday", "Thursday", "Friday"}

    mySlice = append(mySlice, myOtherSlice...)
    fmt.Println(mySlice)

    mySlice = append(mySlice[:2], mySlice[3:]...)
    fmt.Println(mySlice)

}
```

delete from a slice

Terminal

```
09_delete $ go run main.go
[Monday Tuesday Wednesday Thursday Friday]
[Monday Tuesday Thursday Friday]
09_delete $
```

exercise

write a program that
creates a slice of ints using shorthand notation
then prints the slice

exercise

write a program that
creates a slice of strings using shorthand notation
then prints the slice

exercise

create a slice of ints using make;
set the length to 5 and capacity to 10

exercise

append a slice to a slice

exercise

delete an element from a slice

exercise

create a slice
then make your program throw an
“index out of range” error

map

key, value

Map types

A map is an unordered group of elements of one type, called the element type, indexed by a set of unique *keys* of another type, called the key type. The value of an uninitialized map is `nil`.

```
MapType     = "map" "[" KeyType "]" ElementType .
KeyType     = Type .
```

The comparison operators == and != must be fully defined for operands of the key type; thus the key type must not be a function, map, or slice. If the key type is an interface type, these comparison operators must be defined for the dynamic key values; failure will cause a run-time panic.

```
map[string]int
map[*T]struct{ x, y float64 }
map[string]interface{}
```

The number of map elements is called its length. For a map `m`, it can be discovered using the built-in function `len` and may change during execution. Elements may be added during execution using assignments and retrieved with index expressions; they may be removed with the `delete` built-in function.

A new, empty map value is made using the built-in function `make`, which takes the map type and an optional capacity hint as arguments:

```
make(map[string]int)
make(map[string]int, 100)
```

The initial capacity does not bound its size: maps grow to accommodate the number of items stored in them, with the exception of `nil` maps. A `nil` map is equivalent to an empty map except that no elements may be added.

```
myGreeting := map[string]string{
    "Tim":     "Good morning!",
    "Jenny":   "Bonjour!",
    "Medhi":   "Buenos dias!",
    "Marcus":  "Bongiorno!",
    "Julian":  "Ohayo!",
    "Sushant": "Selamat pagi!",
    "Jose":    "Gutten morgen!",
}

fmt.Println(myGreeting["Jenny"])
```

```
favorite := make(map[string]string)
favorite["breakfast"] = "eggs"
```

maps

- basic info
  - key:value
  - maps keys to values
  - called “dictionaries” in some languages
  - built into the language (not an additional library you must import) so they’re first-class citizens
- Map Keys
  - need to be unique
  - the type used for a key needs to have the equality operator defined for it
    - the type must allow equals comparisons
    - can’t use these types
      - slice
      - map
- Maps are Reference Types
  - they behave like pointers
  - when you pass a map variable to a function
    - any changes to that mapped variable in the function
    - change that original mapped variable outside the function
- Maps are Not Thread Safe
  - best to avoid using maps concurrently

main.go

```go
package main

import "fmt"

func main() {

    // Maps - Shorthand Notation
    myGreeting := map[string]string{
        "Tim":     "Good morning!",
        "Jenny":   "Bonjour!",
        "Medhi":   "Buenos dias!",
        "Marcus":  "Bongiorno!",
        "Julian":  "Ohayo!",
        "Sushant": "Selamat pagi!",
        "Jose":    "Gutten morgen!",
    }

    fmt.Println(myGreeting["Jenny"])
}
```

shorthand
for creating a map

Terminal

```
01 $ go run main.go
Bonjour!
01 $
```

adding an entry to a map

main.go

```go
package main

import "fmt"

func main() {

    // Maps - Shorthand Notation
    myGreeting := map[string]string{
        "Tim":     "Good morning!",
        "Jenny":   "Bonjour!",
        "Medhi":   "Buenos dias!",
        "Marcus":  "Bongiorno!",
        "Julian":  "Ohayo!",
        "Sushant": "Selamat pagi!",
        "Jose":    "Gutten morgen!",
    }

    myGreeting["Harleen"] = "Howdy"

    fmt.Println(myGreeting["Harleen"])
}
```

Terminal

```
02_adding-element $ go run main.go
Howdy
02_adding-element $
```

main.go

```go
package main

import "fmt"

func main() {

    // Maps - Shorthand Notation
    myGreeting := map[string]string{
        "Tim":     "Good morning!",
        "Jenny":   "Bonjour!",
        "Medhi":   "Buenos dias!",
        "Marcus":  "Bongiorno!",
        "Julian":  "Ohayo!",
        "Sushant": "Selamat pagi!",
        "Jose":    "Gutten morgen!",
    }

    myGreeting["Harleen"] = "Howdy"

    fmt.Println(len(myGreeting))
    fmt.Println(myGreeting)
}
```

Printing map len
Printing map

Terminal

```
03_printing-map $ go run main.go
8
map[Sushant:Selamat pagi! Jose:Gutten morgen! Harleen:Howdy Tim:Good morning! Jenny:Bonjour! Medhi:Buenos dias! Marcus:Bongiorno! Julian:Ohayo!]
03_printing-map $
```

main.go

```go
package main

import "fmt"

func main() {

    // Maps - Shorthand Notation
    myGreeting := map[string]string{
        "Tim":     "Good morning!",
        "Jenny":   "Bonjour!",
        "Medhi":   "Buenos dias!",
        "Marcus":  "Bongiorno!",
        "Julian":  "Ohayo!",
        "Sushant": "Selamat pagi!",
        "Jose":    "Gutten morgen!",
    }

    myGreeting["Harleen"] = "Howdy"
    fmt.Println(myGreeting["Harleen"])
    myGreeting["Harleen"] = "Gidday"
    fmt.Println(myGreeting["Harleen"])
}

```

updating an entry

Terminal

```
04_updating-entry $ go run main.go
Howdy
Gidday
04_updating-entry $
```

delete an entry

main.go

```go
package main

import "fmt"

func main() {

    myGreeting := map[int]string{
        0: "Good morning!",
        1: "Bonjour!",
        2: "Buenos dias!",
        3: "Bongiorno!",
    }

    fmt.Println(myGreeting)
    delete(myGreeting, 2)
    fmt.Println(myGreeting)
}
```

Terminal

```
05_deleting-entry $ go run main.go
map[0:Good morning! 1:Bonjour! 2:Buenos dias! 3:Bongiorno!]
map[0:Good morning! 1:Bonjour! 3:Bongiorno!]
05_deleting-entry $
```

main.go

```go
func main() {

    myGreeting := map[int]string{
        0: "Good morning!",
        1: "Bonjour!",
        2: "Buenos dias!",
        3: "Bongiorno!",
    }

    fmt.Println(myGreeting)

    if val, exists := myGreeting[2]; exists {
        delete(myGreeting, 2)
        fmt.Println("val: ", val)
        fmt.Println("exists: ", exists)
    } else {
        fmt.Println("That value doesn't exist.")
        fmt.Println("val: ", val)
        fmt.Println("exists: ", exists)
    }

    fmt.Println(myGreeting)
}
```

check for existence
comma ok idiom

Terminal

```
06_comma-ok-idiom $ go run main.go
map[3:Bongiorno! 0:Good morning! 1:Bonjour! 2:Buenos dias!]
val:  Buenos dias!
exists:  true
map[0:Good morning! 1:Bonjour! 3:Bongiorno!]
06_comma-ok-idiom $
```

main.go

```go
package main

import "fmt"

func main() {

    myGreeting := map[int]string{
        0: "Good morning!",
        1: "Bonjour!",
        2: "Buenos dias!",
        3: "Bongiorno!",
    }

    fmt.Println(myGreeting)

    if val, exists := myGreeting[7]; exists {
        delete(myGreeting, 7)
        fmt.Println("val: ", val)
        fmt.Println("exists: ", exists)
    } else {
        fmt.Println("That value doesn't exist.")
        fmt.Println("val: ", val)
        fmt.Println("exists: ", exists)
    }

    fmt.Println(myGreeting)
}
```

check for existence comma ok idiom

```
rminal
07_comma-ok-idiom_val-not-exists $ go run main.go
map[0:Good morning! 1:Bonjour! 2:Buenos dias! 3:Bongiorno!]
That value doesn't exist.
val:
exists:  false
map[3:Bongiorno! 0:Good morning! 1:Bonjour! 2:Buenos dias!]
07_comma-ok-idiom_val-not-exists $
```

main.go

```go
package main

import "fmt"

func main() {

    myGreeting := map[int]string{
        0: "Good morning!",
        1: "Bonjour!",
        2: "Buenos dias!",
        3: "Bongiorno!",
    }

    for key, val := range myGreeting {
        fmt.Println(key, " - ", val)
    }
}
```

range loop

Terminal

```
08_loop-range $ go run main.go
1  -  Bonjour!
2  -  Buenos dias!
3  -  Bongiorno!
0  -  Good morning!
08_loop-range $
```

main.go

```go
package main

import "fmt"

func main() {

    favorite := make(map[string]string)
    fmt.Println(favorite)
    fmt.Println(len(favorite))
    favorite["breakfast"] = "eggs"
    fmt.Println(favorite)
    fmt.Println(len(favorite))
}
```

make
a map

Terminal

```
09_make $ go run main.go
map[]
0
map[breakfast:eggs]
1
09_make $
```

exercise

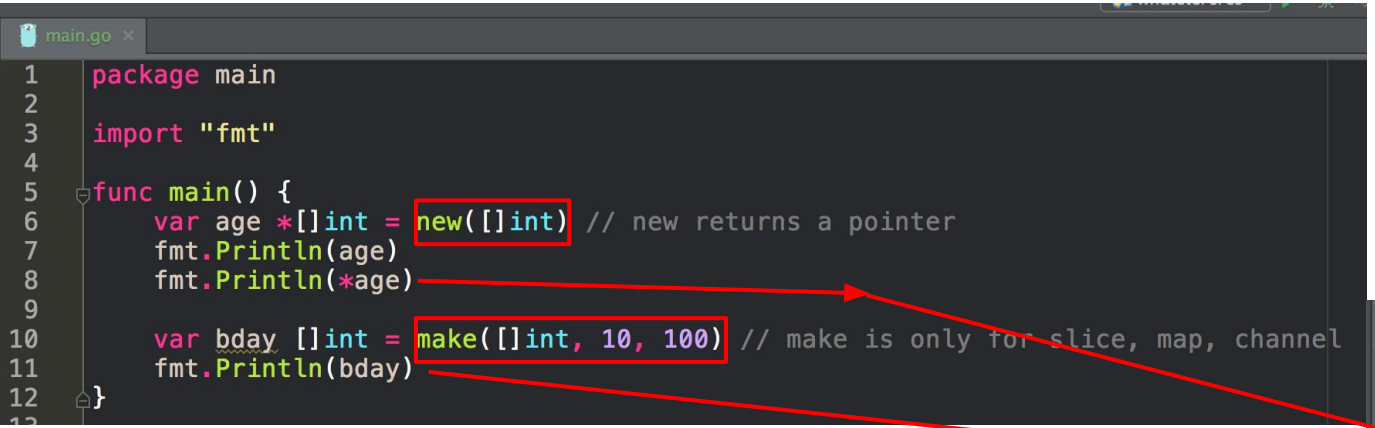
create a program that:

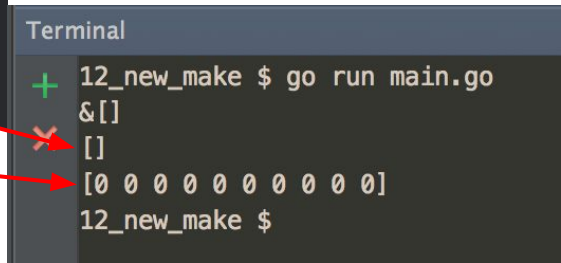
- creates a **map** using shorthand notation
- adds an entry to the map
- changes an entry in the map
- deletes an entry in the map
- prints all of the entries in the map using **range**
- prints the **len** of the map
- uses the **comma ok idiom**

make vs new

make vs new

- make
  - slices, maps, channels
  - allocates memory
  - initializes
    - puts 0 or empty string into values
- new
  - returns a pointer
    - newly allocated
    - zeroed value
      - “returns a pointer to a newly allocated zeroed value of type T”





make vs new

- make
  - slices, maps, channels
  - allocates memory
  - initializes
    - puts 0 or empty string into values

- new
  - returns a pointer
    - newly allocated
    - zeroed value
      - “returns a pointer to a newly allocated zeroed value of type T”

main.go

```
1  package main
2
3  import "fmt"
4
5  func main() {
6      var age *int = new(int) // new returns a pointer
7      fmt.Println(age)
8      fmt.Println(*age)
9  }
10
```

Terminal

```
10_new $ go run main.go
0x20818a220
0
10_new $
```

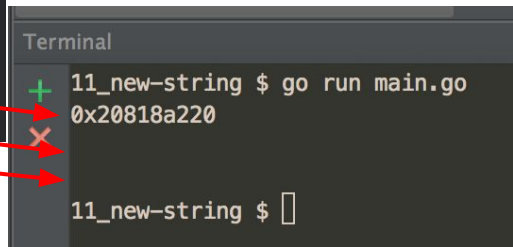
make vs new

- make
  - slices, maps, channels
  - allocates memory
  - initializes
    - puts 0 or empty string into values

- new
  - returns a pointer
    - newly allocated
    - zeroed value
      - “returns a pointer to a newly allocated zeroed value of type T”

main.go

```
1  package main
2
3  import "fmt"
4
5  func main() {
6      var name *string = new(string) // new returns a pointer
7      fmt.Println(name)
8      fmt.Println(*name)
9      fmt.Println("") // this is what *name is, an empty string
10 }
```



zero value

false for booleans, 0 for integers, 0.0 for floats, "" for strings
nil for pointers, functions, interfaces, slices, channels, and maps

https://golang.org/ref/spec#The_zero_value

make vs new

- make
  - slices, maps, channels
  - allocates memory
  - initializes
    - puts 0 or empty string into values

- new
  - returns a pointer
    - newly allocated
    - zeroed value
      - “returns a pointer to a newly allocated zeroed value of type T”

main.go

```
1  package main
2
3  import "fmt"
4
5  func main() {
6      var member *bool = new(bool) // new returns a pointer
7      fmt.Println(member)
8      fmt.Println(*member)
9  }
10
```

Terminal

```
12_new-bool $ go run main.go
0x20818a1f9
false
12_new-bool $
```

make vs new

- make
  - slices, maps, channels
  - allocates memory
  - initializes
    - puts 0 or empty string into values

- new
  - returns a pointer
    - newly allocated
    - zeroed value
      - “returns a pointer to a newly allocated zeroed value of type T”

main.go

```
1  package main
2
3  import "fmt"
4
5  func main() {
6      var members *[]string = new([]string) // new returns a p
7      fmt.Println(members)
8      fmt.Println(*members)
9  }
```

Terminal

```
13_new-slice $ go run main.go
&[]
[]
13_new-slice $
```

make vs new

- make
  - slices, maps, channels
  - allocates memory
  - initializes
    - puts 0 or empty string into values

- new
  - returns a pointer
    - newly allocated
    - zeroed value
      - “returns a pointer to a newly allocated zeroed value of type T”

main.go

```
1  package main
2
3  import "fmt"
4
5  func main() {
6      var members *map[int]string = new(map[int]string) // new returns a pointer
7      fmt.Println(members)
8      fmt.Println(*members)
9  }
10
```

```
al
4_new-map $ go run main.go
&map[]
map[]
14_new-map $
```

make vs new

- make
  - **slices**, **maps**, **channels**
  - allocates memory
- initializes
  - puts 0 or empty string into values

- new
  - returns a pointer
    - newly allocated
    - zeroed value
      - “returns a pointer to a newly allocated zeroed value of type T”

main.go

```
1  package main
2
3  import "fmt"
4
5  func main() {
6      var bday int = make(int) // make is only for slice, map, channel
7      fmt.Println(bday)
8  }
9
```

```
15_error_invalid-code $ go run main.go
# command-line-arguments
./main.go:6: cannot make type int
15_error_invalid-code $
```

- make
  - slices, maps

make vs new

- make
  - slices, maps, channels
  - allocates memory
- initializes
  - puts 0 or empty string into values

- new
  - returns a pointer
    - newly allocated
    - zeroed value
      - “returns a pointer to a newly allocated zeroed value of type T”

main.go

```
1  package main
2
3  import "fmt"
4
5  func main() {
6      var age *[]int = new([]int) // new returns a pointer
7      fmt.Println(age)
8      fmt.Println(*age)
9
10     var bday []int = make([]int, 10, 100) // make is only for slice, map, channel
11     fmt.Println(bday)
12 }
13
```

```
minal
08_new_make_slice-int $ go run main.go
&[]
[]
[0 0 0 0 0 0 0 0 0 0]
08_new_make_slice-int $
```

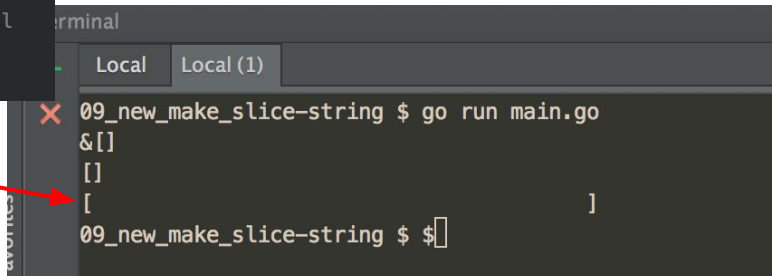
make vs new

- make
  - slices, maps, channels
  - allocates memory
- initializes
  - puts 0 or empty string into values

- new
  - returns a pointer
    - newly allocated
    - zeroed value
      - “returns a pointer to a newly allocated zeroed value of type T”

main.go

```
1  package main
2
3  import "fmt"
4
5  func main() {
6      var members *[]string = new([]string) // new returns a pointer
7      fmt.Println(members)
8      fmt.Println(*members)
9
10     var staff []string = make([]string, 40, 100) // make is only for slice, map, channel
11     fmt.Println(staff)
12 }
13
```



make vs new

- make
  - slices, maps, channels
  - allocates memory
- initializes
  - puts 0 or empty string into values

- new
  - returns a pointer
    - newly allocated
    - zeroed value
      - “returns a pointer to a newly allocated zeroed value of type T”

main.go

```
1  package main
2
3  import "fmt"
4
5  func main() {
6      var age *map[int]int = new(map[int]int) // new returns a pointer
7      fmt.Println(age)
8      fmt.Println(*age)
9
10     var bday map[int]int = make(map[int]int) // make is only for slice, map, channel
11     fmt.Println(bday)
12     fmt.Println(bday[0])
13     fmt.Println(bday[10])
14     fmt.Println(bday[1000])
15     fmt.Println(bday[99999999999999])
16 }
```

Terminal

```
10_new_make_map-int-int $ go run main.go
&map[]
map[]
map[]
0
0
0
0
10_new_make_map-int-int $
```

Allocation with `make`

Back to allocation. The built-in function `make(T, args)` serves a purpose different from `new(T)`. It creates slices, maps, and channels only, and it returns an *initialized* (not *zeroed*) value of type `T` (not `*T`). The reason for the distinction is that these three types represent, under the covers, references to data structures that must be initialized before use. A slice, for example, is a three-item descriptor containing a pointer to the data (inside an array), the length, and the capacity, and until those items are initialized, the slice is `nil`. For slices, maps, and channels, `make` initializes the internal data structure and prepares the value for use. For instance,

```
make([]int, 10, 100)
```

allocates an array of 100 ints and then creates a slice structure with length 10 and a capacity of 100 pointing at the first 10 elements of the array. (When making a slice, the capacity can be omitted; see the section on slices for more information.) In contrast, `new([]int)` returns a pointer to a newly allocated, zeroed slice structure, that is, a pointer to a `nil` slice value.

These examples illustrate the difference between `new` and `make`.

```
var p *[]int = new([]int)       // allocates slice structure; *p == nil; rarely useful
var v  []int = make([]int, 100) // the slice v now refers to a new array of 100 ints

// Unnecessarily complex:
var p *[]int = new([]int)
*p = make([]int, 100, 100)

// Idiomatic:
v := make([]int, 100)
```

Remember that `make` applies only to maps, slices and channels and does not return a pointer. To obtain an explicit pointer allocate with `new` or take the address of a variable explicitly.

Allocation with `new`

Go has two allocation primitives, the built-in functions `new` and `make`. They do different things and apply to different types, which can be confusing, but the rules are simple. Let's talk about `new` first. It's a built-in function that allocates memory, but unlike its namesakes in some other languages it does not *initialize* the memory, it only *zeros* it. That is, `new(T)` allocates zeroed storage for a new item of type `T` and returns its address, a value of type `*T`. In Go terminology, it returns a pointer to a newly allocated zero value of type `T`.

Since the memory returned by `new` is zeroed, it's helpful to arrange when designing your data structures that the zero value of each type can be used without further initialization. This means a user of the data structure can create one with `new` and get right to work. For example, the documentation for `bytes.Buffer` states that "the zero value for `Buffer` is an empty buffer ready to use." Similarly, `sync.Mutex` does not have an explicit constructor or `Init` method. Instead, the zero value for a `sync.Mutex` is defined to be an unlocked mutex.

exercise

create a program that:

- declares a variable of type **int** using **new**
  - (note: this is not idiomatic go code to create an int this way)
- print out the memory address of the variable
- print out the value of the variable
- true or false:
  - **new** returned a pointer

exercise

create a program that:

- declares a variable of type **string** using **new**
  - (note: this is not idiomatic go code to create a string this way)
- print out the memory address of the variable
- print out the value of the variable
- true or false:
  - **new** returned a pointer

exercise

create a program that:

- declares a variable of type **bool** using **new**
  - (note: this is not idiomatic go code to create a bool this way)
- print out the memory address of the variable
- print out the value of the variable
- true or false:
  - **new** returned a pointer

exercise

create a program that:

- declares a variable of type **[]int** using **make**
- print out the value of the variable
- true or false:
  - **make** returned a pointer

exercise

create a program that:

- declares a variable of type **map[int]string** using **make**
- print out the value of the variable
- true or false:
  - **make** returned a pointer

exercise

What are the zeroed values for int, string, bool?

exercise

What are the zeroed values for slice and map

exercise

Explain the difference between **make** and **new**

struct

grouped fields

main.go

```go
package main

import "fmt"

type person struct {
    name string
    age  int
}

func main() {
    fmt.Println(person{"James", 20})
}
```

create a struct and **initialize** it

Terminal

```
01_struct $ go run main.go
{James 20}
01_struct $
```

main.go

```go
package main

import "fmt"

type person struct {
    name string
    age  int
}

func main() {
    p1 := person{"James", 20}
    fmt.Println(p1)
    fmt.Println(p1.name)
    fmt.Println(p1.age)
}
```

accessing fields using dot notation

Terminal

```
02_struct_dot-notation $ go run main.go
{James 20}
James
20
02_struct_dot-notation $
```

initializing two variables
of type person struct

main.go

```go
package main

import "fmt"

type person struct {
    name string
    age  int
}

func main() {
    p1 := person{"James", 20}
    p2 := person{"Ian", 45}
    fmt.Println(p1.name)
    fmt.Println(p2.name)
}
```

Terminal

```
03_struct_multiple-instances $ go run main.go
James
Ian
03_struct_multiple-instances $
```

main.go

```go
package main

import "fmt"

type person struct {
    name string
    age  int
}

func main() {
    p1 := person{name: "James", age: 20}
    fmt.Println(p1.name)
}
```

initialize a variable
naming the fields

Terminal

```
04_struct_name-fields $ go run main.go
James
04_struct_name-fields $
```

main.go

```go
package main

import "fmt"

type person struct {
    name string
    age  int
}

func main() {
    p1 := person{name: "James"}
    fmt.Println(p1.name)
    fmt.Println(p1.age)
}
```

initialize a variable
omitting fields

Terminal

```
05_struct_omitted-fields-zeroed $ go run main.go
James
0
05_struct_omitted-fields-zeroed $
```

main.go

```go
package main

import "fmt"

type person struct {
    name string
    age  int
}

func main() {
    p1 := person{name: "James"}
    fmt.Printf("%T\n", p1)
}
```

viewing the type of p1

Terminal

```
06_struct_variable-type $ go run main.go
main.person
06_struct_variable-type $
```

We have created our own type which is interesting to think about!

main.go

```go
package main

import "fmt"

type person struct {
    name string
    age  int
}

func main() {
    p1 := &person{"James", 20}
    fmt.Println(p1)
    fmt.Printf("%T\n", p1)
    fmt.Println(p1.name)
    fmt.Println(p1.age)
}
```

taking the address of a struct
p1 is of type *person

Terminal

```
06_struct_pointer-to-struct $ go run main.go
&{James 20}
*main.person
James
20
06_struct_pointer-to-struct $
```

main.go

```go
package main

import "fmt"

type person struct {
    name string
    age  int
}

func main() {
    p1 := new(person)
    p1.name = "James"
    p1.age = 20
    fmt.Println(p1)
    fmt.Printf("%T\n", p1)
    fmt.Println(p1.name)
    fmt.Println(p1.age)
}
```

- new
  - returns a pointer
    - newly allocated
    - zeroed value
      - “returns a pointer to a newly allocated zeroed value of type T”

Terminal

```
07_struct_new-pointer-to-struct $ go run main.go
&{James 20}
*main.person
James
20
07_struct_new-pointer-to-struct $
```

question

Can we use **make** with a struct?

make is for **slices**, **maps**, and **channels**

no

question

Can we use **make** with a struct?

- make
  - slices, maps, channels
  - allocates memory
- initializes
    - puts 0 or empty string into values

main.go

```go
package main

import "fmt"

type person struct {
    name string
    age  int
}

func main() {
    p1 := &person{"James", 20}
    fmt.Println(p1.name)
    p1.name = "Ian"
    fmt.Println(p1.name)
}
```

Terminal

```
07_struct_mutable $ go run main.go
James
Ian
07_struct_mutable $
```

main.go

```go
package main

import "fmt"

type person struct {
    name string
    age  int
}

func main() {
    p1 := person{"James", 20}
    fmt.Println(p1.name)
    p1.name = "Ian"
    fmt.Println(p1.name)
}
```

Terminal

```
08_struct_mutable $ go run main.go
James
Ian
08_struct_mutable $
```

Struct types

A struct is a sequence of named elements, called fields, each of which has a name and a type. Field names may be specified explicitly (IdentifierList) or implicitly (AnonymousField). Within a struct, non-blank field names must be unique.

```
StructType     = "struct" "{" { FieldDecl ";" } "}" .
FieldDecl      = (IdentifierList Type | AnonymousField) [ Tag ] .
AnonymousField = [ "*" ] TypeName .
Tag            = string_lit .
```

```
// An empty struct.
struct {}
```

```
// A struct with 6 fields.
struct {
        x, y int
        u float32
        _ float32  // padding
        A *[]int
        F func()
}
```

A field declared with a type but no explicit field name is an *anonymous field*, also called an *embedded* field or an embedding of the type in the struct. An embedded type must be specified as a type name `T` or as a pointer to a non-interface type name `*T`, and `T` itself may not be a pointer type. The unqualified type name acts as the field name.

```
// A struct with four anonymous fields of type T1, *T2, P.T3 and *P.T4
struct {
        T1        // field name is T1
        *T2       // field name is T2
        P.T3      // field name is T3
        *P.T4     // field name is T4
        x, y int  // field names are x and y
}
```

exercise

create a program that:

- defines a struct type to hold customer info
- initialize two variables using that struct type
- uses dot-notation to print a field from each of the variables
- changes the value of one of the fields
- prints the changed field

exercise

Can you use **new** to create a variable of a **struct** type?

exercise

Can you use **make** to create a variable of a **struct** type?

Review

- **slice** vs. **slicing** vs. **index access**
- **slice**
  - **list**
  - mySlice := []int{1, 3, 5, 7, 9, 11}
  - greeting := make([]string, 3, 5)
  - greeting = append(greeting, "Hello")
  - mySlice = append(mySlice, myOtherSlice...)
  - mySlice = append(mySlice[:2], mySlice[3:]...)
- **map**
  - **key, value**
    - key type, element type
  - initializing
    - myMap := map[int]string{<entries>}
    - otras := make(map[string]string)
  - adding new entry
    - otras[“new key”] = “new value”
  - changing entry
    - otras[“new key”] = “newer value”
  - delete entry
    - delete(otras, “new key”)
- **comma ok idiom**
  - if val, ok := myGreeting[2]; ok {
    fmt.Println("val: ", val)
    fmt.Println("exists: ", exists)
    }
- **make**
  - slices, maps, channels
  - allocates memory
  - initializes
    - puts 0 or empty string into values
- **new**
  - returns a pointer
    - newly allocated
    - zeroed value
      - “returns a pointer to a newly allocated zeroed value of type T”
- **struct**
  - **grouped fields**
  - type person struct { name string age int }
  - p1 := person{name: "James"}

Review Questions

slice

- Why would you want to use make when creating a slice?

slice vs map vs struct

- Describe the difference between the above data structures. Give an example of data that would be stored in each type.