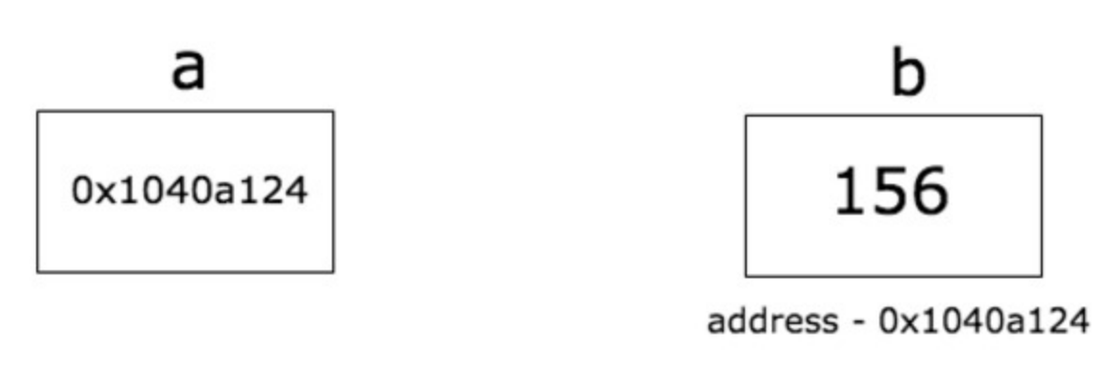
Part 15: Pointers

14 MAY 2017

Welcome to tutorial no. 15 in Golang tutorial series.

What is a pointer

A pointer is a variable which stores the memory address of another variable.



In the above illustration, variable `b` has value `156` and is stored at memory address `0x1040a124`. The variable `a` holds the address of `b`. Now `a` is said to point to `b`.

Declaring pointers

***T** is the type of the pointer variable which points to a value of type **T**.

Lets write some code.

```
1  package main
2
3  import (
4      "fmt"
5  )
6
7  func main() {
8      b := 255
9      var a *int = &b
10     fmt.Printf("Type of a is %T\n", a)
11     fmt.Println("address of b is", a)
12 }
```

The **&** operator is used to get the address of a variable. In line no. 9 of the above program we are assigning the address of `b` to `a` whose type is `*int`. Now a is said to point to b. When we print the value in `a`, the address of `b` will be printed. This program outputs

```
Type of a is *int
address of b is 0x1040a124
```

You might get a different address for b since the location of b can be anywhere in memory.

Get the free Golang tools cheat sheet

zero value of a pointer

The zero value of a pointer is `nil`.

```
package main

import (
    "fmt"
)

func main() {
    a := 25
    var b *int
    if b == nil {
        fmt.Println("b is", b)
        b = &a
        fmt.Println("b after initialization is", b
    }
}
```

b is initially nil in the above program and then later its assigned to the address of a. This program outputs

```
b is <nil>
b after initialisation is 0x1040a124
```

Dereferencing a pointer

Dereferencing a pointer means accessing the value of the variable which the pointer points to. `*a` is the syntax to deference a.

Lets see how this works in a program.

```
1  package main
2  import (
3      "fmt"
4  )
5
6  func main() {
7      b := 255
8      a := &b
9      fmt.Println("address of b is", a)
10     fmt.Println("value of b is", *a)
11 }
```

In line no 10 of the above program, we deference `a` and print the value of it. As expected it prints the value of b. The output of the program is

```
address of b is 0x1040a124
value of b is 255
```

Lets write one more program where we change the value in b using the pointer.

```
1  package main
2
3  import (
4      "fmt"
5  )
6
7  func main() {
8      b := 255
9      a := &b
10     fmt.Println("address of b is", a)
11     fmt.Println("value of b is", *a)
12     *a++
13     fmt.Println("new value of b is", b)
14 }
```

In line no. 12 of the above program, we increment the value pointed by a by 1 which changes the value of b *since a points to b*. Hence the value of b becomes 256. The output of the program is

```
address of b is 0x1040a124
value of b is 255
new value of b is 256
```

Passing pointer to a function

```
1  package main
2
3  import (
4      "fmt"
5  )
6
7  func change(val *int) {
8      *val = 55
9  }
10 func main() {
11     a := 58
12     fmt.Println("value of a before function call is
13     b := &a
14     change(b)
15     fmt.Println("value of a after function call is'
16 }
```

In the above program, in line no. 14 we are passing the pointer variable b which holds the address of a to the function `change`. Inside `change` function, value of a is changed using dereference in line no 8. This program outputs,

```
value of a before function call is 58
value of a after function call is 55
```

Do not pass a pointer to an array as a argument to a function. Use slice instead.

Lets assume that we want to make some modifications to an array inside the function and also the changes made to that array inside the function should be visible to the caller. One way of doing this is to pass a pointer to an array as an argument to the function.

```
1  package main
2
3  import (
4      "fmt"
5  )
6
7  func modify(arr *[3]int) {
8      (*arr)[0] = 90
9  }
10
11 func main() {
12     a := [3]int{89, 90, 91}
13     modify(&a)
14     fmt.Println(a)
15 }
```

In line no. 13 of the above program, we are passing the address of the array `a` to the `modify` function. In line no.8 in the `modify` function we are dereferencing arr and assigning `90` to the first element of the array. This program outputs `[90 90 91]`

a[x] is shorthand for (*a)[x]. So (*arr)[0] in the above program can be replaced by arr[0]. Lets rewrite the above program using this shorthand syntax.

```
package main

import (
    "fmt"
)

func modify(arr *[3]int) {
    arr[0] = 90
}

func main() {
    a := [3]int{89, 90, 91}
    modify(&a)
    fmt.Println(a)
}
```

This program also outputs `[90 90 91]`

Although this way of passing a pointer to an array as a argument to a function and making modification to it works, it is not the idiomatic way of achieving this in Go. We have slices for this.

Lets rewrite the same program using slices.

```
1  package main
2
3  import (
4      "fmt"
5  )
6
7  func modify(sls []int) {
8      sls[0] = 90
9  }
10
11 func main() {
12     a := [3]int{89, 90, 91}
13     modify(a[:])
14     fmt.Println(a)
15 }
```

In line no.13 of the program above, we pass a slice to the `modify` function. The first element of the slice is changed to `90` inside the `modify` function. This program also outputs `[90 90 91]`. **So forget about passing pointers to arrays around and use slices instead :)**. This code is much more clean and is idiomatic Go :).

Go does not support pointer arithmetic

Go does not support pointer arithmetic which is present in other languages like C.

```
1  package main
2
3  func main() {
4      b := [...]int{109, 110, 111}
5      p := &b
6      p++
7  }
```

The above program will throw compilation error **main.go:6: invalid operation: p++ (non-numeric type *[3]int)**

I have created a single program in github which covers everything we have discussed.

Thats it for pointers in Go. Have a good day.

Next tutorial - Structures

Get the free Golang tools cheat sheet

Naveen Ramanathan
iOS developer at dietco.de and Golang enthusiast.

Share this post