DIVYANSHU SHEKHAR

Golang Pointers Struct and Array

by Divyanshu Shekhar in Golang on May 10, 2020 0

In this blog, we will learn about **Golang Pointers** Struct and Array. The **Pointers** are one of the most important topics in **Golang** and **Pointers** are also very important in other languages like C, C++, etc.

Let's see what will be in this **Golang Pointers** Struct and Array Blog:

- What are **Pointers in Golang**?
- **How to create Golang Pointers**?
- What is **Dereferencing Pointers**? and, **How to Dereference Pointers in Golang**?
- **The New Function in Golang**
- **Nil Type in Golang**
- Built-in **Golang Internal Pointers** and How they work?

Before Learning **Golang Pointers** Struct and Array, make sure your **Golang Basics** are clear:

Golang Pointers

Pointers in Golang are variables that store the memory address of the variable to which it points.

In Simple words, **Pointers are simple variables**, it doesn't hold any kind of value but stores another variables memory address or location.

Golang Pointers | Source – Divyanshu Shekhar

Golang Memory Address

A variable's memory address can be known using the "**address of**" Symbol (**&**).

Syntax:

```
&<variable_name>
```

The **Address of** Symbol is mainly used to know the memory location of a variable and allow pointers to point at that address.

Example:

```
a := 17
var b int = 18
fmt.Println(&a)
fmt.Println(&b)
```

Output:

```
0xc0000100a0 // memory address of variable a
0xc0000100a8 // memory address of variable b
```

&a and **&b** returns the memory address/location of variable a and b.

Golang Pointers Creation

1. Long Format

```
var <Pointer_name> *<data_type> = &<variable_to_point>
```

Example:

```
var b *int = &a
```

2. Short Format

```
<pointer_name> := &<variable_to_point>
```

Example:

```
b := &a
```

Golang Pointers Example

```
a := 17
var b *int = &a
fmt.Println("Memory Address of variable a : ", &a)
fmt.Println("Value of variable a: ", a)
fmt.Println("Memory Address of pointer b : ", &b)

fmt.Println("Value of pointer b: ", b)
```

Output:

```
Memory Address of variable a : 0xc0000100a0
Value of variable a: 17
Memory Address of pointer b : 0xc000006028
Value of pointer b: 0xc0000100a0
```

Value of pointer b = Memory Address of variable a = 0xc0000100a0

The Output Shows that the pointer 'b' holds the **memory address** of variable 'a', and the pointer 'b' itself has a memory address.

This Output Proves Two Things about **Golang Pointers**:

1. **Golang Pointers are simple Variables** (because they hold a specific memory address)
2. **Golang Pointers holds the memory address to which they point** (Pointer 'b' has a's memory address as value)

Golang Pointer Declaration | Source – Divyanshu Shekhar

Golang Pointers Dereferencing

We looked at what is a pointer and how Golang Pointers are created? Now let's look at how to dereference a pointer to get the value of the variable to which the pointer points.

Syntax:

*<pointer_name>

Example:

```
a := 17
var b *int = &a
fmt.Println("Value of variable a: ", a)
fmt.Println("Dereferenced pointer b: ", *b)
```

Output:

```
Value of variable a: 17
Dereferenced pointer b: 17
```

When dereferencing the pointer, the value of the variable is received as the pointer points to its memory address.

Dereferencing a pointer is done at the compilation stage, the dereferenced pointer asks the memory address to present it with the value that is contained in that memory location.

Let's see what happens if the value of the variable to which the pointer points is changed.

```
a := 17
var b *int = &a
fmt.Println("Before Changing Variable a")
fmt.Println("Value of variable a: ", a)
fmt.Println("Dereferenced pointer b: ", *b)
a = 13
fmt.Println("After Changing Variable a")
fmt.Println("Value of variable a: ", a)
fmt.Println("Dereferenced pointer b: ", *b)
```

Output:

```
Value of variable a: 17
Dereferenced pointer b: 17
After Changing Variable a
Value of variable a: 13
Dereferenced pointer b: 13
```

Golang Dereferencing pointer to change the value

```
a := 17
var b *int = &a
fmt.Println("Before Changing Variable a")
fmt.Println("Value of variable a: ", a)
fmt.Println("Dereferenced pointer b: ", *b)
*b = 84
fmt.Println("After Changing Variable Using Dereferencing Pointer")
fmt.Println("Value of variable a: ", a)
```

```
    fmt.Println("Dereferenced pointer b: ", *b)
```

Output:

```
Value of variable a: 17
Dereferenced pointer b: 17
After Changing Variable Using Dereferencing Pointer
Value of variable a: 84
Dereferenced pointer b: 84
```

Golang Pointer Arithmetic

```
a := [3]int{1, 2, 3}
b := &a[1]
c := &a[2]
fmt.Printf("%v, %p, %p\n", a, b, c)
```

In this Example, variable a holds an array with size 3 and values 1,2 and 3, Variable b and c points to the second and third element in a's array respectively.

Golang Printf tags:

- **%v** – Value of the array
- **%p** – Pointer value (memory address which it holds)
- **\n** – Newline

Output:

```
[1 2 3], 0xc00011c148, 0xc00011c150
```

The output shows us that the difference between the memory address of the second array element and third element is 2. This actually tells us that the int type takes 2 bytes space. This is based on the type of Go compiler.

Let's see if we can do arithmetic operation on pointer b to get the value of pointer c. (By adding 2 bytes to the memory address hold by pointer b.

```
a := [3]int{1, 2, 3}
b := &a[1] + 2
c := &a[2]
fmt.Printf("%v, %p, %p", a, b, c)
```

```
Error:- invalid operation: &a[1] + 2 (mismatched types *int and int)
```

Golang doesn't allow these kind of pointer arithmetic operations.

Go Pointers vs C Pointers

C/C++ Language allows Arithmetic Operations on Pointers.

C Example:

C Pointer Arithmetic Operation

```
int  var[] = {10, 100, 200};
int  i, *p;
p = var;

for ( i = 0; i < 3; i++) {
   printf("%d\t", i, *p );

   /* increment to next array element*/
   p++;
}
```

Output:

```
10 100 200
```

Arithmetic Operation on Golang Pointers are not allowed.

Example:

Go Code

```
a := [3]int{1, 2, 3}
b := &a
for i := 0; i < 3; i++ {
    fmt.Println(i, *b)
    b++
}
```

Output:

```
invalid operation: b++ (non-numeric type *[3]int)
```

Golang Unsafe Package

Golang by itself doesn't allow us to implement Arithmetic Operations on Pointers but if a developer needs this functionality to make an application, can use the Golang unsafe package to do some advanced pointer arithmetic operations.

Golang Pointers Struct

Go Pointers can also be used to point to a struct.

Example:

```
package main

import (
    "fmt"
)

type Student struct {
    name   string
    rollno int
}

func main() {

    var s *Student
    s = &Student{name: "Divyanshu Shekhar", rollno: 17}
    fmt.Println(s)

}
```

Output:

```
&{Divyanshu Shekhar 17}
```

Golang New Function

Golang's New function is used to create pointers.

```
1  package main
2
3  import (
4      "fmt"
5  )
6
7  type Student struct {
8      name   string
9      rollno int
10 }
11
12 func main() {
13
14     var s *Student
15     s = new(Student)
16     fmt.Println(s)
17
18 }
```

Output:

```
&{ 0}
```

While using New Function, initialization is not possible, it is done afterwards.

The output shows that the pointer variable s contains 0, but it is not true. In Golang when you don't initialize any variable and try to print its value it is 0 as default. This has already been discussed in the Golang Variables tutorial.

Let's see what's the default initialized value of a pointer in Golang!

```
var s *Student
    // printing uninitialized pointer variable
fmt.Println(s)
s = new(Student)
```

```
        // printing initialized pointer variable
    fmt.Println(s)
```

Output:

```
<nil> // uninitialized pointer variable
&{ 0} // initialized pointer variable with New Function
```

The uninitialized pointer variable holds nil value.

Golang Pointers Struct

Syntax 1:

(*<pointer_name>).<Struct_field_name> = <Value>

Example:

(*s).name = “Divyanshu Shekhar”

The dot (.) has higher precedence so, in order to be safe, the dereference pointers are wrapped inside the parenthesis. But there is another syntax which is quite simple.

Syntax 2:

<pointer_name>.<Struct_field_name> = <Value>

Example:

s.name = “Hritul Priya”

This might be confusing as “s” is just a pointer variable and it doesn’t have any field named “name”, then how the value is declared to the struct’s field.

Actually Go has made it simple for the users to work with pointers and most of the work is done by the compiler.

```
var s *Student
fmt.Println(s)
s = new(Student)
fmt.Println(s)
    // Both the Syntax works the same way
(*s).name = "Hritul Priya"
s.rollno = 13
fmt.Println(s)
```

Output:

```
<nil>
&{ 0}
&{Hritul Priya 13}
```

Golang Pointer to Array

Its been already been discussed in Arrays, Slices , Map and Struct Tutorials.

Arrays are Value type so they wholly get copied and changes made in the copied variable doesn't affect the original Array.

Example:

```
a := [3]int{1, 2, 3}
b := a
fmt.Println(a, b)
a[0] = 17
fmt.Println(a, b)
```

Output:

```
[1 2 3] [1 2 3]
[17 2 3] [1 2 3]
```

But, Slices in Go are reference types and uses pointers internally.

```
a := []int{1, 2, 3}
b := a
fmt.Println(a, b)
a[0] = 17
fmt.Println(a, b)
```

Output:

```
[1 2 3] [1 2 3]
[17 2 3] [17 2 3]
```

Maps are also reference types while Structs are value types.

Hope you all like this!

Also Read **Why Golang is called Future of Server-side Language?**

Get to know more about Go Pointers from the Official Go Docs.

Tags: Go, Golang, Internal Pointers, New Function, Nil, Pointers

Previous:
Golang Defer Panic Recover With Examples

Next:
Golang Variadic Arguments With Example

Recent Posts

Categories

Software Development (1)

Web Development (1)

Pages

Blog

Contact Me

Home

Privacy Policy

Sitemap