

Structs

01 MAY 2020

Welcome to tutorial no. 16 in our Golang tutorial series.

What is a struct?

A struct is a user-defined type that represents a collection of fields. It can be used in places where it makes sense to group the data into a single unit rather than having each of them as separate values.

For instance, an employee has a firstName, lastName and age. It makes sense to group these three properties into a single struct named <code>Employee</code>.

Declaring a struct

```
type Employee struct {
firstName string
lastName string
age int
}
```

The above snippet declares a struct type Employee with fields

firstName, lastName and age. The above Employee struct is called a **named struct** because it creates a new data type named Employee using which Employee structs can be created.

Go Tutorial - LThis structed in also bemade more compact by declaring udly published with Ghost 2021 Privacy Policy

fields that belong to the same type in a single line followed by the type name. In the above struct firstName and lastName belong to the same type string and hence the struct can be rewritten as

Although the above syntax saves a few lines of code, it doesn't make the field declarations explicit. Please refrain from using the above syntax.

Creating named structs

Let's declare a **named struct Employee** using the following simple program.

```
package main
1
2
     import (
3
         "fmt"
4
5
6
7
     type Employee struct {
         firstName string
8
         lastName string
9
         age
                    int
10
         salary
                    int
11
12
13
```

```
func main() {

    //creating struct specifying field names
    emp1 := Employee{
        firstName: "Sam",
        age: 25,
        salary: 500,
        lastName: "Anderson",
    }

    //creating struct without specifying field names
    emp2 := Employee{"Thomas", "Paul", 29, 800}

    fmt.Println("Employee 1", emp1)
    fmt.Println("Employee 2", emp2)
}
```

In line no.7 of the above program, we create a named struct type <code>Employee</code>. In line no.17 of the above program, the <code>empl</code> struct is defined by specifying the value for each field name. The order of the fields need not necessarily be the same as that of the order of the field names while declaring the struct type. In this case, we have changed the position of <code>lastName</code> and moved it to the end. This will work without any problems.

In line 25. of the above program, emp2 is defined by omitting the field names. In this case, it is necessary to maintain the order of fields to be the same as specified in the struct declaration. Please refrain from using this syntax since it makes it difficult to figure out which value is for which field. We specified this format here just to understand that this is also a valid syntax:)

4/22/2021, 3:16 PM

The above program prints

```
Employee 1 {Sam Anderson 25 500}
Employee 2 {Thomas Paul 29 800}
```

Get the free Golang tools cheat sheet

Creating anonymous structs

It is possible to declare structs without creating a new data type. These types of structs are called **anonymous structs**.

```
package main
 1
 2
     import (
 3
         "fmt"
4
 5
 6
     func main() {
7
         emp3 := struct {
8
             firstName string
9
             lastName string
10
                        int
             age
11
             salary int
12
         }{
13
             firstName: "Andreah",
14
             lastName:
                         "Nikola",
15
             age:
                         31,
16
             salary:
                         5000,
17
18
19
         fmt.Println("Employee 3", emp3)
20
```

```
}
```

In line no 8. of the above program, an **anonymous struct variable** [emp3] is defined. As we have already mentioned, this struct is called anonymous because it only creates a new struct <u>variable</u> [emp3] and does not define any new struct type like named structs.

This program outputs,

```
Employee 3 {Andreah Nikola 31 5000}
```

Accessing individual fields of a struct

The dot ... operator is used to access the individual fields of a struct.

```
package main
1
2
     import (
3
         "fmt"
4
5
6
     type Employee struct {
7
         firstName string
8
         lastName string
9
                    int
         age
10
         salary
                    int
11
12
13
```

```
func main() {
    emp6 := Employee{
        firstName: "Sam",
        lastName: "Anderson",
        age: 55,
        salary: 6000,
    }
    fmt.Println("First Name:", emp6.firstName)
    fmt.Println("Last Name:", emp6.lastName)
    fmt.Println("Age:", emp6.age)
    fmt.Printf("Salary: $%d\n", emp6.salary)
    emp6.salary = 6500
    fmt.Printf("New Salary: $%d", emp6.salary)
}
```

emp6.firstName in the above program accesses the firstName field of the emp6 struct. In line no. 25 we modify the salary of the employee. This program prints,

```
First Name: Sam

Last Name: Anderson

Age: 55

Salary: $6000

New Salary: $6500
```

Zero value of a struct

When a struct is defined and it is not explicitly initialized with any value, the fields of the struct are assigned their zero values by default.

```
package main
 1
 2
     import (
 3
         "fmt"
4
 5
 6
     type Employee struct {
7
         firstName string
8
         lastName string
9
                    int
         age
10
         salary
                   int
11
     }
12
13
     func main() {
14
         var emp4 Employee //zero valued struct
15
         fmt.Println("First Name:", emp4.firstName)
16
         fmt.Println("Last Name:", emp4.lastName)
17
         fmt.Println("Age:", emp4.age)
18
         fmt.Println("Salary:", emp4.salary)
19
20
```

The above program defines <code>emp4</code> but it is not initialized with any value. Hence <code>firstName</code> and <code>lastName</code> are assigned the zero values of <u>string</u> which is an empty string <code>""</code> and <code>age</code>, <code>salary</code> are assigned the zero values of int which is 0. This program prints,

```
First Name:
Last Name:
Age: 0
Salary: 0
```

It is also possible to specify values for some fields and ignore the rest. In this case, the ignored fields are assigned zero values.

```
package main
1
 2
     import (
 3
         "fmt"
4
 5
 6
     type Employee struct {
 7
         firstName string
8
         lastName string
9
                    int
         age
10
         salary
                    int
11
     }
12
13
     func main() {
14
         emp5 := Employee{
15
             firstName: "John",
16
             lastName: "Paul",
17
18
         fmt.Println("First Name:", emp5.firstName)
19
         fmt.Println("Last Name:", emp5.lastName)
20
         fmt.Println("Age:", emp5.age)
21
         fmt.Println("Salary:", emp5.salary)
22
23
```

Run in playground

In the above program in line. no 16 and 17, firstName and lastName are initialized whereas age and salary are not. Hence age and salary are assigned their zero values. This program outputs,

```
First Name: John
Last Name: Paul
Age: 0
Salary: 0
```

Pointers to a struct

It is also possible to create pointers to a struct.

```
package main
1
 2
     import (
 3
         "fmt"
4
 5
 6
    type Employee struct {
7
         firstName string
8
         lastName string
9
                    int
         age
10
         salary
                    int
11
     }
12
13
    func main() {
14
         emp8 := &Employee{
15
             firstName: "Sam",
16
             lastName: "Anderson",
17
             age:
                         55,
18
             salary:
                         6000,
19
20
         fmt.Println("First Name:", (*emp8).firstName)
21
         fmt.Println("Age:", (*emp8).age)
22
23
```

Run in playground

emp8 in the above program is a pointer to the Employee
struct. (*emp8).firstName is the syntax to access the
firstName field of the emp8 struct. This program prints,

```
First Name: Sam
Age: 55
```

The Go language gives us the option to use emp8.firstName instead of the explicit dereference (*emp8).firstName to access the firstName field.

```
package main
1
 2
     import (
 3
         "fmt"
4
 5
 6
     type Employee struct {
7
         firstName string
8
         lastName string
9
                    int
         age
10
         salary
                    int
11
     }
12
13
     func main() {
14
         emp8 := &Employee{
15
             firstName: "Sam",
16
                         "Anderson",
             lastName:
17
                         55,
             age:
18
             salary:
                         6000,
19
20
         fmt.Println("First Name:", emp8.firstName)
21
         fmt.Println("Age:", emp8.age)
22
23
```

We have used emp8.firstName to access the firstName field in the above program and this program also outputs,

```
First Name: Sam

Age: 55
```

Get the free Golang tools cheat sheet

Anonymous fields

It is possible to create structs with fields that contain only a type without the field name. These kinds of fields are called anonymous fields.

The snippet below creates a struct Person which has two anonymous fields string and int

```
1  type Person struct {
2    string
3    int
4  }
```

Even though anonymous fields do not have an explicit name, by default the name of an anonymous field is the name of its type. For example in the case of the Person struct above, although the fields are anonymous, by default they take the name of the type of the fields. So Person struct has 2 fields with name string and int.

```
package main
1
2
     import (
3
          "fmt"
4
5
6
     type Person struct {
7
          string
8
          int
9
10
```

```
func main() {
    p1 := Person{
        string: "naveen",
        int: 50,
    }
    fmt.Println(p1.string)
    fmt.Println(p1.int)
}
```

In line no. 17 and 18 of the above program, we access the anonymous fields of the Person struct using their types as field name which is string and int respectively. The output of the above program is,

```
naveen
50
```

Nested structs

It is possible that a struct contains a field which in turn is a struct. These kinds of structs are called nested structs.

```
package main

import (
    "fmt"

}

type Address struct {
    city string
```

```
state string
type Person struct {
    name
            string
    age
            int
    address Address
func main() {
    p := Person{
        name: "Naveen",
        age: 50,
        address: Address{
            city: "Chicago",
            state: "Illinois",
        },
    fmt.Println("Name:", p.name)
   fmt.Println("Age:", p.age)
   fmt.Println("City:", p.address.city)
   fmt.Println("State:", p.address.state)
```

The Person struct in the above program has a field address which in turn is a struct. This program prints

```
Name: Naveen

Age: 50

City: Chicago

State: Illinois
```

Promoted fields

Fields that belong to an anonymous struct field in a struct are called promoted fields since they can be accessed as if they belong to the struct which holds the anonymous struct field. I can understand that this definition is quite complex so let's dive right into some code to understand this:).

```
type Address struct {
1
        city string
2
        state string
3
4
    type Person struct {
5
        name string
6
7
        age int
        Address
8
9
```

In the above code snippet, the Person struct has an anonymous field Address which is a struct. Now the fields of the Address namely city and state are called promoted fields since they can be accessed as if they are directly declared in the Person struct itself.

```
package main
1
 2
     import (
 3
          "fmt"
4
 5
 6
 7
     type Address struct {
         city string
 8
         state string
9
10
```

```
type Person struct {
    name string
    age int
    Address
func main() {
    p := Person{
        name: "Naveen",
        age: 50,
        Address: Address{
            city: "Chicago",
            state: "Illinois",
        },
    fmt.Println("Name:", p.name)
   fmt.Println("Age:", p.age)
    fmt.Println("City:", p.city) //city is promoted
   fmt.Println("State:", p.state) //state is promoted
}
```

In line no. 29 and 30 of the program above, the promoted fields <code>city</code> and <code>state</code> are accessed as if they are declared in the struct <code>p</code> itself using the syntax <code>p.city</code> and <code>p.state</code>. This program prints,

```
Name: Naveen

Age: 50

City: Chicago

State: Illinois
```

Exported structs and fields

If a struct type starts with a capital letter, then it is an exported type and it can be accessed from other packages. Similarly, if the fields of a struct start with caps, they can be accessed from other packages.

Let's write a program that has custom packages to understand this better.

Create a folder named structs in your Documents directory.

Please feel free to create it anywhere you like. I prefer my

Documents directory.

```
mkdir ~/Documents/structs
```

Let's create a go module named structs.

```
cd ~/Documents/structs/
go mod init structs
```

Create another directory computer inside structs.

```
mkdir computer
```

Inside the computer directory, create a file spec.go with the following contents.

```
package computer

type Spec struct { //exported struct
    Maker string //exported field
    Price int //exported field
    model string //unexported field

}
```

The above snippet creates a <u>package computer</u> which contains an exported struct type <code>spec</code> with two exported fields <code>Maker</code> and <code>Price</code> and one unexported field <code>model</code>. Let's import this package from the main package and use the <code>spec</code> struct.

Create a file named main.go inside the structs directory and write the following program in main.go

```
package main
1
2
     import (
3
         "structs/computer"
4
         "fmt"
5
6
7
     func main() {
8
         spec := computer.Spec {
9
                  Maker: "apple",
10
                  Price: 50000,
11
12
         fmt.Println("Maker:", spec.Maker)
13
         fmt.Println("Price:", spec.Price)
14
15
```

The structs folder should have the following structure,

In line no. 4 of the program above, we import the computer package. In line no. 13 and 14, we access the two exported fields Maker and Price of the struct Spec. This program can be run by executing the commands go install followed by structs command. If you are not sure about how to run a Go program, please visit https://golangbot.com/hello-world-gomod/#1goinstall to know more.

```
go install structs
```

Running the above commands will print,

```
Maker: apple
Price: 50000
```

If we try to access the unexported field model, the compiler will complain. Replace the contents of main.go with the following code.

```
package main
```

In line no. 12 of the above program, we try to access the unexported field model. Running this program will result in compilation error.

```
# structs
./main.go:12:13: unknown field 'model' in struct
literal of type computer.Spec
```

Since model field is unexported, it cannot be accessed from other packages.

Structs Equality

Structs are value types and are comparable if each of their fields are comparable. Two struct variables are considered equal if their corresponding fields are equal.

```
1 package main
```

```
import (
    "fmt"
type name struct {
    firstName string
    lastName string
func main() {
    name1 := name{
        firstName: "Steve",
        lastName: "Jobs",
    name2 := name{
        firstName: "Steve",
        lastName: "Jobs",
    if name1 == name2 {
        fmt.Println("name1 and name2 are equal")
    } else {
        fmt.Println("name1 and name2 are not equal")
    name3 := name{
        firstName: "Steve",
        lastName: "Jobs",
    }
    name4 := name{
        firstName: "Steve",
    }
    if name3 == name4 {
        fmt.Println("name3 and name4 are equal")
    } else {
        fmt.Println("name3 and name4 are not equal")
    }
}
```

In the above program, name struct type contain two string fields. Since strings are comparable, it is possible to compare two struct variables of type name.

In the above program <code>name1</code> and <code>name2</code> are equal whereas <code>name3</code> and <code>name4</code> are not. This program will output,

```
name1 and name2 are equal name3 and name4 are not equal
```

Struct variables are not comparable if they contain fields that are not comparable (Thanks to <u>alasijia</u> from reddit for pointing this out).

```
package main
1
2
     import (
3
         "fmt"
4
5
6
     type image struct {
7
         data map[int]int
8
     }
9
10
     func main() {
11
         image1 := image{
12
              data: map[int]int{
13
                  0: 155,
14
              }}
15
         image2 := image{
16
```

```
data: map[int]int{
      0: 155,
    }}
if image1 == image2 {
    fmt.Println("image1 and image2 are equal")
}
```

In the program above <code>image</code> struct type contains a field <code>data</code> which is of type <code>map</code>. maps are not comparable, hence <code>image1</code> and <code>image2</code> cannot be compared. If you run this program, the compilation will fail with error

```
./prog.go:20:12: invalid operation: image1 == image2
(struct containing map[int]int cannot be compared)
```

Thanks for reading. Please leave your comments and feedback.

Next tutorial - Methods

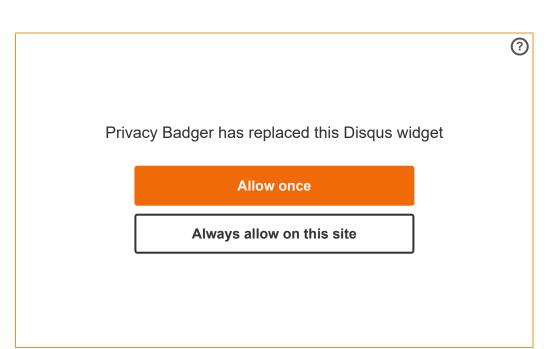
Like my tutorials? Please support the content.

Get the free Golang tools cheat sheet

Naveen Ramanathan

Share this post

Naveen Ramanathan is a software engineer with interests in Go, Docker, Kubernetes, Swift, Python, and Web Assembly. If you would like to hire him, please mail to naveen[at]golangbot[dot]com.

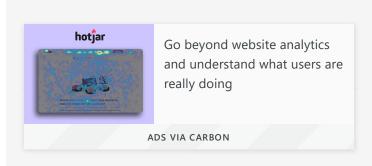


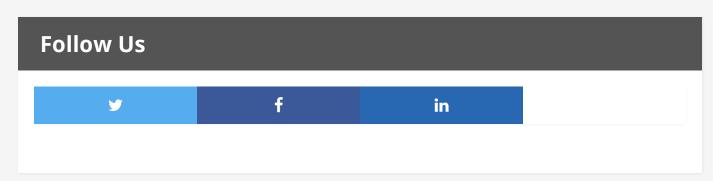
Support Me

Please support my work by donating. Your donations will help me with the running costs of the website and also my personal time investment.









Popular Articles

Goroutines

This tutorial discusses how concurrency is achieved in Go using goroutines.

Channels

This article explains how channels can be used to establish communication between goroutines.

Arrays and Slices

A detailed tutorial about arrays and slices covering the internal implementation details too.

Structs

An in-depth tutorial about declaring and defining structs. It also covers anonymous structs, promoted fields and nested structs.

First Class Functions

A tutorial explaining how to use anonymous functions, user-defined functions, higher order functions and closures in Go.

Interfaces

Learn how interfaces are declared and implemented and also get to know the use of interfaces in Go.

