

GOLANGBOT.COM

[GOLANG TUTORIAL - TABLE OF CONTENTS](#)

[SUPPORT THIS WEBSITE](#)

[HIRE ME](#)

[ABOUT](#)

[ADVERTISE HERE](#)

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 `Employee`.

Declaring a struct

```
1  type Employee struct {  
2      firstName string  
3      lastName  string  
4      age       int  
5  }
```

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 - Learn to Go the easy way with code examples. Proudly published with Ghost 2021 Privacy Policy

This struct can also be made more compact by declaring 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

```
1 type Employee struct {
2     firstName, lastName string
3     age                int
4 }
```

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.

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

[Run in playground](#)

In line no.7 of the above program, we create a named struct type `Employee`. In line no.17 of the above program, the `emp1` 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 `lastName` 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 :)

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**.

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

```
}
```

Run in playground

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 variable `emp3` and does not define any new struct type like named structs.

This program outputs,

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

Accessing individual fields of a struct

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

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

[Run in playground](#)

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.

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

[Run in playground](#)

The above program defines `emp4` but it is not initialized with any value. Hence `firstName` and `lastName` are assigned the zero values of `string` which is an empty string `""` and `age`, `salary` 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.

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

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

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

[Run in playground](#)

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`.

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

```
}

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

[Run in playground](#)

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.

```
1 package main
2
3 import (
4     "fmt"
5 )
6
7 type Address struct {
8     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)
}
```

[Run in playground](#)

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 :).

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

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

[Run in playground](#)

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

```
1 package computer
2
3 type Spec struct { //exported struct
4     Maker string //exported field
5     Price int //exported field
6     model string //unexported field
7
8 }
```

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

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

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

The `structs` folder should have the following structure,

```
├─ structs
│   └─ computer
│       └─ spec.go
│       └─ go.mod
│       └─ main.go
```

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.

```
1 | package main
2 |
```

```
import (  
    "structs/computer"  
    "fmt"  
)  
  
func main() {  
    spec := computer.Spec {  
        Maker: "apple",  
        Price: 50000,  
        model: "Mac Mini",  
    }  
    fmt.Println("Maker:", spec.Maker)  
    fmt.Println("Price:", spec.Price)  
}
```

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")  
    }  
}
```

Run in playground

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 `name1` and `name2` are equal whereas `name3` and `name4` 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 [alasijia](#) from reddit for pointing this out).

```
1  package main
2
3  import (
4      "fmt"
5  )
6
7  type image struct {
8      data map[int]int
9  }
10
11 func main() {
12     image1 := image{
13         data: map[int]int{
14             0: 155,
15         }}
16     image2 := image{
17
```

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

Run in playground

In the program above `image` struct type contains a field `data` which is of type `map`. maps are not comparable, hence `image1` and `image2` 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

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.

Share this post



Privacy Badger has replaced this Disqus widget

Allow once


Always allow on this site

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.

Donate on  patreon





Go beyond website analytics
and understand what users are
really doing

ADS VIA CARBON

Follow Us



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.

Newsletter

Join Our Newsletter

Signup for our newsletter and get the **Golang tools cheat sheet for free.**



Subscribe