Introduction to Struct

This lesson introduces structs, addressing rudimentary concepts such as declaration and memory allocation.

WE'LL COVER THE FOLLOWING

Go supports *user-defined* or *custom* types in the form of alias types or structs. A struct tries to represent a real-world entity with its properties. Structs are *composite* types to use when you want to define a type that consists of several properties each having their type and value, grouping pieces of data together. Then, one can access that data as if it were part of a single entity.

Structs are *value* types and are constructed with the `new` function. The component pieces of data that constitute the struct type are called `fields`. A field has a type and a name. Field names within a struct must be unique.

The concept was called **ADT** (**A**bstract **D**ata **T**ype) in older texts on software engineering. It was also called a *record* in older languages like Cobol, and it also exists under the same name of struct in the C-family of languages and in the OO languages as a lightweight-class without methods.

However, because Go does not have the concept of a class, the struct type has a much more important place in Go.

Definition of a struct

The general format of the definition of a struct is as follows:

```
type identifier struct {
  field1 type1
  field2 type2
```

```
  ...
}
```

Also, `type T struct { a, b int }` is legal syntax and is more suited for simple structs. The fields in this struct have names, like `field1`, `field2`, and so on. If the field is never used in code, it can be named `_`.

These fields can be of any type, even structs themselves, functions or interfaces. Because a struct is a value, we can declare a variable of the struct type, and give the values of its fields, like:

```
var s T
s.a = 5
s.b = 8
```

An array could be seen as a sort of struct but with indexed rather than named fields.

Using `new()` function

Memory for a new struct variable is allocated with the `new` function, which returns a pointer to the allocated storage:

```
var t *T = new(T)
```

This can be put on different lines if needed (e.g., when the declaration has to be package scope, but the allocation is not needed at the start):

```
var t *T
t = new(T)
```

The idiom to write this shorter is: `t := new(T)`; the variable `t` is a pointer to `T`. At this point, the fields contain the zero-values according to their types. However, declaring `var t T` also allocates and zero-initializes memory for `t`, but now `t` is of type `T`. In both cases, `t` in OO jargon is commonly called an *instance* or *object* of the type `T`, but in Go it is merely a *value* of type `T`.

The fields can be given a different value by using the *dot-notation*, as is custom in OO-languages:

```
structname.fieldname = value
```

The values of the struct-fields can be retrieved with the same notation:

```
structname.fieldname
```

This is called a **selector** in Go. In order to access the fields of a struct, whether the variable is of the struct type or a pointer to the struct type, we use the same selector-notation:

```
type myStruct struct { i int, j float32, k string }
var v myStruct // v has struct type
var p *myStruct // p is a pointer to a struct
v.i
p.i
```

An even shorter notation and the idiomatic way to initialize a struct value (a struct-literal) is the following:

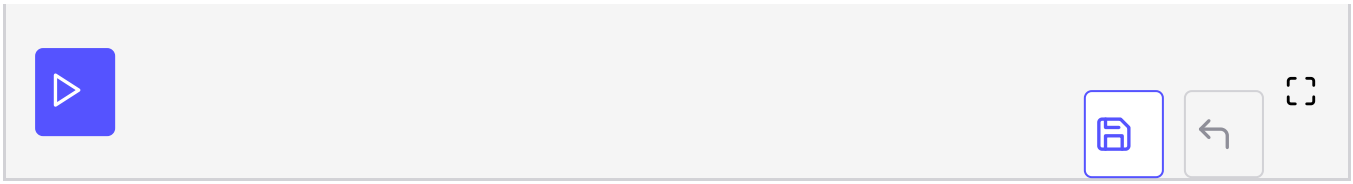
```
v := &myStruct{10, 15.5, "Chris"} // this means that `v` is of type *myStr
uct
```

or:

```
var mt myStruct
mt = myStruct{10, 15.5, "Chris"}
```

A simple example is given below:

```
package main
import "fmt"

type struct1 struct {   // struct definition
  i1 int
  f1 float32
  str string
}

func main() {
  ms := new(struct1)    // making a struct1 type variable
  // Filling fields of the struct of struct1 type
  ms.i1 = 10
  ms.f1 = 15.5
  ms.str = "Chris"
  fmt.Printf("The int is: %d\n", ms.i1)
  fmt.Printf("The float is: %f\n", ms.f1)
  fmt.Printf("The string is: %s\n", ms.str)
  fmt.Println(ms)
}
```



Making a Struct

In the above program, look at **line 4**. We are declaring a struct of type `struct1`. It has *three* different fields. First, we have an *integer* `i1` (see **line 5**), then we have a *float32* type variable `f1` (see **line 6**) and lastly, there is a *string* variable `str` (see **line 7**). This means, if we made a variable of type `struct1`, it would hold these three different fields to which we can assign values by our own choice. Now, look at the `main` function. At **line 11**, we make a `struct1` type struct `ms` using the `new` function as: `ms := new(struct1)`. In the next three lines, we are assigning values to the fields of `ms`. At **line 13**, we assign `ms.i1` the value of **10**. At **line 14**, we assign `ms.f1` the value of **15.5**. At **line 15**, we assign `ms.str` the value of **Chris**. In the next lines (from **line 16** to **line 19**), we are printing the fields' values for `ms` to verify the results.

In the following example, we see that you can also initialize values by *preceding* them with the field names. So, `new(Type)` and `&Type{}` are *equivalent* expressions. A typical example of a struct is a time-interval (with a *start* and *end* time expressed here in seconds):

```
type Interval struct {
  start int
  end int
}
```

Here, are some initializations:

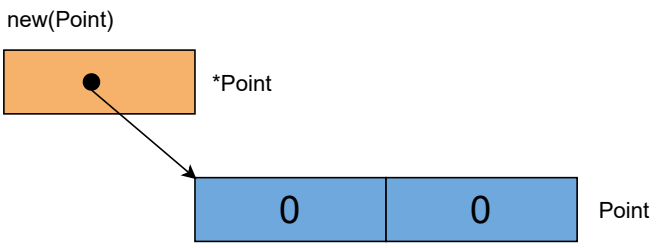
```
inter := Interval{0,3} //(A)
inter2 := Interval{end:5, start:1} //(B)
inter3 := Interval{end:5} //(C)
```

In case **A**, the values given in the literal must be exactly in the same order as the fields are defined in the struct; the `&` is not mandatory. Case **B** shows another possibility where the field names with a `:` precede the values; in that case, the order of the field names can be different from the order of their definition. Also, if fields are omitted like in case **C**, zero-values are assigned.

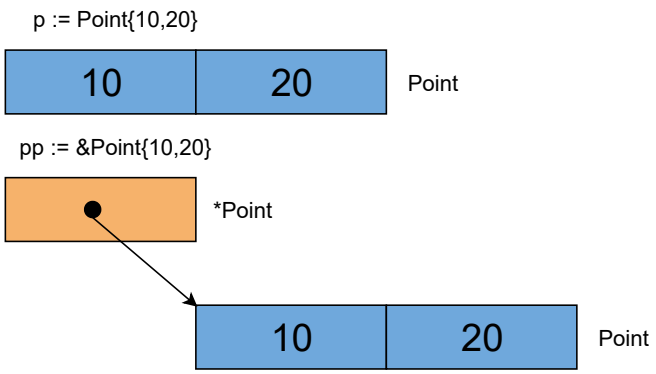
The naming of the struct type and its fields adheres to the *Visibility rule*. An

The naming of the struct type and its fields adheres to the *Visibility-rule*. An exported struct type may have a mix of fields where some are exported and others not.

The following figure clarifies the memory layout of a struct value and a pointer to a struct for the following struct type: `type Point struct { x, y int }`



Memory Layout of a Struct, Initialized with new()

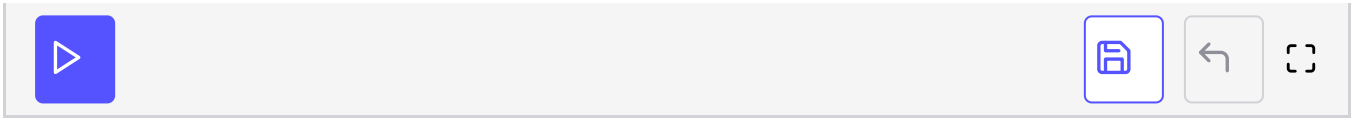


Memory Layout of a Struct, Initialized as Struct Literal

The following example shows how to pass a struct as a parameter:

```
package main
import (
  "fmt"
  "strings"
)

type Person struct {  // struct definition
  firstName string
  lastName string
}

func upPerson (p *Person) { // function using struct as a parameter
  p.firstName = strings.ToUpper(p.firstName)
  p.lastName = strings.ToUpper(p.lastName)
}

func main() {
  // 1- struct as a value type:
  var pers1 Person
  pers1.firstName = "Chris"
  pers1.lastName = "Woodward"
  upPerson(&pers1)
  fmt.Printf("The name of the person is %s %s\n", pers1.firstName, pers1.lastName)

  // 2 - struct as a pointer:
  pers2 := new(Person)
```

```
  pers2.firstName = "Chris"
  pers2.lastName = "Woodward"
  (*pers2).lastName = "Woodward" // this is also valid
  upPerson(pers2)
  fmt.Printf("The name of the person is %s %s\n", pers2.firstName, pers2.lastName)

  // 3 - struct as a literal:
  pers3 := &Person{"Chris","Woodward"}
  upPerson(pers3)
  fmt.Printf("The name of the person is %s %s\n", pers3.firstName, pers3.lastName)
}
```



Passing Struct as a Parameter

In the above program, look at **line** 7. We are declaring a struct of type `Person`. It has *two* different fields: a *string* called `firstName` (see **line 8**) and a *string* called `lastName` (see **line 9**). Look at the header of the function `upPerson` at **line 12**: `func upPerson (p *Person)`. Here, a pointer to the `Person` type variable is passed as a parameter, which means if the values of the fields are changed inside the function for `Person` type struct, the effects will be visible from outside the function too. Nothing is being returned from this function. Look at **line 13** and **14**, we are converting both the strings to uppercase. Now, let's see the `main` function, where we use *three* different methods to call `upPerson` function.

Let's see the first case. At **line 19**, we declare a `Person` type variable `pers1` as a *value* type. In the next two lines, we fill the fields `firstName` as **Chris** and `lastName` as **Woodward** of `pers1` using `.` selector. At **line 22**, we call the `upPerson` function on `pers1`. The `upPerson` accepts *pointer* to the `Person` type, but `pers1` is of *value* type, so we pass `pers1` as: `upPerson(&pers1)`. Because `&pers1`, is the pointer (address) to `pers1`.

Let's see the second case. At **line 26**, we declare a `Person` type variable `pers2` as a pointer type. In the next two lines, we fill the fields `firstName` as **Chris** and `lastName` as **Woodward** of `pers1` using the `.` selector. At **line 30**, we call the `upPerson` function on `pers2`. The `upPerson` accepts a pointer to the `Person` type and `pers2` is already of pointer type, so we pass `pers2` as it is: `upPerson(pers2)`.

Let's see the third case. At **line 34**, we declare a `Person` type variable `pers3` as a *struct literal* type as: `pers3 := &Person{"Chris","Woodward"}`. At **line 35**, we

call the `upPerson` function on `pers3`. As `upPerson` accepts a *pointer* to the

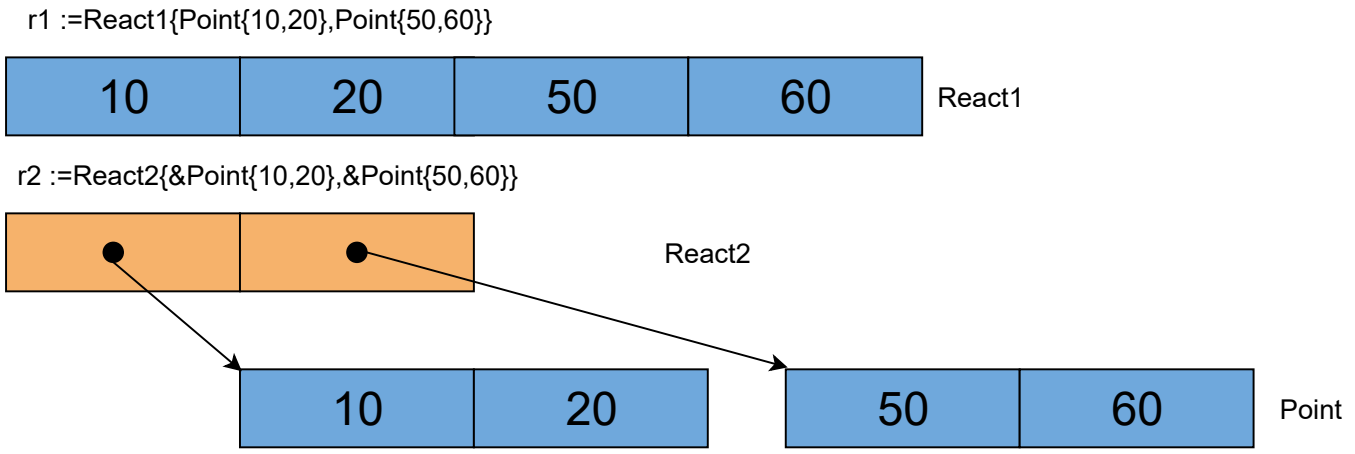
`Person` type and `pers3` is already of *pointer* type, so we pass `pers3` as it is: `upPerson(pers3)`.

At **lines 23**, **31** and **36**, we are printing the fields of `pers1`, `pers2`, and `pers3`, respectively. For all three of them, `firstName` will now be **CHRIS** instead of **Chris**, and `lastName` will now be **WOODWARD** instead of **Woodward**.

Struct of structs

Structs in Go and the data they contain, even when a struct contains other structs, form a contiguous block in memory. It gives a huge performance benefit. This is unlike in Java with its reference types, where an object and its contained objects can be in different parts of memory. In Go, this is also the case with pointers. This is clearly illustrated in the following example:

```
type Rect1 struct { Min, Max Point }
type Rect2 struct { Min, Max *Point }
```



Memory Layout of a Struct of Structs

That's it for the introduction. Now let's dive into some advanced concepts of structs in Go.