Advanced Structs Concepts

This lesson provides detailed information on structs and how other data structures are related to structs in Go.

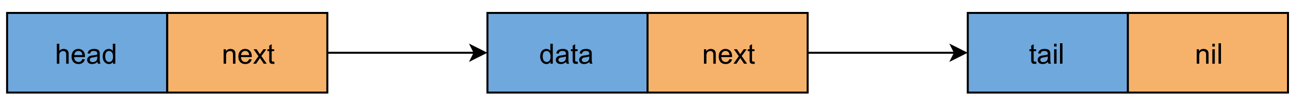
WE'LL COVER THE FOLLOWING

Recursive structs

A struct type can be defined in terms of itself. This is particularly useful when the struct variable is an element of a *linked list* or a *binary tree*, commonly called a **node**. In that case, the node contains links (the addresses) to the neighboring nodes.

In the following examples, *next* for a list and *left* and *right* for a tree are pointers to another Node-variable.

Linked List



Linked List as Recursive Struct

The *data* field contains useful information (for example a float64), and *next* points to the successor node; in Go-code:

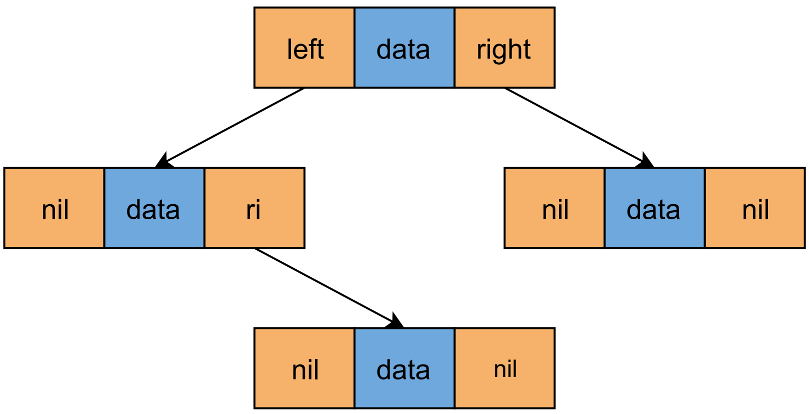
```
type Node struct {
    data float64
    next *Node
```

```
}
```

The first element of the list is called the *head*, and it points to the 2nd element. The last element is called the *tail*; it doesn't point to any successor, so its *next* field has value *nil*. Of course, in a real list, we would have many data-nodes. The list can grow or shrink dynamically. In the same way, you could define a doubly-linked list with a predecessor node field *pr* and a successor field *su*.

```go
type Node struct {
    pr *Node
    data float64
    su *Node
}
```

Binary Tree



Binary Tree as a Recursive Struct

Here, each node has at most two links to other nodes: the left and the right. Both of them can propagate this further. The top element of the tree is called the *root*. The bottom layer of nodes, which have no more nodes beneath them, are called the *leaves*. A leaf node has *nil*-values for the `left` and `right` pointers. We could define the type like this:

```go
type Tree struct {
    left *Tree
    data float64
    right *Tree
}
```

Size of a struct

If you have a struct type `T` and you quickly want to see how many bytes a value occupies in memory, use:

```
size := unsafe.Sizeof(T{})
```

There is a difference between the size of a struct containing another struct and the size of a struct containing a pointer to another struct. This is illustrated in the following code snippet:

```
package main
import (
  "fmt"
  "unsafe"  // to use function Sizeof()
)

type T1 struct {
  a, b int64
}

type T2 struct {
  t1 *T1  // pointer to T1
}

type T3 struct {
  t1 T1   // value type of T1
}

func main() {
  fmt.Println("Size of T1:", unsafe.Sizeof(T1{}))       // T1 value type
  fmt.Println("Size of T2:", unsafe.Sizeof(T2{&T1{}}))  // T2 containing pointer to T1
  fmt.Println("Size of T3:", unsafe.Sizeof(T3{}))       // Value of T3
}
```

Size of Struct

In the program above, at **line 4**, we import the package `unsafe` to use the function `Sizeof()` that tells the size of the parameter passed to it. At **line 7**, we declare a struct of type `T1`, which has two fields of type `int64`; `a` and `b`. At **line 11**, we declare another struct of type `T2`, which has one field, and is a pointer to the variable of type `T1` called `t1`. At **line 15**, we declare another struct of type `T3` with one field `t1`, which is a value of type `T1`.

Now, look at the `main`. At **line 20**, we are printing the size of `T1`. The size of one `int64` type number is **8** bytes. Since the struct `T1` has two `int64` type numbers, the total size will be **16**.

At **line 21**, we are printing the size of `T2{&T1{}}`. Since it's a pointer or address type, the size of the pointer is the same, regardless of what data type they are pointing to. On a 32-bit machine, the size of the pointer is **4** bytes, where on a 64-bit machine, the size of the pointer is **8** bytes. In this case, the size of the pointer is also **8** bytes.

At **line 22**, we are printing the size of `T3`. Where `T3` holds a value of type `T1`, and the size of `T1` is **16**. So, the size of `T3` will also be **16** bytes.

Conversion of structs

As we have already seen, conversion in Go follows strict rules. When we have a struct type and define an alias type for it, both types have the same underlying type and can be converted into one another. Also note the compile-error cases which denote impossible assignments or conversions. Look at the following implementation of the conversion:

Environment Variables

Key:	Value:
GOROOT	/usr/local/go
GOPATH	//root/usr/local/go/src
PATH	//root/usr/local/go/src/bin:/usr/local/go...

```
package main
import (
	"fmt"
)

type number struct {
	f	float32
}

type nr number   // new distinct type
type nrAlias = number // alias type

func main() {
	a := number{5.0}
	b := nr{5.0}
	c := nrAlias{5.0}
	// var i float32 = b   // compile-error: cannot use b (type nr) as type float32 in as
	// var i = float32(b)  // compile-error: cannot convert b (type nr) to type float32
	// var d number = b    // compile-error: cannot use b (type nr) as type number in ass
	// needs a conversion:
	var d = number(b)
	// an alias doesn't need conversion:
	var e = b
	fmt.Println(a, b, c, d, e)
}
```

Click the **RUN** button and wait for the terminal to start. Type `go run main.go` and press ENTER. In case you make any changes to the file you have to press **RUN** again.

In the above code, at **line 6**, we declare struct `number` with one field of type *float32* `f`. Then, at **line 10**, we create another distinct type (aliasing) for `number` as `nr`. In the next line, we alias the type `number` as `nrAlias`.

Now, look at `main`. At **line 14**, we are creating a variable of type `number`, as `a` and setting its `f` as **5.0**. In the next line, we are creating another variable of type `number` but by using its alias `nr` as `b` and setting its `f` also as **5.0**. At **line 16**, we are creating another variable of type `number` but by using its alias `nrAlias` as `c` and setting its `f` also as **5.0**.

Look at the commented part (from **line 17** to line **20**). At **line 17**, we are creating a new variable `i` of type **float32** and setting it equal to `b`. It will generate an error because `b` type is `nr`, and it can't be assigned to a variable of type *float32*. Similarly, in the next line, we are again creating a new variable `i` of type **float32** but by setting it equal to `float32(b)`. It will generate an error because `b` type is `nr`, and it can't be directly converted to *float32*. At **line 19**, we are creating a new variable `d` of type `number` and setting it equal to `b`. It will generate an error because `b` type is `nr`, and it can't be assigned to a variable of type `number`, no matter whether `nr` is the alias of type `number`.

Look at **line 21**, to make a `number` type variable equal to `nr`; we have to convert the type as: `var d = number(b)`. Such a conversion makes `d` as a `number` type variable with fields equal to the field of `b` (of type `nr`). In the next line, we are making another `nr` type variable `e` and setting it equal to `b`. For such a case, we don't need any conversion, and all the fields of `b` and `e` would be the same. At **line 24**, we are printing all the variables : `a`, `b`, `c`, `d` and `e`. All variables will give **{5}** as an output. Because for all `f` is **5.0**.

Now, that you are familiar with advanced concepts, let's study the concept of factories related to structs in the next lesson.