Chapter 4

Objective

- **Pointers to a struct**
- **Anonymous struct Fields**
- **Nested structs**
- **Promoted struct Fields**
- **Structs Equality**
- **Mathods vs Functions**
- **Methods with Same Names**
- **Pointer receivers vs value receivers**
- **When to use pointer receiver and when to use value receiver**
- **Value Receivers vs Value Arguments**
- **Pointer receivers in methods vs pointer arguments in functions.**
- **Methods on non struct types**
- **Linked list Data structure**
- **Using new Function To Create Link List**
- **Create Link List using Method**
- **Link List Container in Go**
- **LIFO (Stack)**
- **Stack of int Slice**
- **Create Stack Of Strings Using Slice**

This page is left blank intentionally

Unit 1

Pointers to a struct

- Struct fields can be accessed through a **struct** pointer.
- The struct field X can be accessed by pointer **(*ptr).X**
- Golang also allows **ptr.X** to access the **struct field** without the explicit dereference.
- Golang does not support **arrow operator** that C/C++ has (**->**)

```go
package main // Example 4-1
import "fmt"

type Employee struct {
   firstName, lastName string
   age, salary         int
}

func main() {
   // emp holds the address of Employee
   emp := &Employee{"Sam", "Anderson", 55, 6000}

   // Using dereferencing (*)
   fmt.Println("First Name:", (*emp).firstName)
   fmt.Println("Age:", (*emp).age)

   // Using just dot operator
   fmt.Println("First Name:", emp.firstName)
   fmt.Println("Age:", emp.age)
}
```

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

Anonymous struct Fields

- Anonymous struct fields are those fields that only contain **data type** without the field name
- By default the name of an anonymous field is the name of its type.

```go
package main // Example 4-2
import "fmt"

type Person struct {
   string // Anonymous Field
   int    // Anonymous Field
}

func main() {
   admin := Person{"John", 50}
   fmt.Println(admin)
   AnonymousFields()
}

// Access the anonymous fields of the Person struct using their types
// as field name which is "string" and "int" respectively.
func AnonymousFields() {
   var admin Person
   admin.string = "Jack"
   admin.int = 50
   fmt.Println(admin)
}
```

Output:

```
{John 50}
{Jack 50}
```

Nested structs

- In other computer languages it is called composition, Has-A relationship.
- The structs are nested if one structure name is used within other sturct
- One struct contains a field which in turn is a other struct.

```
package main // Example 4-3
import "fmt"

type Address struct {
   city, state string
}
type Person struct {
   name    string
   age     int
   address Address // nested struct
}

func main() {
   var admin Person
   admin.name = "Jack"
   admin.age = 50
   admin.address = Address{ // initilize nested struct
      city:  "Chicago",
      state: "Illinois",
   }

   fmt.Println("Name:", admin.name)
   fmt.Println("Age:", admin.age)
   fmt.Println("City:", admin.address.city)
   fmt.Println("State:", admin.address.state)
}
```

Output:

```
Name: Jack
Age: 50
City: Chicago
State: Illinois
```

Promoted struct Fields

- Anonymous struct fields in a structure are called **promoted** fields
- They can be accessed as if they belong to the structure which holds the **anonymous** struct field

```go
package main // Example 4-4
import "fmt"

type Address struct {
   city, state string
}
type Person struct {
   name    string
   age     int
   Address // Anonymous struct field is called promoted field
}

func main() {
   var admin Person
   admin.name = "John"
   admin.age = 50
   admin.Address = Address{ // initialize Anonymous struct fields
      city:  "Chicago",
      state: "Illinois",
   }
   fmt.Println("Name:", admin.name)
   fmt.Println("Age:", admin.age)
   fmt.Println("City:", admin.city)   //city is promoted field
   fmt.Println("State:", admin.state) //state is promoted field
}
```

Output:

```
Name: John
Age: 50
City: Chicago
State: Illinois
```

Structs Equality

- Two struct variables are considered equal if their corresponding fields are equal.
- Struct variables are not comparable if they contain fields which are not comparable
- If struct contains map type field, the two variable of struct cannot be comparable.

```
package main // Example 4-5
import "fmt"

type name struct {
   firstName string
   lastName  string
}

func main() {
   admin1 := name{"Steve", "Jobs"}
   admin2 := name{"Steve", "Jobs"}
   if admin1 == admin2 {
      fmt.Println("admin1 and admin2 are equal")
   } else {
      fmt.Println("admin1 and admin2 are not equal")
   }

   admin3 := name{firstName: "Steve", lastName: "Jobs"}
   admin4 := name{}
   admin4.firstName = "Steve"
   if admin3 == admin4 { // not comparable
      fmt.Println("admin3 and admin4 are equal")
   } else {
      fmt.Println("admin3 and admin4 are not equal")
   }
}
```

Output:

```
admin1 and admin2 are equal
admin3 and admin4 are not equal
```

struct cannot be comparable if it has **map data type**.

```
 1 package main // Example 4-6
 2  import "fmt"
 3
 4 type image struct {
 5    // struct containing map[int]int cannot be compared
 6    data map[int]int
 7 }
 8
 9 func main() {
10    image1 := image{data: map[int]int{
11       0: 155,
12    }}
13    image2 := image{data: map[int]int{
14       0: 155,
15    }}
16
17    fmt.Println(image1,image2)
18
19    // variables image1 and image2 are not comparable
20    if image1 == image2 {
21       fmt.Println("image1 and image2 are equal")
22    }
23 }
```

Output:

```
# command-line-arguments
./ex4-06.go:20:12: invalid operation: image1 == image2
(struct containing map[int]int cannot be compared)
```

Unit 2

Mathods vs Functions

- A method is just a function with a special **receiver** type that is written between the func keyword and the method name.
- The receiver can be either **struct** type or **non struct** type.
- The receiver is available for access inside the body of a method.
- The following is the syntax to create a method.

```
func (t Type) methodName(parameter list) {
}
```

- The above snippet creates a method named methodName which has receiver type Type.

```
package main // Example 4-7
import "fmt"

type Employee struct {
   name     string
   salary   int
   currency string
}

// showSalary() is a method that has Employee as the receiver
// type method has access to the receiver e Employee inside it
func (e Employee) showSalary() {
   fmt.Printf("Salary of %s is %s%d\n", e.name, e.currency, e.salary)
}

func main() {
   emp := Employee{
      name:     "Mark Tyler",
      salary:   5000,
      currency: "$",
   }

   //Call showSalary() method of Employee type
   emp.showSalary()
}
```

Output:

```
Salary of Mark Tyler is $5000
```

- **Function cannot be attached to a type, like methods do.**
- By default type is passed to a function parameter by value.

```
// Passing struct type as a function parameter.
package main // Example 4-8
import "fmt"

type Employee struct {
   name     string
   salary   int
   currency string
}

// showSalary() is a function that takes Employee variable
// as a parameter and has access to the e Employee inside showSalary
func showSalary(e Employee) {
   fmt.Printf("Salary of %s is %s%d\n", e.name, e.currency, e.salary)
}

func main() {
   emp := Employee{
      name:     "Mark Tyler",
      salary:   5000,
      currency: "$",
   }

   //Call showSalary(emp) function
   showSalary(emp) // pass Employee type variable
}
```

```
Output:
Salary of Mark Tyler is $5000
```

Methods with Same Names

- Methods with same name can be defined on different types whereas **functions** with the **same names** are **not allowed**.

```
package main // Example 4-9
import (
   "fmt"
   "math"
)

type Rectangle struct {
   length int
   width  int
}

type Circle struct {
   radius float64
}

func (r Rectangle) Area() int { // Receiver is r
   return r.length * r.width
}

func (c Circle) Area() float64 { // Receiver is c
   return math.Pi * c.radius * c.radius
}

func main() {
   r := Rectangle{
      length: 10,
      width:  5,
   }

   fmt.Printf("Area of rectangle %d\n", r.Area())
   c := Circle{
      radius: 12,
   }

   fmt.Printf("Area of circle %f\n", c.Area())
}
```

Output:

```
Area of rectangle 50
Area of circle 452.389342
```

Unit 3

Pointer Receivers vs Value Receivers

- Golang allows creating methods with **pointer** receivers.
- With **pointer receivers** changes made inside a method are visible to the caller.
- The **value receivers** changes in a method are not visible to its caller.

```
package main // Example 4-10
import "fmt"

type Employee struct {
   name string
   age  int
}

// Method with value receiver
func (e Employee) changeName(newName string) {
   e.name = newName
}

// Method with pointer receiver
func (e *Employee) changeAge(newAge int) {
   e.age = newAge
}

func main() {
   e := Employee{
      name: "Mark Andrew",
      age:  50,
   }

   fmt.Printf("Name before call to changeName: %s", e.name)
   e.changeName("Michael Andrew")
   fmt.Printf("\nName after call to changeName: %s", e.name)

   fmt.Printf("\n\nAge before call to changeAge: %d", e.age)

   // & is not needed. Go gives the option to just use dot (.)
   (&e).changeAge(51)
   fmt.Printf("\nAge after call to changeAge: %d\n", e.age)
}
```

Output:

```
Name before call to changeName: Mark Andrew
Name after call to changeName: Mark Andrew

Age before call to changeAge: 50
Age after call to changeAge: 51
```


When to use pointer receiver and when to use value receiver

- Generally pointer receivers can be used when changes made to the receiver inside the method should be visible to the caller.
- Pointers receivers can also be used in places where it is expensive to copy a data structure.
- If a pointer receiver is used for sturct type, the struct type will not be copied, only an address to it will be used in the method.

Value Receivers vs Value Arguments

- Value receivers in methods vs value arguments in functions
- When a function has a value argument, it will accept only a value argument.
- When a method has a value receiver, it will accept both pointer and value receivers.

```
package main // Example 4-11
import "fmt"

type rectangle struct {
   length int
   width int
}

func area(r rectangle) { // accepts a value argument
   fmt.Printf("Area Function result: %d\n", (r.length * r.width))
}

// method func (r rectangle) area() accepts a value receiver.
func (r rectangle) area() {
  fmt.Printf("Area Method result: %d\n", (r.length * r.width))
}

func main() {
   r := rectangle{
      length: 10,
      width:  5,
   }

  area(r) // calling area function, argument is pass by value
   r.area() //  calling area method
   p := &r // a pointer p to r

  // compilation error, cannot use p (type *rectangle) as type
   // rectangle in argument to area
   //area(p) // function area only takes pass by value
   p.area() //calling value receiver with a pointer
  // accepts only a value receiver using the pointer receiver p.
  // In Go for convenience it allows p.area() instead of (*p).area()
}
```

Output:

```
Area Function result: 50
Area Method result: 50
Area Method result: 50
```

Pointer receivers in methods vs pointer arguments in functions.

- functions with pointer arguments will accept only pointers
- methods with pointer receivers will accept both value and pointer receiver.

```
package main // Example 4-12

import "fmt"

type rectangle struct {
   length int
   width  int
}

func boundary(r *rectangle) {
   fmt.Println("boundary function output:", 2*(r.length+r.width))

}

func (r *rectangle) boundary() {
   fmt.Println("boundary method output:", 2*(r.length+r.width))
}

func main() {
   r := rectangle{
      length: 10,
      width:  5,
   }
   p := &r // address of r
   boundary(p)    // Function Call
   p.boundary()   // Method Call

   // boundary function argument is type pointer, cannot pass r
   // boundary(r)

   r.boundary() //calling pointer receiver with a value
```

Output:

```
boundary function output: 30
boundary method output: 30
boundary method output: 30
```

Unit 4

Methods on non struct types

- Methods and non struct types must be in the same package.
- Following add method has built in type int, and it is not in the same package
- The program will throw compilation error : cannot define new methods on non-local type int

```
package main

func (a int) add(b int) {
}

func main() {

}
```

- Create a type alias for the built-in type int
  ```
  type myInt int
  ```
- Create a method with this type alias as the receiver.
  ```
  func (a myInt) add(b myInt) myInt { return a + b }
  ```

```
package main // Example 4-13
import "fmt"

type myInt int // create a type alias myInt for int

func (a myInt) add(b myInt) myInt {
   return a + b
}


func main() {
   num1 := myInt(5)
   num2 := myInt(10)
   sum := num1.add(num2)
   fmt.Println("Sum is", sum)
}
```

Output:

```
Sum is 15
```

Unit 5

Linked list Data structure

- Data structures and algorithms are the bread and butter of computer science.
- Linked lists are one of the simpler data structures
- a linked list is a linear collection of data elements, in which linear order is not given by their physical placement in memory.
- A linear data structure is the one where it's elements form a sequence of some sort.
- In link list each element points to the next element.
- It is a data structure consisting of a group of nodes which together represent a sequence.
- Each node is composed of data and a pointer that holds the address of the next node in the sequence.

Using new Function To Create Link List

- The built-in new(T) function allocates “zeroed” storage for a new item of type T.
- After allocation of storage it returns its address, a value of type *T
- The new function allows to create each node that is linked to the next node to create a list
- The new function always allocates memory for each node from the heap.

```
package main // Example 4-14
import "fmt"

type node struct {
   data int
   next *node
}

func main() {
   //currPtr := &node{data: 1, next:nil}
   //currPtr := &node{}

   currPtr := new(node)
   tailPtr := currPtr
   headPtr := currPtr
   currPtr.data = 1
   currPtr.next = nil

   tailPtr.next = new(node)
   //tailPtr.next = &node{}
   currPtr = tailPtr.next
   currPtr.data = 2
   currPtr.next = nil
   tailPtr = currPtr

   tailPtr.next = new(node)
   //tailPtr.next = &node{}
   currPtr = tailPtr.next
   currPtr.data = 3
   currPtr.next = nil
   tailPtr = currPtr
```

```go
    currPtr = headPtr
    for currPtr != nil {
       deleteThisNode := currPtr
       fmt.Println("Data: ", currPtr.data)
       fmt.Println(deleteThisNode, currPtr)
       currPtr = currPtr.next
       fmt.Println(deleteThisNode, currPtr)
       // deleteThisNode.next = nil
       deleteThisNode = nil
       fmt.Println(deleteThisNode, currPtr)
    }

    fmt.Println("Current", currPtr, headPtr, tailPtr)

}
```

Output:

```
Data:  1
&{1 0xc0000101f0} &{1 0xc0000101f0}
&{1 0xc0000101f0} &{2 0xc000010200}
<nil> &{2 0xc000010200}
Data:  2
&{2 0xc000010200} &{2 0xc000010200}
&{2 0xc000010200} &{3 <nil>}
<nil> &{3 <nil>}
Data:  3
&{3 <nil>} &{3 <nil>}
&{3 <nil>} <nil>
<nil> <nil>
Current <nil> &{1 0xc0000101f0} &{3 <nil>}
```

Create Link List using Method

```
package main // Example 4-15
import "fmt"

// A Node contains data and a link to the next node.
// The ‘next’ field is same type as the struct, which is legal
// because it's a pointer. Otherwise it'd give an error about
// "invalid recursive type Node".
type Node struct {
    data int
    next *Node
}

type List struct {
    head *Node
}

func (l *List) Append(newNode *Node) {
    if l.head == nil {
        l.head = newNode
        return
    }

    currentNode := l.head
    for currentNode.next != nil {
        currentNode = currentNode.next
    }
    currentNode.next = newNode
}

```

```go
func main() {
    l := &List{}
    l.Append(&Node{data: 10})
    l.Append(&Node{data: 20})
    l.Append(&Node{data: 30})

    fmt.Printf("first=%+v\n", l.head)
    fmt.Printf("second=%+v\n", l.head.next)
    fmt.Printf("third=%+v\n\n", l.head.next.next)

    // Better yet, loop through the list
    // instead of manually chaining .next's
    for e := l.head; e != nil; e = e.next {
        fmt.Printf("e=%+v\n", e)
    }
}

```

Output:

```
first=&{data:10 next:0xc0000101f0}
second=&{data:20 next:0xc000010200}
third=&{data:30 next:<nil>}

e=&{data:10 next:0xc0000101f0}
e=&{data:20 next:0xc000010200}
e=&{data:30 next:<nil>}
```

Unit 6

Link List Container in Golang

- Go uses the container/list package to support link list.

```go
package main // Example 4-16

import (
	"container/list"
	"fmt"
)

func main() {
	// create a new link list
	alist := list.New()

	fmt.Println("Size before : ", alist.Len())

	// push element into list
	alist.PushBack("a")
	alist.PushBack("b")
	alist.PushBack("c")
	// list size after
	fmt.Println("Size after insert(push): ", alist.Len())

	// list elements
	for e := alist.Front(); e != nil; e = e.Next() {
		fmt.Println(e.Value.(string))
	}

	// pop 3 elements
	alist.Remove(alist.Front())
	alist.Remove(alist.Front())
	alist.Remove(alist.Front())
	// list size after
	fmt.Println("Size after remove(pop) : ", alist.Len())

}
```

Output:

```
Size before :  0
Size after insert(push):  3
a
b
c
Size after remove(pop) :  0
```

Unit 7

LIFO (Stack)

- LIFO (Last In First Out) uses stack data structure.
- A stack is a container of objects that are inserted and removed according to the last-in first-out (LIFO) principle.
- In the pushdown stacks two operations are allowed, push and pop.
- push the item into the stack, and pop the item out of the stack.
- non-struct type receiver must be present in the same package as method definitions are present.

```go
package main // Example 4-17
import "fmt"

type stack []int // non-struct type definition

func (s stack) empty() bool { return len(s) == 0 }
func (s stack) Peek() int   { return s[len(s)-1] }
func (s *stack) push(i int) { (*s) = append((*s), i) }

func (s *stack) pop() int {
   d := (*s)[len(*s)-1]
   (*s) = (*s)[:len(*s)-1]
   return d
}

func main() {
   var s stack

   fmt.Println("Push Items")
   for i := 0; i < 3; i++ {
      s.push(i + 100)
      fmt.Printf("len=%d peek=%d\n", len(s), s.Peek())
   }

   fmt.Println("\nPop Items")
   for !s.empty() {
      i := s.pop()
      fmt.Printf("len=%d pop=%d\n", len(s), i)
   }

   // Example of Slice
   var myslice []int
   myslice = append(myslice, 8, 10, 19, 13)
fmt.Printf("\nlen = %d cap=%d value=%v\n",len(myslice),cap(myslice), myslice)
}
```

Output:

```
Push Items
len=1 peek=100
len=2 peek=101
len=3 peek=102

Pop Items
len=2 pop=102
len=1 pop=101
len=0 pop=100

len = 4 cap=4 value=[8 10 19 13]
```

Create Stack Of Strings Using Slice

```go
package main  // Example 4-18
import "fmt"

type Stack []string // non-struct type definition

// IsEmpty: check if stack is empty
func (s *Stack) IsEmpty() bool {
   return len(*s) == 0
}

// Push a new value onto the stack
func (s *Stack) Push(str string) {
   *s = append(*s, str) // Simply append the new value to the end of the stack
}

// Remove and return top element of stack. Return false if stack is empty.
func (s *Stack) Pop() (string, bool) {
   if s.IsEmpty() {
      return "", false
   } else {
      index := len(*s) - 1   // Get the index of the top most element.
      element := (*s)[index] // Index into the slice and obtain the element.
      *s = (*s)[:index]      // Remove it from the stack by slicing it off.
      return element, true
   }
}

func main() {
   var stack Stack // create a stack variable of type Stack

   stack.Push("golang")
   stack.Push("with")
   stack.Push("fun")

   for len(stack) > 0 {
      x, y := stack.Pop()
      if y == true {
         fmt.Println(x)
      }
   }
}
```

```
Output:
        fun
        with
        golang
```