User defined types

Objectives

- Understand
  - Structs
  - Self referential structs
  - Recursion
  - Struct embedding
  - Encapsulation

Structs

- A struct is an aggregate data type that groups together zero or more named values of arbitrary types as a single entity
  - Each value is called a field
- Fields are usually written one per line
  - Field-name Type
  - Consecutive fields of the same type may be combined
    - ID, Code int32
  - The name of a struct field is exported if it begins with a capital letter
    - A struct type may contain a mixture of exported and unexported fields
- Struct types usually appear within the declaration of a named type like Moto
- If all the fields of a struct are comparable, the struct itself is comparable
  - using == or !=
- A value of a struct type can be written using a struct literal
  - type Point struct{ X, Y int }
    p1 := Point{1, 2} //Fields init in order
    p2 := Point{Y: 2} //X retains 0 value

```
1   package main
2
3   import "fmt"
4
5   func main() {
6       type Moto struct {
7           ID    int64
8           Make  string
9           Model string
10          Price float32
11      }
12      var x Moto
13      x.ID = 12
14      x.Make = "Honda"
15      x.Model = "VFR750"
16      x.Price = 2310.0
17      fmt.Println(x)
18  }
19
```

```
PROBLEMS    OUTPUT    DEBUG CONSOLE
2017/04/09 23:00:50 server.go:73:
2017/04/09 23:00:50 debugger.go:68
API server listening at: 127.0.0.1
2017/04/09 23:00:50 debugger.go:41
{12 Honda VFR750 2310}
```

Self referential structs

- A named struct type S can't declare a field of type S
  - S may declare a field of the pointer type *S
  - This allows for recursive data structures like linked lists and trees
- Go is a call by value language
  - Pointers must be passed to modify most external objects

```
PROBLEMS  OUTPUT  DEBUG CONSOLE  TERMINAL
2017/04/09 23:27:13 server.go:73: Using API v1
2017/04/09 23:27:13 debugger.go:68: launching pro
API server listening at: 127.0.0.1:2345
2017/04/09 23:27:13 debugger.go:414: continuing
{100 0xc04204a440 0xc04204a420}
5
55
100
135
178
```

```go
package main

import "fmt"

type node struct {
    id          int64
    left, right *node
}

func insert(root *node, i int64) *node {
    if nil == root {
        return &node{i, nil, nil}
    }
    if i < root.id {
        root.left = insert(root.left, i)
    } else if i > root.id {
        root.right = insert(root.right, i)
    }
    return root
}

func treePrint(root *node) {
    if nil != root.left {
        treePrint(root.left)
    }
    fmt.Println(root.id)
    if nil != root.right {
        treePrint(root.right)
    }
}

func main() {
    root := node{100, nil, nil}
    insert(&root, 178)
    insert(&root, 5)
    insert(&root, 55)
    insert(&root, 135)
    fmt.Println(root)
    treePrint(&root)
}
```

Struct embedding

- Struct embedding allows one named struct type be used as an anonymous field of another struct type
  - Syntactic shortcut where x.f can stand for x.d.e.f
  - Declare a field with a type but no name
    - Called anonymous fields
      - The fields actually have the name of the type but that name can be omitted
    - The type of the field must be a named type or a pointer to a named type
      - type Point struct {X, Y int }
type Circle struct { Point; Radius int }
type Wheel struct { Circle; Spokes int }
      - var w Wheel w.X = 8 //equivalent to w.Circle.Point.X = 8
- Literals follow the shape of the type:
  - w = Wheel{ Circle{ Point{ 8, 8}, 5}, 20}
- Anonymous fields have implicit names so you can't have two anonymous fields of the same type
- Because the name of the field is implicitly determined by its type, so too is the visibility of the field

JSON

- Go supports several encoding libraries
  - encoding/json
  - encoding/xml
- JavaScript Object Notation (JSON) is a standard notation for sending and receiving structured information
- The basic JSON types are
  - numbers (in decimal or scientific notation)
  - booleans (true or false)
  - strings
- These basic types may be combined recursively using JSON arrays and objects
  - JSON array is an ordered sequence of values, written as a comma-separated list enclosed in square brackets
    - JSON arrays are used to encode Go arrays and slices
  - JSON object is a mapping from strings to values, written as a sequence of name:value pairs separated by commas and surrounded by braces
    - JSON objects are used to encode Go maps (with string keys) and structs
- The json.Marshall() method accepts a Go object and returns its JSON representation
  - MarshalIndent() provides human friendly formatting
  - Only exported fields are marshaled

```go
package main

import "fmt"
import "encoding/json"

type Moto struct {
    ID          int64
    Make, Model string
}

func main() {
    b1 := Moto{34, "Honda", "VFR750"}
    data, _ := json.Marshal(b1)
    fmt.Printf("%s\n", data)
    data, _ = json.MarshalIndent(b1, "", "    ")
    fmt.Printf("%s\n", data)
}
```

```
PROBLEMS  OUTPUT  DEBUG CONSOLE  TERMINAL
2017/04/10 00:18:40 server.go:73: Using API v1
2017/04/10 00:18:40 debugger.go:68: launching process w
API server listening at: 127.0.0.1:2345
2017/04/10 00:18:40 debugger.go:414: continuing
{"ID":34,"Make":"Honda","Model":"VFR750"}
{
    "ID": 34,
    "Make": "Honda",
    "Model": "VFR750"
}
```

JSON Field Tags

- A field tag is a string of metadata associated with the field of a struct
  - Year int `json:"released"`
- A field tag is conventionally interpreted as a space-separated list of key:"value" pairs
  - since they contain double quotes they are usually written with raw string literals.
- The first part of the JSON field tag specifies an alternative JSON name for the Go field
  - Like total_count for a Go field named TotalCount
- Additional option, omitempty, which indicates that no JSON output should be produced if the field has the zero value
  - Comma separated from the alt field name but within the double quotes

```go
package main

import "fmt"
import "encoding/json"

type Moto struct {
    ID          int64 `json:"model_no"`
    Make, Model string
}

func main() {
    b1 := Moto{34, "Honda", "VFR750"}
    data, _ := json.Marshal(b1)
    fmt.Printf("%s\n", data)
    data, _ = json.MarshalIndent(b1, "", "    ")
    fmt.Printf("%s\n", data)
}
```

```
PROBLEMS  OUTPUT  DEBUG CONSOLE  TERMINAL
2017/04/10 00:30:50 server.go:73: Using API v1
2017/04/10 00:30:50 debugger.go:68: launching process
API server listening at: 127.0.0.1:2345
2017/04/10 00:30:50 debugger.go:414: continuing
{"model_no":34,"Make":"Honda","Model":"VFR750"}
{
    "model_no": 34,
    "Make": "Honda",
    "Model": "VFR750"
}
```

Unmarshalling

- JSON text can be unmarshalled back into Go structures
  - json.Unmarshal()
  - Unmarshal takes the text to unmarshal and a point to the struct to unmarshal into
  - If the operation fails an error is returned
- The unmarshal operation will populate the elements of the Go struct that match JSON fields
  - All fields need not be present
  - Unmatched fields retain their zero value
  - Fields to unmarshal to must have initial caps (json will not unmarshall internal fields)
  - JSON field matching is case insensitive

```go
package main

import "fmt"
import "encoding/json"

type Moto struct {
    ID          int64
    Make, Model string
}

func main() {
    b1 := Moto{34, "Honda", "VFR750"}
    data, _ := json.Marshal(b1)
    fmt.Printf("%s\n", data)
    var modelInfo struct {
        ID    int64
        Model string
    }
    json.Unmarshal(data, &modelInfo)
    fmt.Println(modelInfo)
}
```

```
PROBLEMS  OUTPUT  DEBUG CONSOLE  TERMINAL
2017/04/10 00:37:56 server.go:73: Using API v1
2017/04/10 00:37:56 debugger.go:68: launching p
API server listening at: 127.0.0.1:2345
2017/04/10 00:37:57 debugger.go:414: continuing
{"ID":34,"Make":"Honda","Model":"VFR750"}
{34 VFR750}
```

Templates

- Go offers several packages which provide a mechanism for substituting the values of variables into a template
  - text/template
  - html/template
- A template is a string or file containing one or more portions enclosed in double braces: {{...}}
  - Called *actions*
  - Each action contains an expression in the template language
    - printing values
    - selecting struct fields
    - calling functions
    - expressing control flow (if-else/range loops)
    - Even instantiating other templates
- https://golang.org/pkg/text/template/

```
1  package main
2
3  import (
4      "os"
5      "text/template"
6  )
7
8  type Person struct {
9      Name string
10 }
11
12 func main() {
13     t := template.New("hello template")
14     t, _ = t.Parse("hello {{.Name}}!")
15
16     p := Person{Name: "world"}
17
18     t.Execute(os.Stdout, p)
19 }
20
```

```
PROBLEMS  OUTPUT  DEBUG CONSOLE  TERMINAL
2017/04/10 00:55:48 server.go:73: Using API v1
2017/04/10 00:55:48 debugger.go:68: launching pro
API server listening at: 127.0.0.1:2345
2017/04/10 00:55:48 debugger.go:414: continuing
hello world!
```

Summary

- Go includes a range of features for creating UDTs
  - Structs
  - Self referential structs
  - Recursion
  - Struct embedding
  - Encapsulation

Lab: User Defined Types

- Working with structs