Introduction to Programming in Go

8. Types and Structs

Module Topics

1. User Defined Types
2. Defining Structs
3. Working with Structs
4. Embedding Structs

User Defined Types

User Defined Types

1. We can create new types by using the type keyword.
2. Aliases allow us to create "clones" of existing types.
3. We can create complex data types using type.
4. Structs in Go work much like structs in C and C++.
5. Structs are part of how we implement OO in Go.

Aliases

1. Aliases allow us to create a copy of a type. For example, Go defines:

```
type rune int32
type byte uint8
```

2 An alias belongs to the package that it is defined in.

3. Aliases are their own type, not just an alternate name.

4. Useful when we define methods in the next module .

Defining Structs

Defining Structs

1. Structs are defined using the following syntax:

```
type sname struct {
   field1 type1
   field2 type2
   ...
   }
```

2. Structs are types so variables can be defined to be the type corresponding to a struct definition:

```
   var v1 sname
   var v2 sname
```

Creating Structs

```
// Example 08-01 Basic struct definition
...

type point struct{ x, y int }
type employee struct {
   fname, lname string
   id           int
   job          string
   salary       int           }

func main() {
   var anil employee
   var p point
   fmt.Println("anil=", anil, "p=", p)
}
```

```
[Module058$ go run ex08-01.go
anil= {  0  0} p= {0 0}
```

Initializing Structs

1. We can use literals to initialize structs:

```
var anil = employee {“Anil”, “Patel”, 8971,
   “Developer”, 100000}

p := point{4,2}
```

2. Selected fields may be initialized:

```
var anil = employee{id: 8971, fname: “Anil”,
                    lname: “Patel”}
```

3. If field names are provided, the order does not matter.

4. Any field not named in the initial list is set to its default zero value.

Initialization of Structs

```
// Example 08-02 Struct initialization
...
type point struct{ x, y int }
type employee struct { ... }

func main() {
   var p = point{2, 3}
   fmt.Println("p =", p)

   anil := employee{"Anil", "Patel", 9891,
                    "Developer", 100000}
   greta := employee{id: 8897, fname: "Greta",
                                 lname: "Smith"}

   fmt.Println("anil =", anil)
   fmt.Println("greta =", greta)
}
```

```
[Module08]$ go run ex08-02.go
p = {2 3}
anil = {Anil Patel 9891 Developer 100000}
grea = {Greta Smith 8897  0}
```

Using New

1. new() is used to allocate memory for a struct.

2. new() returns a pointer to the newly created object.

```
var greta *employee = new(employee)
```

3. The underlying object is accessed with the de-referencing operator *.

4 Short way for calling the new operator and then initializing the fields of the new point object:

```
p := &point{4,5}
```

Using New

For Clarity:

```
bob := employee{0,0}
```
bob is an object

```
ptr_to_bob := &bob
```
ptr_to_bob is the address of bob

```
ptr_to_sue := &employee{fname: "Sue"}
```

```
*(ptr_to_sue)
```
is the sue object pointed to by ptr_to_sue

Using New

```
// Example 08-03 Using new
...
type point struct{ x, y int }

func main() {

   var pp1 *point = new(point) // pp1 is a pointer
   fmt.Println("pp1 =", pp1, "*pp=", *pp1)

   p := point{3, 4} // p is a variable
   pp := &p         // pp is a pointer
   fmt.Println("pp =", pp, "*pp=", *pp, "p=", p)

   pp2 := &point{5, 6} //pp2 is a pointer
   fmt.Println("pp2 =", pp2, "*pp2=", *pp2)
}
```

```
[Module08]$ go run ex08-03.go
pp1 = &{0 0} *pp= {0 0}
pp = &{3 4} *pp= {3 4} p= {3 4}
pp2 = &{5 6} *pp2= {5 6}
```

Working with Structs

Accessing Fields

```
// Example 08-04 Field access
...

type point struct{ x, y int }

func main() {
   p := point{2, 3}    // object
   pp := &point{4, 5}    // pointer
   fmt.Println("x coord of p", p.x)
   fmt.Println("x coord of pp", pp.x)
   pp.y = -1
   p.y = 0
   fmt.Println("p=", p, " pp=", *pp)
}
```

```
[Module08]$ go run ex08-04.go
x coord of p 2
x coord of pp 4
p= {2 0}  pp= {4 -1}
```

Comparing Structs

1. The operator == is defined on a struct if all the fields
   in the struct have the == operator defined for their type.

2. Two structs are equivalent if they are the same type and
   each corresponding field in the two structs is also equivalent.

3. Care has to be taken when working with pointers

4. If pp1 and pp2 are pointers to the same type of struct:

   pp1 == pp2 means "Are we both pointing to the same object?"

   *pp1 == *pp2 means "Are the objects we are pointing to equivalent?"

Struct Comparisons

```
// Example 08-05 Comparisons
...
type point struct{ x, y int }

func main() {
   p1 := point{2, 3}
   p2 := point{4, 5}
   p3 := point{2, 3}
   fmt.Println("p1 == p2? ", p1 == p2)
   fmt.Println("p1 == p3? ", p1 == p3)
   pp1 := &point{1, 1}
   pp2 := &point{1, 1}
   fmt.Println("pp1 == pp2? ", pp1 == pp2)
   fmt.Println("*pp1 == *pp? ", *pp1 == *pp2)
}
```

```
[Module08]$ go run ex08-05.go
p1 == p2?  false
p1 == p3?  true
pp1 == pp2?  false
*pp1 == *pp?  true
```

Struct Pointers

1. Structs are value types just like arrays are.
2. We use slices to make passing arrays around more effective.
3. We use pointers when using structs for the same reason.
4. It is more efficient to pass pointers to structs and not the struct object.

Struct Pointers as Parameters

```
// Example 08-06 Struct pointers
...
type point struct{ x, y int }

func swap1(p point) {
   p.x, p.y = p.y, p.x
   fmt.Println("After executing swap1 p=", p)
}
func swap2(p *point) {
   p.x, p.y = p.y, p.x
}
func main() {
   a := point{1, 2}
   fmt.Println("Original a =", a)
   swap1(a)
   fmt.Println("After swap1 a =", a)
   swap2(&a)
   fmt.Println("After swap2 a =", a)
}
```

```
[Module08]$ go run ex08-06.go
Original a = {1 2}
After executing swap1 p= {2 1}
After swap1 a = {1 2}
After swap2 a = {2 1}
```

Embedding Structs

Embedding Structs

```
// Example 08-07 Embedded structs 1
...
type point struct{ x, y int }

type circle struct {
   center point
   radius float32
}

func main() {
   c := circle{point{50, 32}, 13.0}
   fmt.Println("c=", c)
   c.center.x = 0
   fmt.Println("c=", c)
}
```

```
[Module08]$ go run ex08-07.go
c= {{50 32} 13}
c= {{0 32} 13}
```

Anonymous Fields

1. An anonymous field is one that only has a type but no name.

2. Fields in the inner struct act as if they are fields in the outer struct.

3. There cannot be two anonymous fields of the same type otherwise there would be no way to tell them apart.

4. Names in the outer struct shadow names in the inner struct

Anonymous Fields

```
// Example 08-08 Anonymous fields
...
type point struct{ x, y int }
type circle struct {
   point
   bool
   radius float32
}

func main() {
   c := circle{point{50, 32}, false, 3.0}
   fmt.Println("c=", c)
   c.x = 0
   c.bool = true
   fmt.Println("c=", c)
}
```

```
[Module08]$ go run ex08-08.go
c= {{50 32} false 3}
c= {{0 32} true 3}
```

Pointers as fields

```
// Example 08-09 Pointers as fields
...
type employee struct {
   fname, lname string
   id           int
   job          string
   salary       int
   boss         *employee
}

func main() {
   greta := employee{fname: "Greta", lname: "Smith"}
   anil := employee{fname: "Greta", lname: "Smith",
                                          boss: &greta}
   fmt.Println("Anil's boss is", anil.boss.fname)
}
```

```
[Module08]$ go run ex08-09.go
Anil's boss is Greta
```

Pseudo-Constructors

```
// Example 08-10 Struct factory
...
type point struct{ x, y int }

func makePoint(x, y int) *point {
   if x < 0 {
      x = -x
   }
   if y < 0  {
      y = -y
   }
   return &point{x, y}
}

func main() {
   p1 := makePoint(1, 1)
   fmt.Println(*p1)
   p1 = makePoint(-4, -9)
   fmt.Println(*p1)
}
```

```
[Module08]$ go run ex08-10.go
{1 1}
{4 9}
```

Lab 8: Structs