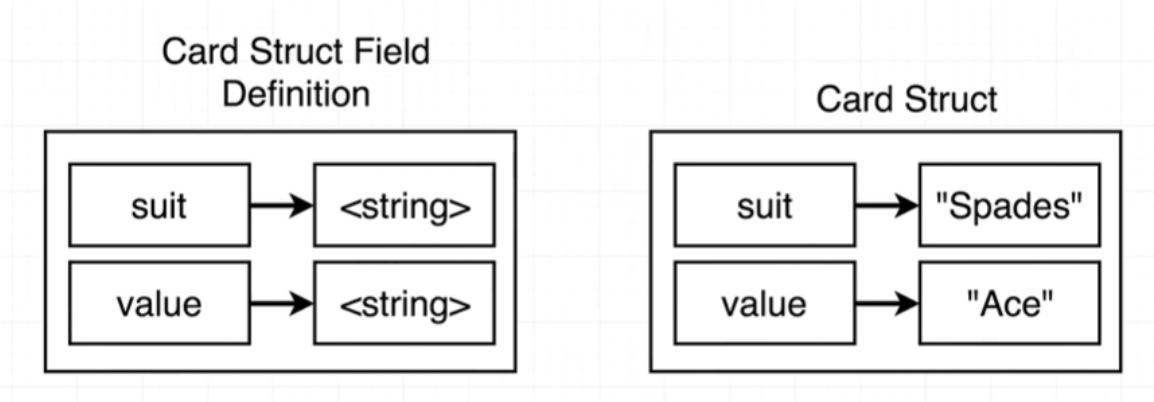
Organizing Data with Structs

Created	@Feb 20, 2021 11:55 PM
Tags	

Structs in Go

- Structs are a data structure defined by a collection of properties that are related together



- This code defines a **struct** called **person** with two `string` properties *firstName* and *lastName*
- A variable **alex** is then declared
  - "Alex" is applied to *firstName*
  - "Anderson" is applied to *lastName*

```
type person struct {
  firstName string
  lastName string
}

func main() {
  alex := person{"Alex", "Anderson"}
}
```

- This syntax is **not** ideal
- It's better way to define **alex** is with this explicit syntax

```
func main() {
  alex := person{firstName:"Alex", lastName: "Anderson"}
}
```

- Individual fields can be assigned/updated in this way:

```
alex.firstName = "Alex"
alex.lastName = "Anderson"
```


A struct's field names and values can be printed with `fmt.Printf("%+v", alex)`

- If any fields in a struct are not defined, they will be defaulted to **zero values** based on type
  - `string` | `""`
  - `int` | `0`
  - `float` | `0.0`
  - `bool` | `false`

Type	Zero Value
string	""
int	0
float	0
bool	false

Embedding Structs

- Structs can be nested within each other like so:
- When declaring multi-line structs every line **must have a comma**, even if it's the last property
- In the below code a field is still declared of type **contactInfo** but its name is also called *contactInfo*

```
type person struct {
  firstName string
  lastName  string
  contactInfo
}
```

```
type contactInfo struct {
  email   string
  zipCode int
}

type person struct {
  firstName string
  lastName  string
  contact   contactInfo
}

func main() {
  jim := person{
    firstName: "Jim",
    lastName:  "Party",
    contact: contactInfo{
      email:   "jim@email.com",
      zipCode: 80210,
    },
  }
}
```

Receiver Functions / Pass by Value

- Receiver functions can be set up with structs the same way as custom types
- This `print()` function can be called by any **person** like so:
  - `jim.print()`

```
func (p person) print() {
  fmt.Printf("%+v", p)
}
```

- Calling `jim.updateName("jimmy")` will **not** change the *firstName* field of variable **jim** because the variable passes a value not a pointer

```
func (p person) updateName(newFirstName string) {
  p.firstName = newFirstName
}
```


Go is a pass by value language

Pointers

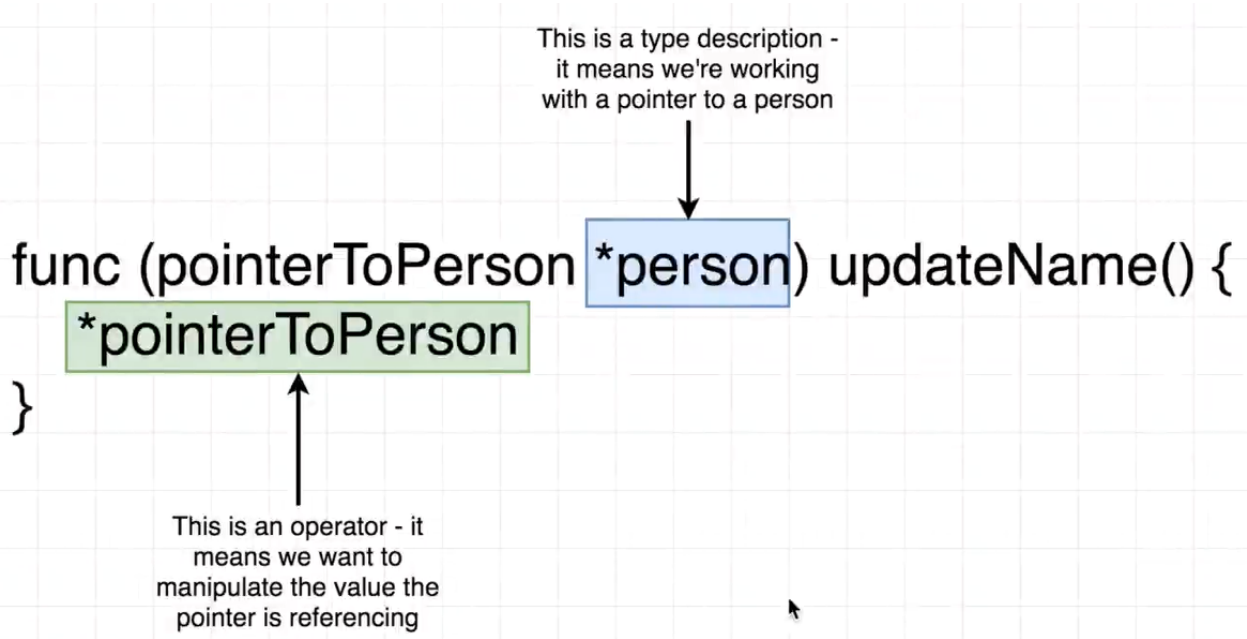
- This code uses a **pointer** to update the value of **jim**

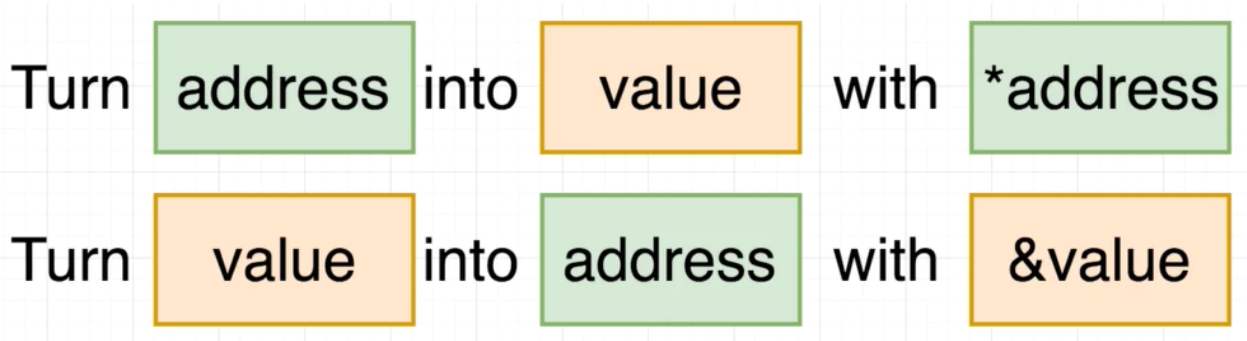
```
jimPointer := &jim
jimPointer.updateName("jimmy")
jim.print()

func (pointerToPerson *person) updateName (newFirstName string) {
  (*pointerToPerson).firstName = newFirstName
}
```

- The `&` operator in `&variable` yields the memory address of the value the variable is pointing to
- The `*` operator in `*pointer` yields the value at the memory address being pointed to

- When you see a `*` where a ***type*** should be, it's a type description - saying the type is a pointer





Pointer Shortcut

- Rather than writing the code on the right, there is a shortcut where if the function takes a **pointer** as a parameter and it is passed a value, it will instead pass by reference

```
jimPointer := &jim
jimPointer.updateName("jimmy")
jim.print()
```

- All you need is `jim.updateName("jimmy")` as long as the function is defined properly with a pointer

Value Types and Reference Types

- When working with **slices** go acts with pass by reference
- Slices are technically still copied, but a slice is really comprised of an array, and a structure that records the length of the slice, the capacity of the slice, and a reference to said underlying array.
- Slices are a **Reference Type**