

# Initializing

This lesson discusses initializing variables using the new expression in Go

## WE'LL COVER THE FOLLOWING ^

- Using the **new** Expression
- Resources

## Using the **new** Expression #

Now that we know the different types variables can take, we will look into initializing variables. Go supports the **new** expression to allocate a zeroed value of the requested type and to return a pointer to it.

```
x := new(int)
```



As seen in [Section: Structs](#) a common way to “initialize” a variable containing a struct or a reference to one, is to create a struct literal. Another option is to create a constructor. This is usually done when the zero value isn’t good enough and you need to set some default field values for instance.

Note that following expressions using **new** and an empty struct literal are equivalent and result in the same kind of allocation/initialization:

### Environment Variables ^

Key:	Value:
GOPATH	/go

```
package main
```

```
import (  
    "fmt"  
)
```

```
type Bootcamp struct {
```



```
type Bootcamp struct {  
    Lat float64  
    Lon float64  
}  
  
func main() {  
    x := new(Bootcamp)  
    y := &Bootcamp{}  
    fmt.Println(*x == *y)  
}
```



Note that [slices](#), [maps](#) and [channels](#) are usually allocated using `make` so the data structure these types are built upon can be initialized.

## Resources #

- [Allocation with `new` - effective Go](#)
- [Composite Literals - effective Go](#)
- [Allocation with `make` - effective Go](#)

Now that we have gone over initializing variables in Go, we will look at how we can use this knowledge to implement composition, the alternative to inheritance in Go.