# Tips on Constants in Go

Useful tips and tricks for programming in Go.

WE'LL COVER THE FOLLOWING



## Iota: Elegant Constants #

Some concepts have names, and sometimes we care about those names, even (or especially) in our code.

```
const (
    CCVisa            = "Visa"
    CCMasterCard      = "MasterCard"
    CCAmericanExpress = "American Express"
)
```

At other times, we only care to distinguish one thing from the other. There are times when there's no inherently meaningful value for a thing. For example, if we're storing products in a database table we probably don't want to store their category as a string. We don't care how the categories are named, and besides, marketing changes the names all the time.

We care only that they're distinct from each other.

```
const (
    CategoryBooks    = 0
    CategoryHealth   = 1
```

```
    CategoryClothing = 2
)
```

Instead of 0, 1, and 2 we could have chosen 17, 43, and 61. The values are arbitrary.

Constants are important but they can be hard to reason about and difficult to maintain. In some languages like Ruby developers often just avoid them. In Go, constants have many interesting subtleties that, when used well, can make the code both elegant and maintainable.

## Auto Increment #

A handy idiom for this in golang is to use the iota identifier, which simplifies constant definitions that use incrementing numbers, giving the categories exactly the same values as above.

```
const (
    CategoryBooks = iota // 0
    CategoryHealth       // 1
    CategoryClothing     // 2
)
```

## Custom Types #

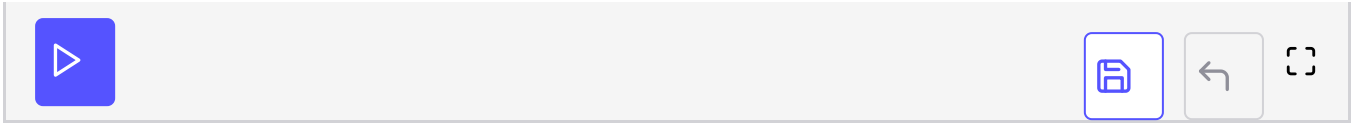
Auto-incrementing constants are often combined with a custom type, allowing you to lean on the compiler.

```
type Stereotype int

const (
    TypicalNoob Stereotype = iota // 0
    TypicalHipster                // 1
    TypicalUnixWizard             // 2
    TypicalStartupFounder         // 3
)
```

If a function is defined to take an int as an argument rather than a Stereotype, it will blow up at compile-time if you pass it a Stereotype:

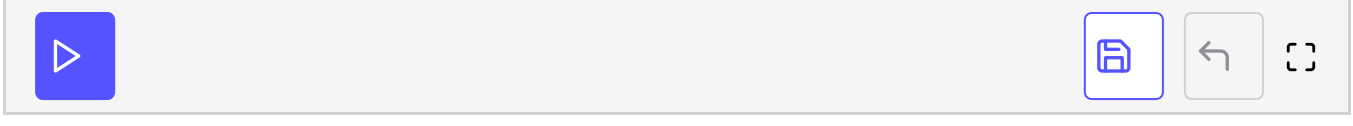
```
func CountAllTheThings(i int) string {
                return fmt.Sprintf("there are %d things", i)
}

func main() {
    n := TypicalHipster
```

```
    fmt.Println(CountAllTheThings(n)) // cannot use TypicalHipster (type Stereotype) as type
}
```



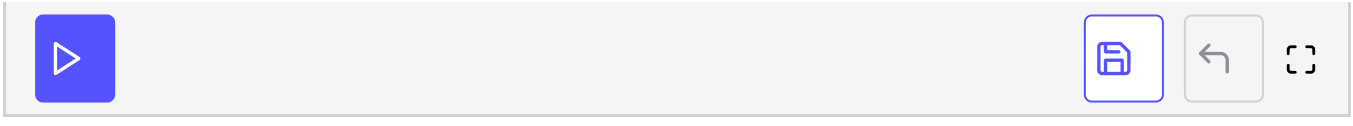
The inverse is also true. Given a function that takes a Stereotype as an argument, you **can't** pass it an int:

```
func SoSayethThe(character Stereotype) string {
    var s string
    switch character {
    case TypicalNoob:
        s = "I'm a confused ninja rockstar."
    case TypicalHipster:
        s = "Everything was better we programmed uphill and barefoot in the snow on the SUTX
    case TypicalUnixWizard:
        s = "sudo grep awk sed %!#?!!1!"
    case TypicalStartupFounder:
        s = "exploit compelling convergence to syndicate geo-targeted solutions"
    }
    return s
}

func main() {
    i := 2
    fmt.Println(SoSayethThe(i)) // cannot use i (type int) as type Stereotype in argument to
}
```



There's a dramatic twist, however. You could pass a number constant, and it would work:

```
func main() {
    fmt.Println(SoSayethThe(0))
}

// output:
// I'm a confused ninja rockstar.
```



This is because constants in Go are loosely typed until they are used in a strict context.

## Skipping Values #

Imagine that you're dealing with consumer audio output. The audio might not have any output whatsoever, or it could be mono, stereo, or surround.

There's some underlying logic to defining no output as 0, mono as 1, and stereo as 2, where the value is the number of channels provided.

So what value do you give Dolby 5.1 surround?

On the one hand, it's 6 channel output, but on the other hand, only 5 of those channels are full bandwidth channels (hence the 5.1 designation – with the .1 referring to the low-frequency effects channel).

Either way, we don't want to simply increment to 3.

We can use underscores to skip past the unwanted values.

```
type AudioOutput int

const (
    OutMute AudioOutput = iota // 0
    OutMono                    // 1
    OutStereo                  // 2
    _
    _
    OutSurround                // 5
)
```

## Expressions #

The iota can do more than just increment. Or rather, iota always increments, but it can be used in expressions, storing the resulting value in the constant.

Here we're creating constants to be used as a bitmask.

```
type Allergen int

const (
    IgEggs Allergen = 1 << iota // 1 << 0 which is 00000001
    IgChocolate                 // 1 << 1 which is 00000010
    IgNuts                      // 1 << 2 which is 00000100
    IgStrawberries              // 1 << 3 which is 00001000
    IgShellfish                 // 1 << 4 which is 00010000
)
```

This works because when you have only an identifier on a line in a const group, it will take the previous expression and reapply it, with the incremented iota. In the language of the spec, this is called *implicit repetition*

*of the last non-empty expression list*.

If you're allergic to eggs, chocolate, and shellfish, and flip those bits to the "on" position (mapping the bits right to left), then you get a bit value of 00010011, which corresponds to 19 in decimal.

```
fmt.Println(IgEggs | IgChocolate | IgShellfish)

// output:
// 19
```

There's a great example in Effective Go for defining orders of magnitude:

```
type ByteSize float64

const (
    _           = iota              // ignore first value by assigning to blank identifier
    KB ByteSize = 1 << (10 * iota)  // 1 << (10*1)
    MB                              // 1 << (10*2)
    GB                              // 1 << (10*3)
    TB                              // 1 << (10*4)
    PB                              // 1 << (10*5)
    EB                              // 1 << (10*6)
    ZB                              // 1 << (10*7)
    YB                              // 1 << (10*8)
)
```

Today I learned that after zettabyte we get yottabyte. #TIL

**But Wait, There's More**

What happens if you define two constants on the same line? What is the value of Banana? 2 or 3? And what about Durian?

```
const (
    Apple, Banana = iota + 1, iota + 2
    Cherimoya, Durian
    Elderberry, Fig
)
```

The iota increments on the next line, rather than as soon as it gets referenced.

```
// Apple: 1
// Banana: 2
// Cherimoya: 2
// Durian: 3
// Elderberry: 3
```

```
// Fig: 4
```

Which is messed up, because now you have constants with the same value.

**So, Yeah**

There's a lot more to be said about constants in Go, and you should probably read Rob Pike's blog post on the subject over on the golang blog.

*This was first published by Katrina Owen on the Splice blog.*

## Web resources #

- Dave Cheney maintains a list of resources for new Go developers.