Constant Time | Dave Cheney

by Dave Cheney

This essay is a derived from my dotGo 2019 presentation about my favourite feature in Go.

Many years ago Rob Pike remarked,

> "Numbers are just numbers, you'll never see `0x80ULL` in a `.go` source file".
>
> —*Rob Pike, The Go Programming Language*

Beyond this pithy observation lies the fascinating world of Go's constants. Something that is perhaps taken for granted because, as Rob noted, is Go numbers–constants–just work.
In this post I intend to show you a few things that perhaps you didn't know about Go's `const` keyword.

What's so great about constants?

To kick things off, why are constants good? Three things spring to mind:

Immutability. Constants are one of the few ways we have in Go to express immutability to the compiler.

Clarity. Constants give us a way to extract magic numbers from our code, giving them names and semantic meaning.

Performance. The ability to express to the compiler that something will not change is key as it unlocks optimisations such as constant folding, constant propagation, branch and dead code elimination.

But these are generic use cases for constants, they apply to any language. Let's talk about some of the properties of Go's constants.

A Challenge

To introduce the power of Go's constants let's try a little challenge: declare a *constant* whose value is the number of bits in the natural machine word.

~~We can't use `unsafe.Sizeof` as it is not a constant expression~~[1]. We could use a build tag and laboriously record the natural word size of each Go platform, or we could do something like this:

const uintSize = 32 << (^uint(0) >> 32 & 1)

There are many versions of this expression in Go codebases. They all work roughly the same way. If we're on a 64 bit platform then the exclusive or of the number zero–all zero bits–is a number with all bits set, sixty four of them to be exact.

1111111111111111111111111111111111111111111111111111111111111111

If we shift that value thirty two bits to the right, we get another value with thirty two ones in it.

0000000000000000000000000000000011111111111111111111111111111111

Anding that with a number with one bit in the final position give us, the same thing, `1`,

0000000000000000000000000000000011111111111111111111111111111111 & 1 = 1

Finally we shift the number thirty two one place to the right, giving us 64[2].

32 << 1 = 64

This expression is an example of a *constant expression*. All of these operations happen at compile time and the result of the expression is itself a constant. If you look in the in runtime package, in particular the garbage collector, you’ll see how constant expressions are used to set up complex invariants based on the word size of the machine the code is compiled on.

So, this is a neat party trick, but most compilers will do this kind of constant folding at compile time for you. Let’s step it up a notch.

Constants are values

In Go, constants are values and each value has a type. In Go, user defined types can declare their own methods. Thus, a constant value can have a method set. If you’re surprised by this, let me show you an example that you probably use every day.

```
const timeout = 500 * time.Millisecond
fmt.Println("The timeout is", timeout) // 500ms
```

In the example the untyped literal constant `500` is multiplied by `time.Millisecond`, itself a constant of type `time.Duration`. The rule for assignments in Go are, unless otherwise declared, the type on the left hand side of the assignment operator is inferred from the type on the right.`500` is an untyped constant so it is converted to a `time.Duration` then multiplied with the constant `time.Millisecond`.

Thus `timeout` is a constant of type `time.Duration` which holds the value `500000000`.
Why then does `fmt.Println` print `500ms`, not `500000000`?

The answer is `time.Duration` has a `String` method. Thus any `time.Duration` value, even a constant, knows how to pretty print itself.

Now we know that constant values are typed, and because types can declare methods, we can derive that *constant values can fulfil interfaces*. In fact we just saw an example of this. `fmt.Println` doesn’t assert that a value has a `String` method, it asserts the value implements the `Stringer` interface.

Let’s talk a little about how we can use this property to make our Go code better, and to do that I’m going to take a brief digression into the Singleton pattern.

Singletons

I’m generally not a fan of the singleton pattern, in Go or any language. Singletons complicate testing and create unnecessary coupling between packages. I feel the singleton pattern is often used *not* to create a singular instance of a thing, but instead to create a place to coordinate registration. `net/http.DefaultServeMux` is a good example of this pattern.

```
package http

// DefaultServeMux is the default ServeMux used by Serve.
var DefaultServeMux = &defaultServeMux

var defaultServeMux ServeMux
```

There is nothing singular about `http.defaultServerMux`, nothing prevents you from creating another `ServeMux`. In fact the `http` package provides a helper that will create as many `ServeMux`‘s as you want.

```
// NewServeMux allocates and returns a new ServeMux.
func NewServeMux() *ServeMux { return new(ServeMux) }
```

`http.DefaultServeMux` is not a singleton. Never the less there is a case for things which are truely singletons because they can only represent a single thing. A good example of this are the file descriptors of a process; 0, 1, and 2 which represent stdin, stdout, and stderr respectively.

It doesn’t matter what names you give them, `1` is always stdout, and there can only ever be one file descriptor `1`. Thus these two operations are identical:

```
fmt.Fprintf(os.Stdout, "Hello dotGo\n")
syscall.Write(1, []byte("Hello dotGo\n"))
```

So let’s look at how the `os` package defines `Stdin`, `Stdout`, and `Stderr`:

```
package os

var (
	Stdin  = NewFile(uintptr(syscall.Stdin), "/dev/stdin")
	Stdout = NewFile(uintptr(syscall.Stdout), "/dev/stdout")
	Stderr = NewFile(uintptr(syscall.Stderr), "/dev/stderr")
)
```

There are a few problems with this declaration. Firstly their type is `*os.File` not the respective `io.Reader` or `io.Writer` interfaces. People have long complained that this makes replacing them with alternatives problematic. However the notion of replacing these variables is precisely the point of this digression. Can you safely change the value of `os.Stdout` once your program is running without causing a data race?

I argue that, in the general case, you cannot. In general, if something is unsafe to do, as programmers we shouldn't let our users think that it is safe, lest they begin to depend on that behaviour.

Could we change the definition of `os.Stdout` and friends so that they retain the observable behaviour of reading and writing, but remain immutable? It turns out, we can do this easily with constants.

```
type readfd int

func (r readfd) Read(buf []byte) (int, error) {
     return syscall.Read(int(r), buf)
}

type writefd int

func (w writefd) Write(buf []byte) (int, error) {
      return syscall.Write(int(w), buf)
}

const (
      Stdin  = readfd(0)
      Stdout = writefd(1)
      Stderr = writefd(2)
)

func main() {
      fmt.Fprintf(Stdout, "Hello world")
}
```

In fact this change causes only one compilation failure in the standard library.[3]

Sentinel error values

Another case of things which look like constants but really aren't, are sentinel error values. `io.EOF`, `sql.ErrNoRows`, `crypto/x509.ErrUnsupportedAlgorithm`, and so on are all examples of sentinel error values. They all fall into a category of *expected* errors, and because they are expected, you're expected to check for them.

To compare the error you have with the one you were expecting, you need to import the package that defines that error. Because, by definition, sentinel errors are exported public variables, any code that imports, for example, the `io` package could change the value of `io.EOF`.

```
package nelson

import "io"

func init() {
	io.EOF = nil // haha!
}
```

I'll say that again. If I know the name of `io.EOF` I can import the package that declares it, which I must if I want to compare it to my error, and thus I could change `io.EOF`'s value. Historically convention and a bit of dumb luck discourages people from writing code that does this, but technically there is nothing to prevent you from doing so.

Replacing `io.EOF` is probably going to be detected almost immediately. But replacing a less frequently used sentinel error may cause some interesting side effects:

```
package innocent

import "crypto/rsa"

func init() {
	rsa.ErrVerification = nil // 🤔
}
```

If you were hoping the race detector will spot this subterfuge, I suggest you talk to the folks writing testing frameworks who replace `os.Stdout` without it triggering the race detector.

Fungibility

I want to digress for a moment to talk about *the* most important property of constants. Constants aren't just immutable, its not enough that we cannot overwrite their declaration,
Constants are *fungible*. This is a tremendously important property that doesn't get nearly enough attention.

Fungible means identical. Money is a great example of fungibility. If you were to lend me 10 bucks, and I later pay you back, the fact that you gave me a 10 dollar note and I returned to you 10 one dollar bills, with respect to its operation as a financial instrument, is irrelevant. Things which are fungible are by definition equal and equality is a powerful property we can leverage for our programs.

```
var myEOF = errors.New("EOF") // io/io.go line 38
fmt.Println(myEOF == io.EOF)  // false
```

Putting aside the effect of malicious actors in your code base the key design challenge with sentinel errors is they behave like *singletons*, not *constants*. Even if we follow the exact procedure used by the `io` package to create our own EOF value, `myEOF` and `io.EOF` are not equal. `myEOF` and `io.EOF` are not fungible, they cannot be interchanged. Programs can spot the difference.

When you combine the lack of immutability, the lack of fungibility, the lack of equality, you have a set of weird behaviours stemming from the fact that sentinel error values in Go are not constant expressions. But what if they were?

Constant errors

Ideally a sentinel error value should behave as a constant. It should be immutable and fungible. Let’s recap how the built in `error` interface works in Go.

```
type error interface {
        Error() string
}
```

Any type with an `Error() string` method fulfils the `error` interface. This includes user defined types, it includes types derived from primitives like string, and it includes constant strings. With that background, consider this error implementation:

```
type Error string

func (e Error) Error() string {
        return string(e)
}
```

We can use this error type as a constant expression:

```
const err = Error("EOF")
```

Unlike `errors.errorString`, which is a struct, a compact struct literal initialiser is not a constant expression and cannot be used.

```
const err2 = errors.errorString{"EOF"} // doesn't compile
```

As constants of this `Error` type are not variables, they are immutable.

```
const err = Error("EOF")
err = Error("not EOF")   // doesn't compile
```

Additionally, two constant strings are always equal if their contents are equal:

```
const str1 = "EOF"
const str2 = "EOF"
fmt.Println(str1 == str2) // true
```

which means two constants of a type derived from string with the same contents are also equal.

```
type Error string

const err1 = Error("EOF")
const err2 = Error("EOF")
fmt.Println(err1 == err2) // true```
```

Said another way, equal constant `Error` values are the same, in the way that the literal constant `1` is the same as every other literal constant `1`.

Now we have all the pieces we need to make sentinel errors, like `io.EOF`, and `rsa.ErrVerfication`, immutable, fungible, constant expressions.

```
% git diff
diff --git a/src/io/io.go b/src/io/io.go
index 2010770e6a..355653b4b8 100644
--- a/src/io/io.go
+++ b/src/io/io.go
@@ -35,7 +35,12 @@ var ErrShortBuffer = errors.New("short buffer")
 // If the EOF occurs unexpectedly in a structured data stream,
 // the appropriate error is either ErrUnexpectedEOF or some other error
 // giving more detail.
-var EOF = errors.New("EOF")
+const EOF = ioError("EOF")
+
+type ioError string
+
+func (e ioError) Error() string { return string(e) }
```

This change is probably a bit of a stretch for the Go 1 contract, but there is no reason you cannot adopt a constant error pattern for your sentinel errors in the packages that you write.

In summary

Go’s constants are powerful. If you only think of them as immutable numbers, you’re missing out. Go’s constants let us compose programs that are more correct and harder to misuse.

Today I’ve outlined three ways to use constants that are more than your typical immutable number.

Now it’s over to you, I’m excited to see where you can take these ideas.

Several commenters have written to remind me that this is not correct. `unsafe.Sizeof` *is* a constant expression. I’m sorry, I’m not sure what I was thinking.

I’ll leave it as an exercise for you to do the math for a 32 bit word.

Ironically this failure is in the `testing` package which is trying to do exactly the replacement this section warns about.