The Go Blog

Constants

25 August 2014

Introduction

Go is a statically typed language that does not permit operations that mix numeric types. You can't add a `float64` to an `int`, or even an `int32` to an `int`. Yet it is legal to write `1e6*time.Second` or `math.Exp(1)` or even `1<<('\t'+2.0)`. In **Go**, **constants**, unlike variables, behave pretty much like regular numbers. This post explains why that is and what it means.

Background: C

In the early days of thinking about **Go**, we talked about a number of problems caused by the way **C** and its descendants let you mix and match numeric types. Many mysterious bugs, crashes, and portability problems are caused by expressions that combine integers of different sizes and "signedness". Although to a seasoned **C** programmer the result of a calculation like

```
unsigned int u = 1e9;
long signed int i = -1;
... i + u ...
```

may be familiar, it isn't *a priori* obvious. How big is the result? What is its value? Is it signed or unsigned?

Nasty bugs lurk here.

C has a set of rules called "the usual arithmetic conversions" and it is an indicator of their subtlety that they have changed over the years (introducing yet more bugs, retroactively).

Next article

Deploying **Go** servers with Docker

Previous article

Go at OSCON

Links

golang.org
Install **Go**
A Tour of **Go**
Go Documentation
Go Mailing List
Go on Google+
Go+ Community
Go on Twitter

Blog index

When designing **Go**, we decided to avoid this minefield by mandating that there is *no* mixing of numeric types. If you want to add `i` and `u`, you must be explicit about what you want the result to be. Given

```
var u uint
var i int
```

you can write either `uint(i)+u` or `i+int(u)`, with both the meaning and type of the addition clearly expressed, but unlike in **C** you cannot write `i+u`. You can't even mix `int` and `int32`, even when `int` is a 32-bit type.

This strictness eliminates a common cause of bugs and other failures. It is a vital property of **Go**. But it has a cost: it sometimes requires programmers to decorate their code with clumsy numeric conversions to express their meaning clearly.

And what about **constants**? Given the declarations above, what would make it legal to write `i = 0` or `u = 0`? What is the *type* of `0`? It would be unreasonable to require **constants** to have type conversions in simple contexts such as `i = int(0)`.

We soon realized the answer lay in making numeric **constants** work differently from how they behave in other **C**-like languages. After much thinking and experimentation, we came up with a design that we believe feels right almost always, freeing the programmer from converting **constants** all the time yet being able to write things like `math.Sqrt(2)` without being chided by the compiler.

In short, **constants** in **Go** just work, most of the time anyway. Let's see how that happens.

Terminology

First, a quick definition. In **Go**, `const` is a keyword introducing a name for a scalar value such as `2` or `3.14159` or `"scrumptious"`. Such values, named or otherwise, are called ***constants*** in **Go**. **Constants** can also be created by expressions built from **constants**, such as `2+3` or `2+3i` or `math.Pi/2` or `("go"+"pher")`.

Some languages don't have **constants**, and others have a more general definition of constant or application of the word `const`. In **C** and C++, for instance, `const` is a type qualifier that can codify more intricate properties of more intricate values.

But in **Go**, a constant is just a simple, unchanging value, and from here on we're talking only about **Go**.

String constants

There are many kinds of numeric **constants**—integers, floats, runes, signed, unsigned, imaginary, complex—so let's start with a simpler form of constant: strings. String **constants** are easy to understand and provide a smaller space in which to explore the type issues of **constants** in **Go**.

A string constant encloses some text between double quotes. (**Go** also has raw string literals, enclosed by backquotes ``, but for the purpose of this discussion they have all the same properties.) Here is a string constant:

```
"Hello, 世界"
```

(For much more detail about the representation and interpretation of strings, see this blog post.)

What type does this string constant have? The obvious answer is `string`, but that is *wrong*.

This is an *untyped string constant*, which is to say it is a constant textual value that does not yet have a fixed type. Yes, it's a string, but it's not a **Go** value of type `string`. It remains an untyped string constant even when given a name:

```
const hello = "Hello, 世界"
```

After this declaration, `hello` is also an untyped string constant. An untyped constant is just a value, one not yet given a defined type that would force it to obey the strict rules that prevent combining differently typed values.

It is this notion of an *untyped* constant that makes it possible for us to use **constants** in **Go** with great freedom.

So what, then, is a *typed* string constant? It's one that's been given a type, like this:

```
const typedHello string = "Hello, 世界"
```

Notice that the declaration of `typedHello` has an explicit `string` type before the equals sign. This means that `typedHello` has **Go** type `string`, and cannot be assigned to a **Go** variable of a different type. That is to say, this code works:

```
var s string
s = typedHello
fmt.Println(s)
```

but this does not:

```
type MyString string
var m MyString
m = typedHello // Type error
fmt.Println(m)
```

The variable `m` has type `MyString` and cannot be assigned a value of a different type. It can only be assigned values of type `MyString`, like this:

```
const myStringHello MyString = "Hello, 世界"
m = myStringHello // OK
fmt.Println(m)
```

or by forcing the issue with a conversion, like this:

```
m = MyString(typedHello)
fmt.Println(m)
```

Returning to our *untyped* string constant, it has the helpful property that, since it has no type, assigning it to a typed variable does not cause a type error. That is, we can write

```
m = "Hello, 世界"
```

or

```
m = hello
```

because, unlike the typed **constants** `typedHello` and `myStringHello`, the untyped **constants** `"Hello, 世界"` and `hello` *have no type*. Assigning them to a variable of any type compatible with strings works without error.

These untyped string **constants** are strings, of course, so they can only be used where a string is allowed, but they do not have *type* `string`.

Default type

As a **Go** programmer, you have certainly seen many declarations like

```
str := "Hello, 世界"
```

and by now you might be asking, "if the constant is untyped, how does `str` get a type in this variable declaration?" The answer is that an untyped constant has a default type, an implicit type that it transfers to a value if a type is needed where none is provided. For untyped string **constants**, that default type is obviously `string`, so

```
str := "Hello, 世界"
```

or

```
var str = "Hello, 世界"
```

means exactly the same as

```
var str string = "Hello, 世界"
```

One way to think about untyped **constants** is that they live in a kind of ideal space of values, a space less restrictive than **Go**'s full type system. But to do anything with them, we need to assign them to variables, and when that happens the *variable* (not the constant itself) needs a type, and the constant can tell the variable what type it should have. In this example, `str` becomes a value of type `string` because the untyped string constant gives the declaration its default type, `string`.

In such a declaration, a variable is declared with a type and initial value. Sometimes when we use a constant, however, the destination of the value is not so clear. For instance consider this statement:

```
    fmt.Printf("%s", "Hello, 世界")
```

The signature of `fmt.Printf` is

```
func Printf(format string, a ...interface{}) (n int, err error)
```

which is to say its arguments (after the format string) are interface values. What happens when `fmt.Printf` is called with an untyped constant is that an interface value is created to pass as an argument, and the concrete type stored for that argument is the default type of the constant. This process is analogous to what we saw earlier when declaring an initialized value using an untyped string constant.

You can see the result in this example, which uses the format `%v` to print the value and `%T` to print the type of the value being passed to `fmt.Printf`:

```
    fmt.Printf("%T: %v\n", "Hello, 世界", "Hello, 世界")
    fmt.Printf("%T: %v\n", hello, hello)
```

If the constant has a type, that goes into the interface, as this example shows:

```
    fmt.Printf("%T: %v\n", myStringHello, myStringHello)
```

(For more information about how interface values work, see the first sections of this blog post.)

In summary, a typed constant obeys all the rules of typed values in **Go**. On the other hand, an untyped constant does not carry a **Go** type in the same way and can be mixed and matched more

freely. It does, however, have a default type that is exposed when, and only when, no other type information is available.

Default type determined by syntax

The default type of an untyped constant is determined by its syntax. For string **constants**, the only possible implicit type is `string`. For numeric **constants**, the implicit type has more variety. Integer **constants** default to `int`, floating-point **constants** `float64`, rune **constants** to `rune` (an alias for `int32`), and imaginary **constants** to `complex128`. Here's our canonical print statement used repeatedly to show the default types in action:

```
fmt.Printf("%T %v\n", 0, 0)
fmt.Printf("%T %v\n", 0.0, 0.0)
fmt.Printf("%T %v\n", 'x', 'x')
fmt.Printf("%T %v\n", 0i, 0i)
```

(Exercise: Explain the result for `'x'`.)

Booleans

Everything we said about untyped string **constants** can be said for untyped boolean **constants**. The values `true` and `false` are untyped boolean **constants** that can be assigned to any boolean variable, but once given a type, boolean variables cannot be mixed:

```
type MyBool bool
const True = true
const TypedTrue bool = true
var mb MyBool
mb = true      // OK
mb = True      // OK
mb = TypedTrue // Bad
fmt.Println(mb)
```

Run the example and see what happens, then comment out the "Bad" line and run it again. The pattern here follows exactly that of string **constants**.

Floats

Floating-point **constants** are just like boolean **constants** in most respects. Our standard example works as expected in translation:

```
type MyFloat64 float64
const Zero = 0.0
const TypedZero float64 = 0.0
var mf MyFloat64
mf = 0.0       // OK
mf = Zero      // OK
mf = TypedZero // Bad
fmt.Println(mf)
```

One wrinkle is that there are *two* floating-point types in **Go**: `float32` and `float64`. The default type for a floating-point constant is `float64`, although an untyped floating-point constant can be assigned to a `float32` value just fine:

```
var f32 float32
f32 = 0.0
f32 = Zero      // OK: Zero is untyped
```

```
f32 = TypedZero // Bad: TypedZero is float64 not float32.
fmt.Println(f32)
```

Floating-point values are a good place to introduce the concept of overflow, or the range of values.

Numeric **constants** live in an arbitrary-precision numeric space; they are just regular numbers. But when they are assigned to a variable the value must be able to fit in the destination. We can declare a constant with a very large value:

```
const Huge = 1e1000
```

—that's just a number, after all—but we can't assign it or even print it. This statement won't even compile:

```
fmt.Println(Huge)
```

The error is, "constant 1.00000e+1000 overflows float64", which is true. But `Huge` might be useful: we can use it in expressions with other **constants** and use the value of those expressions if the result can be represented in the range of a `float64`. The statement,

```
fmt.Println(Huge / 1e999)
```

prints `10`, as one would expect.

In a related way, floating-point **constants** may have very high precision, so that arithmetic involving them is more accurate. The **constants** defined in the math package are given with many more digits than are available in a `float64`. Here is the definition of `math.Pi`:

```
Pi    = 3.14159265358979323846264338327950288419716939937510582097494459
```

When that value is assigned to a variable, some of the precision will be lost; the assignment will create the `float64` (or `float32`) value closest to the high-precision value. This snippet

```
pi := math.Pi
fmt.Println(pi)
```

prints `3.141592653589793`.

Having so many digits available means that calculations like `Pi/2` or other more intricate evaluations can carry more precision until the result is assigned, making calculations involving **constants** easier to write without losing precision. It also means that there is no occasion in which the floating-point corner cases like infinities, soft underflows, and `NaNs` arise in constant expressions. (Division by a constant zero is a compile-time error, and when everything is a number there's no such thing as "not a number".)

Complex numbers

Complex **constants** behave a lot like floating-point **constants**. Here's a version of our now-familiar litany translated into complex numbers:

```
type MyComplex128 complex128
const I = (0.0 + 1.0i)
const TypedI complex128 = (0.0 + 1.0i)
var mc MyComplex128
mc = (0.0 + 1.0i) // OK
mc = I            // OK
mc = TypedI       // Bad
fmt.Println(mc)
```

The default type of a complex number is `complex128`, the larger-precision version composed of two `float64` values.

For clarity in our example, we wrote out the full expression `(0.0+1.0i)`, but this value can be shortened to `0.0+1.0i`, `1.0i` or even `1i`.

Let's play a trick. We know that in **Go**, a numeric constant is just a number. What if that number is a complex number with no imaginary part, that is, a real? Here's one:

```
const Two = 2.0 + 0i
```

That's an untyped complex constant. Even though it has no imaginary part, the *syntax* of the expression defines it to have default type `complex128`. Therefore, if we use it to declare a variable, the default type will be `complex128`. The snippet

```
s := Two
fmt.Printf("%T: %v\n", s, s)
```

prints `complex128: (2+0i)`. But numerically, `Two` can be stored in a scalar floating-point number, a `float64` or `float32`, with no loss of information. Thus we can assign `Two` to a `float64`, either in an initialization or an assignment, without problems:

```
var f float64
var g float64 = Two
f = Two
fmt.Println(f, "and", g)
```

The output is `2 and 2`. Even though `Two` is a complex constant, it can be assigned to scalar floating-point variables. This ability for a constant to "cross" types like this will prove useful.

Integers

At last we come to integers. They have more moving parts—many sizes, signed or unsigned, and more—but they play by the same rules. For the last time, here is our familiar example, using just `int` this time:

```
type MyInt int
const Three = 3
const TypedThree int = 3
var mi MyInt
mi = 3          // OK
mi = Three      // OK
mi = TypedThree // Bad
fmt.Println(mi)
```

The same example could be built for any of the integer types, which are:

```
int int8 int16 int32 int64
uint uint8 uint16 uint32 uint64
uintptr
```

(plus the aliases `byte` for `uint8` and `rune` for `int32`). That's a lot, but the pattern in the way **constants** work should be familiar enough by now that you can see how things will play out.

As mentioned above, integers come in a couple of forms and each form has its own default type: `int` for simple **constants** like `123` or `0xFF` or `-14` and `rune` for quoted characters like 'a', '世' or '\r'.

No constant form has as its default type an unsigned integer type. However, the flexibility of untyped **constants** means we can initialize unsigned integer variables using simple **constants** as long as we are clear about the type. It's analogous to how we can initialize a `float64` using a complex number with zero imaginary part. Here are several different ways to initialize a `uint`; all are equivalent, but all must mention the type explicitly for the result to be unsigned.

```
var u uint = 17
var u = uint(17)
u := uint(17)
```

Similarly to the range issue mentioned in the section on floating-point values, not all integer values can fit in all integer types. There are two problems that might arise: the value might be too large, or it might be a negative value being assigned to an unsigned integer type. For instance, `int8` has range -128 through 127, so **constants** outside of that range can never be assigned to a variable of type `int8`:

```
var i8 int8 = 128 // Error: too large.
```

Similarly, `uint8`, also known as `byte`, has range 0 through 255, so a large or negative constant cannot be assigned to a `uint8`:

```
var u8 uint8 = -1 // Error: negative value.
```

This type-checking can catch mistakes like this one:

```
type Char byte
var c Char = '世' // Error: '世' has value 0x4e16, too large.
```

If the compiler complains about your use of a constant, it's likely a real bug like this.

An exercise: The largest unsigned int

Here is an informative little exercise. How do we express a constant representing the largest value that fits in a `uint`? If we were talking about `uint32` rather than `uint`, we could write

```
const MaxUint32 = 1<<32 - 1
```

but we want `uint`, not `uint32`. The `int` and `uint` types have equal unspecified numbers of bits, either 32 or 64. Since the number of bits available depends on the architecture, we can't just write down a single value.

Fans of two's-complement arithmetic, which **Go**'s integers are defined to use, know that the representation of `-1` has all its bits set to 1, so the bit pattern of `-1` is internally the same as that of the largest unsigned integer. We therefore might think we could write

```
const MaxUint uint = -1 // Error: negative value
```

but that is illegal because -1 cannot be represented by an unsigned variable; `-1` is not in the range of unsigned values. A conversion won't help either, for the same reason:

```
const MaxUint uint = uint(-1) // Error: negative value
```

Even though at run-time a value of -1 can be converted to an unsigned integer, the rules for constant conversions forbid this kind of coercion at compile time. That is to say, this works:

```
var u uint
var v = -1
```

```
u = uint(v)
```

but only because `v` is a variable; if we made `v` a constant, even an untyped constant, we'd be back in forbidden territory:

```
var u uint
const v = -1
u = uint(v) // Error: negative value
```

We return to our previous approach, but instead of `-1` we try `^0`, the bitwise negation of an arbitrary number of zero bits. But that fails too, for a similar reason: In the space of numeric values, `^0` represents an infinite number of ones, so we lose information if we assign that to any fixed-size integer:

```
const MaxUint uint = ^0 // Error: overflow
```

How then do we represent the largest unsigned integer as a constant?

The key is to constrain the operation to the number of bits in a `uint` and avoiding values, such as negative numbers, that are not representable in a `uint`. The simplest `uint` value is the typed constant `uint(0)`. If `uints` have 32 or 64 bits, `uint(0)` has 32 or 64 zero bits accordingly. If we invert each of those bits, we'll get the correct number of one bits, which is the largest `uint` value.

Therefore we don't flip the bits of the untyped constant `0`, we flip the bits of the typed constant `uint(0)`. Here, then, is our constant:

```
const MaxUint = ^uint(0)
fmt.Printf("%x\n", MaxUint)
```

Whatever the number of bits it takes to represent a `uint` in the current execution environment (on the playground, it's 32), this constant correctly represents the largest value a variable of type `uint` can hold.

If you understand the analysis that got us to this result, you understand all the important points about **constants** in **Go**.

Numbers

The concept of untyped **constants** in **Go** means that all the numeric **constants**, whether integer, floating-point, complex, or even character values, live in a kind of unified space. It's when we bring them to the computational world of variables, assignments, and operations that the actual types matter. But as long as we stay in the world of numeric **constants**, we can mix and match values as we like. All these **constants** have numeric value 1:

```
1
1.000
1e3-99.0*10-9
'\x01'
'\u0001'
'b' - 'a'
1.0+3i-3.0i
```

Therefore, although they have different implicit default types, written as untyped **constants** they can be assigned to a variable of any integer type:

```
var f float32 = 1
var i int = 1.000
var u uint32 = 1e3 - 99.0*10.0 - 9
```

```
var c float64 = '\x01'
var p uintptr = '\u0001'
var r complex64 = 'b' - 'a'
var b byte = 1.0 + 3i - 3.0i

fmt.Println(f, i, u, c, p, r, b)
```

The output from this snippet is: `1 1 1 1 1 (1+0i) 1`.

You can even do nutty stuff like

```
var f = 'a' * 1.5
fmt.Println(f)
```

which yields 145.5, which is pointless except to prove a point.

But the real point of these rules is flexibility. That flexibility means that, despite the fact that in **Go** it is illegal in the same expression to mix floating-point and integer variables, or even `int` and `int32` variables, it is fine to write

```
sqrt2 := math.Sqrt(2)
```

or

```
const millisecond = time.Second/1e3
```

or

```
bigBufferWithHeader := make([]byte, 512+1e6)
```

and have the results mean what you expect.

Because in **Go**, numeric **constants** work as you expect: like numbers.

By Rob Pike