Lecture 6: Types and Expressions

Types

Since the computer only operates on bits, we need a way to specify how to interpret particular sets of bits. Do these bits represent an integer? a string? a real number?

Go (and many programming languages) do this via types, which we saw a little bit last time. Now, we'll see more of the types that Go has built in.

Number-based types

<table>
<thead>
<tr>
<th>Type</th>
<th>Data</th>
<th>Examples</th>
</tr>
</thead>
<tbody>
<tr>
<td>int</td>
<td>Positive or negative integers</td>
<td>3,-200,40,42</td>
</tr>
<tr>
<td>uint</td>
<td>Non-negative integers (u = unsigned)</td>
<td>0, 3, 7, 11, 13</td>
</tr>
<tr>
<td>bool</td>
<td>Holds true or false</td>
<td>true</td>
</tr>
<tr>
<td>float64</td>
<td>Real, floating point number</td>
<td>3.14159, 12e-3, 0.23</td>
</tr>
<tr>
<td>complex128</td>
<td>Complex number (real, imaginary)</td>
<td>10 + 3i</td>
</tr>
<tr>
<td>string</td>
<td>Holds a sequence of characters</td>
<td>&quot;Hello, world&quot;</td>
</tr>
</tbody>
</table>

Some example variable definition:

```go
var m uint = 10
var small bool = true
var big bool = m > 10
var e, pi float64 = 2.7182818285, 3.14
var name string = "Carl"
var root complex64 = 3 + 7i
```
Explicit values for variables are called literals.

- **Integer literals:** a sequence of digits 0, 1, 2, ..., 9

```plaintext
1 72
2 6402
3 000734
4 0xFF  // hexadecimal literals start with 0x
```

- **String literals:** a sequence of characters between quotes "

```plaintext
1 "Hi there"
2 "😄"  // unicode characters supported in strings
3 "1+¾=4"
4 "3.14159"  // this is a string NOT a number
```

- **bool (Boolean) literals:** either `true` or `false`

```plaintext
1 true
2 false
```

- **Floating point (real) literals:** a number with a "." or "e"

```plaintext
1 7.
2 7.0
3 .32456
4 1.21212121
5 12E2
6 10E+3
7 1e-2
```

aEb means \( a \times 10^b \).

- **Imaginary literal:** floating point literal with `i` after it

```plaintext
1 7.0i
2 7i
3 1e-5i
```
Everything in a Go expression must have the same type.

Type conversions

You can change the type of a variable in an expression by type casting.

You use the syntax: \texttt{TYPE(EXPR)} to change the type of \texttt{EXPR} to \texttt{TYPE}.

You know that time is 7.2, but Go doesn’t know that, so it trusts you that you want to change time to an int.

When converting a floating point number to an int, Go will throw away the fractional part.

Conversion challenges

Q: What values do \texttt{c} and \texttt{d} have?
Q: What value does q have?

Variables have limited range

<table>
<thead>
<tr>
<th>Type</th>
<th>Min</th>
<th>Max</th>
</tr>
</thead>
<tbody>
<tr>
<td>int</td>
<td>-9223372036854775808</td>
<td>9223372036854775807</td>
</tr>
<tr>
<td>uint</td>
<td>0</td>
<td>18446744073709551615</td>
</tr>
<tr>
<td>float64</td>
<td>-1.797693134862315708145274237317043567981e+30</td>
<td>1.797693134862315708145274237317043567981e+308</td>
</tr>
</tbody>
</table>

Question: What does this print?

```
var i int = 9223372036854775807
fmt.Println(i+1)
```

Go has types that let you specify how many bits they use:
<table>
<thead>
<tr>
<th>Type</th>
<th>Number of bits</th>
</tr>
</thead>
<tbody>
<tr>
<td>int</td>
<td>32 or 64 depending on your computer</td>
</tr>
<tr>
<td>uint</td>
<td>32 or 64 depending on your computer (but always same size as int)</td>
</tr>
<tr>
<td>int8 / uint8</td>
<td>8</td>
</tr>
<tr>
<td>int16 / uint16</td>
<td>16</td>
</tr>
<tr>
<td>int32 / uint32</td>
<td>32</td>
</tr>
<tr>
<td>int64 / uint64</td>
<td>64</td>
</tr>
<tr>
<td>float32</td>
<td>32</td>
</tr>
<tr>
<td>float64</td>
<td>64</td>
</tr>
<tr>
<td>complex64</td>
<td>32 for each of the real and imaginary parts</td>
</tr>
<tr>
<td>complex128</td>
<td>64 for each of the real and imaginary parts</td>
</tr>
<tr>
<td>byte</td>
<td>another word for int8</td>
</tr>
<tr>
<td>rune</td>
<td>another word for int32</td>
</tr>
</tbody>
</table>

Tip: use int, float64, and complex128 unless you have memory limitations.

**string types**

A variable that can hold a string has type `string`:

```go
var name string = "Carl"
```

**Summary of types**

- Types in an expression must agree.
- Be sure you don’t corrupt your data by converting to the wrong type.
- Everything else is basically details that you have to know to program, but that shouldn’t be forefront in your mind.
Expressions & Operators

Operations on integers

\[ a + b \] addition  \hspace{2cm} \[ a - b \] subtraction

\[ a \times b \] multiplication  \hspace{2cm} \[ -a \] negation

\[ +a \] doesn't do anything, but available for symmetry with -

\[ a / b \] integer division: \( 2/3 = 0; \ 10/3 = 3; \ -10/3 = -3 \)
results are truncated toward 0

\[ a \mod b \] modulus (aka remainder):
\[ 13 \mod 2 = 1; \ 10 \mod 2 = 0; \ 10 \mod 3 = 1 \]
\[ -10 \mod 3 = -1 \] (since \( -10 = 3 \times (-3) - 1 \))

Increment and decrement on integers

Adding and subtracting 1 is so common there is a special notation for it:

\[ a++ \] is the same as \[ a = a + 1 \]

\[ a-- \] is the same as \[ a = a - 1 \]

Operations on real values

The standard mathematical operations on real values are supported:
Real-valued division is of limited precision:

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>2.0/3.0 = 0.6666666666666666</td>
</tr>
<tr>
<td>2</td>
<td>10.0/3.0 = 3.3333333333333335</td>
</tr>
<tr>
<td>3</td>
<td>-10.0/3.0 = -3.3333333333333335</td>
</tr>
</tbody>
</table>

Example expressions

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>a + b / 3 + 2</td>
</tr>
<tr>
<td>2</td>
<td>(a+b) / 3 + 2</td>
</tr>
<tr>
<td>3</td>
<td>-a*(3+c - d)</td>
</tr>
</tbody>
</table>

Boolean variables and operators

A **Boolean** variable is a variable that can hold two possible values: either `true` or `false`.

Boolean operators combine boolean variables:

```
<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>a &amp; b</td>
<td><strong>true</strong> if and only if a and b are both <strong>true</strong></td>
</tr>
<tr>
<td>a</td>
<td></td>
</tr>
</tbody>
</table>
```

Go also has comparison operators, the result of which is a Boolean value:

```
<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>a &lt; b a &gt; b</td>
<td>equals</td>
</tr>
<tr>
<td>a &lt;= b a &gt;= b</td>
<td>not equals</td>
</tr>
<tr>
<td>a == b a != b</td>
<td></td>
</tr>
</tbody>
</table>
```

Examples with Boolean values:
Examples, true or false?

<table>
<thead>
<tr>
<th>Expression</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>a &gt; 10 &amp;&amp; b &gt; 20</code></td>
<td><code>false</code></td>
</tr>
<tr>
<td><code>a == 10 &amp;&amp; b &lt; 100 &amp;&amp; a*b &gt; 1000</code></td>
<td><code>false</code></td>
</tr>
<tr>
<td>`a &gt; 20</td>
<td></td>
</tr>
<tr>
<td><code>a==10 &amp;&amp; b=50</code></td>
<td><code>syntax error!</code></td>
</tr>
<tr>
<td>`a==10 &amp;&amp; b &gt;= 100</td>
<td></td>
</tr>
<tr>
<td>`b==50</td>
<td></td>
</tr>
<tr>
<td>`a&gt;5 &amp;&amp; b&gt;20</td>
<td></td>
</tr>
<tr>
<td>`a&gt;5</td>
<td></td>
</tr>
<tr>
<td>`a&gt;5</td>
<td></td>
</tr>
</tbody>
</table>

Packages

- Packages are collections of functions you can use in your program.
- Go provides many built-in packages: see [http://golang.org/pkg/](http://golang.org/pkg/)
- Enable the use of a package with:

```go
import "packageName"
```

at the top of your program.

- If you need to import lots of packages, you can write:

```go
import (
    "package1"
    "package2"
    "package3"
)
```

at the top of your program.

- The `fmt` package provides the `fmt.Print` and `fmt.Println` functions we’ve used a lot.

The `math` package
The **math** package contains many functions related to mathematical operations. For example:

```go
func Abs(x float64) float64
func Min(x, y float64) float64
func Pow(x, y float64) float64
```

To use the math package, put `import "math"` at the top of your program. Then these functions are available with the following syntax:

```go
math.Abs(x)
math.Min(x, y)
math.Pow(x, y)
```

When using a function `F` from a package `X`, you write `X.F`.

### The **strings** package

The **strings** package provides many functions to operate on strings. This example tests whether one string has another as a substring:

```go
var a string = "hi, there"
var b string = "the"
if strings.Contains(a, b) {
    fmt.Println("String a contains string b!")
}
```

- A very large part of programming in practice is looking up how to use functions in existing packages.

### The **strconv** package and **error**

Suppose we have an string "42" that we want to convert to an `int`. We cannot do:

```go
var s string = "42"
var x int64
x = int64(s) // WRONG!
```
The reason we can't do this is that converting a string to an integer involves more than just relabeling its type. It requires interpreting the string in a way we understand as decimal notation. Instead, we have to use a function to do this conversion.

What's going on here? `strconv.ParseInt` is a function that takes 3 parameters:

- the string to convert
- the base of the integer that the string is written in
- the number of bits used to represent the integer

It also returns 2 values:

- the value in the string as an `int64`
- an error value that indicates whether the operation worked

What is an `error` value? `error` is a type that can hold either an error code or the special value `nil` indicating no error.

**Test yourself:** What happens if you run this code:

```go
var x int64
var err error
x, err = strconv.ParseInt("StampyCat", 10, 64)
fmt.Println("x=", x, "err=", err)
```

Finally, note that `strconv.ParseInt` always returns an `int64` --- if you want a different size int, you must convert the result you get.

The `strconv` package contains many functions to convert between various types and `string`s.

Next time, we'll see how to handle the case when you get an error like the one above.

**On your own:** What happens when you run this code:

```go
var x string
x = string(0x1f601)
fmt.Println(x)
```
Summary

- Every variable has a type that tells Go how to interpret the bits that represent that variable.
- In Go, everything in an expression must have the same type.
- You can convert between types by using the type name like a function: int(myFloatVar).
- Consistent with other types: string(103) reinterprets the number 103 as a string. It does not turn 103 into a string "103" of the decimal representation of 103.
- Packages provide lots of useful pre-defined functions, one of which is to convert numbers into strings (and vice versa).

Glossary

- **package**: a group of functions that you can import to use in your program
- **literal**: an explicit value like 23, "hi", or 2.3 in your program
- **type casting**: changing the type of an expression
- **error**: is a special type that holds an error code
- **Boolean**: a type that can hold true or false