Types & Expressions
02-201 / 02-601
Types
# Variables beyond Integers

You can declare variables of several different built-in types:

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

Example variables declarations:

```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…9
  - 72
  - 6402
  - 000734

- **String literals**: a sequence of characters between quotes "
  - "Hi there"
  - "😊"
  - "1+3=4"
  - "3.14159"

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

- **Floating point (real) literals**: a number with a "." or "e"
  - 7.
  - 7.0
  - .32456
  - 1.21212121
  - 12E2
  - 10E+3
  - 1e-2

- **Imaginary literal**: floating point literal with “i” after it
  - 7.0i
  - 7i
  - 1e-5i

---

**Tip:** Use 7.0 and 0.32456 instead of 7. or .32456 (easier to read)

Unicode strings are supported.
Expressions and Variables Must Have the Same Type

```go
code-block-
var a float64 = 3.0  // ok!
var b float64 = "4.0" // ERROR! "4.0" is a string
var c int = 3.0      // ERROR! 3.0 is not an int
var d string = 7000  // ERROR!
var e int = 2        // ok
var f uint = e       // ERROR! e is an int not a uint
var ok bool = 0      // ERROR! 0 is not a bool
var ok2 bool = e > 1 // ok: boolean expression
```

Everything in a Go expression must have the same type.

(this is different than C, C++, Java, which are more forgiving about types)
Can Convert Between Types

Use `type(expression)` to convert `expression` to type:

```go
var a float64 = 3.0 // ok!
var c int = int(3.0) // ok: 3.0 converted to int
var g int = int(3.2) // ERROR! can’t convert 3.2 to int
var e int = 2 // ok
var f uint = uint(e) // ok: e is converted to uint
var ok bool = bool(0) // ERROR! can’t covert ints to bools
```

Go really tries to avoid changing the value of a constant.

```go
var time float64 = 7.2 // ok
var r int = time // ERROR! time not an int
var round int = int(time) // ok!!! round will equal 7
```

*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 c and d have?
Answer:
\[
\begin{align*}
c & = 7 \\
d & = -13
\end{align*}
\]

Q: What value does q have?
Answer: it depends on your computer, but on mine:
\[
q = 18446744073709551546
\]

What’s going on here?

- Go (and nearly all programming languages) represent negative integers in a format called “twos-complement”
- Converting a negative number to a \texttt{uint} simply reinterprets the bits used in twos-complement to represent a positive integer
- This is almost never what you want.
- Lesson: converting a negative integer to a uint is probably a bug!

More details later
```go
var i uint = 10
for ; i >= 0; i = i - 1 {
    fmt.Println(i)
}
```

Q: How many times does this loop iterate?

Answer: it never stops!

What’s going on here?

• This is a common kind of bug

• `uints` can’t be negative:

```go
var q uint = 0
q = q - 1
fmt.Println(q)
```

• This prints: 18446744073709551615

• `q` can’t be negative, so "wraps around" to the largest possible `int`
# 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+308</td>
<td>1.797693134862315708145274237317043567981e+308</td>
</tr>
</tbody>
</table>

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

Q: What does the above print?

**Answer:**

```
-9223372036854775808
```

`i+1` is too big for an **int**, so it wraps around to the smallest possible **int**. This is called *overflow* and it's usually a bug!

**Lesson:** if you have very big or very small numbers, you have to do something special.
Binary Numbers

Base 10 (decimal) notation:

\[
\begin{array}{cccc}
4 & 2 & 5 & 6 \\
\downarrow & \downarrow & \downarrow & \downarrow \\
6 \times 10^0 & 5 \times 10^1 & 2 \times 10^2 & 4 \times 10^3 \\
\hline
\end{array}
\]

\[4256\]

Computers store the numbers in binary because it has transistors that can encode 0 and 1 efficiently.

Each 0 and 1 is a *bit*.

Built-in number types each have a maximum number of bits.

Base 2 (binary) notation:

\[
\begin{array}{cccccccccccc}
1 & 0 & 0 & 0 & 0 & 1 & 0 & 1 & 0 & 0 & 0 & 0 \\
\downarrow & \downarrow & \downarrow & \downarrow & \downarrow & \downarrow & \downarrow & \downarrow & \downarrow & \downarrow & \downarrow & \downarrow \\
0 \times 2^0 & 0 \times 2^1 & 0 \times 2^2 & 1 \times 2^3 & 0 \times 2^4 & 0 \times 2^5 & 1 \times 2^6 & 0 \times 2^7 & 0 \times 2^8 & 0 \times 2^9 & 1 \times 2^{10} & 0 \times 2^{11} & 1 \times 2^{12} \\
\hline
\end{array}
\]

\[4256 = 1000010100000\]
## Variables with Different Ranges

<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.
You can often omit specifying the type if Go can guess what type it should be:

```go
type Inference in var and := Statements

var a = 3     // a will be an int
var b = uint(a*a)  // b will be a uint
var d = "hi there" // d will be a string
var ok = true     // ok will be a bool
var ok2 = e > 1   // ok2 will be a bool
var z = 3.0       // a will be a float64
q := 1e-23        // q will be a float64
var theta = 3 + 2.0i // theta will be a complex128
var t2 = 2*theta  // t2 will be complex128
result := f(a,b,d) // will be the return type of f
```

Integer literals (3, -10, etc.) cause ints to be inferred.

The larger float and complex size types (float64, complex128) are always used when inferring from literals of those types.
The Lesson 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
**Integer Operations**

- **Addition**: $a + b$
- **Subtraction**: $a - b$
- **Multiplication**: $a \times b$
- **Negation**: $-a$
- **Integer Division**: $a \div b$
- **Modulus** (aka remainder): $a \mod b$

**Integer Division**:
- $2/3 = 0$
- $10/3 = 3$
- $-10/3 = -3$

Results are truncated toward 0.

**Modulus** (aka remainder):
- $13 \mod 2 = 1$
- $10 \mod 2 = 0$
- $10 \mod 3 = 1$
- $-10 \mod 3 = -1$ (since $-10 = 3(-3) - 1$)

**Equation**:
- If $q = x / y$ and $r = x \mod y$ then $x = q \times y + r$ and $|r| < |y|$
Increment and Decrement Statements

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 \[ a = a - 1 \]

This is particularly useful in for loops:

```
for i := 1; i <= n; i++ {
    // body of for loop
}
```
Float and Complex Operators

\[ a + b \quad a - b \quad a \times b \quad -a \quad +a \]

\[ a / b \]

floating point division:
- \[ 2.0/3.0 = 0.6666666666666666 \]
- \[ 10.0/3.0 = 3.3333333333333335 \]
- \[ -10.0/3.0 = -3.3333333333333335 \]

results are of limited precision
b can’t be 0
Boolean Operators

- `a && b` is `true` if and only if `a` and `b` are both `true`.
- `a || b` is `true` if and only if one of `a` or `b` is `true`.

Comparison Operators

- `a < b` 
- `a > b` 
- `a <= b` 
- `a >= b` 
- `a == b` equals 
- `a != b` not equals
Operator Precedence

\[ x \times y + z = (x\times y) + z \]

<table>
<thead>
<tr>
<th>Precedence</th>
<th>Operator</th>
</tr>
</thead>
<tbody>
<tr>
<td>5</td>
<td>* / %</td>
</tr>
<tr>
<td>4</td>
<td>+ -</td>
</tr>
<tr>
<td>3</td>
<td>== != &lt; &lt;= &gt; &gt;=</td>
</tr>
<tr>
<td>2</td>
<td>&amp;&amp;</td>
</tr>
<tr>
<td>1</td>
<td></td>
</tr>
</tbody>
</table>

applied first

applied last

Use () to group operators and change the order they are applied: \( x\times(y+z) \)

Tip: don’t remember the order of operations: always use () to make the order explicit.
Example Expressions

\[ a + \frac{b}{3} + 2 \]

\[ \frac{(a+b)}{3} + 2 \]

\[ -a(3+c - d) \]
Strings & The Motivation for Types
String Operator

\[ s_1 + s_2 \text{ \hspace{1cm} concatenation} \]

Places one string after another:

```
"hi" + "there" == "hithere"
"what's " + "up" + " doc?" == "what's up doc?"
```

Question: What's the value of `a`?

```
c := 42
a := "hi" + string(c)
```

Answer: this is an error, since "hi" has type `string` and 3 has type `int`. 
String Representations

The name “string” is meant to suggest a sequence of characters strung together.

Hello, World!

Each item in a string is called a character.

Characters are stored as binary numbers (as is all data in the computer).

The type of a variable tells your program how to interpret those numbers as data that makes sense in the world.

- **int** means “interpret these binary digits as an integer”
- **float64** means “interpret these binary digits as a real number”
- **string** means “interpret these binary digits as a sequence of characters”

Question: What's the value of `a`?

```go
a := "hi" + string(42)
```
# ASCII Chart

(You don’t need to know these numbers)

<table>
<thead>
<tr>
<th>Binary</th>
<th>Dec</th>
<th>Glyph</th>
</tr>
</thead>
<tbody>
<tr>
<td>110 0000</td>
<td>96</td>
<td>`</td>
</tr>
<tr>
<td>110 0001</td>
<td>97</td>
<td>a</td>
</tr>
<tr>
<td>110 0010</td>
<td>98</td>
<td>b</td>
</tr>
<tr>
<td>110 0011</td>
<td>99</td>
<td>c</td>
</tr>
<tr>
<td>110 0100</td>
<td>100</td>
<td>d</td>
</tr>
<tr>
<td>110 0101</td>
<td>101</td>
<td>e</td>
</tr>
<tr>
<td>110 0110</td>
<td>102</td>
<td>f</td>
</tr>
<tr>
<td>110 0111</td>
<td>103</td>
<td>g</td>
</tr>
<tr>
<td>110 1000</td>
<td>104</td>
<td>h</td>
</tr>
<tr>
<td>110 1001</td>
<td>105</td>
<td>i</td>
</tr>
<tr>
<td>110 1010</td>
<td>106</td>
<td>j</td>
</tr>
<tr>
<td>110 1011</td>
<td>107</td>
<td>k</td>
</tr>
<tr>
<td>110 1100</td>
<td>108</td>
<td>l</td>
</tr>
<tr>
<td>110 1101</td>
<td>109</td>
<td>m</td>
</tr>
<tr>
<td>110 1110</td>
<td>110</td>
<td>n</td>
</tr>
<tr>
<td>110 1111</td>
<td>111</td>
<td>o</td>
</tr>
<tr>
<td>111 0000</td>
<td>112</td>
<td>p</td>
</tr>
<tr>
<td>111 0001</td>
<td>113</td>
<td>q</td>
</tr>
<tr>
<td>111 0010</td>
<td>114</td>
<td>r</td>
</tr>
<tr>
<td>111 0011</td>
<td>115</td>
<td>s</td>
</tr>
<tr>
<td>111 0100</td>
<td>116</td>
<td>t</td>
</tr>
<tr>
<td>111 0101</td>
<td>117</td>
<td>u</td>
</tr>
<tr>
<td>111 0110</td>
<td>118</td>
<td>v</td>
</tr>
<tr>
<td>111 0111</td>
<td>119</td>
<td>w</td>
</tr>
<tr>
<td>111 1000</td>
<td>120</td>
<td>x</td>
</tr>
<tr>
<td>111 1001</td>
<td>121</td>
<td>y</td>
</tr>
<tr>
<td>111 1010</td>
<td>122</td>
<td>z</td>
</tr>
<tr>
<td>111 1011</td>
<td>123</td>
<td></td>
</tr>
<tr>
<td>111 1100</td>
<td>124</td>
<td></td>
</tr>
<tr>
<td>111 1101</td>
<td>125</td>
<td></td>
</tr>
<tr>
<td>111 1110</td>
<td>126</td>
<td>~</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Binary</th>
<th>Dec</th>
<th>Glyph</th>
</tr>
</thead>
<tbody>
<tr>
<td>100 0000</td>
<td>64</td>
<td>@</td>
</tr>
<tr>
<td>100 0001</td>
<td>65</td>
<td>A</td>
</tr>
<tr>
<td>100 0010</td>
<td>66</td>
<td>B</td>
</tr>
<tr>
<td>100 0011</td>
<td>67</td>
<td>C</td>
</tr>
<tr>
<td>100 0100</td>
<td>68</td>
<td>D</td>
</tr>
<tr>
<td>100 0101</td>
<td>69</td>
<td>E</td>
</tr>
<tr>
<td>100 0110</td>
<td>70</td>
<td>F</td>
</tr>
<tr>
<td>100 0111</td>
<td>71</td>
<td>G</td>
</tr>
<tr>
<td>100 1000</td>
<td>72</td>
<td>H</td>
</tr>
<tr>
<td>100 1001</td>
<td>73</td>
<td>I</td>
</tr>
<tr>
<td>100 1010</td>
<td>74</td>
<td>J</td>
</tr>
<tr>
<td>100 1011</td>
<td>75</td>
<td>K</td>
</tr>
<tr>
<td>100 1100</td>
<td>76</td>
<td>L</td>
</tr>
<tr>
<td>100 1101</td>
<td>77</td>
<td>M</td>
</tr>
<tr>
<td>100 1110</td>
<td>78</td>
<td>N</td>
</tr>
<tr>
<td>100 1111</td>
<td>79</td>
<td>O</td>
</tr>
<tr>
<td>101 0000</td>
<td>80</td>
<td>P</td>
</tr>
<tr>
<td>101 0001</td>
<td>81</td>
<td>Q</td>
</tr>
<tr>
<td>101 0010</td>
<td>82</td>
<td>R</td>
</tr>
<tr>
<td>101 0011</td>
<td>83</td>
<td>S</td>
</tr>
<tr>
<td>101 0100</td>
<td>84</td>
<td>T</td>
</tr>
<tr>
<td>101 0101</td>
<td>85</td>
<td>U</td>
</tr>
<tr>
<td>101 0110</td>
<td>86</td>
<td>V</td>
</tr>
<tr>
<td>101 0111</td>
<td>87</td>
<td>W</td>
</tr>
<tr>
<td>101 1000</td>
<td>88</td>
<td>X</td>
</tr>
<tr>
<td>101 1001</td>
<td>89</td>
<td>Y</td>
</tr>
<tr>
<td>101 1010</td>
<td>90</td>
<td>Z</td>
</tr>
<tr>
<td>101 1011</td>
<td>91</td>
<td></td>
</tr>
<tr>
<td>101 1100</td>
<td>92</td>
<td>\</td>
</tr>
<tr>
<td>101 1101</td>
<td>93</td>
<td></td>
</tr>
<tr>
<td>101 1110</td>
<td>94</td>
<td>^</td>
</tr>
<tr>
<td>101 1111</td>
<td>95</td>
<td>_</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Binary</th>
<th>Dec</th>
<th>Glyph</th>
</tr>
</thead>
<tbody>
<tr>
<td>010 0000</td>
<td>32</td>
<td>(space)</td>
</tr>
<tr>
<td>010 0001</td>
<td>33</td>
<td>!</td>
</tr>
<tr>
<td>010 0010</td>
<td>34</td>
<td>*</td>
</tr>
<tr>
<td>010 0011</td>
<td>35</td>
<td>#</td>
</tr>
<tr>
<td>010 0100</td>
<td>36</td>
<td>$</td>
</tr>
<tr>
<td>010 0101</td>
<td>37</td>
<td>%</td>
</tr>
<tr>
<td>010 0110</td>
<td>38</td>
<td>&amp;</td>
</tr>
<tr>
<td>010 0111</td>
<td>39</td>
<td></td>
</tr>
<tr>
<td>010 1000</td>
<td>40</td>
<td>(</td>
</tr>
<tr>
<td>010 1001</td>
<td>41</td>
<td>)</td>
</tr>
<tr>
<td>010 1010</td>
<td>42</td>
<td></td>
</tr>
<tr>
<td>010 1011</td>
<td>43</td>
<td>+</td>
</tr>
<tr>
<td>010 1100</td>
<td>44</td>
<td>.</td>
</tr>
<tr>
<td>010 1101</td>
<td>45</td>
<td>-</td>
</tr>
<tr>
<td>010 1110</td>
<td>46</td>
<td></td>
</tr>
<tr>
<td>011 0000</td>
<td>48</td>
<td>0</td>
</tr>
<tr>
<td>011 0001</td>
<td>49</td>
<td>1</td>
</tr>
<tr>
<td>011 0010</td>
<td>50</td>
<td>2</td>
</tr>
<tr>
<td>011 0011</td>
<td>51</td>
<td>3</td>
</tr>
<tr>
<td>011 0100</td>
<td>52</td>
<td>4</td>
</tr>
<tr>
<td>011 0101</td>
<td>53</td>
<td>5</td>
</tr>
<tr>
<td>011 0110</td>
<td>54</td>
<td>6</td>
</tr>
<tr>
<td>011 0111</td>
<td>55</td>
<td>7</td>
</tr>
<tr>
<td>011 1000</td>
<td>56</td>
<td>8</td>
</tr>
<tr>
<td>011 1001</td>
<td>57</td>
<td>9</td>
</tr>
<tr>
<td>011 1010</td>
<td>58</td>
<td>:</td>
</tr>
<tr>
<td>011 1011</td>
<td>59</td>
<td>;</td>
</tr>
<tr>
<td>011 1100</td>
<td>60</td>
<td>&lt;</td>
</tr>
<tr>
<td>011 1101</td>
<td>61</td>
<td>=</td>
</tr>
<tr>
<td>011 1110</td>
<td>62</td>
<td>&gt;</td>
</tr>
<tr>
<td>011 1111</td>
<td>63</td>
<td>?</td>
</tr>
</tbody>
</table>

**Logical interpretation:**

Representation in the computer:

```
Hello, World!
```

```
72 101 108 108 111 44 32 87 111 114 108 100 33
```
Packages
Packages

• Packages are collections of functions you can use in your program.

• Go provides many built-in packages →

   Enable the use of a package with:

   ```go
   import "packageName"
   ```

   at the top of your program.

• Get a list of built-in packages at: http://golang.org/pkg/

• fmt package provides the fmt.Print and fmt.Println functions we’ve used a lot.
String Package

• Lets you manipulate strings.

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

• This example tests whether one string has another as a substring:

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

Back to our problem: how do we get this:

```go
a := "hi" + `string(42)`
```
to do what we want?

Inside of the strconv package, there are these functions:

- `func FormatBool(b bool) string`
- `func FormatFloat(f float64, fmt byte, prec, bitSize int) string`
- `func FormatUint(i int64, base int) string`
- `func Itoa(i int) string`

FormatInt also allows you to set the base (10, etc.) of the representation.

Using the Itoa() function gives us our desired result:

```go
a := "hi" + strconv.Itoa(42)
```
math Package

Might find:

math.Abs()
math.Pow10()

functions useful for the KthDigit question on homework 1.
Types & Expressions 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).
Go Summary

90% of programming is the combination of these things:

- **functions**: basic building block: define new “things” the computer can do
- **variables**: units of data that your program can manipulate
- **types**: tell Go what kind of data is in each variable
- **if...else**: select statements to execute based on some condition
- **for**: repeat statements while some condition is true