Contents

1 Enumeration of types

1.1 Basic types

- bool
- string
- uint
- uint8
- uint16
- uint32
- uint64
- uintptr
- int
- int8
- int16
- int32
- int64
- float32
- float64
- complex64
- complex128
- byte (alias for uint8)
- rune (alias for int32)

1.2 Composite types

- array
- struct
- pointer
- function
- interface
- slice
- map
- channel

2 Description of basic types

2.1 Boolean

- The predeclared boolean type is `bool`.
- Two predeclared constants: `true` and `false`.

2.2 String

- The `string` type is a sequence of bytes, roughly equivalent to a read-only slice of bytes.
- Strings are immutable and have a constant length.
- The length of the string can be accessed with the builtin `len` function. Note that this returns the number of bytes in the string, not characters.
- The zero-value for strings is `""`.
- See https://blog.golang.org/strings

2.3 Unsigned integer types

2.3.1 Types

- uint
  - Predeclared numeric type. Either 32 or 64 bits.
- uint8
- uint16
- uint32
- uint64
- uintptr
  - An unsigned integer large enough to store the uninterpreted bits of a pointer value. Used to subvert type system.
  - See https://golang.org/pkg/unsafe/

2.3.2 Operators

```
+    sum
-    difference
*    product
/    quotient
%    remainder

&    bitwise AND
|    bitwise OR
^    bitwise XOR
&^   bit clear (AND NOT)

<<   left shift
>>   right shift
```

- The operands MUST be the same type i.e. cannot add uint and uint32, even if they are the same number of bits.

2.3.3 Overflow

- Constant declarations will raise an exception on overflow i.e.

  ```
  var x uint8 = 300
  ```

  will raise an exception.

- The operations +, -, *, and << may overflow unsigned integers. The operations are computed modulo 2^X, with 'X' as in 'uintX'. As a result, integer overflow wraparound may occur.

2.3.4 Notes

- The size (in bits) of integers is 'X' in 'uintX'. 'uint8' is 8 bits, 'uint16' is 16 bits, and so on.
- The range of values for 'uintX' is $[0, 2^X - 1]$.
- The size of 'uint' is either 32 or 64 bits. The size is implementation specific. It can be expected to be the word size of the CPU, but this is NOT guaranteed.
- The zero-value for unsigned integers is `0`.

2.4 Signed integer types

2.4.1 Types

- int
  - Predeclared numeric type. Either 32 or 64 bits.
- int8
- int16
- int32
- int64

2.4.2 Operators

```
+    sum
-    difference
*    product
/    quotient
%    remainder

&    bitwise AND
|    bitwise OR
^    bitwise XOR
&^   bit clear (AND NOT)

<<   left shift (NOTE BELOW)
>>   right shift (NOTE BELOW)
```

- The operands MUST be the same type i.e. cannot add int and int32, even if they are the same number of bits.
- NOTE: The right operand of left and right shift must be an unsigned integer

2.4.3 Overflow

- Constant declarations will raise an exception on overflow i.e.

  ```
  var x int8 = 300
  ```

  will raise an exception.

- The operations +, -, *, and << may legally overflow signed integers.

2.4.4 Notes

- Signed integers are represented using two's-complement.
- The size of 'int' is either 32 or 64 bits. The size is implementation specific. It can be expected to be the word size of the CPU, but this is NOT guaranteed.
- The zero-value for signed integers is `0`.

2.5 Floating-point types

2.5.1 Types

- float32
- float64

2.5.2 Operators

```
+    sum
-    difference
*    product
/    quotient
```

- The operands MUST be the same type i.e. cannot add float32 and float64.

2.5.3 Notes

- Follows the IEEE-754 standard for floating points.
- The size of 'floatX' is 'X' bits.
- The zero-value for floating points is `0.0`.

2.6 Complex-number types

Builtin support for complex numbers

2.6.1 Types

- complex32
  - Uses float32 for real and imaginary parts
- complex64
  - Uses float64 for real and imaginary parts

2.6.2 Operators

```
+    sum
-    difference
*    product
/    quotient
```

2.6.3 Notes

- Typically constructed using the builtin `complex` function. See https://golang.org/pkg/builtin/#complex
- Real and imaginary parts can be accessed using the builtin `real` and `imag` functions. See https://golang.org/pkg/builtin/#real and https://golang.org/pkg/builtin/#imag
- Additional function support provided in the "math/cmplx" package. See https://golang.org/pkg/math/cmplx/

2.7 Aliased types

- byte (alias for uint8)
- rune (alias for int32)
  - An integer value identifying a Unicode code point.

3 Description of composite types

3.1 Array

A sequence of elements of a certain type. For example, `[32]int8` is an array of int8 with 32 elements.

- Arrays are copy on assignment. When assigned to a new variable, all elements are copied over.
- Arrays are pass by value. When passed into a function, a new array is created and passed in.
- Upon creation, the elements of an array are set to their zero-value. For example, `[2]float32` will create an array `[0.0, 0.0]`.

3.2 Struct

A collection of named and unnamed fields, each of which has a type. Similar to a C struct.

- Upon creation, the fields of a struct are set to their zero-value.
- See https://golang.org/ref/spec#Struct_types

3.3 Pointer

A pointer to an address in memory for a certain type.

- Does not support pointer arithmetic.
- Can be deferenced with `*`.
- The zero-value for pointers is `nil`.

3.4 Function

- Functions are first-class citizens. They can be assigned to variables, returned from other functions, and so on.
- The zero-value for functions is `nil`.
- For examples, see https://golang.org/doc/codewalk/functions/
- For reference specification, see https://golang.org/ref/spec#Function_types

3.5 Interface

A collection of function signatures that defines an interface.

- A type is said to implement an interface if it has a method for each function signature.
- The empty interface, `interface{}` is implemented by all types.
- The zero-value for interfaces is `nil`.
- See https://golang.org/ref/spec#Interface_types

3.6 Slice

A slice consists of a reference to the underlying array, the length of the slice, and the capacity of the underlying array.

- Use the builtin functions `cap` and `len` to get the capacity and length.
- See https://blog.golang.org/go-slices-usage-and-internals

3.7 Map

Unordered collection of key:value pairs.

- The key type must have `==` and `!=` defined.
- The zero-value for maps is `nil`.
- Typically initialized with the builtin `make` function. For example, `var m map[int]string = make(map[int]string)`
- See https://blog.golang.org/go-maps-in-action

3.8 Channel

Mechanism for message-passing. For application by concurrent applications.

- A channel consists of a direction (send/receive/both), the type it handles, and the capacity for its buffer.
- The zero-value for channels is `nil`.
- See https://golang.org/ref/spec#Channel_types

4 Other Notes

4.1 Zero values

Variables cannot be uninitialized. Whenever a variable is created, it is initialized to the type's zero value. The zero values of the builtin types are listed below.

Type	Zero Value
All integer types	0
Floating points	0.0
Booleans	false
Strings	""
Pointers	nil
Functions	nil
Interfaces	nil
Slices	nil
Channels	nil
Maps	nil

Initialization is recursive, so a struct is initialized with all of it's fields set to their zero values. The same goes for complex types, where the real and imaginary values are set to zero. See https://golang.org/ref/spec#The_zero_value

4.2 Nil

`nil` is the zero-value for reference types, which includes pointers, functions, interfaces, slices, channels, and maps. Nil has no type.

4.3 Error

`error` is the builtin interface type that specifies any error condition. See https://golang.org/pkg/builtin/#error.

4.4 Precedence

```
Precedence    Operator
    5             *  /  %  <<  >>  &  &^
    4             +  -  |  ^
    3             ==  !=  <  <=  >  >=
    2             &&
    1             ||
```

See https://golang.org/ref/spec#Operators

4.5 Memory allocation

When possible, the Go compilers will allocate variables that are local to a function in that function's stack frame. However, if the compiler cannot prove that the variable is not referenced after the function returns, then the compiler must allocate the variable on the garbage-collected heap to avoid dangling pointer errors. See https://golang.org/doc/faq#stack_or_heap

4.6 Type conversions

See https://golang.org/ref/spec#Conversions