2

Program Structure

In Go, as in any other programming language, one builds large programs from a small set of basic constructs. Variables store values. Simple expressions are combined into larger ones with operations like addition and subtraction. Basic types are collected into aggregates like arrays and structs. Expressions are used in statements whose execution order is determined by control-flow statements like `if` and `for`. Statements are grouped into functions for isolation and reuse. Functions are gathered into source files and packages.

We saw examples of most of these in the previous chapter. In this chapter, we'll go into more detail about the basic structural elements of a Go program. The example programs are intentionally simple, so we can focus on the language without getting sidetracked by complicated algorithms or data structures.

2.1. Names

The names of Go functions, variables, constants, types, statement labels, and packages follow a simple rule: a name begins with a letter (that is, anything that Unicode deems a letter) or an underscore and may have any number of additional letters, digits, and underscores. Case matters: `heapSort` and `Heapsort` are different names.

Go has 25 *keywords* like `if` and `switch` that may be used only where the syntax permits; they can't be used as names.

```
break       default      func         interface    select
case        defer        go           map          struct
chan        else         goto         package      switch
const       fallthrough  if           range        type
continue    for          import       return       var
```

In addition, there are about three dozen *predeclared* names like `int` and `true` for built-in constants, types, and functions:

```
Constants:   true false iota nil

Types:       int int8 int16 int32 int64
             uint uint8 uint16 uint32 uint64 uintptr
             float32 float64 complex128 complex64
             bool byte rune string error

Functions:   make len cap new append copy close delete
             complex real imag
             panic recover
```

These names are not reserved, so you may use them in declarations. We'll see a handful of places where redeclaring one of them makes sense, but beware of the potential for confusion.

If an entity is declared within a function, it is *local* to that function. If declared outside of a function, however, it is visible in all files of the package to which it belongs. The case of the first letter of a name determines its visibility across package boundaries. If the name begins with an upper-case letter, it is *exported*, which means that it is visible and accessible outside of its own package and may be referred to by other parts of the program, as with `Printf` in the `fmt` package. Package names themselves are always in lower case.

There is no limit on name length, but convention and style in Go programs lean toward short names, especially for local variables with small scopes; you are much more likely to see variables named `i` than `theLoopIndex`. Generally, the larger the scope of a name, the longer and more meaningful it should be.

Stylistically, Go programmers use "camel case" when forming names by combining words; that is, interior capital letters are preferred over interior underscores. Thus the standard libraries have functions with names like `QuoteRuneToASCII` and `parseRequestLine` but never `quote_rune_to_ASCII` or `parse_request_line`. The letters of acronyms and initialisms like ASCII and HTML are always rendered in the same case, so a function might be called `htmlEscape`, `HTMLEscape`, or `escapeHTML`, but not `escapeHtml`.

2.2. Declarations

A *declaration* names a program entity and specifies some or all of its properties. There are four major kinds of declarations: `var`, `const`, `type`, and `func`. We'll talk about variables and types in this chapter, constants in Chapter 3, and functions in Chapter 5.

A Go program is stored in one or more files whose names end in `.go`. Each file begins with a `package` declaration that says what package the file is part of. The `package` declaration is followed by any `import` declarations, and then a sequence of *package-level* declarations of types, variables, constants, and functions, in any order. For example, this program declares a constant, a function, and a couple of variables:

gopl.io/ch2/boiling

```
// Boiling prints the boiling point of water.
package main

import "fmt"

const boilingF = 212.0

func main() {
    var f = boilingF
    var c = (f - 32) * 5 / 9
    fmt.Printf("boiling point = %g°F or %g°C\n", f, c)
    // Output:
    // boiling point = 212°F or 100°C
}
```

The constant `boilingF` is a package-level declaration (as is `main`), whereas the variables `f` and `c` are local to the function `main`. The name of each package-level entity is visible not only throughout the source file that contains its declaration, but throughout all the files of the package. By contrast, local declarations are visible only within the function in which they are declared and perhaps only within a small part of it.

A function declaration has a name, a list of parameters (the variables whose values are provided by the function's callers), an optional list of results, and the function body, which contains the statements that define what the function does. The result list is omitted if the function does not return anything. Execution of the function begins with the first statement and continues until it encounters a return statement or reaches the end of a function that has no results. Control and any results are then returned to the caller.

We've seen a fair number of functions already and there are lots more to come, including an extensive discussion in Chapter 5, so this is only a sketch. The function `fToC` below encapsulates the temperature conversion logic so that it is defined only once but may be used from multiple places. Here `main` calls it twice, using the values of two different local constants:

gopl.io/ch2/ftoc

```
// Ftoc prints two Fahrenheit-to-Celsius conversions.
package main

import "fmt"

func main() {
    const freezingF, boilingF = 32.0, 212.0
    fmt.Printf("%g°F = %g°C\n", freezingF, fToC(freezingF)) // "32°F = 0°C"
    fmt.Printf("%g°F = %g°C\n", boilingF, fToC(boilingF))   // "212°F = 100°C"
}

func fToC(f float64) float64 {
    return (f - 32) * 5 / 9
}
```

2.3. Variables

A `var` declaration creates a variable of a particular type, attaches a name to it, and sets its initial value. Each declaration has the general form

```
var name type = expression
```

Either the type or the `= expression` part may be omitted, but not both. If the type is omitted, it is determined by the initializer expression. If the expression is omitted, the initial value is the *zero value* for the type, which is `0` for numbers, `false` for booleans, `""` for strings, and `nil` for interfaces and reference types (slice, pointer, map, channel, function). The zero value of an aggregate type like an array or a struct has the zero value of all of its elements or fields.

The zero-value mechanism ensures that a variable always holds a well-defined value of its type; in Go there is no such thing as an uninitialized variable. This simplifies code and often ensures sensible behavior of boundary conditions without extra work. For example,

```
var s string
fmt.Println(s) // ""
```

prints an empty string, rather than causing some kind of error or unpredictable behavior. Go programmers often go to some effort to make the zero value of a more complicated type meaningful, so that variables begin life in a useful state.

It is possible to declare and optionally initialize a set of variables in a single declaration, with a matching list of expressions. Omitting the type allows declaration of multiple variables of different types:

```
var i, j, k int                 // int, int, int
var b, f, s = true, 2.3, "four" // bool, float64, string
```

Initializers may be literal values or arbitrary expressions. Package-level variables are initialized before `main` begins (§2.6.2), and local variables are initialized as their declarations are encountered during function execution.

A set of variables can also be initialized by calling a function that returns multiple values:

```
var f, err = os.Open(name) // os.Open returns a file and an error
```

2.3.1. Short Variable Declarations

Within a function, an alternate form called a *short variable declaration* may be used to declare and initialize local variables. It takes the form `name := expression`, and the type of `name` is determined by the type of `expression`. Here are three of the many short variable declarations in the `lissajous` function (§1.4):

```
anim := gif.GIF{LoopCount: nframes}
freq := rand.Float64() * 3.0
t := 0.0
```

Because of their brevity and flexibility, short variable declarations are used to declare and initialize the majority of local variables. A `var` declaration tends to be reserved for local variables that need an explicit type that differs from that of the initializer expression, or for when the variable will be assigned a value later and its initial value is unimportant.

```
i := 100                  // an int
var boiling float64 = 100 // a float64

var names []string
var err error
var p Point
```

As with `var` declarations, multiple variables may be declared and initialized in the same short variable declaration,

```
i, j := 0, 1
```

but declarations with multiple initializer expressions should be used only when they help readability, such as for short and natural groupings like the initialization part of a `for` loop.

Keep in mind that `:=` is a declaration, whereas `=` is an assignment. A multi-variable declaration should not be confused with a *tuple assignment* (§2.4.1), in which each variable on the left-hand side is assigned the corresponding value from the right-hand side:

```
i, j = j, i // swap values of i and j
```

Like ordinary `var` declarations, short variable declarations may be used for calls to functions like `os.Open` that return two or more values:

```
f, err := os.Open(name)
if err != nil {
    return err
}
// ...use f...
f.Close()
```

One subtle but important point: a short variable declaration does not necessarily *declare* all the variables on its left-hand side. If some of them were already declared in the *same* lexical block (§2.7), then the short variable declaration acts like an *assignment* to those variables.

In the code below, the first statement declares both `in` and `err`. The second declares `out` but only assigns a value to the existing `err` variable.

```
in, err := os.Open(infile)
// ...
out, err := os.Create(outfile)
```

A short variable declaration must declare at least one new variable, however, so this code will not compile:

```
f, err := os.Open(infile)
// ...
f, err := os.Create(outfile) // compile error: no new variables
```

The fix is to use an ordinary assignment for the second statement.

A short variable declaration acts like an assignment only to variables that were already declared in the same lexical block; declarations in an outer block are ignored. We'll see examples of this at the end of the chapter.

2.3.2. Pointers

A *variable* is a piece of storage containing a value. Variables created by declarations are identified by a name, such as `x`, but many variables are identified only by expressions like `x[i]` or `x.f`. All these expressions read the value of a variable, except when they appear on the left-hand side of an assignment, in which case a new value is assigned to the variable.

A *pointer* value is the *address* of a variable. A pointer is thus the location at which a value is stored. Not every value has an address, but every variable does. With a pointer, we can read or update the value of a variable *indirectly*, without using or even knowing the name of the variable, if indeed it has a name.

If a variable is declared `var x int`, the expression `&x` ("address of `x`") yields a pointer to an integer variable, that is, a value of type `*int`, which is pronounced "pointer to int." If this value is called `p`, we say "`p` points to `x`," or equivalently "`p` contains the address of `x`." The variable to which `p` points is written `*p`. The expression `*p` yields the value of that variable, an `int`, but since `*p` denotes a variable, it may also appear on the left-hand side of an assignment, in which case the assignment updates the variable.

```
x := 1
p := &x         // p, of type *int, points to x
fmt.Println(*p) // "1"
*p = 2          // equivalent to x = 2
fmt.Println(x)  // "2"
```

Each component of a variable of aggregate type—a field of a struct or an element of an array—is also a variable and thus has an address too.

Variables are sometimes described as *addressable* values. Expressions that denote variables are the only expressions to which the *address-of* operator `&` may be applied.

The zero value for a pointer of any type is `nil`. The test `p != nil` is true if `p` points to a variable. Pointers are comparable; two pointers are equal if and only if they point to the same variable or both are nil.

```
var x, y int
fmt.Println(&x == &x, &x == &y, &x == nil) // "true false false"
```

It is perfectly safe for a function to return the address of a local variable. For instance, in the code below, the local variable `v` created by this particular call to `f` will remain in existence even after the call has returned, and the pointer `p` will still refer to it:

```
var p = f()

func f() *int {
    v := 1
    return &v
}
```

Each call of `f` returns a distinct value:

```
fmt.Println(f() == f()) // "false"
```

Because a pointer contains the address of a variable, passing a pointer argument to a function makes it possible for the function to update the variable that was indirectly passed. For example, this function increments the variable that its argument points to and returns the new value of the variable so it may be used in an expression:

```
func incr(p *int) int {
    *p++ // increments what p points to; does not change p
    return *p
}

v := 1
incr(&v)              // side effect: v is now 2
fmt.Println(incr(&v)) // "3" (and v is 3)
```

Each time we take the address of a variable or copy a pointer, we create new *aliases* or ways to identify the same variable. For example, `*p` is an alias for `v`. Pointer aliasing is useful because it allows us to access a variable without using its name, but this is a double-edged sword: to find all the statements that access a variable, we have to know all its aliases. It's not just pointers that create aliases; aliasing also occurs when we copy values of other reference types like slices, maps, and channels, and even structs, arrays, and interfaces that contain these types.

Pointers are key to the `flag` package, which uses a program's command-line arguments to set the values of certain variables distributed throughout the program. To illustrate, this variation on the earlier `echo` command takes two optional flags: `-n` causes `echo` to omit the trailing newline that would normally be printed, and `-s sep` causes it to separate the output arguments by the contents of the string `sep` instead of the default single space. Since this is our fourth version, the package is called `gopl.io/ch2/echo4`.

gopl.io/ch2/echo4

```
// Echo4 prints its command-line arguments.
package main

import (
    "flag"
    "fmt"
    "strings"
)

var n = flag.Bool("n", false, "omit trailing newline")
var sep = flag.String("s", " ", "separator")
```

```
func main() {
    flag.Parse()
    fmt.Print(strings.Join(flag.Args(), *sep))
    if !*n {
        fmt.Println()
    }
}
```

The function `flag.Bool` creates a new flag variable of type `bool`. It takes three arguments: the name of the flag (`"n"`), the variable's default value (`false`), and a message that will be printed if the user provides an invalid argument, an invalid flag, or `-h` or `-help`. Similarly, `flag.String` takes a name, a default value, and a message, and creates a `string` variable. The variables `sep` and `n` are pointers to the flag variables, which must be accessed indirectly as `*sep` and `*n`.

When the program is run, it must call `flag.Parse` before the flags are used, to update the flag variables from their default values. The non-flag arguments are available from `flag.Args()` as a slice of strings. If `flag.Parse` encounters an error, it prints a usage message and calls `os.Exit(2)` to terminate the program.

Let's run some test cases on `echo`:

```
$ go build gopl.io/ch2/echo4
$ ./echo4 a bc def
a bc def
$ ./echo4 -s / a bc def
a/bc/def
$ ./echo4 -n a bc def
a bc def$
$ ./echo4 -help
Usage of ./echo4:
  -n    omit trailing newline
  -s string
        separator (default " ")
```

2.3.3. The new Function

Another way to create a variable is to use the built-in function `new`. The expression `new(T)` creates an *unnamed variable* of type `T`, initializes it to the zero value of `T`, and returns its address, which is a value of type `*T`.

```
p := new(int)   // p, of type *int, points to an unnamed int variable
fmt.Println(*p) // "0"
*p = 2          // sets the unnamed int to 2
fmt.Println(*p) // "2"
```

A variable created with `new` is no different from an ordinary local variable whose address is taken, except that there's no need to invent (and declare) a dummy name, and we can use `new(T)` in an expression. Thus `new` is only a syntactic convenience, not a fundamental notion:

the two `newInt` functions below have identical behaviors.

```
func newInt() *int {
    return new(int)
}
```

```
func newInt() *int {
    var dummy int
    return &dummy
}
```

Each call to `new` returns a distinct variable with a unique address:

```
p := new(int)
q := new(int)
fmt.Println(p == q) // "false"
```

There is one exception to this rule: two variables whose type carries no information and is therefore of size zero, such as `struct{}` or `[0]int`, may, depending on the implementation, have the same address.

The `new` function is relatively rarely used because the most common unnamed variables are of struct types, for which the struct literal syntax (§4.4.1) is more flexible.

Since `new` is a predeclared function, not a keyword, it's possible to redefine the name for something else within a function, for example:

```
func delta(old, new int) int { return new - old }
```

Of course, within `delta`, the built-in `new` function is unavailable.

2.3.4. Lifetime of Variables

The *lifetime* of a variable is the interval of time during which it exists as the program executes. The lifetime of a package-level variable is the entire execution of the program. By contrast, local variables have dynamic lifetimes: a new instance is created each time the declaration statement is executed, and the variable lives on until it becomes *unreachable*, at which point its storage may be recycled. Function parameters and results are local variables too; they are created each time their enclosing function is called.

For example, in this excerpt from the Lissajous program of Section 1.4,

```
for t := 0.0; t < cycles*2*math.Pi; t += res {
    x := math.Sin(t)
    y := math.Sin(t*freq + phase)
    img.SetColorIndex(size+int(x*size+0.5), size+int(y*size+0.5),
        blackIndex)
}
```

the variable `t` is created each time the `for` loop begins, and new variables `x` and `y` are created on each iteration of the loop.

How does the garbage collector know that a variable's storage can be reclaimed? The full story is much more detailed than we need here, but the basic idea is that every package-level variable, and every local variable of each currently active function, can potentially be the start or

root of a path to the variable in question, following pointers and other kinds of references that ultimately lead to the variable. If no such path exists, the variable has become unreachable, so it can no longer affect the rest of the computation.

Because the lifetime of a variable is determined only by whether or not it is reachable, a local variable may outlive a single iteration of the enclosing loop. It may continue to exist even after its enclosing function has returned.

A compiler may choose to allocate local variables on the heap or on the stack but, perhaps surprisingly, this choice is not determined by whether `var` or `new` was used to declare the variable.

```
var global *int

func f() {
    var x int
    x = 1
    global = &x
}

func g() {
    y := new(int)
    *y = 1
}
```

Here, `x` must be heap-allocated because it is still reachable from the variable `global` after `f` has returned, despite being declared as a local variable; we say `x` *escapes from* `f`. Conversely, when `g` returns, the variable `*y` becomes unreachable and can be recycled. Since `*y` does not escape from `g`, it's safe for the compiler to allocate `*y` on the stack, even though it was allocated with `new`. In any case, the notion of escaping is not something that you need to worry about in order to write correct code, though it's good to keep in mind during performance optimization, since each variable that escapes requires an extra memory allocation.

Garbage collection is a tremendous help in writing correct programs, but it does not relieve you of the burden of thinking about memory. You don't need to explicitly allocate and free memory, but to write efficient programs you still need to be aware of the lifetime of variables. For example, keeping unnecessary pointers to short-lived objects within long-lived objects, especially global variables, will prevent the garbage collector from reclaiming the short-lived objects.

2.4. Assignments

The value held by a variable is updated by an assignment statement, which in its simplest form has a variable on the left of the = sign and an expression on the right.

```
x = 1                       // named variable
*p = true                   // indirect variable
person.name = "bob"         // struct field
count[x] = count[x] * scale // array or slice or map element
```

Each of the arithmetic and bitwise binary operators has a corresponding *assignment operator* allowing, for example, the last statement to be rewritten as

```
count[x] *= scale
```

which saves us from having to repeat (and re-evaluate) the expression for the variable.

Numeric variables can also be incremented and decremented by `++` and `--` statements:

```
v := 1
v++    // same as v = v + 1; v becomes 2
v--    // same as v = v - 1; v becomes 1 again
```

2.4.1. Tuple Assignment

Another form of assignment, known as *tuple assignment*, allows several variables to be assigned at once. All of the right-hand side expressions are evaluated before any of the variables are updated, making this form most useful when some of the variables appear on both sides of the assignment, as happens, for example, when swapping the values of two variables:

```
x, y = y, x

a[i], a[j] = a[j], a[i]
```

or when computing the greatest common divisor (GCD) of two integers:

```
func gcd(x, y int) int {
    for y != 0 {
        x, y = y, x%y
    }
    return x
}
```

or when computing the n-th Fibonacci number iteratively:

```
func fib(n int) int {
    x, y := 0, 1
    for i := 0; i < n; i++ {
        x, y = y, x+y
    }
    return x
}
```

Tuple assignment can also make a sequence of trivial assignments more compact,

```
i, j, k = 2, 3, 5
```

though as a matter of style, avoid the tuple form if the expressions are complex; a sequence of separate statements is easier to read.

Certain expressions, such as a call to a function with multiple results, produce several values. When such a call is used in an assignment statement, the left-hand side must have as many variables as the function has results.

```
f, err = os.Open("foo.txt")  // function call returns two values
```

Often, functions use these additional results to indicate some kind of error, either by returning an `error` as in the call to `os.Open`, or a `bool`, usually called `ok`. As we'll see in later chapters,

there are three operators that sometimes behave this way too. If a map lookup (§4.3), type assertion (§7.10), or channel receive (§8.4.2) appears in an assignment in which two results are expected, each produces an additional boolean result:

```
v, ok = m[key]             // map lookup
v, ok = x.(T)              // type assertion
v, ok = <-ch               // channel receive
```

As with variable declarations, we can assign unwanted values to the blank identifier:

```
_, err = io.Copy(dst, src) // discard byte count
_, ok = x.(T)              // check type but discard result
```

2.4.2. Assignability

Assignment statements are an explicit form of assignment, but there are many places in a program where an assignment occurs *implicitly*: a function call implicitly assigns the argument values to the corresponding parameter variables; a `return` statement implicitly assigns the `return` operands to the corresponding result variables; and a literal expression for a composite type (§4.2) such as this slice:

```
medals := []string{"gold", "silver", "bronze"}
```

implicitly assigns each element, as if it had been written like this:

```
medals[0] = "gold"
medals[1] = "silver"
medals[2] = "bronze"
```

The elements of maps and channels, though not ordinary variables, are also subject to similar implicit assignments.

An assignment, explicit or implicit, is always legal if the left-hand side (the variable) and the right-hand side (the value) have the same type. More generally, the assignment is legal only if the value is *assignable* to the type of the variable.

The rule for *assignability* has cases for various types, so we'll explain the relevant case as we introduce each new type. For the types we've discussed so far, the rules are simple: the types must exactly match, and `nil` may be assigned to any variable of interface or reference type. Constants (§3.6) have more flexible rules for assignability that avoid the need for most explicit conversions.

Whether two values may be compared with `==` and `!=` is related to assignability: in any comparison, the first operand must be assignable to the type of the second operand, or vice versa. As with assignability, we'll explain the relevant cases for *comparability* when we present each new type.

2.5. Type Declarations

The type of a variable or expression defines the characteristics of the values it may take on, such as their size (number of bits or number of elements, perhaps), how they are represented internally, the intrinsic operations that can be performed on them, and the methods associated with them.

In any program there are variables that share the same representation but signify very different concepts. For instance, an `int` could be used to represent a loop index, a timestamp, a file descriptor, or a month; a `float64` could represent a velocity in meters per second or a temperature in one of several scales; and a `string` could represent a password or the name of a color.

A `type` declaration defines a new *named type* that has the same *underlying type* as an existing type. The named type provides a way to separate different and perhaps incompatible uses of the underlying type so that they can't be mixed unintentionally.

```
type name underlying-type
```

Type declarations most often appear at package level, where the named type is visible throughout the package, and if the name is exported (it starts with an upper-case letter), it's accessible from other packages as well.

To illustrate type declarations, let's turn the different temperature scales into different types:

gopl.io/ch2/tempconv0

```
// Package tempconv performs Celsius and Fahrenheit temperature computations.
package tempconv

import "fmt"

type Celsius float64
type Fahrenheit float64

const (
    AbsoluteZeroC Celsius = -273.15
    FreezingC     Celsius = 0
    BoilingC      Celsius = 100
)

func CToF(c Celsius) Fahrenheit { return Fahrenheit(c*9/5 + 32) }

func FToC(f Fahrenheit) Celsius { return Celsius((f - 32) * 5 / 9) }
```

This package defines two types, `Celsius` and `Fahrenheit`, for the two units of temperature. Even though both have the same underlying type, `float64`, they are not the same type, so they cannot be compared or combined in arithmetic expressions. Distinguishing the types makes it possible to avoid errors like inadvertently combining temperatures in the two different scales; an explicit type *conversion* like `Celsius(t)` or `Fahrenheit(t)` is required to convert from a `float64`. `Celsius(t)` and `Fahrenheit(t)` are conversions, not function calls. They don't change the value or representation in any way, but they make the change of meaning explicit. On the other hand, the functions `CToF` and `FToC` convert between the two scales; they *do* return different values.

For every type `T`, there is a corresponding conversion operation `T(x)` that converts the value `x` to type `T`. A conversion from one type to another is allowed if both have the same underlying type, or if both are unnamed pointer types that point to variables of the same underlying type; these conversions change the type but not the representation of the value. If `x` is assignable to `T`, a conversion is permitted but is usually redundant,

Conversions are also allowed between numeric types, and between string and some slice types, as we will see in the next chapter. These conversions may change the representation of the value. For instance, converting a floating-point number to an integer discards any fractional part, and converting a string to a `[]byte` slice allocates a copy of the string data. In any case, a conversion never fails at run time.

The underlying type of a named type determines its structure and representation, and also the set of intrinsic operations it supports, which are the same as if the underlying type had been used directly. That means that arithmetic operators work the same for `Celsius` and `Fahrenheit` as they do for `float64`, as you might expect.

```
fmt.Printf("%g\n", BoilingC-FreezingC) // "100" °C
boilingF := CToF(BoilingC)
fmt.Printf("%g\n", boilingF-CToF(FreezingC)) // "180" °F
fmt.Printf("%g\n", boilingF-FreezingC)       // compile error: type mismatch
```

Comparison operators like `==` and `<` can also be used to compare a value of a named type to another of the same type, or to a value of the underlying type. But two values of different named types cannot be compared directly:

```
var c Celsius
var f Fahrenheit
fmt.Println(c == 0)          // "true"
fmt.Println(f >= 0)          // "true"
fmt.Println(c == f)          // compile error: type mismatch
fmt.Println(c == Celsius(f)) // "true"!
```

Note the last case carefully. In spite of its name, the type conversion `Celsius(f)` does not change the value of its argument, just its type. The test is true because `c` and `f` are both zero.

A named type may provide notational convenience if it helps avoid writing out complex types over and over again. The advantage is small when the underlying type is simple like `float64`, but big for complicated types, as we will see when we discuss structs.

Named types also make it possible to define new behaviors for values of the type. These behaviors are expressed as a set of functions associated with the type, called the type's *methods*. We'll look at methods in detail in Chapter 6 but will give a taste of the mechanism here.

The declaration below, in which the `Celsius` parameter `c` appears before the function name, associates with the `Celsius` type a method named `String` that returns `c`'s numeric value followed by `°C`:

```
func (c Celsius) String() string { return fmt.Sprintf("%g°C", c) }
```

Many types declare a `String` method of this form because it controls how values of the type appear when printed as a string by the `fmt` package, as we will see in Section 7.1.

```
c := FToC(212.0)
fmt.Println(c.String()) // "100°C"
fmt.Printf("%v\n", c)   // "100°C"; no need to call String explicitly
fmt.Printf("%s\n", c)   // "100°C"
fmt.Println(c)          // "100°C"
fmt.Printf("%g\n", c)   // "100"; does not call String
fmt.Println(float64(c)) // "100"; does not call String
```

2.6. Packages and Files

Packages in Go serve the same purposes as libraries or modules in other languages, supporting modularity, encapsulation, separate compilation, and reuse. The source code for a package resides in one or more `.go` files, usually in a directory whose name ends with the import path; for instance, the files of the `gopl.io/ch1/helloworld` package are stored in directory `$GOPATH/src/gopl.io/ch1/helloworld`.

Each package serves as a separate *name space* for its declarations. Within the `image` package, for example, the identifier `Decode` refers to a different function than does the same identifier in the `unicode/utf16` package. To refer to a function from outside its package, we must *qualify* the identifier to make explicit whether we mean `image.Decode` or `utf16.Decode`.

Packages also let us hide information by controlling which names are visible outside the package, or *exported*. In Go, a simple rule governs which identifiers are exported and which are not: exported identifiers start with an upper-case letter.

To illustrate the basics, suppose that our temperature conversion software has become popular and we want to make it available to the Go community as a new package. How do we do that?

Let's create a package called `gopl.io/ch2/tempconv`, a variation on the previous example. (Here we've made an exception to our usual rule of numbering examples in sequence, so that the package path can be more realistic.) The package itself is stored in two files to show how declarations in separate files of a package are accessed; in real life, a tiny package like this would need only one file.

We have put the declarations of the types, their constants, and their methods in `tempconv.go`:

gopl.io/ch2/tempconv

```
// Package tempconv performs Celsius and Fahrenheit conversions.
package tempconv

import "fmt"

type Celsius float64
type Fahrenheit float64
```

```
const (
    AbsoluteZeroC Celsius = -273.15
    FreezingC     Celsius = 0
    BoilingC      Celsius = 100
)

func (c Celsius) String() string    { return fmt.Sprintf("%g°C", c) }
func (f Fahrenheit) String() string { return fmt.Sprintf("%g°F", f) }
```

and the conversion functions in `conv.go`:

```
package tempconv

// CToF converts a Celsius temperature to Fahrenheit.
func CToF(c Celsius) Fahrenheit { return Fahrenheit(c*9/5 + 32) }

// FToC converts a Fahrenheit temperature to Celsius.
func FToC(f Fahrenheit) Celsius { return Celsius((f - 32) * 5 / 9) }
```

Each file starts with a `package` declaration that defines the package name. When the package is imported, its members are referred to as `tempconv.CToF` and so on. Package-level names like the types and constants declared in one file of a package are visible to all the other files of the package, as if the source code were all in a single file. Note that `tempconv.go` imports `fmt`, but `conv.go` does not, because it does not use anything from `fmt`.

Because the package-level `const` names begin with upper-case letters, they too are accessible with qualified names like `tempconv.AbsoluteZeroC`:

```
fmt.Printf("Brrrr! %v\n", tempconv.AbsoluteZeroC) // "Brrrr! -273.15°C"
```

To convert a Celsius temperature to Fahrenheit in a package that imports `gopl.io/ch2/tempconv`, we can write the following code:

```
fmt.Println(tempconv.CToF(tempconv.BoilingC)) // "212°F"
```

The *doc comment* (§10.7.4) immediately preceding the `package` declaration documents the package as a whole. Conventionally, it should start with a summary sentence in the style illustrated. Only one file in each package should have a package doc comment. Extensive doc comments are often placed in a file of their own, conventionally called `doc.go`.

Exercise 2.1: Add types, constants, and functions to `tempconv` for processing temperatures in the Kelvin scale, where zero Kelvin is −273.15°C and a difference of 1K has the same magnitude as 1°C.

2.6.1. Imports

Within a Go program, every package is identified by a unique string called its *import path*. These are the strings that appear in an `import` declaration like `"gopl.io/ch2/tempconv"`. The language specification doesn't define where these strings come from or what they mean; it's up to the tools to interpret them. When using the `go` tool (Chapter 10), an import path denotes a directory containing one or more Go source files that together make up the package.

In addition to its import path, each package has a *package name*, which is the short (and not necessarily unique) name that appears in its `package` declaration. By convention, a package's name matches the last segment of its import path, making it easy to predict that the package name of `gopl.io/ch2/tempconv` is `tempconv`.

To use `gopl.io/ch2/tempconv`, we must import it:

gopl.io/ch2/cf

```
// Cf converts its numeric argument to Celsius and Fahrenheit.
package main

import (
    "fmt"
    "os"
    "strconv"

    "gopl.io/ch2/tempconv"
)

func main() {
    for _, arg := range os.Args[1:] {
        t, err := strconv.ParseFloat(arg, 64)
        if err != nil {
            fmt.Fprintf(os.Stderr, "cf: %v\n", err)
            os.Exit(1)
        }
        f := tempconv.Fahrenheit(t)
        c := tempconv.Celsius(t)
        fmt.Printf("%s = %s, %s = %s\n",
            f, tempconv.FToC(f), c, tempconv.CToF(c))
    }
}
```

The import declaration binds a short name to the imported package that may be used to refer to its contents throughout the file. The `import` above lets us refer to names within `gopl.io/ch2/tempconv` by using a *qualified identifier* like `tempconv.CToF`. By default, the short name is the package name—`tempconv` in this case—but an import declaration may specify an alternative name to avoid a conflict (§10.3).

The `cf` program converts a single numeric command-line argument to its value in both Celsius and Fahrenheit:

```
$ go build gopl.io/ch2/cf
$ ./cf 32
32°F = 0°C, 32°C = 89.6°F
$ ./cf 212
212°F = 100°C, 212°C = 413.6°F
$ ./cf -40
-40°F = -40°C, -40°C = -40°F
```

It is an error to import a package and then not refer to it. This check helps eliminate dependencies that become unnecessary as the code evolves, although it can be a nuisance during

debugging, since commenting out a line of code like `log.Print("got here!")` may remove the sole reference to the package name `log`, causing the compiler to emit an error. In this situation, you need to comment out or delete the unnecessary `import`.

Better still, use the `golang.org/x/tools/cmd/goimports` tool, which automatically inserts and removes packages from the import declaration as necessary; most editors can be configured to run `goimports` each time you save a file. Like the `gofmt` tool, it also pretty-prints Go source files in the canonical format.

Exercise 2.2: Write a general-purpose unit-conversion program analogous to `cf` that reads numbers from its command-line arguments or from the standard input if there are no arguments, and converts each number into units like temperature in Celsius and Fahrenheit, length in feet and meters, weight in pounds and kilograms, and the like.

2.6.2. Package Initialization

Package initialization begins by initializing package-level variables in the order in which they are declared, except that dependencies are resolved first:

```
var a = b + c     // a initialized third, to 3
var b = f()       // b initialized second, to 2, by calling f
var c = 1         // c initialized first, to 1

func f() int { return c + 1 }
```

If the package has multiple `.go` files, they are initialized in the order in which the files are given to the compiler; the `go` tool sorts `.go` files by name before invoking the compiler.

Each variable declared at package level starts life with the value of its initializer expression, if any, but for some variables, like tables of data, an initializer expression may not be the simplest way to set its initial value. In that case, the `init` function mechanism may be simpler. Any file may contain any number of functions whose declaration is just

```
func init() { /* ... */ }
```

Such `init` functions can't be called or referenced, but otherwise they are normal functions. Within each file, `init` functions are automatically executed when the program starts, in the order in which they are declared.

One package is initialized at a time, in the order of imports in the program, dependencies first, so a package `p` importing `q` can be sure that `q` is fully initialized before `p`'s initialization begins. Initialization proceeds from the bottom up; the `main` package is the last to be initialized. In this manner, all packages are fully initialized before the application's `main` function begins.

The package below defines a function `PopCount` that returns the number of set bits, that is, bits whose value is 1, in a `uint64` value, which is called its *population count*. It uses an `init` function to precompute a table of results, `pc`, for each possible 8-bit value so that the `PopCount` function needn't take 64 steps but can just return the sum of eight table lookups. (This is definitely *not* the fastest algorithm for counting bits, but it's convenient for illustrating `init`

functions, and for showing how to precompute a table of values, which is often a useful programming technique.)

gopl.io/ch2/popcount

```
package popcount

// pc[i] is the population count of i.
var pc [256]byte

func init() {
    for i := range pc {
        pc[i] = pc[i/2] + byte(i&1)
    }
}

// PopCount returns the population count (number of set bits) of x.
func PopCount(x uint64) int {
    return int(pc[byte(x>>(0*8))] +
        pc[byte(x>>(1*8))] +
        pc[byte(x>>(2*8))] +
        pc[byte(x>>(3*8))] +
        pc[byte(x>>(4*8))] +
        pc[byte(x>>(5*8))] +
        pc[byte(x>>(6*8))] +
        pc[byte(x>>(7*8))])
}
```

Note that the `range` loop in `init` uses only the index; the value is unnecessary and thus need not be included. The loop could also have been written as

```
for i, _ := range pc {
```

We'll see other uses of `init` functions in the next section and in Section 10.5.

Exercise 2.3: Rewrite `PopCount` to use a loop instead of a single expression. Compare the performance of the two versions. (Section 11.4 shows how to compare the performance of different implementations systematically.)

Exercise 2.4: Write a version of `PopCount` that counts bits by shifting its argument through 64 bit positions, testing the rightmost bit each time. Compare its performance to the table-lookup version.

Exercise 2.5: The expression `x&(x-1)` clears the rightmost non-zero bit of `x`. Write a version of `PopCount` that counts bits by using this fact, and assess its performance.

2.7. Scope

A declaration associates a name with a program entity, such as a function or a variable. The *scope* of a declaration is the part of the source code where a use of the declared name refers to that declaration.

Don't confuse scope with lifetime. The scope of a declaration is a region of the program text; it is a compile-time property. The lifetime of a variable is the range of time during execution when the variable can be referred to by other parts of the program; it is a run-time property.

A syntactic *block* is a sequence of statements enclosed in braces like those that surround the body of a function or loop. A name declared inside a syntactic block is not visible outside that block. The block encloses its declarations and determines their scope. We can generalize this notion of blocks to include other groupings of declarations that are not explicitly surrounded by braces in the source code; we'll call them all *lexical blocks*. There is a lexical block for the entire source code, called the *universe block*; for each package; for each file; for each `for`, `if`, and `switch` statement; for each case in a `switch` or `select` statement; and, of course, for each explicit syntactic block.

A declaration's lexical block determines its scope, which may be large or small. The declarations of built-in types, functions, and constants like `int`, `len`, and `true` are in the universe block and can be referred to throughout the entire program. Declarations outside any function, that is, at *package level*, can be referred to from any file in the same package. Imported packages, such as `fmt` in the `tempconv` example, are declared at the *file level*, so they can be referred to from the same file, but not from another file in the same package without another `import`. Many declarations, like that of the variable `c` in the `tempconv.CToF` function, are *local*, so they can be referred to only from within the same function or perhaps just a part of it.

The scope of a control-flow label, as used by `break`, `continue`, and `goto` statements, is the entire enclosing function.

A program may contain multiple declarations of the same name so long as each declaration is in a different lexical block. For example, you can declare a local variable with the same name as a package-level variable. Or, as shown in Section 2.3.3, you can declare a function parameter called `new`, even though a function of this name is predeclared in the universe block. Don't overdo it, though; the larger the scope of the redeclaration, the more likely you are to surprise the reader.

When the compiler encounters a reference to a name, it looks for a declaration, starting with the innermost enclosing lexical block and working up to the universe block. If the compiler finds no declaration, it reports an "undeclared name" error. If a name is declared in both an outer block and an inner block, the inner declaration will be found first. In that case, the inner declaration is said to *shadow* or *hide* the outer one, making it inaccessible:

```
func f() {}

var g = "g"

func main() {
    f := "f"
    fmt.Println(f) // "f"; local var f shadows package-level func f
    fmt.Println(g) // "g"; package-level var
    fmt.Println(h) // compile error: undefined: h
}
```

Within a function, lexical blocks may be nested to arbitrary depth, so one local declaration can shadow another. Most blocks are created by control-flow constructs like `if` statements and `for` loops. The program below has three different variables called `x` because each declaration appears in a different lexical block. (This example illustrates scope rules, not good style!)

```
func main() {
    x := "hello!"
    for i := 0; i < len(x); i++ {
        x := x[i]
        if x != '!' {
            x := x + 'A' - 'a'
            fmt.Printf("%c", x) // "HELLO" (one letter per iteration)
        }
    }
}
```

The expressions `x[i]` and `x + 'A' - 'a'` each refer to a declaration of `x` from an outer block; we'll explain that in a moment. (Note that the latter expression is *not* equivalent to `unicode.ToUpper`.)

As mentioned above, not all lexical blocks correspond to explicit brace-delimited sequences of statements; some are merely implied. The `for` loop above creates two lexical blocks: the explicit block for the loop body, and an implicit block that additionally encloses the variables declared by the initialization clause, such as `i`. The scope of a variable declared in the implicit block is the condition, post-statement (`i++`), and body of the `for` statement.

The example below also has three variables named `x`, each declared in a different block—one in the function body, one in the `for` statement's block, and one in the loop body—but only two of the blocks are explicit:

```
func main() {
    x := "hello"
    for _, x := range x {
        x := x + 'A' - 'a'
        fmt.Printf("%c", x) // "HELLO" (one letter per iteration)
    }
}
```

Like `for` loops, `if` statements and `switch` statements also create implicit blocks in addition to their body blocks. The code in the following `if-else` chain shows the scope of `x` and `y`:

```
if x := f(); x == 0 {
    fmt.Println(x)
} else if y := g(x); x == y {
    fmt.Println(x, y)
} else {
    fmt.Println(x, y)
}
fmt.Println(x, y) // compile error: x and y are not visible here
```

The second `if` statement is nested within the first, so variables declared within the first statement's initializer are visible within the second. Similar rules apply to each case of a switch statement: there is a block for the condition and a block for each case body.

At the package level, the order in which declarations appear has no effect on their scope, so a declaration may refer to itself or to another that follows it, letting us declare recursive or mutually recursive types and functions. The compiler will report an error if a constant or variable declaration refers to itself, however.

In this program:

```
if f, err := os.Open(fname); err != nil { // compile error: unused: f
    return err
}
f.ReadByte() // compile error: undefined f
f.Close()    // compile error: undefined f
```

the scope of `f` is just the `if` statement, so `f` is not accessible to the statements that follow, resulting in compiler errors. Depending on the compiler, you may get an additional error reporting that the local variable `f` was never used.

Thus it is often necessary to declare `f` before the condition so that it is accessible after:

```
f, err := os.Open(fname)
if err != nil {
    return err
}
f.ReadByte()
f.Close()
```

You may be tempted to avoid declaring `f` and `err` in the outer block by moving the calls to `ReadByte` and `Close` inside an `else` block:

```
if f, err := os.Open(fname); err != nil {
    return err
} else {
    // f and err are visible here too
    f.ReadByte()
    f.Close()
}
```

but normal practice in Go is to deal with the error in the `if` block and then return, so that the successful execution path is not indented.

Short variable declarations demand an awareness of scope. Consider the program below, which starts by obtaining its current working directory and saving it in a package-level variable. This could be done by calling `os.Getwd` in function `main`, but it might be better to separate this concern from the primary logic, especially if failing to get the directory is a fatal error. The function `log.Fatalf` prints a message and calls `os.Exit(1)`.

```
var cwd string

func init() {
    cwd, err := os.Getwd() // compile error: unused: cwd
    if err != nil {
        log.Fatalf("os.Getwd failed: %v", err)
    }
}
```

Since neither `cwd` nor `err` is already declared in the `init` function's block, the `:=` statement declares both of them as local variables. The inner declaration of `cwd` makes the outer one inaccessible, so the statement does not update the package-level `cwd` variable as intended.

Current Go compilers detect that the local `cwd` variable is never used and report this as an error, but they are not strictly required to perform this check. Furthermore, a minor change, such as the addition of a logging statement that refers to the local `cwd` would defeat the check.

```
var cwd string

func init() {
    cwd, err := os.Getwd() // NOTE: wrong!
    if err != nil {
        log.Fatalf("os.Getwd failed: %v", err)
    }
    log.Printf("Working directory = %s", cwd)
}
```

The global `cwd` variable remains uninitialized, and the apparently normal log output obfuscates the bug.

There are a number of ways to deal with this potential problem. The most direct is to avoid `:=` by declaring `err` in a separate `var` declaration:

```
var cwd string

func init() {
    var err error
    cwd, err = os.Getwd()
    if err != nil {
        log.Fatalf("os.Getwd failed: %v", err)
    }
}
```

We've now seen how packages, files, declarations, and statements express the structure of programs. In the next two chapters, we'll look at the structure of data.

This page intentionally left blank