The Go Programming Language

Alan A. A. Donovan
Brian W. Kernighan

ADDISON-WESLEY PROFESSIONAL COMPUTING SERIES

The Go Programming Language

```
type Employee struct {
    ID            int
    Name, Address string
    DoB           time.Time
    Position      string
    Salary        int
    ManagerID     int
}
```

Field order is significant to type identity. Had we also combined the declaration of the `Position` field (also a string), or interchanged `Name` and `Address`, we would be defining a different struct type. Typically we only combine the declarations of related fields.

The name of a struct field is exported if it begins with a capital letter; this is Go's main access control mechanism. A struct type may contain a mixture of exported and unexported fields.

Struct types tend to be verbose because they often involve a line for each field. Although we could write out the whole type each time it is needed, the repetition would get tiresome. Instead, struct types usually appear within the declaration of a named type like `Employee`.

A named struct type `S` can't declare a field of the same type `S`: an aggregate value cannot contain itself. (An analogous restriction applies to arrays.) But `S` may declare a field of the pointer type `*S`, which lets us create recursive data structures like linked lists and trees. This is illustrated in the code below, which uses a binary tree to implement an insertion sort:

gopl.io/ch4/treesort

```
type tree struct {
    value       int
    left, right *tree
}

// Sort sorts values in place.
func Sort(values []int) {
    var root *tree
    for _, v := range values {
        root = add(root, v)
    }
    appendValues(values[:0], root)
}

// appendValues appends the elements of t to values in order
// and returns the resulting slice.
func appendValues(values []int, t *tree) []int {
    if t != nil {
        values = appendValues(values, t.left)
        values = append(values, t.value)
        values = appendValues(values, t.right)
    }
    return values
}
```

```
func add(t *tree, value int) *tree {
    if t == nil {
        // Equivalent to return &tree{value: value}.
        t = new(tree)
        t.value = value
        return t
    }
    if value < t.value {
        t.left = add(t.left, value)
    } else {
        t.right = add(t.right, value)
    }
    return t
}
```

The zero value for a struct is composed of the zero values of each of its fields. It is usually desirable that the zero value be a natural or sensible default. For example, in `bytes.Buffer`, the initial value of the struct is a ready-to-use empty buffer, and the zero value of `sync.Mutex`, which we'll see in Chapter 9, is a ready-to-use unlocked mutex. Sometimes this sensible initial behavior happens for free, but sometimes the type designer has to work at it.

The struct type with no fields is called the *empty struct*, written `struct{}`. It has size zero and carries no information but may be useful nonetheless. Some Go programmers use it instead of `bool` as the value type of a map that represents a set, to emphasize that only the keys are significant, but the space saving is marginal and the syntax more cumbersome, so we generally avoid it.

```
seen := make(map[string]struct{}) // set of strings
// ...
if _, ok := seen[s]; !ok {
    seen[s] = struct{}{}
    // ...first time seeing s...
}
```

4.4.1. Struct Literals

A value of a struct type can be written using a *struct literal* that specifies values for its fields.

```
type Point struct{ X, Y int }

p := Point{1, 2}
```

There are two forms of struct literal. The first form, shown above, requires that a value be specified for *every* field, in the right order. It burdens the writer (and reader) with remembering exactly what the fields are, and it makes the code fragile should the set of fields later grow or be reordered. Accordingly, this form tends to be used only within the package that defines the struct type, or with smaller struct types for which there is an obvious field ordering convention, like `image.Point{x, y}` or `color.RGBA{red, green, blue, alpha}`.

More often, the second form is used, in which a struct value is initialized by listing some or all of the field names and their corresponding values, as in this statement from the Lissajous program of Section 1.4:

```
anim := gif.GIF{LoopCount: nframes}
```

If a field is omitted in this kind of literal, it is set to the zero value for its type. Because names are provided, the order of fields doesn't matter.

The two forms cannot be mixed in the same literal. Nor can you use the (order-based) first form of literal to sneak around the rule that unexported identifiers may not be referred to from another package.

```
package p
type T struct{ a, b int } // a and b are not exported
```

```
package q
import "p"
var _ = p.T{a: 1, b: 2} // compile error: can't reference a, b
var _ = p.T{1, 2}       // compile error: can't reference a, b
```

Although the last line above doesn't mention the unexported field identifiers, it's really using them implicitly, so it's not allowed.

Struct values can be passed as arguments to functions and returned from them. For instance, this function scales a `Point` by a specified factor:

```
func Scale(p Point, factor int) Point {
    return Point{p.X * factor, p.Y * factor}
}
```

```
fmt.Println(Scale(Point{1, 2}, 5)) // "{5 10}"
```

For efficiency, larger struct types are usually passed to or returned from functions indirectly using a pointer,

```
func Bonus(e *Employee, percent int) int {
    return e.Salary * percent / 100
}
```

and this is required if the function must modify its argument, since in a call-by-value language like Go, the called function receives only a copy of an argument, not a reference to the original argument.

```
func AwardAnnualRaise(e *Employee) {
    e.Salary = e.Salary * 105 / 100
}
```

Because structs are so commonly dealt with through pointers, it's possible to use this shorthand notation to create and initialize a `struct` variable and obtain its address:

```
pp := &Point{1, 2}
```

It is exactly equivalent to

```
pp := new(Point)
*pp = Point{1, 2}
```

but `&Point{1, 2}` can be used directly within an expression, such as a function call.

4.4.2. Comparing Structs

If all the fields of a struct are comparable, the struct itself is comparable, so two expressions of that type may be compared using `==` or `!=`. The `==` operation compares the corresponding fields of the two structs in order, so the two printed expressions below are equivalent:

```
type Point struct{ X, Y int }

p := Point{1, 2}
q := Point{2, 1}
fmt.Println(p.X == q.X && p.Y == q.Y) // "false"
fmt.Println(p == q)                   // "false"
```

Comparable struct types, like other comparable types, may be used as the key type of a map.

```
type address struct {
    hostname string
    port     int
}

hits := make(map[address]int)
hits[address{"golang.org", 443}]++
```

4.4.3. Struct Embedding and Anonymous Fields

In this section, we'll see how Go's unusual *struct embedding* mechanism lets us use one named struct type as an *anonymous field* of another struct type, providing a convenient syntactic shortcut so that a simple dot expression like `x.f` can stand for a chain of fields like `x.d.e.f`.

Consider a 2-D drawing program that provides a library of shapes, such as rectangles, ellipses, stars, and wheels. Here are two of the types it might define:

```
type Circle struct {
    X, Y, Radius int
}

type Wheel struct {
    X, Y, Radius, Spokes int
}
```

A `Circle` has fields for the `X` and `Y` coordinates of its center, and a `Radius`. A `Wheel` has all the features of a `Circle`, plus `Spokes`, the number of inscribed radial spokes. Let's create a wheel:

```
var w Wheel
w.X = 8
w.Y = 8
w.Radius = 5
w.Spokes = 20
```

As the set of shapes grows, we're bound to notice similarities and repetition among them, so it may be convenient to factor out their common parts:

```
type Point struct {
    X, Y int
}

type Circle struct {
    Center Point
    Radius int
}

type Wheel struct {
    Circle Circle
    Spokes int
}
```

The application may be clearer for it, but this change makes accessing the fields of a `Wheel` more verbose:

```
var w Wheel
w.Circle.Center.X = 8
w.Circle.Center.Y = 8
w.Circle.Radius = 5
w.Spokes = 20
```

Go lets us declare a field with a type but no name; such fields are called *anonymous fields*. The type of the field must be a named type or a pointer to a named type. Below, `Circle` and `Wheel` have one anonymous field each. We say that a `Point` is *embedded* within `Circle`, and a `Circle` is embedded within `Wheel`.

```
type Circle struct {
    Point
    Radius int
}

type Wheel struct {
    Circle
    Spokes int
}
```

Thanks to embedding, we can refer to the names at the leaves of the implicit tree without giving the intervening names:

```
var w Wheel
w.X = 8        // equivalent to w.Circle.Point.X = 8
w.Y = 8        // equivalent to w.Circle.Point.Y = 8
w.Radius = 5   // equivalent to w.Circle.Radius = 5
w.Spokes = 20
```

The explicit forms shown in the comments above are still valid, however, showing that "anonymous field" is something of a misnomer. The fields `Circle` and `Point` do have names—that of the named type—but those names are optional in dot expressions. We may omit any or all of the anonymous fields when selecting their subfields.

Unfortunately, there's no corresponding shorthand for the struct literal syntax, so neither of these will compile:

```
w = Wheel{8, 8, 5, 20}                       // compile error: unknown fields
w = Wheel{X: 8, Y: 8, Radius: 5, Spokes: 20} // compile error: unknown fields
```

The struct literal must follow the shape of the type declaration, so we must use one of the two forms below, which are equivalent to each other:

gopl.io/ch4/embed

```
w = Wheel{Circle{Point{8, 8}, 5}, 20}

w = Wheel{
    Circle: Circle{
        Point:  Point{X: 8, Y: 8},
        Radius: 5,
    },
    Spokes: 20, // NOTE: trailing comma necessary here (and at Radius)
}

fmt.Printf("%#v\n", w)
// Output:
// Wheel{Circle:Circle{Point:Point{X:8, Y:8}, Radius:5}, Spokes:20}

w.X = 42

fmt.Printf("%#v\n", w)
// Output:
// Wheel{Circle:Circle{Point:Point{X:42, Y:8}, Radius:5}, Spokes:20}
```

Notice how the `#` adverb causes `Printf`'s `%v` verb to display values in a form similar to Go syntax. For struct values, this form includes the name of each field.

Because "anonymous" fields do have implicit names, you can't have two anonymous fields of the same type since their names would conflict. And because the name of the field is implicitly determined by its type, so too is the visibility of the field. In the examples above, the `Point` and `Circle` anonymous fields are exported. Had they been unexported (`point` and `circle`), we could still use the shorthand form

```
w.X = 8 // equivalent to w.circle.point.X = 8
```

but the explicit long form shown in the comment would be forbidden outside the declaring package because `circle` and `point` would be inaccessible.