Capítulo 2

Estructura del programa

En **Go**, como en cualquier otro lenguaje de programación, se construyen grandes programas con un pequeño conjunto de construcciones básicas. Las variables almacenan valores. Las expresiones simples se combinan con las más grandes con operaciones como la suma y la resta. Los tipos básicos se recogen en agregados como arreglos y estructuras. Las expresiones se utilizan en sentencias cuya orden de ejecución se determina por las sentencias de control del flujo como **if** y **for**. Las sentencias se agrupan en funciones para su aislamiento y reutilización. Las funciones se agrupan en los archivos fuente en paquetes.

Vimos ejemplos de la mayoría de éstos en el capítulo anterior. En este capítulo, vamos a entrar en más detalles acerca de los elementos estructurales básicos de un programa de **Go**. Los programas de ejemplo son intencionalmente simples, por lo que podemos centrarnos en el lenguaje sin desviarnos por complicados algoritmos y estructuras de datos.

2.1. Nombres

Los nombres de las funciones, variables, constantes, tipos, etiquetas de sentencias y paquetes de **Go** siguen una regla simple: un nombre comienza con una letra (es decir, cualquier cosa que Unicode considere una letra) o un guion bajo y puede tener cualquier número de letras, dígitos y guiones bajos adicionales. Las mayúsculas importan: **HeapSort** y **heapsort** son diferentes nombres.

Go tiene **25** palabras clave como **if** y **switch** que puede ser utilizadas sólo cuando la sintaxis lo permita; no pueden ser utilizados como nombres.

```
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
```

Además, hay cerca de tres docenas de nombres predeclarados como **int** y **true** para las constantes, tipos y funciones incorporadas:

Constantes	true false iota nil
Tipos	int int8 int16 int32 int64 uint uint8 uint16 uint32 uint64 uintptr float32 float64 complex128 complex64 bool byte rune string error
Funciones	make len cap new append copy close delete complex real imag panic recover

Estos nombres no están reservados, por lo que se pueden utilizar en las declaraciones. Veremos un puñado de lugares donde redeclararlos tiene sentido, pero cuidado con el potencial de confusión.

Si una entidad es declarada dentro de una función, es **local** a esa función. Sin embargo, si se declara fuera de una función, es visible en todos los archivos del paquete al que pertenece. La primera letra del nombre determina su visibilidad a través de los límites del paquete. Si el nombre comienza con una letra mayúscula, se **exporta**, lo que significa que es visible y accesible fuera de su propio paquete y puede ser referida por otras partes del programa, como **Printf** en el paquete **fmt**. Los nombres de paquetes en sí son siempre en minúsculas.

No hay límite en la longitud del nombre, pero la convención y el estilo en los programas **Go** se inclinan por nombres cortos, especialmente para las variables locales con pequeños alcances; es mucho más probable ver variables con nombre **i** que con nombre **theLoopIndex**. En general, cuanto mayor sea el alcance de un nombre, más largo y más significativo debe ser.

Estilísticamente, los programadores de **Go** utilizan "**camel case**" cuando forman nombres mediante la combinación de palabras; es decir, las letras mayúsculas interiores se prefieren a los guiones interiores. Por lo tanto, las librerías estándar tienen funciones con nombres como **QuoteRuneToASCII** y **parseRequestLine**, pero nunca **quote_rune_to_ASCII** o **parse_request_line**. Las letras de acrónimos y siglas como **ASCII** y **HTML** siempre se prestan al mismo caso, por lo que una función podrá ser llamada **htmlEscape**, **HTMLEscape**, o **escapeHTML**, pero no **escapeHtml**.

2.2. Declaraciones

Una declaración nombra una entidad del programa y especifica todas o algunas de sus propiedades. Hay cuatro tipos principales de declaraciones: **var**, **const**, **type**, y **func**. Hablaremos de las variables y los tipos en este capítulo, de las constantes en el **Capítulo 3**, y de las funciones en el **Capítulo 5**.

Un programa **Go** se almacena en uno o más archivos cuyos nombres terminan en **.go**. Cada archivo comienza con una declaración **package** que dice de qué paquete es parte el archivo. La declaración **package** es seguida por declaraciones **import**, y luego una secuencia de declaraciones de tipos, variables, constantes y funciones a nivel de paquete, en cualquier orden. Por ejemplo, este programa declara una constante, una función, y un par de variables:

```
gopl.io/ch2/boiling
// Boiling imprime el punto de ebullición del agua.
package main

import "fmt"

const boilingF = 212.0

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

La constante **boilingF** es una declaración de nivel de paquete (que es **main**), mientras que las variables **f** y **c** son locales de la función **main**. El nombre de cada entidad de nivel de paquete es visible no sólo en todo el archivo fuente que contiene su declaración, sino a través de todos los archivos del paquete. Por el contrario, las declaraciones locales son visibles sólo dentro de la función en la que se declaran y tal vez sólo en una pequeña parte de ella.

Una declaración de función tiene un nombre, una lista de parámetros (las variables cuyos valores son proporcionados por los llamadores a la función), una lista opcional de resultados, y el cuerpo de la función, que contiene las declaraciones que definen lo que hace la función. La lista de resultados se omite si la función no devuelve nada. La ejecución de la función comienza con la primera instrucción y continúa hasta que se encuentra una sentencia **return** o se alcanza el final de una función que no tiene resultados. El control y cualquier resultado luego se devuelven al que llamó.

Hemos visto un buen número de funciones ya y hay muchas más para ver, incluyendo un tratado extenso en el **Capítulo 5**, así que esto es sólo un esbozo. La siguiente función **fToC** encapsula la lógica de conversión de temperatura de modo que se define sólo una vez, pero puede utilizarse desde múltiples lugares. Aquí **main** llama dos veces, utilizando los valores de dos constantes locales diferentes:

```
gopl.io/ch2/ftoc
// Ftoc Imprime dos conversiones Fahrenheit-to-Celsius.
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

Una declaración **var** crea una variable de un tipo particular, une un nombre a la misma, y define su valor inicial. Cada declaración tiene la forma general

```
var name type = expression
```

El tipo o la parte de expresión **=** pueden ser omitidos, pero no ambos. Si se omite el tipo, se determina por la expresión inicializadora. Si se omite la expresión, el valor inicial es el valor cero para el tipo, que es **0** para los números, **false** para booleanos, **""** para los **strings**, y **nil** para las interfaces y los tipos de referencia (**slice**, puntero, mapa, canal, función). El valor cero de un tipo agregado como un arreglo o estructura tiene el valor cero de todos sus elementos o campos.

El mecanismo de valor cero asegura que una variable siempre contenga un valor bien definido de su tipo; en **Go** no existe tal cosa como una variable sin inicializar. Esto simplifica el código y, a menudo garantiza un comportamiento sensible de las condiciones del entorno y sin trabajo adicional. Por ejemplo,

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

imprime una cadena vacía, en lugar de causar algún tipo de error o un comportamiento impredecible. Los programadores de **Go** suelen hacer un esfuerzo para hacer que el valor cero de un tipo más complicado sea significativo, por lo que las variables comienzan su vida en un estado útil.

Es posible declarar y opcionalmente inicializar un conjunto de variables en una sola declaración, con una lista de coincidencias de expresiones. La omisión del tipo permite la declaración de variables múltiples de diferentes tipos:

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

Los inicializadores pueden ser valores literales o expresiones arbitrarias. Las variables a nivel de paquete se inicializan antes de que comience **main** (**§2.6.2**), y las variables locales son inicializadas a medida que se encuentran sus declaraciones durante la ejecución de la función.

Un conjunto de variables también se puede inicializar llamando a una función que devuelve varios valores:

```
var f, err = os.Open(name) // os.Open devuelve un archivo y un error
```

2.3.1. Declaraciones cortas de variables

Se puede usar dentro de una función, una forma alternativa llamada declaración corta de variable para declarar e inicializar variables locales. Toma la forma **nombre := expresión**, y el tipo de nombre es determinado por el tipo de expresión. Aquí están tres de las muchas declaraciones cortas de variables en la función **lissajous** (**§1.4**):

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

Debido a su brevedad y flexibilidad, las declaraciones de variables cortas se utilizan para declarar e inicializar la mayoría de las variables locales. Una declaración **var** tiende a ser reservada para las variables locales que necesitan un tipo explícito que difiere de la de la expresión de inicialización, o cuando a la variable se le asignará un valor más adelante y su valor inicial no es importante.

```
i := 100 // como int
var boiling float64 = 100 // como float64

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

Al igual que con las declaraciones **var**, pueden ser declaradas e inicializadas múltiples variables en la misma declaración corta de variables,

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

pero las declaraciones con múltiples expresiones de inicialización deben utilizarse sólo cuando ayudan a la legibilidad, como para agrupaciones cortas y naturales como la parte de inicialización de un bucle **for**.

Tener en cuenta que **: =** es una declaración, mientras que **=** es una asignación. Una declaración de múltiples variables no debe confundirse con una asignación de tupla (**§2.4.1**), en la que se asigna a cada variable de la parte izquierda el valor correspondiente de la parte derecha:

```
i, j = j, i // intercambia valores de i y j
```

Igual que las declaraciones ordinarias **var**, las declaraciones cortas de variables pueden utilizarse para llamadas a funciones como **os.Open** que devuelven dos o más valores:

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

Un punto sutil pero importante: una declaración corta de variable no necesariamente declara todas las variables en su lado izquierdo. Si alguna de ellas ya fue declarada en el mismo bloque de léxico (**§2.7**), la declaración corta de variables actúa como una asignación de esas variables.

En el código siguiente, la primera sentencia declara **in** y **err**. La segunda declara **out** pero sólo asigna un valor a la variable existente **err**.

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

Una declaración corta de variables debe declarar al menos una nueva variable, por lo que este código no se compilará:

```
f, err := os.Open(infile)
// ...
f, err := os.Create(outfile) // error de compilación: no hay variables nuevas
```

La solución es utilizar una asignación ordinaria para la segunda declaración.

Una declaración corta de variables actúa como una asignación única de las variables que ya se declararon en el mismo bloque; las declaraciones en un bloque exterior se ignoran. Veremos ejemplos de esto al final del capítulo.

2.3.2. Punteros

Una variable es una pieza de almacenamiento que contiene un valor. Las variables creadas por las declaraciones se identifican por un nombre, como **x**, pero muchas variables son identificadas solamente por expresiones como **x[i]** o **x.f**. Todas estas expresiones leen el valor de una variable, excepto cuando aparecen en el lado izquierdo de una asignación, en cuyo caso se le asigna un nuevo valor a la variable.

Un valor de puntero es la dirección de una variable. Un puntero es por lo tanto la ubicación en la que se almacena un valor. No todos los valores tienen una dirección, pero todas las variables si la tienen. Con un puntero, podemos leer o actualizar el valor de una variable indirectamente, sin utilizar o incluso sin saber el nombre de la variable, si es que tiene un nombre.

Si se declara una variable **var x int**, la expresión **&x** ("dirección de **x**'') obtiene un puntero a una variable entera, es decir, un valor del tipo ***int**, que se pronuncia "puntero a **int**''. Si este valor se denomina **p**, decimos "**p** apunta a **x**'', o de forma equivalente "**p** contiene la dirección **x**''. La variable a la que **p** apunta se escribe como ***p**. La expresión ***p** produce el valor de esa variable, un **int**, pero como ***p** denota una variable, también puede aparecer en el lado izquierdo de una asignación, en cuyo caso la asignación actualiza la variable.

```
x := 1
p := &x // p, de tipo *int, apunta a x
fmt.Println(*p) // "1"
*p = 2 // equivalente a x = 2
fmt.Println(x) // "2"
```

Cada componente de una variable de tipo agregado, (un campo de una estructura o un elemento de un arreglo), es también una variable y por lo tanto también tiene una dirección.

Las variables se describen a veces como valores direccionables. Las expresiones que denotan variables son las únicas expresiones a las que se les puede aplicar el operador de dirección **&**.

El valor cero para un puntero de cualquier tipo es **nil**. La prueba **p!=nil** es **true** si **p** apunta a una variable. Los punteros son comparables; dos punteros son iguales si y sólo si apuntan a la misma variable o ambos son **nil**.

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

Es perfectamente seguro para una función devolver la dirección de una variable local. Por ejemplo, en el código siguiente, la variable local **v** creada por esta llamada a **f** seguirá existiendo incluso después de que haya retornado la llamada, y el puntero **p** aún se referirá a ella:

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

Cada llamada de **f** devuelve un valor distinto:

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

Debido a que un puntero contiene la dirección de una variable, al pasar un argumento puntero a una función hace posible que la función pueda actualizar la variable que se pasó indirectamente. Por ejemplo, esta función incrementa la variable que apunta su argumento y devuelve el nuevo valor de la variable por lo que se puede utilizar en una expresión:

```
func incr(p *int) int {
    *p++ // incrementa a lo que apunta p; no cambia p
    return *p
}

v := 1
incr(&v) // efecto lateral: v ahora es 2
fmt.Println(incr(&v)) // "3" (y v es 3)
```

Cada vez que se toma la dirección de una variable o se copia un puntero, creamos nuevos alias o maneras de identificar a la misma variable. Por ejemplo, ***p** es un alias para **v**. Los alias de puntero son útiles porque nos permite acceder a una variable sin necesidad de utilizar su nombre, pero esto es un arma de doble filo: para encontrar todas las sentencias que acceden a una variable, tenemos que conocer todos sus alias. No sólo los punteros crean alias; también se crean alias cuando copiamos valores de otros tipos de referencia como **slices**, mapas y canales, e incluso estructuras, arreglos e interfaces que contienen estos tipos.

Los punteros son clave para el paquete **flag**, que utiliza argumentos de línea de comandos de un programa para establecer los valores de ciertas variables distribuidas a lo largo del programa. Para ilustrarlo, esta variación del viejo comando **echo** tiene dos indicadores opcionales: **-n** causa que **echo** de omita el salto de línea final que normalmente se imprime, y **-s sep** hace que se separe los argumentos de salida por el contenido del **string sep** en lugar del simple espacio predeterminado. Dado que esta es nuestra cuarta versión, el paquete se llama **gopl.io/ch2/echo4**.

```
gopl.io/ch2/echo4
// Echo4 Imprime los argumentos de línea de comandos.
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()
    }
}
```

La función **flag.Bool** crea una nueva variable **flag** de tipo **bool**. Toma tres argumentos: el nombre del **flag** ("**n**"), el valor predeterminado de la variable (**false**), y un mensaje que se imprimirá si el usuario proporciona un argumento inválido, un **flag** inválido o **-h** o **-help**. Del mismo modo, **flag.String** toma un nombre, un valor por defecto, y un mensaje, y crea una variable **string**. Las variables **sep** y **n** son punteros a las variables **flag**, las cuales pueden ser accedidas indirectamente como ***sep** y ***n**.

Cuando se ejecuta el programa, se debe llamar a **flag.Parse** antes de utilizar los **flags**, para actualizar las variables de **flag** con sus valores por defecto. Los argumentos que no son **flag** están disponibles a partir **flag.Args()** como un **slice** de **strings**. Si **flag.Parse** encuentra un error, se imprime un mensaje de uso y llama a **os.Exit(2)** para finalizar el programa.

Vamos a ejecutar algunos casos de prueba de **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. La función new

Otra forma de crear una variable es utilizar la función incorporada **new**. La expresión **new(T)** crea un variable sin nombre de tipo **T**, la inicializa con el valor cero de **T**, y devuelve su dirección, que es un valor de tipo ***T**.

```
p := new(int) // p, de tipo *int, apunta a una variable int sin nombre
fmt.Println(*p) // "0"
*p = 2 // asigna el valoe 2 a la variable int sin nombre
fmt.Println(*p) // "2"
```

Una variable creada con **new** no es diferente de una variable local ordinaria cuya dirección está tomada, salvo que no hay necesidad de inventar (y declarar) un nombre ficticio, y podemos usar **new(T)** en una expresión. De este modo **new** no es más que una conveniencia sintáctica, no es una idea fundamental:

Las dos funciones **newInt** a continuación tienen comportamientos idénticos.

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

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

Cada llamada a **new** retorna una variable distinta con una dirección única:

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

Hay una excepción a esta regla: dos variables cuyo tipo no lleva información y por lo tanto son de tamaño cero, como **struct{}** o **[0]int**, puede, dependiendo de la implementación, tener la misma dirección.

La función **new** se utiliza con poca frecuencia debido a que las variables sin nombre más comunes son los tipos de **struct**, para lo cual la sintaxis literal de **struct** es más flexible (**§4.4.1**).

Dado que **new** es una función predeclarada, no es una palabra clave, es posible redefinir el nombre por algo dentro de una función, por ejemplo:

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

Por supuesto, dentro de **delta**, la función incorporada new no está disponible.

2.3.4. Tiempo de vida de las variables

El tiempo de vida de una variable es el intervalo de tiempo durante el cual existe cuando se ejecuta el programa. El tiempo de vida de una variable de nivel de paquete es toda la ejecución del programa. Por el contrario, las variables locales tienen un tiempo de vida dinámico: se crea una nueva instancia cada vez que se ejecuta la sentencia de la declaración, y la variable sigue viva hasta que se vuelve inaccesible, momento en el cual su almacenamiento puede ser reciclado. Los parámetros de función y los resultados son variables locales también; se crean cada vez que la función que las encierra es llamada.

Por ejemplo, en este extracto del programa **Lissajous** de la **Sección 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)
}
```

la variable **t** se crea cada vez que el bucle **for** comienza y se crean nuevas variables **x** e **y** en cada iteración del bucle.

¿Cómo sabe el recolector de basura que almacenamiento de una variable se puede reclamar? La historia completa es mucho más detallada de lo que necesitamos aquí, pero la idea básica es que cada variable de nivel de paquete, y cada variable local de cada función activa en ese momento, potencialmente puede ser el inicio o raíz de una ruta de acceso a la variable en cuestión, a raíz de punteros y otros tipos de referencias que finalmente conducen a la variable. Si no existe tal ruta, la variable se ha convertido en inalcanzable, por lo que ya no puede afectar al resto de la computación.

Debido a que el tiempo de vida de una variable se determina sólo por si es o no es accesible, una variable local puede sobrevivir a una única iteración de un bucle cerrado. Puede seguir existiendo incluso después de que se haya vuelto de la función de encerramiento.

Un compilador puede optar por asignar las variables locales en el **heap** o en el **stack**, pero, sorprendentemente, esta elección no depende de si se utiliza **var** o **new** para declarar la variable.

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

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

Aquí, **x** debe ser asignado en el **heap** porque todavía es accesible la variable global después de que **f** ha retornado, a pesar de ser declarada como una variable local; decimos que **x** escapa desde **f**. A la inversa, cuando **g** retorna, la variable ***y** se convierte en inalcanzable y se pueden reciclar. Como ***y** no se escapa de **g**, es seguro para el compilador asignar ***y** en la pila, a pesar de que se asignó con **new**. En cualquier caso, la noción de escape no es algo que haya que preocuparse para escribir código correcto, a pesar de que es bueno tenerlo en cuenta para la optimización del rendimiento, ya que cada variable que se escapa requiere una asignación de memoria adicional.

La recolección de basura es una tremenda ayuda para escribir programas correctos, pero no nos exime de la carga de pensar en la memoria. No hay necesidad de asignar y liberar de forma explícita memoria, pero para escribir programas eficientes hay que ser conscientes de la duración de las variables. Por ejemplo, mantener punteros a objetos innecesarios de corta duración dentro de objetos de larga duración, especialmente las variables globales, evitará que el recolector de basura recupere los objetos de corta vida.

2.4. Asignaciones

El valor en poder de una variable se actualiza mediante una instrucción de asignación, que en su forma más simple tiene una variable a la izquierda del signo **=** y una expresión a la derecha.

```
x=1 // variable con nombre
*p = true // variable indirecta
person.name = "bob" // campo de struct
count[x] = count[x] * scale // elemento de arreglo slice o map
```

Cada uno de los operadores binarios aritméticas y bit a bit tiene un correspondiente operador de asignación que permite, por ejemplo, que la última sentencia se reescriba como:

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

lo cual nos ahorra tener que repetir (y volver a evaluar) la expresión de la variable. Las variables numéricas se pueden también incrementar y decrementar por sentencias **++** y **--**:

```
v := 1
v++ // lo mismo que v = v + 1; v se convierte en 2
v-- // lo mismo que v = v - 1; v se convierte en 1 de nuevo
```

2.4.1. Asignación de tupla

Otra forma de asignación, conocida como asignación de tupla, permite asignar varias variables a la vez. Todas las expresiones del lado derecho se evalúan antes de que cualesquiera de las variables se actualicen, siendo de esta forma más útil cuando algunas de las variables aparecen en ambos lados de la asignación, como sucede, por ejemplo, al intercambiar los valores de dos variables:

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

o cuando se calcula el máximo común divisor (**GCD**) de dos enteros:

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

o cuando se calcula el n-ésimo número de **Fibonacci** iterativamente:

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

La asignación de tupla también puede hacer una secuencia de tareas triviales más compactas,

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

aunque como una cuestión de estilo, evitar la forma de tupla si las expresiones son complejas; una secuencia de sentencias separadas es más fácil de leer.

Ciertas expresiones, como una llamada a una función con múltiples resultados, producen varios valores. Cuando se utiliza una llamada de este tipo en una sentencia de asignación, el lado izquierdo debe tener tantas variables como resultados tiene la función.

```
f, err = os.Open("foo.txt") // llamada de función que retorna dos valores
```

A menudo, las funciones utilizan estos resultados adicionales para indicar algún tipo de error, ya sea devolviendo un error como en la llamada a **os.Open**, o un **bool**, generalmente llamado **ok**. Veremos en **capítulos** posteriores, que hay tres operadores que a veces se comportan de esta manera también. Si un mapa de búsqueda (**§4.3**), tipo de **assertion** (**§7.10**), o un canal de recepción (**§8.4.2**) aparece en una asignación en la que se espera dos resultados, cada uno produce un resultado booleano adicional:

```
v, ok = m[key] // map lookup
v, ok = x.(T) // tipo assertion
v, ok = <-ch // canal receptor
```

Al igual que con las declaraciones de variables, podemos asignar valores no deseados al identificador en blanco:

```
_, err = io.Copy(dst, src) // descartar byte count
_, ok = x.(T) // probar tipo, pero descartar resultado
```

2.4.2. Asignabilidad

Las sentencias de asignación son una forma explícita de asignación, pero hay muchos lugares en un programa donde se produce una asignación implícita: una llamada de función asigna implícitamente los valores de los argumentos a las variables de los parámetros correspondientes; una sentencia **return** asigna implícitamente los operandos de devolución a las correspondientes variables de resultados; y una expresión literal a un tipo compuesto (**§ 4.2**) como este **slice**:

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

implícitamente se asigna cada elemento, como si hubiera sido escrito así:

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

Los elementos de mapas y canales, aunque no las variables comunes, también están sujetos a asignaciones implícitas similares.

Una asignación, explícita o implícita, siempre es legal si el lado izquierdo (la variable) y el lado derecho (el valor) tienen el mismo tipo. En términos más generales, la asignación es legal sólo si el valor es asignable al tipo de la variable.

La regla para la asignabilidad tiene casos de diversos tipos, por veremos los casos relevantes a medida que introduzcamos cada nuevo tipo. Para los tipos que hemos tratado hasta ahora, las reglas son simples: los tipos deben coincidir exactamente, y puede ser asignado **nil** a cualquier variable de interface o referencia de tipo. Las constantes (**§3.6**) tienen reglas más flexibles para la asignabilidad que evitan la necesidad de conversiones más explícitas.

Si dos valores pueden compararse con **==** y **!=** Se relaciona con la asignabilidad: en cualquier comparación, el primer operando debe ser asignable al tipo del segundo operando, o viceversa. Al igual que con la asignabilidad, explicaremos los casos relevantes para la comparabilidad cuando presentemos cada nuevo tipo.

2.5. Declaraciones de tipo

El tipo de una variable o expresión define las características de los valores que puede asumir, tales como su tamaño (número de bits o número de elementos, tal vez), la forma en que se representan internamente, las operaciones intrínsecas que se pueden realizar en ellos, y los métodos asociados con ellos.

En cualquier programa hay variables que comparten la misma representación, pero significan conceptos muy diferentes. Por ejemplo, un **int** podría ser usado para representar un índice de bucle, una marca de tiempo, un descriptor de archivo, o un mes; un **float64** podría representar una velocidad en metros por segundo o una temperatura en una de varias escalas; y un **string** podría representar una contraseña o el nombre de un color.

Una declaración de tipo define un nuevo tipo con nombre que tiene el mismo tipo subyacente como un tipo existente. El tipo con nombre proporciona una manera de separar usos diferentes y quizás incompatibles del tipo subyacente de modo que no puedan ser mezclado sin querer.

```
type name underlying-type
```

Las declaraciones de tipo aparecen con mayor frecuencia a nivel de paquete, donde el tipo con nombre es visible en todo el paquete, y si se exporta el nombre (que comienza con una letra mayúscula), es accesible desde otros paquetes.

Para ilustrar las declaraciones de tipo, vamos volver a las diferentes escalas de temperatura en diferentes tipos:

```
gopl.io/ch2/tempconv0
// Paquete tempconv que realiza cálculos de temperatura Celsius y Fahrenheit.
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) }
```

Este paquete define dos tipos, **Celsius** y **Fahrenheit**, para las dos unidades de temperatura. A pesar de que ambos tienen el mismo tipo subyacente, **float64**, no son del mismo tipo, por lo que no se pueden comparar o combinar en expresiones aritméticas. Distinguir los tipos hace que sea posible evitar errores como la combinación inadvertida de temperaturas en las dos escalas diferentes; es necesario un tipo explícito de conversión como **Celsius(t)** o **Fahrenheit(t)** para convertir de un **float64**. **Celsius(t)** y **Fahrenheit(t)** son conversiones, no llamadas de función. No cambian el valor o representación de ninguna manera, pero hacen que el cambio sea de significado explícito. Por otro lado, las funciones **CToF** and **FToC** convierten entre las dos escalas; no devuelven valores diferentes.

Para cada tipo **T**, hay una operación de conversión correspondiente **T(x)** que convierte el valor **x** en el tipo **T**. Se permite una conversión de un tipo a otro si ambos tienen el mismo tipo subyacente, o si ambos son tipos de puntero sin nombre que apuntan a variables del mismo tipo subyacente; estas conversiones cambian el tipo, pero no la representación del valor. Si **x** es asignable a **T**, se permite una conversión, pero suele ser redundante,

Las conversiones también se permiten entre tipos numéricos, y entre **strings** y algunos tipos de **slices**, como veremos en el siguiente capítulo. Estas conversiones pueden cambiar la representación del valor. Por ejemplo, la conversión de un número de coma flotante a un entero descarta cualquier parte fraccionaria, y convertir un **string** en un **slice []bytes** asigna una copia de los datos del **string**. En cualquier caso, una conversión nunca falla en tiempo de ejecución.

El tipo subyacente de un tipo con nombre determina su estructura y representación, así como el conjunto de operaciones intrínsecas que le dan soporte, que son las mismas que si el tipo subyacente hubiera sido utilizado directamente. Eso significa que los operadores aritméticos funcionan de la misma para **Celsius** y **Fahrenheit** como lo hacen para **float64**, como se podría esperar.

```
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) // error de compilación: tipo no coincide
```

Los operadores de comparación como **==** y **<** también se puede utilizar para comparar un valor de un tipo con nombre con otro del mismo tipo, o a un valor del tipo subyacente. Sin embargo, dos valores de diferentes tipos con nombre no se pueden comparar directamente:

```
var c Celsius
var f Fahrenheit
fmt.Println(c == 0) // "true"
fmt.Println(f >= 0) // "true"
fmt.Println(c == f) // error de compilación: tipos diferentes
fmt.Println(c == Celsius(f)) // "true"!
```

Tener en cuenta el último caso con cuidado. A pesar de su nombre, la conversión de tipo **Celsius(f)** no cambia el valor de su argumento, sólo su tipo. La prueba es **true** porque **c** y **f** son ambos cero.

Un tipo con nombre puede proporcionar conveniencia de notación si ayuda a evitar escribir tipos complejos una y otra vez. La ventaja es pequeña cuando el tipo subyacente es tan simple como **float64**, pero grande para los tipos complicados, como veremos cuando hablemos de estructuras.

Los tipos con nombre también hacen que sea posible definir nuevos comportamientos para los valores del tipo. Estos comportamientos se expresan como un conjunto de funciones asociadas con el tipo, llamados métodos del tipo. Veremos los métodos en detalle en el **Capítulo 6**, pero aquí conoceremos el mecanismo.

La siguiente declaración, en la que el parámetro **c** de **Celsius** aparece antes del nombre de la función, se asocia con el tipo **Celsius** un método llamado **String** que devuelve un valor numérico **c** seguido **°C**:

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

Muchos tipos declaran un método **String** de esta forma debido a que controlan cómo aparecen los valores del tipo cuando se imprimen en un **string** por el paquete **fmt**, como veremos en la **Sección 7.1**.

```
c := FToC(212.0)
fmt.Println(c.String()) // "100°C"
fmt.Printf("%v\n", c) // "100°C"; no se necesita llamar a String explicitamente
fmt.Printf("%s\n", c) // "100°C"
fmt.Println(c) // "100°C"
fmt.Printf("%g\n", c) // "100"; no llama a String
fmt.Println(float64(c)) // "100"; no llama a String
```

2.6. Paquetes y archivos

Los paquetes en **Go** sirven a los mismos fines que las librerías o módulos en otros lenguajes, soportando modularidad, encapsulación, compilación separada, y reutilización. El código fuente de un paquete reside en uno o más archivos **.go**, por lo general en un directorio cuyo nombre termina con la ruta de importación; por ejemplo, los archivos del paquete **gopl.io/ch1/helloworld** se almacenan en el directorio **$GOPATH/src/gopl.io/ch1/helloworld**.

Cada paquete sirve como en un espacio de nombres separado para sus declaraciones. Por ejemplo, dentro del paquete **image**, el identificador **Decode** se refiere a una función diferente que la que tiene el mismo identificador en el paquete **unicode/utf16**. Para hacer referencia a una función desde fuera de su paquete, hay que calificar el identificador explícitamente si nos referimos a **image.Decode** o a **utf16.Decode**.

Los paquetes también nos permiten ocultar información mediante el control de los nombres que son visibles fuera del paquete, o exportados. En **Go**, una simple regla rige qué identificadores se exportan y cuales no: los identificadores exportados comienzan con una letra mayúscula.

Para ilustrar los conceptos básicos, supongamos que nuestro software de conversión de temperatura ha llegado a ser popular y queremos ponerlo a disposición de la comunidad **Go** como un nuevo paquete. ¿Cómo hacemos eso?

Vamos a crear un paquete llamado **gopl.io/ch2/tempconv**, una variación del ejemplo anterior. (Aquí hemos hecho una excepción a nuestra regla usual de numeración en la secuencia de ejemplos, de modo que la ruta del paquete pueda ser más realista) El propio paquete se almacena en dos archivos para mostrar cómo se accede a las declaraciones en archivos separados de un paquete; en la vida real, un pequeño paquete como éste necesitaría sólo un archivo.

Hemos puesto las declaraciones de los tipos, sus constantes y sus métodos en **tempconv.go**:

```
gopl.io/ch2/tempconv
// Paquete tempconv realiza conversiones de grados Celsius y Fahrenheit.
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) }
```

y las funciones de conversión en **conv.go**:

```
package tempconv

// CTof convierte una temperatura Celsius a Fahrenheit.
func CToF(c Celsius) Fahrenheit { return Fahrenheit(c*9/5 + 32) }
// FToC convierte una temperatura Fahrenheit a Celsius.
func FToC(f Fahrenheit) Celsius { return Celsius((f - 32) * 5 / 9) }
```

Cada archivo comienza con una declaración **package** que define el nombre del paquete. Cuando se importa el paquete, sus miembros se conocen como **tempconv.CToF** y etc. Los nombres de nivel de paquete como los tipos y constantes declaradas en un archivo de un paquete son visibles para todos los otros archivos del paquete, como si el código fuente estuviera todo en un solo archivo. Tener en cuenta que **tempconv.go** importa **fmt**, pero **conv.go** no, ya que no utiliza nada de **fmt**.

Debido a que los nombres const de nivel de paquete comienzan con las letras mayúsculas, también son accesibles con nombres calificados como **tempconv.AbsoluteZeroC**:

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

Para convertir una temperatura **Celsius** a **Fahrenheit** en un paquete que importe **gopl.io/ch2/tempconv**, podemos escribir el siguiente código:

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

El comentario de documentación (**doc comment**) (**§10.7.4**) inmediatamente anterior a la declaración **package** documenta el paquete en su conjunto. Convencionalmente, se debe comenzar con una frase resumida en el estilo ilustrado. Sólo un archivo en cada paquete debe tener un **doc comment**. Los comentarios extensos de documentación se colocan a menudo en un fichero, convencionalmente llamado **doc.go**.

2.6.1. Las importaciones

Dentro de un programa **Go**, cada paquete se identifica por un **string** único llamado ruta de importación (**import path**). Estos son los **strings** que aparecen en una declaración **import** como "**gopl.io/ch2/tempconv**". La especificación del lenguaje no define donde van estos **strings** o lo que significan; incluso las herramientas para interpretarlos. Cuando se utiliza la herramienta **go** (**Capítulo 10**), una ruta de **import** se refiere a un directorio que contiene uno o más archivos fuente **Go** que componen el paquete.

Además de la ruta de importación, cada paquete tiene un nombre de paquete, es un nombre corto (y no necesariamente único) que aparece en su declaración **package**. Por convención, un nombre de paquete coincide con el último segmento de la ruta de importación, por lo que es fácil predecir que el nombre del paquete de **gopl.io/ch2/tempconv** es **tempconv**.

Para utilizar **gopl.io/ch2/tempconv**, tenemos que importarlo:

```
gopl.io/ch2/cf
// Cf convierte su argumento numérico a Celsius y 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))
    }
}
```

La declaración **import** une un nombre corto al paquete importado que puede utilizarse para hacer referencia a su contenido en todo el archivo. El **import** anterior nos permite referimos a nombres dentro de **gopl.io/ch2/tempconv** mediante el uso de un identificador cualificado (**qualified identifier)** como **tempconv.CToF**. Por defecto, el nombre corto es el nombre del paquete, **tempconv** en este caso, pero una declaración **import** puede especificar un nombre alternativo para evitar un conflicto (**§10.3**).

El programa **cf** convierte un único argumento de línea de comando numérico en su valor tanto en grados **Celsius** y **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
```

Es un error importar un paquete y luego no referirse a él. Esta verificación ayuda a eliminar dependencias que se convierten en innecesarias, ya que el código evoluciona, aunque puede ser una molestia durante la depuración, ya que al comentar una línea de código como **log.Print("got here!")** puede borrar la única referencia al nombre del paquete llamado **log**, haciendo que el compilador emita un error. En esta situación, es necesario comentar o eliminar el innecesario **import**.

Mejor aún, utilizar la herramienta **golang.org/x/tools/cmd/goimports**, que inserta y elimina paquetes de forma automática de la declaración **import**, según sea necesario; la mayoría de los editores pueden configurarse para ejecutar **goimports** cada vez que se guarda un archivo. Al igual que la herramienta **gofmt**, también pone los archivos de código fuente grabados de **Go** en el formato canónico.

2.6.2. Inicialización de paquete

La inicialización de paquete comienza inicializando las variables a nivel de paquete en el orden en que se declaran, a excepción de las dependencias, que se resuelven en primer lugar:

```
var a = b + c // a inicializada tercera, a 3
var b = f() // b inicializada segunda, a 2, llamando a f
var c = 1 // c inicializada primera, a 1
func f() int { return c + 1 }
```

Si el paquete tiene múltiples archivos **.go**, se inicializan en el orden en el que se presentan los archivos al compilador; la herramienta **go** ordena los archivos **.go** por nombre antes de invocar el compilador.

Cada variable declarada a nivel de paquete comienza su vida con el valor de su expresión de inicialización, en su caso, algunas variables, como las tablas de datos, una expresión de inicialización puede no ser la forma más sencilla de establecer su valor inicial. En ese caso, el mecanismo de la función **init** puede ser más simple. Cualquier archivo puede contener cualquier número de funciones cuya declaración es sólo

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

Tales funciones **init** no pueden ser llamadas o referenciadas, pero por lo demás son funciones normales. Dentro de cada archivo, las funciones **init** se ejecutan automáticamente cuando se inicia el programa, en el orden en que se declaran.

Un paquete es inicializado, en el orden de las importaciones en el programa, las dependencias primero, por lo que un paquete **p** que importa **q** puede estar seguro de que **q** está totalmente inicializado antes de que comience la inicialización de **p**. La inicialización procede de abajo hacia arriba; el paquete **main** es el último en ser inicializado. De esta manera, todos los paquetes ya han sido inicializados completamente antes de que la función **main** de la aplicación comience.

El paquete a continuación define una función **PopCount** que devuelve el número de bits puestos, es decir, los bits cuyo valor es **1**, en un valor **uint64**, llamado **population count**. Utiliza una función **init** para calcular previamente una tabla de resultados, **pc**, para cada posible valor de **8 bits** de modo que la función **PopCount** no necesita tomar **64**

pasos, sólo puede devolver la suma de ocho consultas de tabla. (Esto es definitivamente no es el algoritmo más rápido para el recuento de bits, pero es conveniente para ilustrar las funciones **init**, y para mostrar cómo calcular previamente una tabla de valores, que es a menudo una técnica de programación útil).

```
gopl.io/ch2/popcount
package popcount
// pc[i] es el recuento de la población de i.
var pc [256]byte

func init() {
    for i := range pc {
        pc[i] = pc[i/2] + byte(i&1)
    }
}
// PopCount devuelve el recuento de la población (número de bits puestos) de 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))])
}
```

Tener en cuenta que **range** del bucle en **init** utiliza sólo el índice; el valor es innecesario y por lo tanto no necesita ser incluido. El bucle también podría haber sido escrito como

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

Veremos otros usos de las funciones **init** en la siguiente sección y en la **Sección 10.5**.

2.7. Alcance

Una declaración asocia un nombre con una entidad de programa, como una función o una variable. El alcance de una declaración es la parte del código fuente donde un uso del nombre declarado hace referencia a dicha declaración.

No confundir alcance con tiempo de vida. El alcance de una declaración es una región del texto del programa; es una propiedad de tiempo de compilación. El tiempo de vida de una variable es el intervalo de tiempo durante la ejecución en el que la variable puede ser referida por otras partes del programa; es una propiedad de tiempo de ejecución.

Un bloque sintáctico es una secuencia de sentencias encerradas entre llaves como las que rodean el cuerpo de una función o un bucle. Un nombre declarado dentro de un bloque sintáctico no es visible fuera de ese bloque. El bloque encierra sus declaraciones y determina su alcance. Podemos generalizar esta noción de bloques para incluir otras agrupaciones de declaraciones que no están rodeadas por llaves de forma explícita en el código fuente; los llamaremos **bloques léxicos**. Hay un bloque léxico para todo el código fuente, llamado el **bloque universo**; para cada paquete; para cada archivo; para cada sentencia **for**, **if** y **switch**; para cada sentencia **case** en un **switch**; y, por supuesto, para cada bloque sintáctico explícito.

Una declaración de bloque léxico determina su alcance, que puede ser grande o pequeño. Las declaraciones de tipos incorporados, funciones y constantes como **int**, **len** y **true** están en el bloque universo y se puede referenciar a lo largo de todo el programa. Las declaraciones fuera de cualquier función, es decir, al nivel de paquete, pueden ser referenciadas desde cualquier archivo en el mismo paquete. Los paquetes importados, tales como **fmt** en el ejemplo **tempconv**, se declaran en el nivel de fichero, para que puedan ser referenciados a partir del mismo archivo, pero no de otro archivo en el mismo paquete con otro **import**. Muchas declaraciones, como la de la variable **c** en la función **tempconv.CToF**, son locales, pueden ser referenciadas sólo desde dentro de la misma función o tal vez en sólo una parte de ella.

El alcance de una etiqueta de flujo de control, como el usado por las sentencias **break**, **continue** y **goto**, es toda la función envolvente.

Un programa puede contener varias declaraciones del mismo nombre, siempre que cada declaración esté en un bloque de léxico diferente. Por ejemplo, se puede declarar una variable local con el mismo nombre que una variable a nivel de paquete. O, como se muestra en la **Sección 2.3.3**, se puede declarar un parámetro de función llamado **new**, a pesar de que una función con este nombre está declarada con anterioridad en el bloque universo. Sin embargo, no hay que excederse; cuanto mayor sea el alcance de la redeclaración, más probabilidades hay de sorprender al lector.

Cuando el compilador encuentra una referencia a un nombre, busca una declaración, empezando por el bloque de léxico más interior que encierra y trabajando hasta el bloque universo. Si el compilador no encuentra ninguna declaración, informa de un error de "nombre no declarado". Si un nombre es declarado tanto en un bloque exterior y uno interior, la declaración interior se encuentra primero. En ese caso, se dice que la declaración interna ensombrece u oculta la exterior, por lo que es inaccesible:

```
func f() {}

var g = "g"

func main() {
    f := "f"
    fmt.Println(f) // "f"; variable local f ensombrece a la función f de nivel de paquete
    fmt.Println(g) // "g"; variable de nivel de paquete
    fmt.Println(h) // error de compilación: h no definido
}
```

Dentro de una función, los bloques léxicos se pueden anidar con una profundidad arbitraria, por lo que una declaración local puede ensombrecer a otra. La mayoría de los bloques son creados con construcciones de flujo de control como sentencias **if** y bucles **for**. El programa siguiente tiene tres variables diferentes llamadas **x**, ya que cada declaración aparece en un bloque léxico diferente. (¡Este ejemplo ilustra las reglas de alcance, no es bueno estilo!)

```
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" (una letra por iteración)
        }
    }
}
```

Las expresiones **x[i]** y **x+'A'-'a'** se refieren a una declaración de **x** en un bloque exterior; lo explicaremos en un momento. (Tener en cuenta que esta última expresión no es equivalente a **unicode.ToUpper**.)

Tal como se mencionó anteriormente, no todos los bloques léxicos corresponden a secuencias de sentencias delimitadas explícitamente por llaves; algunas están meramente implícitas. El bucle **for** anterior crea dos bloques léxicos: el bloque explícito para el cuerpo del bucle, y un bloque implícito que, además, encierra las variables declaradas por la cláusula de inicialización, tales como **i**. El alcance de una variable declarada en el bloque implícito es la condición, la sentencia posterior (**i++**), y el cuerpo de la sentencia **for**.

El siguiente ejemplo también tiene tres variables denominadas **x**, cada una declarada en un bloque diferente, una en el cuerpo de la función, una en la sentencia **for**, y una en el cuerpo del bucle, pero sólo dos de los bloques son explícitos:

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

Al igual que para los bucles, las sentencias **if** y **switch** también crean un bloque implícito, además de sus bloques del cuerpo. El código en la siguiente cadena **if-else** muestra el alcance de **x** e **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) // error de compilación: x e y no son visibles aquí
```

La segunda sentencia i**f** está anidada dentro de la primera, por lo que las variables declaradas dentro de la primera inicialización de la sentencia son visibles dentro de la segunda. Se aplican reglas similares a cada **case** de una sentencia **switch**: hay un bloque para la condición y un bloque para cada **case** del cuerpo.

En el nivel de paquete, el orden en que aparecen las declaraciones no tiene efecto en su alcance, por lo que una declaración puede referirse a sí misma o a otra que le sigue, dejando que podamos declarar tipos y funciones recursivas o mutuamente recursivas. El compilador informará de un error si una declaración de constante o variable se refiere a sí misma, sin embargo, en este programa:

```
if f, err := os.Open(fname); err != nil { // error de compilación: no usado: f
    return err
}
f.ReadByte() // error de compilación: no definido f
f.Close() // error de compilación: no definido f
```

El alcance de **f** es sólo la sentencia **if**, por lo que **f** no es accesible a las sentencias que siguen, lo que resulta en errores de compilación. Dependiendo del compilador, se puede obtener un informe de errores adicional de que la variable local **f** nunca fue utilizada.

Por lo tanto, a menudo es necesario declarar **f** antes de la condición para que sea accesible después:

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

Se puede tener la tentación de evitar declarar **f** y **err** en el bloque exterior moviendo las llamadas a **ReadByte** y **Close** el interior de un bloque **else**:

```
if f, err := os.Open(fname); err != nil {
    return err
} else {
    // f y err son aquí también visibles
    f.ReadByte()
    f.Close()
}
```

Pero en la práctica normal en **Go** es tratar el error en el bloque **if** y luego retornar, por lo que la ruta de ejecución exitosa no está indentada.

Las declaraciones de variables cortas requieren un conocimiento de su alcance. Considerar el siguiente programa, que se inicia mediante la obtención de su directorio de trabajo actual y lo guarda en una variable de nivel de paquete. Esto podría hacerse llamando **os.Getwd** en la función **main**, pero podría ser mejor separar esta preocupación de la lógica primaria, sobre todo si no se puede conseguir el directorio es un error fatal. La función **log.Fatalf** imprime un mensaje y llama **os.Exit(1)**.

```
var cwd string
func init() {
    cwd, err := os.Getwd() // error de compilación: no usado:  cwd
    if err != nil {
        log.Fatalf("os.Getwd failed: %v", err)
    }
}
```

Dado que ni la **cwd** ni **err** se declararon en el bloque de la función **init**, la sentencia **:=** los declara ambos como variables locales. La declaración interna de **cwd** hace que la exterior sea inaccesible, por lo que la sentencia no actualiza la variable de nivel de paquete **cwd** de la forma prevista.

Los compiladores actuales de **Go** detectan que la variable local **cwd** nunca se utiliza y reportar esto como un error, pero no son estrictamente necesarias para realizar esta comprobación. Por otra parte, un cambio menor, tales como la adición de una sentencia de registro que se refiera a la **cwd** local frustraría la prueba.

```
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)
}
```

La variable global **cwd** permanece sin inicializar, y la salida del registro aparentemente normal ofusca el error. Hay varias maneras de hacer frente a este problema potencial. La más directa es evitar **: =** declarando **err** en una declaración **var** separada:

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

Hemos visto cómo los paquetes, los archivos, las declaraciones y las sentencias expresan la estructura de los programas. En los dos capítulos siguientes, veremos la estructura de los datos.