01: Go to go

Publicado por [Uriel Ruelas](https://codingornot.com/author/zainiaz07)

Golang (a.k.a. Go) es un poderoso lenguaje de programación que nos permite resolver una gran cantidad de problemas con algoritmos simples gracias a su sintaxis minimalista. Una característica muy importante del lenguaje sin duda es su concurrencia nativa, sin embargo, no te dejes engañar, Go tiene muchas otras cosas por las que aprender a utilizarlo.

Con las siguientes entradas puedes tener un primer acercamiento a Golang. Al terminar esta primera temporada sabrás lo básico de este lenguaje de programación.

[Iniciando Go for the lulz](https://codingornot.com/01-go-to-go-iniciando-go-for-the-lulz/)

* [Sintaxis, tipo de datos y palabras reservadas](https://codingornot.com/02-go-to-go-sintaxis-tipo-de-datos-y-palabras-reservadas/)
* [Variables](https://codingornot.com/03-go-to-go-variables/)
* [Constantes](https://codingornot.com/04-go-to-go-constantes/)
* [Operadores](https://codingornot.com/05-go-to-go-operadores/)
* [Condicionales](https://codingornot.com/06-go-to-go-condicionales/)
* [Arreglos](https://codingornot.com/07-go-to-go-arreglos/)
* [Ciclos](https://codingornot.com/08-go-to-go-ciclos/)
* [Funciones y ámbitos](https://codingornot.com/08-go-to-go-funciones-y-ambitos/)
* [Apuntadores](https://codingornot.com/10-go-to-go-apuntadores/)
* [Registros (estructuras) en Golang](https://codingornot.com/11-go-to-go-registros-estructuras-en-golang/)
* [Slices (trozos) en Golang](https://codingornot.com/12-go-to-go-slices-trozos-en-golang/)
* [Maps en Golang](https://codingornot.com/13-go-to-go-maps-en-golang/)
* [Recursión y conversión de tipos](https://codingornot.com/14-go-to-go-recursion-y-conversion-de-tipos/)

¡Buenas, chavos! comenzamos con el primero de los capítulos sobre programación en Go que traemos  para ustedes en codingornot. Soy Uriel y yo los acompañaré por este viaje a través de uno de los lenguajes de programación más jóvenes y potentes que hay en la actualidad… Sin más que agregar ¡COMENZAMOS!

¿Qué es Go?

Go es un lenguaje de programación compilado, de tipado estático (las variables se declaran con un tipo desde el principio), concurrente (se pueden realizar múltiples cálculos simultáneos) e imperativo (los estados del programa cambian en base a sentencias). La [página oficial](https://golang.org/) del proyecto puede expandir la definición de Go (a veces referido como Golang), pero para propósitos de los capítulos, la definición dada aquí basta.

El lenguaje fue desarrollado por empleados de Google entre 2007 y 2009 por Robert Griesemer, Rob Pike y Ken Thompson, quienes originalmente lo pensaron como un lenguaje para programación de sistemas. En 2009 fue lanzado al público bajo una licencia de código abierto; desde entonces se ha continuado con el lanzamiento de nuevas versiones pero ahora añadiendo las contribuciones de muchos colaboradores de la comunidad *Open Source*.

¿Por qué debería de aprender Go?

Muchas son las características que hace a Go un lenguaje de programación poderoso con un horizonte de posibilidades casi infinito:

1 –Pocas palabras reservadas que facilitan el aprendizaje.

2 – Incorpora su propio recolector de basura.

3 – Sintaxis minimalista y simple. Con Go, menos es más.

4 – Fácil implementación de concurrencia vía Gorutinas.

5 – Estructurado principalmente en funciones.

6 – Sintaxis similar a C y C++.

7 – Strings y Maps incorporados al lenguaje.

8 – Rápido de compilar.

Aspectos previos a comenzar

Hay que tener en cuenta algunos aspectos que han sido descartados de Go (cada uno descartado por alguna razón en específico) antes de comenzar:

1 – Herencia.

2 – Sobrecarga.

3 – Dependencias circulares entre paquetes.

4 – Aritmética de apuntadores.

5 – Aserciones.

Recordemos, los aspectos anteriores NO están presentes en Go.

La extensión de los archivos de código fuente es “.go”, de aquí en delante cuando se haga algún ejemplo se agregará esa extensión bajo entendimiento que reconocen que es la extensión de Go.

Instalación

Si se mantienen firmes en su idea de aprender Go entonces comencemos con la instalación. El método de instalación varía dependiendo del sistema operativo en el que se trabaje, por tanto te pasar a la página oficial si surge alguna duda extra durante la instalación, ahí podrás encontrar información adicional que podría ser de utilidad.

Instalación en Unix, Linux, Mac OS X o FreeBSD:

1 – Descarga el archivo de instalación desde [aquí](https://golang.org/dl/)

2 – Extraer el archivo en /usr/local/

3 – Agregar /usr/local/go/bin a las variables de entorno PATH:

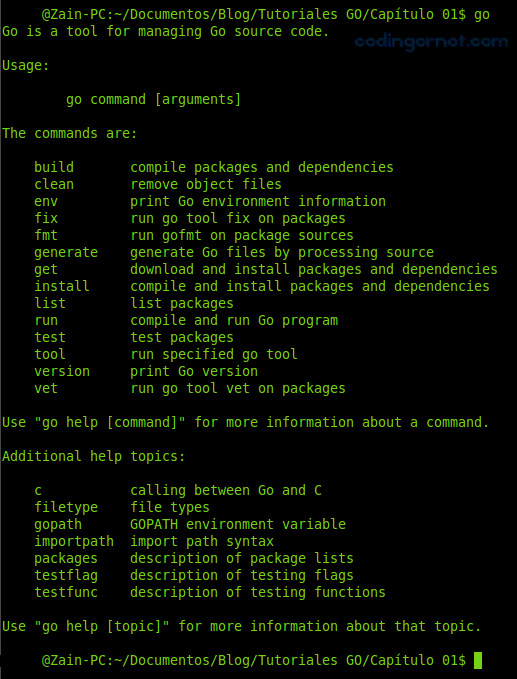
export PATH=$PATH:/usr/local/go/bin

Instalación en Windows:

Basta con ejecutar el archivo .msi que puedes descargar [aquí](https://golang.org/dl/), y seguir las instrucciones. Por defecto se instala en c:\Go. El instalador automáticamente debería de agregar c:\Go\bin en las variables de entorno PATH. Si se tiene abierta alguna ventana de símbolos de sistema (command prompts) se recomienda cerrarla y abrirla de nuevo.

Verificar la instalación

Para constatar que efectivamente se ha descargado e instalado el compilador de Go en nuestro equipo, basta con escribir “go” (sin comillas) desde nuestra terminal y se desplegará una lista de posibles comandos admitidos.



Para Windows el método de verificación es compilar nuestro primer código fuente para constatar que realmente tenemos Go instalado. En los demás sistemas operativos también se recomienda realizar este procedimiento para aprender a compilar un archivo “.go” desde Terminal:

1 – Crearemos una carpeta en alguna ruta de nuestro equipo, puede tener cualquier nombre.  
2 – Abriremos nuestro editor de texto plano e ingresaremos las siguientes líneas de código en un archivo en blanco:

1 package main

2 import "fmt"

3

4 func main() {

5 fmt.Println("¡Has instalado Go correctamente!")

6 }

3 – Guardaremos el archivo como “prueba1.go”. El nombre no importa mucho, lo importante es que su extensión sea “.go”.  
4 – Desde símbolos de sistema (o terminal) nos moveremos por los directorios de nuestra unidad hasta estar en la misma carpeta en la que guardamos nuestro archivo “prueba1.go”.  
5 – Una vez estando en la misma carpeta que nuestro archivo escribiremos el siguiente comando:

go run prueba1.go

Nota: El comando “run” compila y ejecuta el codigo fuente pero no genera un archivo ejecutable. El comando “build” es el que se encarga de generar ejecutables.

6 – Deberíamos de poder leer en terminal “¡Has instalado Go correctamente!”.

¡Felicidades! Has creado tu primer programa en Go. Si no aparece el mensaje y en su lugar se muestra algún mensaje de error, verifica que hayas escrito correctamente el código fuente, o en su defecto que hayas instalado correctamente Go.

02. Sintaxis, tipo de datos y palabras reservadas

por [Uriel Ruelas](https://codingornot.com/author/zainiaz07)

¡Buenas! Continuamos los capítulos de programación en Go. Recordemos que en el capítulo anterior revisábamos algunos aspectos básicos del lenguaje, así como también aprendimos a instalar el compilador en nuestros equipos. En este capítulo continuaremos con algunos aspectos de carácter general de la sintaxis de Go antes de saltar a los temas de estructura de bloques.

Estructura de un archivo .go

Retomando el código de ejemplo del [capítulo anterior](https://codingornot.com/01-go-to-go-iniciando-go-for-the-lulz/) con el cual verificamos la correcta instalación de Go desde símbolos de sistema o terminal, en dicho ejemplo podemos identificar las siguientes secciones en el código:

1 – Declaración de paquetes.  
2 – Paquetes importados.  
3 – Funciones.  
4 – Variables.  
5 – Declaraciones y expresiones.  
6 – Comentarios (no abordados en dicho ejemplo).

Antes de continuar es necesario conocer la sintaxis de los comentarios en Go la cual es:

// para comentar una línea

/\* para comentario en mas de una línea \*/

Todo comentario es ignorado por el compilador independientemente de su contenido. Solamente es de ayuda para el programador.

Los espacios en blanco son un caso similar a los comentarios. Tal y como en C y C++ puedes poner espacios en blanco u otros caracteres similares (tabulares por ejemplo) en cualquier parte del código sólo tomando unas pequeñas consideraciones (**no puede haber caracteres en blanco en el nombre de un identificador(variables,etc)**, en Go también se pueden tomar la mayoría de esas libertades. La libertad más importante de la que se **carece** en Go es la posibilidad de separar la llave de apertura de una sentencia o función por medio de un caracter de nueva línea, es decir:

1 package main

2 import "fmt"

3 func main()

4 { /\* unexpected semicolon or newline before { \*/

5 fmt.Println("Hello, World!")

6 }

lo correcto es:

1 package main

2 import "fmt"

3 func main() {

4 fmt.Println("Hola, mundo!")

5 }

Ahora continuemos con el tema principal, la estructura de los programas. Recordando el ejemplo del capítulo anterior, ahora se ha añadido un comentario previo que identifica cada parte del programa:

1 /\* Todo programa o archivo de Go se inicia con la línea paquete(package) principal(main) \*/

2 package main

3 /\* carga(importación) de los paquetes(grupos) de programas(funciones), preelaborados. En esta

4 oportunidad se carga el paquete **fmt** que contiene el grupo de funciones y métodos de entrada y salida \*/

5 import "fmt"

6 /\* Se procede con la edición de la Función principal que dirige la ejecución del programa \*/

7 func main() { // Llave de inicio del grupo de instrucciones que dirige la ejecución del programa

8 /\*Del paquete **fmt** se toma el método **Println** el cual muestra el texto en pantalla \*/

9 fmt.Println("¡Has instalado Go correctamente!")

10 } // Llave de cierre del grupo de instruciones que dirige la ejecución del programa

Cabe señalar que no existe algún carácter especial que se deba añadir para terminar una línea de código mas que el carácter de fin de línea. Internamente el compilador agrega el punto y coma “;” al final de cada sentencia.

1. fmt.Println("Println imprime el texto entre comillas")
2. fmt.Println("y luego ejecuta un salto de línea")
3. fmt.Println("como lo habrás observado")

Agregando las líneas anteriores al programa (eliminando los comentarios) obtendremos:

1 package main

2 import "fmt"

3 func main() {

4 fmt.Println("¡Has instalado Go correctamente!")

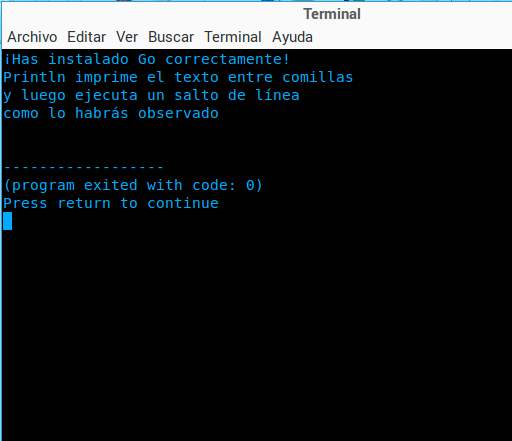
5 fmt.Println("Println imprime el texto entre comillas")

6 fmt.Println("y luego ejecuta un salto de línea")

7 fmt.Println("como lo habrás observado")

8 }

Al ejecutar obtendremos:



Identificadores y palabras reservadas

Al igual que en muchos otros lenguajes de programación, en Go puedes declarar variables, funciones y elementos con un identificador único que tú desees. Un identificador inicia con una letra de la “A”-“Z” (mayúscula o minúscula) o bien con un guión bajo (“\_”) seguido de una combinación de ninguna o varias letras, guiones y dígitos decimales( secuencia de caracteres):

a

\_x9

EstaVariableEsExportada

Se debe de tener cuidado que los identificadores nunca sean iguales a las palabras reservadas. La siguiente es la **lista de palabras reservadas** de Go:

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

Tipos de datos

Como se mencionaba desde el capítulo anterior, Go es un lenguaje de programación con tipado estático y por tanto es necesario declarar las variables antes de usarlas. La razón por la que esto es necesario es porque el compilador necesita conocer cuánto espacio en memoria necesita reservar para variables y cómo interpretar la información guardada en ese espacio de memoria.

Las variables en Go se clasifican como:

|  |  |
| --- | --- |
| Booleanas | Variables lógicas con valor true o false. |
| Numéricas | Representan números enteros o de punto flotante. |
| Cadenas | Secuencia de bytes que representan cadenas de texto inmutable, no es posible cambiar el contenido de una cadena. |
| Derivadas | Punteros, arreglos, estructuras, uniones, funciones, slices, interfaces, Maps, Channels. |

Rangos de valores de enteros y flotantes

Los números enteros que maneja Go son independientes de la arquitectura del procesador, es decir, no cambian los rangos de valores entre arquitecturas de procesadores como ocurre con C. Los tipos de enteros son:

|  |  |  |
| --- | --- | --- |
| **Tipo** | **Descripción** | **Rango** |
| uint8 | Unsigned | 8-bits (0 a 255) |
| uint16 | Unsigned | 16-bits (0 a 65535) |
| uint32 | Unsigned | 32-bits (0 a 4294967295) |
| uint64 | Unsigned | 64-bits (0 a 18446744073709551615) |
| int8 | Signed | 8-bits (-128 a 127) |
| int16 | Signed | 16-bits (-32768 a 32767) |
| int32 | Signed | 32-bits (-2147483648 a 2147483647) |
| int64 | Signed | 64-bits (-9223372036854775808 a 9223372036854775807) |

Por su parte los tipos numéricos con punto flotante son:

|  |  |
| --- | --- |
| **Tipo** | **Descripción** |
| float32 | IEEE-754 32-bit floating-point numbers |
| float64 | IEEE-754 64-bit floating-point numbers |
| complex64 | Números complejos con float32 partes reales e imaginarias |
| complex128 | Números complejos con float64 partes reales e imaginarias |

Además de números decimales enteros y de punto flotante, Go cuenta con otras implementaciones con sus tamaños específicos:

|  |  |
| --- | --- |
| **Tipo** | **Descripción** |
| byte | 8-bits (0 a 255) |
| rune | 32-bits (-2147483648 a 2147483647) |
| uint | 32 o 64 bits |
| int | 32 o 64 bits |
| uintptr | Entero guarda los bits no interpretados de un puntero |

Ya conocemos un poco más sobre la estructura de un programa de Go, sus palabras reservadas y rangos de datos; principalmente el rango de datos es muy útil ya que es bastante común ingresar a una variable el valor que sea cuando estás acostumbrado a lenguajes de tipado dinámico.

03. variables

Hasta este momento deberías conocer al menos un significado aproximado de lo que es una variable. Si no es así, no te preocupes, basta con asimilar que una variable es un valor dentro de nuestro programa que cambia dependiendo de diversos factores. Una variable tiene asignado un espacio en memoria en el que se almacena la información que ésta contiene. Mientras algunos lenguajes como C se han distinguido por su tipado tan estricto; otros lenguajes como Python son altamente reconocidos por su sencillez ya que las variables pueden declararse de cualquier tipo válido y cambiarlo durante la ejecución. Go es similar a C; necesita que todas las variables sean declaradas ya sea con tipo estático o tipo dinámico.

Declaración de variables

Todas las variables en Go tienen un tipo por el cual rigen su valor y su representación. La gran diferencia con otros lenguajes de programación como C, es que Go permite la declaración de variables con un tipo dependiente del tipo de dato de otra variable (esto es conocido como tipo dinámico o inferencia de tipo).

Una declaración en Go sigue el siguiente formato:

**var** variables tipos\_de\_dato(opcional)

Declaración de una sola variable:

var x int

var v4 float32

var nombre string

Declaración de varias variables:

var x, y int

var v1, v2, v3 float

var nombre, apellido string

Inilización de variables

La inicialización de variables puede ocurrir durante cualquier parte de la ejecución del programa o durante la declaración.

Inicialización de variables durante la declaración de variables, especificando o no el tipo de variable :

var x = 3

var peso = 28

var nombre = "Rita"

var y int = 3

var altura float32 = 1.68

var apellido string = "Rita"

Inicialización posterior a la declaración de variables:

var z float32

z = 1.79

var m int

m = 32

var direccion string

direccion = "Calle Principal # 4148"

Inicializar más de una variable en una declaración:

var x, y, z = 3, 4.0 ,"Hola"

Lo anterior es igual a escribir:

1. var x = 3
2. var y = 4.0
3. var z = "Hola"

¿Por qué se han mezclado enteros, flotantes y cadenas? La razón es muy gratificante: porque al inicializar las variables durante su declaración, pueden asignarse diferentes tipos de valores siempre y cuando no se anteponga un tipo de dato trunco… vamos, que no se haga esto:

var x, y, z int = 2, "Hola", 3.4

¿Qué sucede? Sencillo, desde que se antepone un tipo de dato a la inicialización, se le está especificando al compilador que las variables la línea serán tratadas como enteros (int). Una vez que se intenta inicializar **y** con “Hola” ocurre un error con el tipo de datos puesto que **y** es un entero y “Hola” es una cadena de caracteres.

Digamos que necesitamos una variable **X** con el mismo tipo de dato de una ya existente **Y**, pero por alguna razón no sabemos el tipo de dato de la variable **Y**, o en su defecto **X** puede tomar el tipo de dato de **Y** o de **Z** dependiendo del comportamiento del algoritmo. Para casos similares donde nuestras variables nuevas pueden ser de un tipo de dato no conocido, tenemos el tipo dinámico de Go:

1. var x = "Hola a todo el mundo"
2. y:= x

En la primera línea tenemos una declaración de una variable tipo cadena. En la segunda línea es donde podemos identificar una variable con tipo dinámico: **y** toma el valor de **x**, sea cual este sea (por obvias razones también toma el tipo de dato). La inicializaciones con tipo dinámico solamente son posibles con nuevas variables como en el ejemplo anterior donde **y** no había sido declarada antes. No es posible cambiar el tipo de dato de una variable a la que ya se le ha asignado un tipo de dato anteriormente. Las siguientes líneas provocarían un error al momento de compilar:

1. var y, z = 5, “Cadena”
2. /\*¡Error! La palabra reservada 'var no se utiliza
3. con tipos dinámicos\*/
4. var x:=y
5. /\*¡Error!, 'z' ya tiene un tipo de dato, no se
6. puede asignar un tipo nuevo\*/
7. z:=y

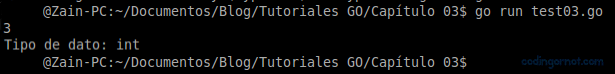
Tipo de dato en ejecución

Gracias al tipo dinámico podría llegar a ser complicado identificar el tipo de dato de una variable, principalmente cuando hablamos de proyectos muy grandes. Go incluye dentro de su biblioteca *fmt* varias funciones de impresión, entre ellas una que nos brinda la opción de conocer el tipo de dato de las variables en ejecución:

1. var x = 3
2. y:= x
3. fmt.Println(x)
4. fmt.Printf("Tipo de dato: %T\n",y)

*Printf* nos permite imprimir en pantalla texto con el formato que el programador especifique, a diferencia de *Println* que imprime texto con un formato por defecto. %T es el especificador de formato, en este caso señala que lo que se imprimirá será el tipo de dato de la variable después de la coma. *Printf* es el equivalente a *printf* de C.

Las líneas anteriores producen la siguiente salida:



Hemos estudiado la declaración e inicialización de variables dentro de Go. Como podemos identificar, a pesar de que el tipo dinámico puede llegar a ser muy útil, realmente el tipado estático de Go continúa casi tan estricto como en C. ¿Qué tan útil es el tipo dinámico? Depende del programador y de los proyectos en los que se utilice, pero sin lugar a dudas es una herramienta con mucho potencial.

04. constantes

As Constant As The Stars

En el capítulo anterior se estudió la declaración e inicialización de variables, pero muchas veces necesitamos manejar valores que ya conocemos antes de la ejecución de nuestro programa. En este capítulo se estudiarán las constantes en Go.

¡As The Stars, As The Stars! ¡Comenzamos!

Go es un lenguaje de tipado estático que no permite las operaciones entre variables de diferentes tipos. No podemos sumar un *int* a un *float*, e incluso no se permite sumar un *int* a un *int32* ya que, por naturaleza, son tipos de datos distintos (aunque sean similares).

En C y otros lenguajes de programación, se permiten operaciones entre diferentes tipos de datos gracias a las conversiones aritméticas automáticas. A veces esas operaciones suelen pasar desapercibidas para los programadores, como aquellas entre *unsigned* y *signed, long y short* etc. Pero se han llegado a identificar algunos problemas de compatibilidad y errores de truncamiento ocasionados por dicha “flexibilidad” que permite mezclar diferentes tipos de datos.

Desde hace mucho tiempo C incluye reglas de conversiones aritméticas que han cambiado a lo largo de sus ciclos de vida para adaptarse a lo que necesitan los programadores. En Go, por el contrario, desde el principio los desarrolladores optaron por prohibir mezclar diferentes tipos de datos; así pues si quieres sumar **x** y **y**, ambos deben de ser del mismo tipo o, en su defecto, especificar de qué tipo será el resultado. Aunque pareciera un poco complicado al principio, realmente es una buena costumbre que no está de más adoptar (¿Ya mencioné que nuestro código se vuelve más legible?).

Contantes en GO

Las constantes en Go, como en otros lenguajes de programación, son valores escalares que no pueden cambiar el valor que almacenan. La palabra reservada ***const*** es la que se utiliza para declarar constantes en Go. Una constante puede ser construida por el usuario directamente, por una expresión (sumas, restas etc.), o por funciones.

Existen constantes de todos los tipos, desde enteros y flotantes, hasta cadenas de texto. Utilizaremos una constante entera para comenzar. Supongamos que tenemos el siguiente número entero: 42

¿Qué tipo de dato tiene ese entero? ¿Entero (int)?

¿La pregunta contiene la respuesta? ¿Seguro?

¿Podrá tratarse de una constante?…

Decir que es un entero lleva algo de razón, sin embargo no es la respuesta correcta. Sí es una constante aunque no tiene ningún tipo asignado, es solamente un entero constante sin tipo (*untyped int constant*). Si deseamos asignar un nombre al entero anterior utilizamos *const*:

const numerito = 42

¿De qué tipo es **numerito**? … Efectivamente, no deja de ser un entero constante sin tipo (*untyped int constant*). La característica principal de una constante sin tipo es que, por naturaleza, no debe de seguir ninguna regla estricta para prevenir la combinación de tipos. *Son las constantes sin tipo las que permiten utilizar valores (casi) libremente en Go.*

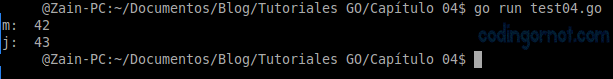
Vale la pena señalar que una constante puede tener un tipo asignado:

const variable int = 42

Tenemos una constante de tipo entero que va a ser regida por todas las reglas de combinación y rango que rigen a un entero.

Como se comentaba en el capítulo anterior, una variable puede tomar un tipo de dato dinámicamente al utilizar **:=** la primera vez que se utiliza dicha variable. Una variable también puede tomar el tipo de dato de una constante:

1. const numerito int = 42
2. var m int
3. m = numerito
4. fmt.Println("m: ", m)
5. j:= numerito
6. j = j + 1
7. fmt.Println("j: ", j)



La razón por la que **j** del ejemplo anterior puede aumentar su valor a pesar de haber sido inicializada por tipo dinámico a *const int*, es porque una variable cuyo tipo es asignado dinámicamente no puede tomar propiedades constantes, es decir, se inicializó solamente como *int*.

Tipo por defecto

Una constante puede no tener un tipo de dato asignado (untyped), sin embargo nos es posible realizar diferentes acciones con ella a tal punto que pareciera que en realidad sí lo tiene. Toda constante sin tipo tiene un tipo de dato implícito asignado, el cual es transferido al valor cuando se necesita realizar alguna acción que requiera un tipo de dato.

El tipo por defecto nos dice que una línea de código como ésta:

cadena:= "Ésta es una cadena"

O como ésta:

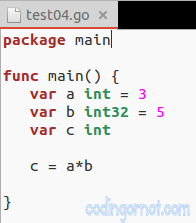
var cadena = "Esta es una cadena"

Realizan lo mismo que una línea como ésta:

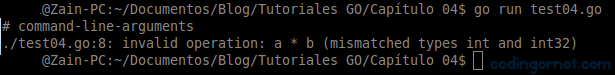
var cadena string = "Esta es una cadena"

¿Cuál fue la utilidad de este capítulo?

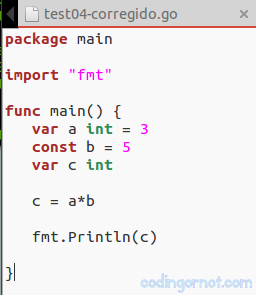
La finalidad principal de este capítulo, además de enseñar sobre el manejo de constantes en GO; es informar a los lectores que la facilidad de otros lenguajes de programación para permitir realizar operaciones entre diferentes tipos de dato, no se encuentra en Go. Los problemas con tipos de datos pueden ser un tema común entre las personas que recién se inician en Go ya que tenemos que éste es un lenguaje de programación muy estricto con sus variables. El siguiente es un ejemplo de un error ocasionado por intentar realizar operaciones entre diferentes tipos de datos:



Salida:



Si utilizamos constantes podemos realizar la operación del ejemplo anterior como se muestra a continuación:



Salida:



En el código fuente corregido hay otros cambios además del uso de constantes, eso porque en el primer ejemplo se inicializó c pero no se utilizó en ningún lugar. Go previene las declaraciones o inicializaciones que no se utilizan. Para lograr ejecutar el ejemplo anterior se debe de utilizar la variable c (al menos imprimirla en terminal).

Más adelante se estudiarán las conversiones de tipos de dato, un tema que facilitará programar algoritmos con muchas variables numéricas que interactúan entre sí pero no pueden ser constantes.

Hemos estudiado las constantes en Go. Aprendimos que una constante puede tener (o no tener) un tipo de dato asignado, que existe un tipo por defecto, las constantes permiten las operaciones entre distintos tipos libremente y gracias a lo estricto que es Go con sus tipos de datos, se pueden evitar algunos de los errores y problemas que lenguajes como C traen consigo desde hace años.

05. operadores

por [Uriel Ruelas](https://codingornot.com/author/zainiaz07)

Todo lenguaje de programación necesita de operadores para realizar las tareas que se necesiten programar, Go no es la excepción. Si ya conoces sobre lenguajes de programación, probablemente muchos de estos operadores te resultarán conocidos e inclusive podrías pensar que son poco necesarios; sin embargo ese podría no ser el caso para otros usuarios quienes, posiblemente, nunca antes hayan programado algo en su vida.

Antes de comenzar a explicar los operadores de Go cabe aclarar que un operador es un símbolo o combinación de símbolos que especifican alguna operación o relación matemática al compilador, es decir, los operadores son el medio para trabajar con nuestras variables. En capítulos anteriores se realizaron pequeños ejercicios donde se utilizaban operadores que suelen ser naturales para casi cualquier persona (por ejemplo una suma +), pero existen otra clase de operadores que resultan de suma importancia cuando se necesita programar algo un poco más complejo.

Go brinda los siguientes tipos de operadores:

* Operadores aritméticos
* Operadores relacionales
* Operadores lógicos
* Operadores de asignación
* Operadores binarios
* Operadores de dirección

 Operadores aritméticos

 Los operadores aritméticos nos permiten realizar operaciones básicas sobre nuestras variables y constantes. Supongamos que tenemos una variable X almacenando un int 10, y una variable Y que almacena un int 5.

* X = 10
* Y = 5

|  |  |  |
| --- | --- | --- |
| **Operador** | **Descripción** | **Ejemplo** |
| + | Suma los operandos de los extremos. | X + Y resulta 15 |
| – | Substrae el operando de la derecha al operando de la izquierda. | X – Y resulta 5 |
| \* | Multiplica los operandos de los extremos. | X \* Y resulta 50 |
| / | División de el número de la izquierda (numerador) entre el número de derecha (denominador). | X / Y resulta 2 |
| % | Operador modular o de residuo de división entera. | X / Y resulta 0 |
| ++ | No esta permitido |  |
| — | No esta permitido |  |

Operadores relacionales

Los operadores relacionales son aquellos que devuelven un valor de verdad (verdadero o falso) cuando se resuelve una expresión:

* X = 1
* Y = 2

|  |  |  |
| --- | --- | --- |
| **Operador** | **Descripción** | **Ejemplo** |
| == | Devuelve verdadero cuando ambos operandos son iguales. Devuelve falso en cualquier otro caso (idéntico a…). | X == Y devuelve falso |
| != | Devuelve verdadero solamente cuando los operandos son distintos (diferente a…). | X != Y devuelve verdadero |
| > | Devuelve verdadero cuando el operando de la izquierda es mayor que el de la derecha (mayor que…). | X > Y devuelve falso |
| < | Devuelve verdadero cuando el operando de la izquierda es menor que el de la derecha (menor que…). | X < Y devuelve falso |
| >= | Devuelve verdadero cuando el operador de la izquierda es mayor o igual al de la derecha (mayor o igual que..). | X >= Y devuelve falso |
| <= | Devuelve verdadero cuando el operador de izquierda es menor o igual a el de la derecha (menor o igual a…). | Z <= Y devuelve verdadero |

Tabla de verdad y concepto de operador lógico

En los subtemas siguientes se trabajará con tablas de verdad. Una tabla de verdad es una herramienta visual que ejemplifica el resultado de un operador lógico con todas las posibles combinaciones de sus operandos.

Además, para entender la forma en la que funciona un operador lógico, es necesario comprender su definición formal. En seguida se definen tres de los operadores fundamentales para este capítulo:

* Conjunción: Es una unión de valores de verdad, su símbolo en Go es &. “A & B” puede ser interpretado como “A y B”.
* Disyunción: Es una separación o desunión de valores de verdad, su símbolo es |. “A | B” se puede interpretar como “A o B”.
* O-exclusiva (XOR): Es una separación de elementos cuyo valor de verdad sea el mismo, su símbolo es ^. “A ^ B”  se interpreta como “o A, o B, pero no ninguno ni ambos”.

La siguiente es la tabla de verdad de los operadores lógicos antes mencionados:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **A** | **B** | **A & B (conjunción)** | **A | B (disyunción)** | **A ^ B (XOR)** |
| 0 | 0 | 0 | 0 | 0 |
| 0 | 1 | 0 | 1 | 1 |
| 1 | 0 | 0 | 1 | 1 |
| 1 | 1 | 1 | 1 | 0 |

Operadores lógicos a nivel de bits (bit a bit)

Los operadores de bits trabajan sobre cadenas de bits a las que aplican conexiones lógicas. A nivel de bits, 0 se considera falso y 1 verdadero.

En la siguiente tabla se explica cómo funcionan las conexiones lógicas de bits en Go:

|  |  |  |  |
| --- | --- | --- | --- |
| **Operando** | **Nombre** | **Descripción** | **Ejemplo** |
| & | Conjunción (AND). | Verdadero (1) mientras ninguno de los operandos sea falso (0). | (0011 0011 & 1100 0011) es igual a: 00000011 |
| | | Disyunción (OR). | Verdadero mientras al menos uno de los operandos sea verdadero. | (0011 0011 | 1100 0011) es igual a: 1111 0011 |
| ^ | Disyunción exclusiva (XOR). | Verdadero mientras los operandos sean distintos. | (0011 0011 ^ 1100 0011) es igual a: 1111 0000 |
| << | Corrimiento de bits a la izquierda. | Los bits son movidos a la izquierda la cantidad de posiciones que se especifique. | (0011 0011 << 2) es igual a: 1100 1100 |
| >> | Corrimiento de bits a la derecha. | Los bits son movidos a la derecha la cantidad de posiciones se especifique. | (0011 0011 >> 2) es igual a: 0000 1100 |

Operadores lógicos

Regularmente se estudian primero los operadores lógicos que los operadores de bits, sin embargo los conocimientos necesarios que se adquieren al estudiar el comportamiento de bits (principalmente la tabla de verdad) son necesarios con los operadores lógicos, sin contar que las operaciones lógicas con bits son de menor nivel que las con expresiones.

Los operadores lógicos funcionan de forma similar a los lógicos binarios, es decir, es necesario conocer el comportamiento del operador en la tabla de verdad para conocer los posibles resultados. Consideremos:

1. var A bool = true

2. var B bool = false

3. var X int = 9

4. var Y = 5

|  |  |  |
| --- | --- | --- |
| **Operador** | **Descripción** | **Ejemplo** |
| && | Operador lógico de conjunción (AND). Compara dos valores bool o expresiones relacionales. | A && B resulta false  X > 0 && Y < 6 resulta true |
| || | Operador lógico de disyunción (OR). Compara dos valores bool o expresiones relacionales. | A || B resulta *true*  X < 0 || Y > 6 resulta false |
| ! | Operador de negación. Invierte (niega) el valor bool del operando. | !A resulta false  !B resulta true |

Operadores de asignación

Los operadores de asignación nos permiten modificar el valor de variables a lo largo de la ejecución de nuestro programa.

|  |  |  |
| --- | --- | --- |
| **Operador** | **Descripción** | **Ejemplo** |
| = | Operador de asignación simple. Asigna el operando de la derecha al operando de la izquierda. | X = Y + Z asigna a X la suma de Y y Z (ni Y ni Z se modifican) |
| += | Operador de suma y asignación. Asigna al operando de la izquierda la suma de sí mismo y el operando de la izquierda. | X += Y asigna a X el valor de X + Y |
| -= | Operador de resta y asignación. Asigna al operando de la izquierda la resta de sí mismo menos el operando de la izquierda. | X -= Y asigna a X el valor de X – Y |
| \*= | Operador de multiplicación y asignación. Asigna al operando de la izquierda la multiplicación de sí mismo y el operando de la izquierda. | X \*= Y asigna a X el valor de X \* Y |
| /= | Operador de división y asignación. Asigna al operando de la izquierda la división de sí mismo entre el operando de la izquierda. | X /= Y asigna a X el valor de X /Y |
| %= | Operador de módulo y asignación. Asigna al operando de la izquierda la resta de sí mismo menos el operando de la izquierda. | X %= Y asigna a X el valor de X % Y |
| <<= | Operador de desplazamiento de bits a la izquierda y asignación. Asigna al operando de la izquierda el resultado de aplicarle un desplazamiento de n bits a la izquierda. | X <<= 2 es igual a escribir X = X<<2 |
| >>= | Operador de desplazamiento de bits a la derecha y asignación. Asigna al operando de la izquierda el resultado de aplicarle un desplazamiento de n bits a la derecha. | X >>= 2 es igual a escribir X = X>>2 |
| &= | Operador de conjunción de bits y asignación. Asigna al operando de la izquierda el resultado de aplicar una conjunción con el operando de la derecha. | X &= Y es igual a escribir X = X&Y |
| |= | Operador de disyunción de bits y asignación. Asigna al operando de la izquierda el resultado de aplicar una disyunción con el operando de la derecha. | X ^= Y es igual a escribir X = X&Y |
| ^= | Operador de disyunción exclusiva de bits y asignación. Asigna al operando de la izquierda el resultado de aplicar una disyunción exclusiva con el operando de la derecha. | X ^= Y es igual a escribir X = X^Y |

Operadores de dirección

Por último, en Go también contamos con operadores de direcciones de variables y apuntadores.

|  |  |  |
| --- | --- | --- |
| **Operador** | **Descripción** | **Ejemplo** |
| & | Regresa la dirección en memoria del operando. | &X regresa la dirección en memoria de X |
| \* | Apuntador a una variable. | \*P apunta a una variable |

Precedencia de operadores

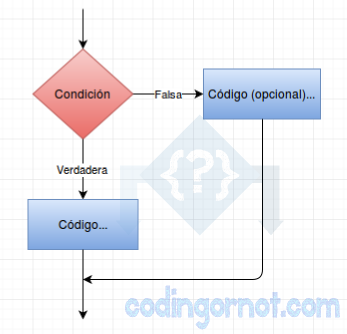
La precedencia de operadores es la que nos indica en qué orden será evaluada la expresión, es decir, es la que nos dice que en una expresión como esta: a + b \* c, la primera operación que se realiza es la multiplicación debido a que el operador multiplicación es de mayor precedencia que el de suma.

Los operadores también tienen asociatividad, es decir, hacia dónde se evalúan sus operandos. La siguiente es una tabla con los operadores estudiados a lo largo de este capítulo (mayor precedencia significa que tiene mayor prioridad de resolución):

|  |  |  |  |
| --- | --- | --- | --- |
| **Precedencia** | **Categoría** | **Operador** | **Asociatividad** |
| 15 | Posfijo | () [] -> . ++ – – | Izquierda a derecha. |
| 14 | Unitario | + – ! ~ ++ – – (type)\* & sizeof | Derecha a izquierda. |
| 13 | Multiplicativo | \* / % | Izquierda a derecha. |
| 12 | Adición | + – | Izquierda a derecha. |
| 11 | Desplazamiento | << >> | Izquierda a derecha. |
| 10 | Relacional | < <= > >= | Izquierda a derecha. |
| 9 | Igualdad | == != | Izquierda a derecha. |
| 8 | AND nivel bits | & | Izquierda a derecha. |
| 7 | XOR nivel bits | ^ | Izquierda a derecha. |
| 6 | OR nivel bits | | | Izquierda a derecha. |
| 5 | AND lógico | && | Izquierda a derecha. |
| 4 | OR lógico | || | Izquierda a derecha. |
| 3 | Condicional | ?: | Derecha a izquierda. |
| 2 | Asignación | = += -= \*= /= %=>>= <<= &= ^= |= | Derecha a izquierda. |
| 1 | Coma | , | Izquierda a derecha. |

06. condicionales

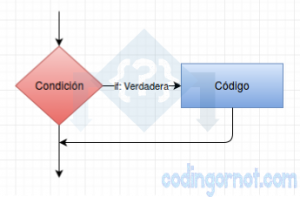
Regularmente cuando programamos nuestros algoritmos, llega un momento donde las acciones que se deben realizar dependerán un poco o mucho del valor o el estado de alguna variable (o constante) que especifiquemos. Por ejemplo: si nuestro programa está destinado a realizar unos cálculos con valores de entrada del usuario, y dependiendo de la magnitud de ese cálculo se sume o reste algún valor **N** para acotar el resultado; en ese caso es necesario recurrir a sentencias condicionales que verifiquen la magnitud de nuestra variable.



Cuando se encuentra una condición, ésta es evaluada; si el resultado es verdadero (*true*), entonces continúa con el código que se haya indicado; si por otro lado el resultado de evaluar la condición es falso (*false*), esa sección de código no se ejecuta y continúa el resto del programa. También es posible especificar una serie de sentencias que solamente se ejecute cuando la condición no se cumpla.

Sentencia *if*

La sentencia ***if*** (si, en inglés) verifica que la expresión que se le indique sea verdadera para entonces ejecutar la sección de código destinada solamente a ejecutarse cuando eso suceda. Si la condicional es falsa, se continúa ejecutando el código de forma secuencial.



Ejemplo:

1. package main
2. import "fmt"
3. func main() {
4. /\*variable local de tipo entero\*/
5. var calificacion int = 5
6. /\*Sentencia if, que verifica si calificación es menor a 6\*/
7. if calificacion < 6 {
8. /\*Si la condición se cumple, imprime\*/
9. fmt.Println("Reprobaste")
10. }
11. fmt.Println("Tu calificación fue de: ", calificacion)
12. }

Salida:

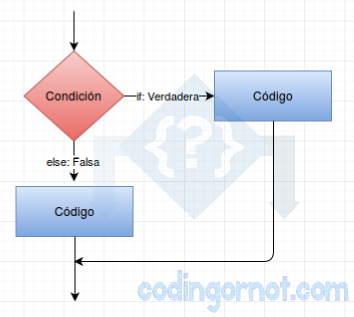
1. Reprobaste
2. Tu calificación fue de: 5

A diferencia de C, en Go es obligatorio incluir las sentencias después de ***if*** entre llaves independientemente de la cantidad de líneas que haya.

Sentencia *if-else*

Cuando necesitamos que el programa ejecute ciertas sentencias cuando la condición es verdadera, pero a su vez necesitamos que ejecute otras sentencias solamente cuando la condición es falsa, necesitamos una sentencia condicional ***if-else***.

El siguiente diagrama ilustra el funcionamiento de if-else:



Ejemplo:

1. package main
2. import "fmt"
3. func main() {
4. /\*variable local de tipo entero\*/
5. var calificacion int = 7
6. /\*Sentencia if, verifica calificación menor 6\*/
7. if calificacion < 6 {
8. /\*Si la condición se cumple\*/
9. fmt.Println("Reprobaste")
10. }else{
11. /\*Si la condición no se cumple\*/
12. fmt.Println("Aprobaste")
13. }
14. fmt.Println("Tu calificación fue de: ", calificacion)
15. }

Salida:

1. Aprobaste
2. Tu calificación fue de: 7

Debe tenerse especial cuidado con la sentencia ***else*** puesto que si no se le localiza en seguida de la llave de cierre de ***if***, podría generar un error sintáctico:

1. if calificacion < 6{
2. /\*Si la condición se cumple\*/
3. fmt.Println("Reprobaste")
4. }
5. else{
6. /\*unexpected semicolon or newline before else\*/
7. fmt.Println("Aprobaste")
8. }

Es posible trabajar con múltiples condiciones que se verifiquen en cadena cuando la anterior no satisface por medio de ***else if*** y de ese modo verificar múltiples escenarios usando solamente sentencias ***if***. Sin embargo debe de tenerse en cuenta que:

* Un ***if*** puede tener 0 o 1 ***else*** que funciona para señalar las sentencias que se ejecutan cuando no se cumplió ninguna de las condiciones anteriores.

* Un if puede tener de 0 a N ***else if*** siempre y cuando no estén después de un ***else*** (ya que el ***else***, en caso de que exista, debe ser la sentencia final).

Si un ***else if*** se cumple, ninguno de los siguientes será siquiera revisado, continúa la ejecución del resto código.

El siguiente ejemplo ilustra el funcionamiento:

1. package main
2. import "fmt"
3. func main() {
4. /\*variable local de tipo entero\*/
5. var calificacion int = 9
6. /\*Sentencia if, verifica calificación menor a 6\*/
7. if calificacion < 6 { /\*Si la condición se cumple\*/ fmt.Println("Reprobaste.") }else if calificacion >= 6 && calificacion <= 8 {
8. /\*Segunda condición\*/
9. fmt.Println("Aprobaste.")
10. }else if calificacion == 9{
11. /\*Tercera condición\*/
12. fmt.Println("Aprobaste. Te fue muy bien.")
13. }else{
14. /\*Si ninguna de las anteriores se cumplió\*/
15. fmt.Println("Felicidades. Aprobaste con calificación perfecta")
16. }
17. fmt.Println("Tu calificación fue de: ", calificacion)
18. }

Salida:

1. Aprobaste. Te fue muy bien.
2. Tu calificación fue de: 9

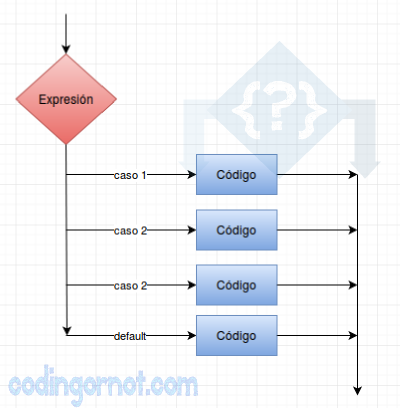
Sentencia switch

La sentencia ***switch*** permite evaluar una variable de prueba contra una lista de valores conocidos como casos (***case***). En Go un ***switch*** puede ser de dos tipos posible:

* **Expresión** – Los casos contienen expresiones que son comparadas con el valor de la variable de prueba.

* **Tipo** – Los casos contienen un tipo de dato que es comparado con el tipo de dato de la variable de prueba.

El siguiente diagrama muestra su funcionamiento:



**Switch de expresiones**

Las siguientes reglas aplican para los ***switch*** que evalúan expresiones:

* Si no se pasa una expresión, el valor por defecto es *true*.
* Puede haber cualquier cantidad de casos siempre y cuando cada caso esté seguido de al menos un valor contra el cual comparar, y dos puntos.
* Si se usa una expresión constante para un caso, debe de ser del mismo tipo que el valor de prueba, y debe de ser una constante o una literal.
* Cuando la variable evaluada es igual a un caso, el código después de los dos puntos se ejecuta. No se utiliza ***break*** para terminar el caso.
* Puede haber un caso por defecto (***default***) para ejecutarse cuando no se cumple ninguno de los otros casos. El caso por defecto debe de ir al final y tampoco necesita ***break***.

Ejemplo:

1. package main
2. import "fmt"
3. func main() {
4. /\*variable local de tipo entero\*/
5. var hora int = 10
6. /\*Se pasa hora como variable de prueba\*/
7. switch hora{
8. /\*Si hora coincide con alguna de las literales especificadas\*/
9. case 1, 2, 3, 4: fmt.Println("Aún es temprano")
10. case 5, 6, 7: fmt.Println("Está atardeciendo")
11. case 8: fmt.Println("Acaba de oscurecer")
12. case 9, 10, 11: fmt.Println("Ya es tarde")
13. default: fmt.Println("Es demasiado tarde")
14. }
15. }

Salida:

Ya es tarde

**Switch de tipo**

El funcionamiento y estructura es similar al ***switch*** de expresiones. Las siguientes reglas aplican para ***switch*** de tipos:

* La expresión usada en una sentencia debe de ser una variable o una ***interface*** (interfaz) de tipo.
* Puede haber cualquier número de casos en un switch. Cada uno es seguido por el valor contra el cual comparar y dos puntos.
* El tipo de dato para un caso debe ser el mismo tipo de dato que la variable de prueba, así como debe de ser de un tipo válido.
* Cuando la variable de prueba es igual a un caso, se ejecuta dicho caso y los demás no son verificados. No es necesario un ***break.***
* Puede haber un caso por defecto (***default***) para ejecutarse cuando no se cumple ninguno de los otros casos. El caso por defecto debe de ir al final y tampoco necesita ***break.***

Ejemplo:

1. package main
2. import "fmt"
3. func main() {
4. /\*variable interface sin tipo asignado\*/
5. var x interface{}
6. switch x.(type){ /\*Retorna el tipo de x\*/
7. /\*Casos\*/
8. case nil: fmt.Println("Es una variable tipo nil")
9. case int: fmt.Println("Es una variable tipo int")
10. case float64: fmt.Println("Es una variable tipo float64")
11. case int64: fmt.Println("Es una variable tipo int64")
12. default: fmt.Println("No es ninguno de los tipos anteriores")
13. }
14. }

Salida:

Es una variable tipo nil

Finalizando…

A lo largo de este capítulo hemos estudiado las sentencias condicionales de Go para aprovecharlas al momento de crear nuestros programas. Cabe aclarar que existe una sentencia más (***select***) pero será abordada más adelante.

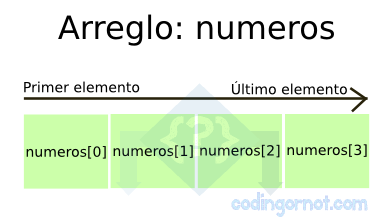
07 arreglos

por [Uriel Ruelas](https://codingornot.com/author/zainiaz07)

Como muchos otros lenguajes de programación, en Go también es posible trabajar con arreglos. Un arreglo es una estructura de datos que permite almacenar una colección de elementos del mismo tipo.

Los arreglos son sumamente útiles cuando necesitamos almacenar información de un mismo tipo de dato bajo un mismo identificador. Como ya se dijo, los arreglos almacenan colecciones de datos, pero suelen ser más sencillos de conceptualizar como una variable que almacena otras variables de un sólo tipo.

Los arreglos trabajan con direcciones de memoria contiguas; para acceder a uno de los elementos, regularmente, nos basta con utilizar el identificador y el número de elemento al que deseamos acceder



Estructura de un arreglo.

**Declarar arreglos en Go**

Para declarar arreglos en Go se necesita conocer la cantidad de miembros de la colección y su tipo de dato. El formato de declaración es el siguiente:

var nombre\_de\_variable [TAMAÑO] tipo\_de\_dato

El **nombre** del arreglo puede ser **cualquier identificador válido de Go**, el **tamaño cualquier número entero mayor a 0**, y el **tipo de dato cualquiera válido de Go**. Supongamos que deseamos almacenar los días de la semana en un arreglo de *string* llamado dias\_semana:

var dias\_semana [7] string

**Inicializar un arreglo en Go**

Los arreglos pueden ser inicializados elemento por elemento con sentencias de asignación:

1. var dias\_semana [0] = "domingo"
2. var dias\_semana [1] = "lunes"
3. var dias\_semana [2] = "martes"
4. var dias\_semana [3] = "miércoles"
5. var dias\_semana [4] = "jueves"
6. var dias\_semana [5] = "viernes"
7. var dias\_semana [6] = "sábado"

**Que se note**: En el ejemplo, las posiciones en el arreglo inician desde 0 y terminan hasta 6, es decir, son 7 elementos (como se especificó en la declaración). A ese número que se utiliza entre corchetes se le conoce como índice y es el que nos proporciona el acceso a las posiciones del arreglo con mucha facilidad. Los arreglos siempre **inician a partir del índice 0**, y por ende, **el último índice es el tamaño del arreglo menos 1**.

También se pueden inicializar al momento de la declaración:

1. var dias\_semana = [] string{"domingo","lunes",
2. "martes", "miércoles", "jueves", "viernes","sábado"}

**Acceder a elementos de un arreglos**

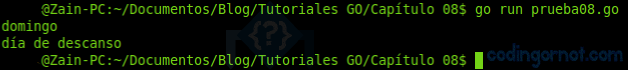
Para acceder a un elemento del arreglo se utiliza el índice de dicho elemento:

1. //Se accede al primer elemento del arreglo
2. fmt.Println(dias\_semana[0])
3. //Se accede al segundo elemento del arreglo
4. fmt.Println(dias\_semana[1])
5. ...
6. //Se accede al quinto elemento del arreglo
7. fmt.Println(dias:semana[4])
8. ...
9. // Se accede al séptimo y último elemento del arreglo
10. fmt.Println(dias\_semana[6])

Los elementos del arreglo pueden ser modificados con la misma mecánica:

1. fmt.Println(dias\_semana[0])
2. dias\_semana[0] = "día de descanso"
3. fmt.Println(dias\_semana[0])

Salida:



**Arreglos multidimensionales**

Los arreglos de los que se ha hablado hasta ahora han sido unidimensionales, es decir, se podía pensar en ellos como una colección en forma de lista de elementos, pero existen otra clase de arreglos a los que se les llama multidimensionales. Los arreglos multidimensionales pueden ser fácilmente comprendidos si se piensa en ellos como si fueran tablas o matrices en las que cada índice especifica una dimensión.

El formato para declarar arreglos multidimensionales es el mismo que el de los unidimensionales. Suponiendo que se desea declarar un arreglo bidimensional de 2×2:

var tabla [2][2] int

Si se desea inicializar el arreglo en declaración puede hacerse lo siguiente:

var tabla = [][]int {{1,2},{3,4}}

El arreglo anterior puede imaginarse como una tabla de 2 filas y 2 columnas

|  |  |  |
| --- | --- | --- |
| **Variable: tabla** | **Columna 0** | **Columna 1** |
| **Fila 0** | 1 | 2 |
| **Fila 1** | 3 | 4 |

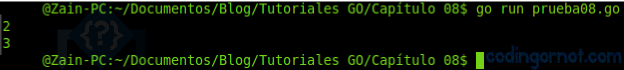
Para acceder a un elemento en específico se utilizan los índices que correspondan al elemento que necesitamos, por ejemplo:

1. var tabla = [][]int {{1,2},{3,4}}
2. fmt.Println(tabla[0][1])Finalizando…

En este capítulo estudiamos los ciclos en Go y cómo estos nos facilitan la programación. En la próxima entrada estudiaremos el uso de las funciones.

1. fmt.Println(tabla[1][0])

Salida:



Para modificar un elemento también se utilizan los índices:

1. tabla[0][1] = 3
2. tabla[1][1] = tabla[0][0] \* tabla[1][0]

**Finalizando…**

En este capítulo se estudiaron los arreglos como una herramienta versátil con múltiples propósitos en nuestros programas. Cabe señalar que, además, nos ayudarán a comprender rápidamente la utilidad de las sentencias que se estudiarán en los próximos capítulos.

08. ciclos

por [Uriel Ruelas](https://codingornot.com/author/zainiaz07)

Yo quería pegarlo 100 veces

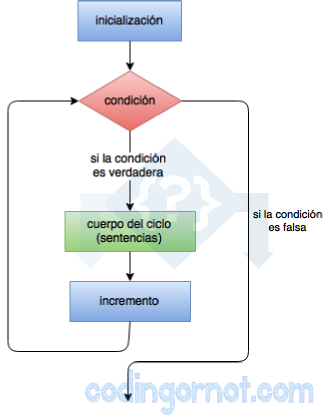
En la [entrada anterior](https://codingornot.com/07-go-to-go-arreglos/) estudiamos los arreglos porque nos serían de utilidad para entender fácilmente este tema. Hay situaciones donde necesitamos ejecutar una sección de código varias veces pero ¡qué flojera copiarlo y pegarlo una y otra vez! Para lograr esa tarea la mayoría de los lenguajes de programación incluyen las sentencias de ciclos.

Los ciclos permiten ejecutar repetidamente una o varias sentencias de forma consecutiva. Pueden ser mezclados con condicionales o, inclusive, con otros ciclos internos para lograr tareas que de otro modo serían casi imposibles.

De verdad quería pegarlo 100 veces

La única sentencia de ciclos existente en Golang (hasta agosto del 2016) es ***for***, contrario a muchos otros lenguajes de programación que permiten algunas como ***while***. La buena noticia es que ***for*** abarca todos los posibles usos que podrían tener otras sentencias similares, así que, realmente no se extraña mucho el ***while***.

Sentencia *for*



La sentencia ***for*** controla una sección de código para que se repita una cierta cantidad de veces especificada al escribir el programa o determinada en tiempo de ejecución. La sintaxis es la siguiente:

Sentencia\_for = "for" [Condición | Cláusula\_for | Rango] Bloque

Sintaxis de Cláusula\_for

Cláusula\_for = [Inicialización ";" Condición ";" Incremento]

Sintaxis de Rango:

Rango = [Lista\_expresiones "=" | Lista\_identificadores ":="] "range" Expresión

Sintaxis de Bloque:

Bloque = "{" Lista\_sentencias "}"

Nota: Es importante colocar la llave de apertura en la misma línea que la sentencia for.

Desglosando la sintaxis anterior:

* Cuando se especifica una condición, el ciclo se ejecutará mientras esta sea verdadera.
* Si se utiliza la sintaxis (inicialización; condición; incremento) ocurre que:

1. La inicialización solamente ocurre una vez y es la primera sentencia que se ejecuta. Se inicializan cualquier cantidad de variables de control del ciclo (aquellas que controlan las repeticiones del ciclo), aunque no es necesario hacerlo siempre y cuando aparezca el punto y coma después de donde debería de estar la inicialización.
2. La condición se evalúa en seguida de la inicialización y antes de repetir cada nuevo ciclo. Si es verdadera entonces el cuerpo se ejecuta, de lo contrario simplemente se salta todo el ciclo.

Después de que se ejecuta el cuerpo, se realiza el incremento. Este actualiza cualesquiera variables de control para de ese modo no incurrir en ciclos infinitos. Esta sección puede ser dejada en blanco siempre y cuando aparezca un punto y coma después de la condición.

1. La condición es evaluada nuevamente. Se vuelve al paso 2.

* En cambio, si se escribe un rango de valores, el ciclo se ejecuta para cada uno de los elementos de ese rango.

Ciclo *for* con una condición de paro

En el siguiente ejemplo se puede apreciar la sintaxis cuando se proporciona solamente una condición para detener el ciclo:

1. package main
2. import "fmt"
3. func main(){
4. var i = 0
5. for i < 10 {
6. fmt.Println("Valor de i:", i)
7. i++
8. }
9. }

Salida:



Ciclo *for* con inicialización, condición e incremento

El siguiente ejemplo ilustra el funcionamiento de esta clase de ciclos:

1. package main
2. import "fmt"
3. func main(){
4. var i int
5. for i=0 ; i < 10; i++ {
6. fmt.Println("Valor de i:", i)
7. }
8. }

Salida:



Ciclo *for* con un rango

Para comprender el funcionamiento de esta clase de for puede utilizarse el ejemplo siguiente:

1. package main
2. import "fmt"
3. func main(){
4. arreglo:=[7]int{0,1,4,6,10,9}
5. for i, j:= range arreglo{
6. fmt.Printf("Valor de j: %d en vuelta #%d\n", j,i)
7. }
8. for i:= range arreglo{
9. fmt.Printf("Valor de i: %d\n", i)
10. }
11. }

Salida:



Sentencia *break*

Go cuenta con una sentencia ***break*** cuyos usos son terminar el ciclo más interno en el que esté situado, o bien terminar un [***case*** de un ***switch***](https://codingornot.com/06-go-to-go-condicionales/).

1. package main
2. import "fmt"
3. func main(){
4. for i:=0 ; i < 10; i++ {
5. fmt.Printf("Valor de i: %d", i)
6. if i == 7{
7. fmt.Printf(" así que saldremos del ciclo...\n")
8. break
9. }
10. fmt.Printf("\n")
11. }
12. }

Salida:



Sentencia *continue*

La sentencia ***continue*** de Go sigue un funcionamiento similar al de ***break*** con la diferencia de que ***continue*** no termina el ciclo por completo, simplemente termina la iteración actual ignorando el código restante y continúa evaluando la condición de la siguiente iteración.

El siguiente ejemplo ilustra el funcionamiento de la sentencia:

1. package main
2. import "fmt"
3. func main(){
4. var i = 0
5. for i < 10 {
6. fmt.Printf("Valor de i: %d", i)
7. if i == 6{
8. fmt.Printf(" sumaremos 3\n")
9. i = i + 3
10. continue
11. }
12. fmt.Printf("...\n")
13. i++
14. }
15. }

Salida:



Sentencia *goto*

Una sentencia ***goto*** permite el “salto” hasta una etiqueta dentro de la misma función. Una etiqueta es un identificador (regularmente escrito con mayúsculas) seguido de dos puntos, el cual sirve para poner “marcas” o “puntos de referencia” en nuestros programas. Si se utiliza goto, el flujo del programa continúa de forma estructurada a partir de la etiqueta que se especificó.

No es recomendable usar sentencia que permita esta clase de movimientos en el código porque dificultan su comprensión, y siempre existe una forma de escribir cualquier programa que utiliza ***goto*** de una forma en que se omita su uso.

El siguiente ejemplo muestra el uso de ***goto***:

1. package main
2. import "fmt"
3. func main() {
4. var i int = 0
5. CICLO: for i < 10 {
6. if i == 6 {
7. i = i + 3
8. fmt.Println("Saltando a etiqueta CICLO con i = i + 3")
9. goto CICLO
10. }
11. fmt.Printf("Valor de i: %d\n", i)
12. i++
13. }
14. }

Salida:



Finalizando…

En este capítulo estudiamos los ciclos en Go y cómo estos nos facilitan la programación. En la próxima entrada estudiaremos el uso de las funciones.

09. funciones y ámbitos

por [Uriel Ruelas](https://codingornot.com/author/zainiaz07)

A veces necesitamos que algunos de nuestros algoritmos se ejecuten varias veces a lo largo de nuestro código, pero resultaría poco práctico copiarlos y pegarlos una y otra vez. Los ciclos podrían parecer una buena alternativa, sin embargo, ¿qué pasaría si esas repeticiones están dispersas a lo largo del código? Para ese tipo de situaciones podemos recurrir a las funciones.

Una función o subrutina es una sección de código que realiza una tarea en específico. Todos los programas cuentan con, al menos, una función main (principal) con la posibilidad de definir más funciones para realizar subtareas. Todas las funciones se diferencian entre ellas, y de las variables, por medio de un identificador único.

La sintaxis de la declaración de funciones en Golang es la siguiente:

1. Declaración\_funciones = "func" Nombre\_función ( Función | Firma )
2. Nombre\_función = identificador
3. Función = Firma Cuerpo\_función
4. Firma = Parámetros [Resultado]
5. Parámetros = "(" [ Lista\_parámetros [ "," ] ] ")"
6. Resultado = Parámetros | Tipo
7. Tipos = Nombre\_tipo | Literal\_tipo | "(" Tipo ")"
8. Cuerpo\_funcion = Bloque
9. Bloque = "{" Lista\_declaraciones "}"

En base a lo anterior, podemos identificar:

* func es la palabra reservada que inicia la declaración de una función.
* Firma es la lista de parámetros.
* Nombre\_función es el identificador de nuestra función.
* Parámetros son los valores que se pasan a una función para trabajar dentro de ella. Puede haber cualquier cantidad de parámetros, o ninguno.
* Tipo es la lista de resultados que devuelve una función por medio de un return. Puede haber cualquier cantidad de resultados, o ninguno. El número de ellos debe coincidir con los devueltos por la sentencia return en cantidad y tipo.
* Cuerpo\_función contiene las instrucciones de la función. Si se especificó un resultado, en esta sección debe de colocarse un return con los correspondientes datos a regresar.

Más información sobre sintaxis aquí y aquí.

Ejemplo de función:

La siguiente función se llama mult, recibe dos parámetros enteros (numero1 y numero2), su tarea es multiplicar esos números que recibió y devolver el producto:

1. func mult(numero1, numero2 int) int{
2. /\*Variable local de la función\*/
3. var resultado int
4. resultado = numero1 \* numero2
5. return resultado
6. }

Llamado a una función

La función del ejemplo anterior teóricamente multiplicaba dos números, sin embargo, si no se le hace un llamado, solamente será código sin uso. Si se llama a una función, el control del programa se transfiere a esta última hasta que se alcanza una sentencia return, o hasta que se terminan las instrucciones dentro de la sección del cuerpo.

Para hacer un llamado a función, se utiliza el identificador de la función (nombre) y se pasan los parámetros necesarios. Si la función regresa algún valor, este puede ser almacenado en variables.

Ejemplo:

1. package main
2. import “fmt”
3. func main(){
4. /\*Variable local\*/
5. var res int
6. /\*Llamado a función\*/
7. fmt.Printf("El resultado de 122\*120: %d\n", mult(122, 120))
8. /\*Llamado con parámetros distintos\*/
9. fmt.Printf("El resultado de 130\*105: %d\n", mult(130, 105))
10. /\*Se almacena el resultado en una variable\*/
11. res = mult(20, 178)
12. fmt.Printf("Resultado almacenado en una variable: %d\n", res)
13. }
14. /\*Función que multiplica dos números\*/
15. func mult(numero1, numero2 int) int{
16. /\*Variable local de la función\*/
17. var resultado int
18. resultado = numero1 \* numero2
19. return resultado
20. }

Funciones que regresan más de un valor

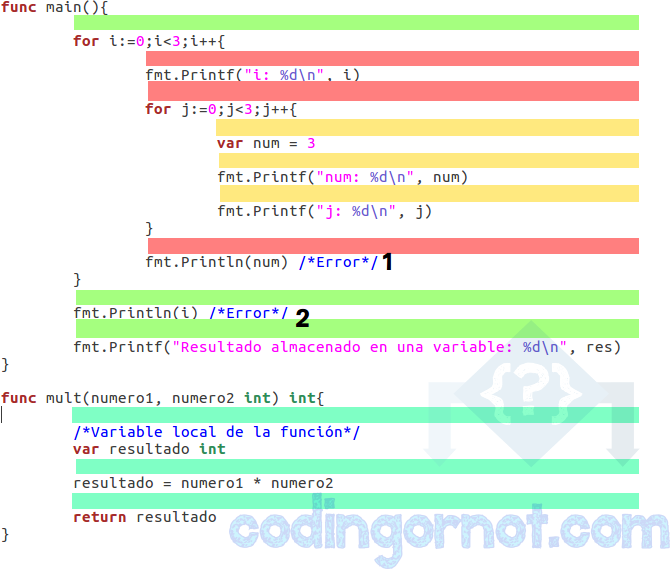
A diferencia de otros lenguajes de programación, las funciones en Golang puede regresar más de un valor con un return. El siguiente ejemplo muestra una función que regresa dos valores, sin embargo, cabe aclarar que puede ser devuelta cualquier cantidad de valores de cualquier tipo:

1. package main
2. import "fmt"
3. func main(){
4. /\*Llamado a la función\*/
5. min, max := minmax(12, 23, 54)
6. fmt.Printf("Min: %d\nMax: %d\n", min, max)
7. }
8. /\*Función para encontrar el MÍNIMO
9. y MÁXIMO entre tres números\*/
10. func minmax(a, b, c int) (int, int){
11. /\*Variables locales\*/
12. var min, max int
13. /\*Se encuentra el MÁXIMO, luego el MÍNIMO\*/
14. /\*Primero verifica si "a" es el MÁXIMO\*/
15. if a > b && a > c{
16. max = a
17. if b < c{
18. min = b
19. }else{
20. min = c
21. }
22. /\*Si "a" no fue el MÁXIMO, prueba con "b"\*/
23. }else if b > a && b > c{
24. max = b
25. if a < c{
26. min = a
27. }else{
28. min = c
29. }
30. /\*Si "a" y "b" no fueron los MÁXIMOS, prueba con "c" \*/
31. }else{
32. max = c
33. if a < b{
34. min = a
35. }else{
36. min = b
37. }
38. }
39. return min, max
40. }

Ámbito de una variable

Nótese que en las funciones de los ejemplos anteriores se utilizó un concepto nuevo: variables locales. Una variable local es aquella a la que solamente se puede acceder desde ciertas partes de nuestros programas. Regularmente a esta característica de los lenguajes de programación se le conoce como ámbito.

El ámbito se refiere a una región del programa en la que una variable existe. En la siguiente imagen se utilizaron colores diferentes para diferenciar ámbitos distintos:

* Las variables del ámbito verde pueden ser consultadas por los ámbitos rojo y amarillo. 
* Las variables del ámbito rojo y amarillo no pueden ser consultadas por el ámbito verde.
* Las variables del ámbito rojo se pueden acceder desde el ámbito amarillo.
* Las variables del ámbito amarillo no se pueden acceder desde el ámbito rojo.
* Las variables del ámbito cian no pueden consultarse desde ningún otro ámbito y viceversa.
* Los número 1 y 2 marcan dos sentencias erróneas que intentan acceder a variables de un ámbito al que no pueden hacerlo.
* En el ámbito amarillo se utiliza la variable i que pertenece al ámbito rojo.

Las variables pueden ser declaradas en 3 lugares distintos:

1. Dentro de una función o bloque de código. En este caso las variables son conocidas como locales, y solamente se puede acceder a ellas dentro del ámbito al que pertenecen y desde otros internos a él.
2. Fuera de todas las funciones. En este caso se les conoce como variables globales y se puede acceder a ellas desde cualquier parte del programa.
3. En la definición de una función. Se les conoce como parámetros formales y solamente se puede acceder a ellas desde la función en la que se declaran.

Haciendo unos pequeños cambios a los ejemplos anteriores, podemos observar los ámbitos en acción:

1. package main
2. import "fmt"
3. /\*Variable global\*/
4. var resglobal int
5. /\*Función principal\*/
6. func main(){
7. /\*Variable local de main\*/
8. var resultado int
9. /\*Modificando una variable global desde main\*/
10. resglobal = 31
11. resultado = mult(20, 178)
12. fmt.Printf("Resultado de multiplicación: %d\n", resultado)
13. }
14. /\*numero1 y numero2 son parámetros formales
15. y su ámbito es local a la función mult\*/
16. func mult(numero1, numero2 int) int{
17. /\*Variable local de la función mult con el
18. mismo nombre que una variable local de main\*/
19. var resultado int
20. resultado = numero1 \* numero2
21. /\*Accediendo a variable global desde función\*/
22. fmt.Printf("Valor en variable global: %d\n", resglobal)
23. return resultado
24. }

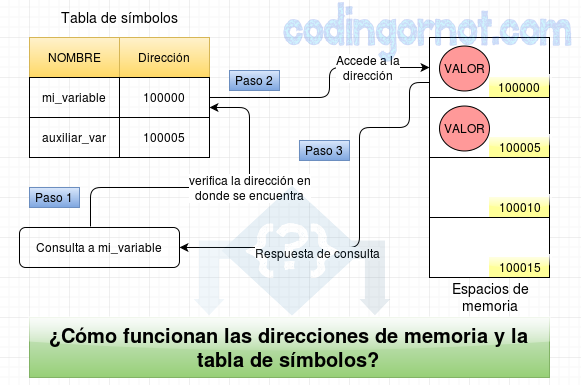
Finalizando…

Las funciones son una herramienta poderosa para mejorar nuestros programas, y en esta nota estudiamos lo necesario para comenzar a realizar pruebas de su funcionamiento. En la próxima entrada será tiempo de aprender un concepto nuevo que nos brindará otra alternativa para trabajar con funciones: los apuntadores.

10. apuntadores

Ha llegado el momento de tratar una de las herramientas más útiles de lenguajes como C y Go: los apuntadores. Un apuntador es una variable que, como su nombre lo indica, apunta a algo, es decir, es una variable que almacena la dirección de memoria de otra variable.

Tal vez sea lógico, pero todas las variables en ejecución se encuentran almacenadas en un espacio de memoria y, a su vez, cada espacio de memoria tiene asignada una dirección; por ende, cada variable tiene asignada una dirección en memoria.



Analizando un poco la imagen anterior, podemos identificar que la tabla de símbolos tiene una entrada para nuestra variable llamada **mi\_variable** en donde se especifica en qué espacio de memoria podemos localizarla. En este caso **mi\_variable** tiene almacenado un valor imaginario VALOR en la dirección de memoria 100000. Este ejemplo es arbitrario, ni las tablas de símbolos son exactamente así, ni las direcciones de memoria se dividen de ese modo (VALOR tampoco es un dato válido).

Apuntadores en Go

Para declarar un apuntador en Go se utiliza la siguiente sintaxis:

1. TipoApuntador = "\*" TipoBase
2. TipoBase = TipoDato

En la siguiente línea de código se declara un apuntador a un entero:

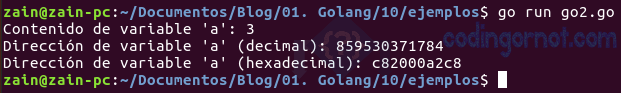
1. var apunta\_entero \*int

¿Cómo acceder a la dirección de una variable?

En la tabla de símbolos todas las variables tienen asignada la dirección de memoria donde se localizan, la cual se puede acceder si le colocamos el operador ampersand (*&*) al inicio de ellas. En el siguiente ejemplo puedes observar dicho operador siendo utilizado para imprimir la dirección de una variable:

1. package main
2. import "fmt"
3. func main(){
4. var mi\_var = 3
5. fmt.Printf("Contenido de variable 'a': %d \n", mi\_var)
6. fmt.Printf("Dirección de variable 'a' (decimal): %d \n", &mi\_var)
7. fmt.Printf("Dirección de variable 'a' (hexadecimal): %x \n", &mi\_var)
8. }

Salida:



¿Cómo utilizar los apuntadores?

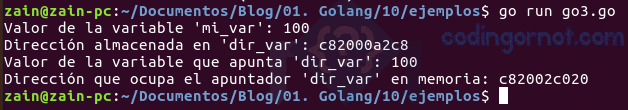
Los pasos para utilizar un apuntador son:

1. Declararlo.
2. Asignarle la dirección de otra variable.
3. Acceder a la variable desde el apuntador.

Para completar el paso 3 es necesario utilizar el operador asterisco (*\**) para consultar el valor de la variable ubicada en la dirección que almacena el apuntador. En el siguiente ejemplo podemos observar esto último:

1. package main
2. import "fmt"
3. func main(){
4. var mi\_var = 100
5. var dir\_var \*int
6. dir\_var = &mi\_var
7. fmt.Printf("Valor de la variable 'mi\_var': %d \n", mi\_var)
8. fmt.Printf("Dirección almacenada en 'dir\_var': %x \n", dir\_var)
9. fmt.Printf("Valor de la variable que apunta 'dir\_var': %d \n", \*dir\_var)
10. fmt.Printf("Dirección que ocupa el apuntador 'dir\_var' en memoria: %x \n", &dir\_var)
11. }

Salida:



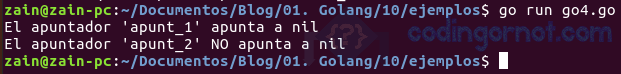
Como se puede observar en la última línea de la salida, los apuntadores también ocupan un lugar en memoria.

Apuntadores a nil

Si no se asigna una dirección a la cual apuntar, por defecto los apuntadores almacenan la dirección de memoria 0 o nil (nula). Dicha constante tiene un significado particular en el lenguaje: cuando se apunta a nil, el compilador lo interpreta como si se intentara acceder a una dirección de memoria inválida. Esta característica puede ser utilizada en conjunto con algunas sentencias como las condicionales:

1. package main
2. import "fmt"
3. func main(){
4. var mi\_var = 100
5. var apunt\_1 \*int
6. var apunt\_2 \*int
7. //Asigna una dirección válida
8. apunt\_2 = &mi\_var
9. if(apunt\_1 == nil){
10. fmt.Println("El apuntador 'apunt\_1' apunta a nil")
11. }else{
12. fmt.Println("El apuntador 'apunt\_1' NO apunta a nil")
13. }
14. if(apunt\_2 == nil){
15. fmt.Println("El apuntador 'apunt\_2' apunta a nil")
16. }else{
17. fmt.Println("El apuntador 'apunt\_2' NO apunta a nil")
18. }
19. }

Salida:



Arreglos de apuntadores

También es posible crear arreglos de apuntadores:

1. package main
2. import "fmt"
3. func main(){
4. mis\_num:= []int{5, 10, 33, 42, 59}
5. var apuntadores[5]\*int
6. //Asigna las direcciones de los elementos a los apuntadores
7. for i:=0; i < 5 ; i++ {
8. apuntadores[i] = &mis\_num[i]
9. fmt.Printf("-----------------------------\n")
10. //Se accede al arreglo 'mis\_num' por medio de apuntadores
11. fmt.Printf("Contenido de mis\_num[%d]: %d \n", i, \*apuntadores[i])
12. fmt.Printf("Dirección de mis\_num[%d]: %x \n", i, apuntadores[i])
13. }
14. }

Salida:

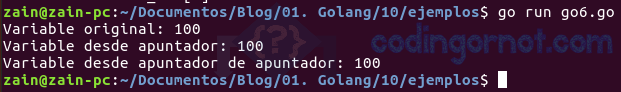


Apuntadores a apuntadores

Por si fuera poco, también es posible crear apuntadores a apuntadores:

1. package main
2. import "fmt"
3. func main(){
4. var val int = 100
5. var ap\_val \*int
6. var ap\_ap\_val \*\*int
7. //Recupera la dirección de la variable
8. ap\_val = &val
9. //Recupera la dirección del apuntador
10. ap\_ap\_val = &ap\_val
11. fmt.Printf("Variable original: %d \n", val)
12. fmt.Printf("Variable desde apuntador: %d \n", \*ap\_val)
13. fmt.Printf("Variable desde apuntador de apuntador: %d \n", \*\*ap\_ap\_val)
14. }

Salida:

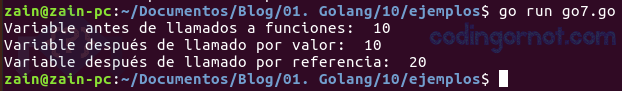


Funciones con paso de parámetros por referencia

Como complemento a la [entrada anterior](https://codingornot.com/08-go-to-go-funciones-y-ambitos/) en la que tratamos el tema de las funciones: es posible utilizar apuntadores con funciones para acceder y modificar las variables originales que se hayan pasado como argumentos, a esto se le conoce como **paso de parámetros por referencia**. En el siguiente ejemplo podemos observar la diferencia entre un llamado por referencia y uno por valor (cuando no se utilizan apuntadores):

1. package main
2. import "fmt"
3. func main(){
4. var mi\_var int = 10
5. fmt.Println("Variable antes de llamados a funciones: ", mi\_var)
6. //Llamado por valor
7. sumaPorValor(mi\_var)
8. fmt.Println("Variable después de llamado por valor: ", mi\_var)
9. //Llamado por referencia
10. sumaPorReferencia(&mi\_var)
11. fmt.Println("Variable después de llamado por referencia: ", mi\_var)
12. }
13. func sumaPorValor(a int){
14. a = a + 10
15. }
16. func sumaPorReferencia(a \*int){
17. \*a = \*a + 10
18. }

Salida:



Finalizando…

Espero que esta entrada te haya servido, no dejes de repasar los apuntadores, te serán de mucha utilidad.

11. Registros (estructuras) en Golang

Anteriormente traté el tema de [arreglos](https://codingornot.com/2016/07/22/07-go-to-go-arreglos/), recordemos que estos nos permiten almacenar una colección de datos del mismo tipo bajo un único identificador.

Si queremos almacenar datos de diferentes miembros de un conjunto, pero dichos datos no corresponden al mismo tipo, podemos recurrir a los ***struct*** o registros.

Un ***struct*** es un registro que guarda datos de diferentes tipos. Imagina que deseas almacenar información de películas. Todas las películas comparten características generales pero con diferentes valores; por ejemplo, todas las películas tienen los siguientes características:

* Título
* Director
* Protagonistas
* Año de estreno

Con un ***struct*** puedes crear fácilmente registros de películas que almacenen los campos anteriores para cada una de ellas.

Sintaxis

Para definir un ***struct*** seguimos la siguiente sintaxis:

1. Struct = "struct" "{" { Campo\_declaraciones ";" } "}"
2. Campo\_declaraciones = (Lista\_identificadores | Campo\_anónimo) [Etiqueta]
3. Campo\_anónimo = [ "\*" ] Nombre\_tipo
4. Etiqueta = Cadena\_literal

Más información al respecto aquí.

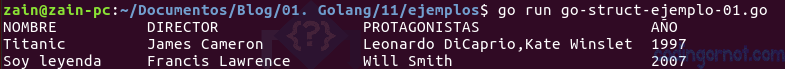
**Nota**: para declarar variables de un ***struct*** que ya ha sido definido, se utiliza la sintaxis de declaración de variables con la que hemos trabajado (ejemplos más adelante):

Ejemplo de funcionamiento

En el siguiente programa defino un ***struct*** que almacenará información de películas, declaro dos variables de tipo **Pelicula**, asigno los datos correspondientes a cada una de ellas y, finalmente, imprimo lo que acabo de almacenar (los tabulares en la impresión se utilizan para aplicar estilo y son completamente arbitrarios).package main

1. import "fmt"
2. //Definición de struct Pelicula
3. type Pelicula struct{
4. titulo string
5. director string
6. protagonistas string
7. año int //Recordemos que Go permite utilizar 'ñ' en identificadores
8. }
9. func main(){
10. /\*Declaración de variables tipo Pelicula\*/
11. var peli01 Pelicula
12. var peli02 Pelicula
13. /\*Llenamos cada variable con la información
14. correspondiente para cada película\*/
15. peli01.titulo = "Titanic"
16. peli01.director = "James Cameron"
17. peli01.protagonistas = "Leonardo DiCaprio,Kate Winslet"
18. peli01.año = 1997
19. peli02.titulo = "Soy leyenda"
20. peli02.director = "Francis Lawrence"
21. peli02.protagonistas = "Will Smith"
22. peli02.año = 2007
23. /\*Imprimimos la información de las películas\*/
24. fmt.Printf("NOMBRE\t\tDIRECTOR\t\tPROTAGONISTAS\t\t\tAÑO\n")
25. fmt.Printf("%s\t\t", peli01.titulo)
26. fmt.Printf("%s\t\t", peli01.director)
27. fmt.Printf("%s\t", peli01.protagonistas)
28. fmt.Printf("%d\n", peli01.año)
29. fmt.Printf("%s\t", peli02.titulo)
30. fmt.Printf("%s\t", peli02.director)
31. fmt.Printf("%s\t\t\t", peli02.protagonistas)
32. fmt.Printf("%d\n", peli02.año)
33. }

Salida:

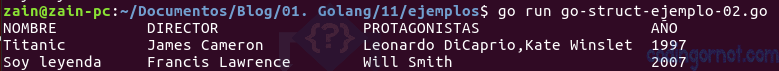


Arreglos de ***struct***

Si no queremos crear variables con diferentes nombres, podemos declarar un arreglo de ***struct***, y de ese modo trabajar bajo un mismo identificador con todos nuestros registros. El siguiente ejemplo demuestra este funcionamiento:

1. package main
2. import "fmt"
3. //Definición
4. type Pelicula struct{
5. titulo string
6. director string
7. protagonistas string
8. año int
9. }
10. func main(){
11. /\*Declaración de arreglo tipo Pelicula\*/
12. var pelis[2] Pelicula
13. /\*Llenamos cada índice con la información
14. correspondiente para cada película\*/
15. pelis[0].titulo = "Titanic"
16. pelis[0].director = "James Cameron"
17. pelis[0].protagonistas = "Leonardo DiCaprio,Kate Winslet"
18. pelis[0].año = 1997
19. pelis[1].titulo = "Soy leyenda"
20. pelis[1].director = "Francis Lawrence"
21. pelis[1].protagonistas = "Will Smith"
22. pelis[1].año = 2007
23. /\*Imprimimos la información de las películas\*/
24. fmt.Printf("NOMBRE\t\tDIRECTOR\t\tPROTAGONISTAS\t\t\tAÑO\n")
25. fmt.Printf("%s\t\t", pelis[0].titulo)
26. fmt.Printf("%s\t\t", pelis[0].director)
27. fmt.Printf("%s\t", pelis[0].protagonistas)
28. fmt.Printf("%d\n", pelis[0].año)
29. fmt.Printf("%s\t", pelis[1].titulo)
30. fmt.Printf("%s\t", pelis[1].director)
31. fmt.Printf("%s\t\t\t", pelis[1].protagonistas)
32. fmt.Printf("%d\n", pelis[1].año)
33. }

Salida:



***struct*** como argumento de una función

Puedes pasar un ***struct*** como argumento a una función tal como si pasaras cualquier otra variable:

1. package main
2. import "fmt"
3. type Pelicula struct{
4. titulo string
5. director string
6. protagonistas string
7. año int
8. }
9. func main(){
10. /\*Declaración de variables tipo Pelicula\*/
11. var pelis[2] Pelicula
12. /\*Llenamos cada variable con la información
13. correspondiente para cada película\*/
14. pelis[0].titulo = "Titanic"
15. pelis[0].director = "James Cameron"
16. pelis[0].protagonistas = "Leonardo DiCaprio,Kate Winslet"
17. pelis[0].año = 1997
18. pelis[1].titulo = "Soy leyenda"
19. pelis[1].director = "Francis Lawrence"
20. pelis[1].protagonistas = "Will Smith"
21. pelis[1].año = 2007
22. /\*Imprimimos la información de las películas\*/
23. imprimePelicula(pelis[0])
24. imprimePelicula(pelis[1])
25. }
26. //Función que imprime la información de una película
27. func imprimePelicula(peli Pelicula){
28. fmt.Printf("NOMBRE: %s\n", peli.titulo)
29. fmt.Printf("DIRECTOR: %s\n", peli.director)
30. fmt.Printf("PROTAGONISTAS: %s\n", peli.protagonistas)
31. fmt.Printf("AÑO: %d\n\n", peli.año)
32. }

Salida:



Apuntadores a ***struct***

También puedes utilizar apuntadores a ***struct***.

Usando el ejemplo anterior, declaremos un apuntador a un ***struct Pelicula***:

1. var ptpeli \*Pelicula

Asignemos una dirección de memoria al apuntador:

1. ptpeli = &peli01

Imprimamos la variable **titulo** del ***struct***:

1. fmt.Println(ptpeli.titulo)

Modifiquemos el ejemplo para que la función **imprimePelicula** reciba un apuntador como argumento:

1. package main
2. import "fmt"
3. type Pelicula struct{
4. titulo string
5. director string
6. protagonistas string
7. año int
8. }
9. func main(){
10. /\*Declaración de variables tipo Pelicula\*/
11. var pelis[2] Pelicula
12. /\*Llenamos cada variable con la información
13. correspondiente para cada película\*/
14. pelis[0].titulo = "Titanic"
15. pelis[0].director = "James Cameron"
16. pelis[0].protagonistas = "Leonardo DiCaprio,Kate Winslet"
17. pelis[0].año = 1997
18. pelis[1].titulo = "Soy leyenda"
19. pelis[1].director = "Francis Lawrence"
20. pelis[1].protagonistas = "Will Smith"
21. pelis[1].año = 2007
22. /\*Para imprimir la información de las películas\*/
23. imprimePelicula(&pelis[0])
24. imprimePelicula(&pelis[1])
25. }
26. //Función que imprime una variable tipo Pelicula por medio de un apuntador
27. func imprimePelicula(pt\_peli \*Pelicula){
28. fmt.Printf("NOMBRE: %s\n", pt\_peli.titulo)
29. fmt.Printf("DIRECTOR: %s\n", pt\_peli.director)
30. fmt.Printf("PROTAGONISTAS: %s\n", pt\_peli.protagonistas)
31. fmt.Printf("AÑO: %d\n\n", pt\_peli.año)
32. }

Salida:



Finalizando…

Esto ha sido todo por esta ocasión, espero que te haya sido de utilidad.

12. slices (trozos) en Golang

Por [Uriel Ruelas](https://codingornot.com/author/zainiaz07)

Cuando estudiamos los arreglos pudiste identificar que, a pesar de ser poderosas herramientas, resultan ser poco flexibles puesto que solamente permiten almacenar la cantidad de datos que sea especificada al momento de declararlos.

Nota: en esta entrada considero que ya has trabajado con arreglos en Go. Si no lo has hecho, te recomiendo leer la entrada al respecto que puedes encontrar en codingornot dando clic [aquí](https://codingornot.com/07-go-to-go-arreglos/).

Go nos proporciona una forma alternativa para trabajar con arreglos por medio de los slices (trozos, rebanadas, porciones etc.) los cuales nos brindan las mismas ventajas de los arreglos pero con algunas funciones y características únicas que los vuelven más flexibles que estos últimos. Inclusive, es considerablemente más común encontrar programas que trabajen con slices a que lo hagan con arreglos¹.

Los slices (trozos)

Antes de continuar, es conveniente aclarar que ni “slice” ni tampoco “slices” son palabras reservadas de Golang, es decir, pueden ser usadas libremente en identificadores porque no son un tipo de dato, sino que son descriptores de un espacio de memoria en el que se almacenan datos contiguos (¡hola, arreglo!), es por eso que mencioné que esta es una forma alternativa para trabajar con arreglos.

La sintaxis que se utiliza para declarar un slice es la siguiente:

1. SliceType = "[" "]" ElementType
2. ElementType = Type
3. Type = TypeName | TypeLit | "(" Type ")"

Más información al respecto aquí.

Ejemplo de declaración de un slice:

var nums1 []int //Slice sin tamaño definido

Como puede notarse, la forma para declarar slices es como se hacía con los arreglos pero sin especificar el tamaño de elementos que almacenarán. También puedes utilizar la función make para la declaración. La sintaxis de dicha función es la siguiente:

func make([]T, len, cap)[]T

* Donde T es el tipo de dato de los elementos que contendrá nuestro slice, len es la cantidad de elementos que tendrá al momento de declararlo (inicializados en 0 automáticamente), mientras que cap es la capacidad total del slice (puede modificarse más adelante). Cuando make es llamado, se reserva espacio en memoria para un arreglo de la capacidad especificada y nuestro slice será el que almacene la información necesaria para realizar operaciones sobre dicho arreglo.

Ejemplo de uso:

1. var nums []int //Declaración de slice
2. nums = make([]int, 3, 3) //Inicialización con 3 elementos y capacidad de 3

O bien, si deseas utilizar solamente una línea de código:

nums := make([]int, 3, 3)

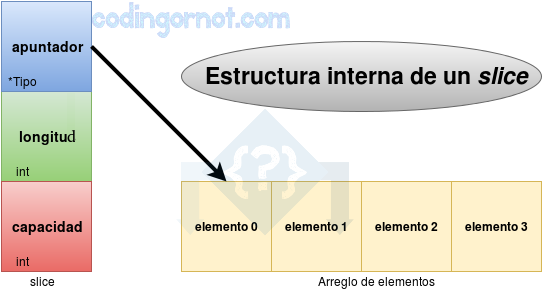
No es necesario especificar la capacidad del slice, si se pasa solamente un parámetro, make establecerá la capacidad igual a la longitud de nuestro arreglo, es decir:

1. //Las dos sentencias siguientes hacen lo mismo
2. nums := make([]int, 3)
3. nums := make([]int, 3, 3)

Recordemos que cuando declaramos que la longitud de nuestro slice sería de tres elementos, estos se inicializaron en 0.

¿Slices o arreglos?

Puede llegar a ser confuso el uso de ambos términos. Podemos comprender internamente cómo funcionan los slices e identificar por qué se habla de arreglos de datos en slices con la siguiente representación:



Ejemplo de slices en acción:

1. package main
2. import "fmt"
3. func main(){
4. //Slice que se declara e inicializa por separado
5. var nums1 []int
6. nums1 = make([]int, 3, 3)
7. //Slice que se declara e inicializa en la misma línea
8. nums2:=make([]int, 4)
9. fmt.Println(nums1)
10. fmt.Println(nums2)
11. }

Salida:



Operaciones y acciones sobre slices

Se puede acceder a los elementos de un slice uno a uno por medio de índices, tal como se hace con los arreglos:

1. //Modificar el elemento en el índice 0
2. nums[0] = 5
3. //Imprimir el elemento en el índice 0
4. fmt.Println(nums[0])

También es posible acceder a diversos índices de un slice a la vez por medio de un rango, como se muestra a continuación:

1. slice[a:b] //Acceder a los alementos desde a-b
2. slice[:b] //Acceder a los alementos desde 0-b
3. slice[a:] //Acceder a los alementos desde a-len(a)
4. slice[:] //Acceder a todos los elementos

* Donde a es el límite inferior (inclusivo) y b el límite superior (exclusivo). Cuando se deja en blanco el primer campo, se considera que se desea acceder desde el índice 0 hasta b; en cambio, si se deja en blanco el segundo campo del rango, se considera que el límite será el último elemento del slice o len(slice).

Puedes inicializar un slice con la información total o parcial de otro gracias a los límites antes mencionados:

1. slice2 := slice1[1:3] //Crea e inicializa slice2 con los índices 1 y 2 de slice1
2. slice3 := slice1[:] //Crea e inicializa slice3 con el contenido total de slice1

¡NOTA IMPORTANTE!: Si se inicializa un slice como en los ejemplos anteriores, es importante tener en cuenta que lo que se crea realmente es un apuntador al espacio de memoria del slice que se usa como información base; por tanto, si se hace un cambio sobre alguno de ellos, el otro también se verá afectado.

slice2:=slice1[:] //slice2 es un apuntador al arreglo de datos de slice1

len(), cap() y copy()

Es posible determinar la cantidad de elementos y capacidad de un arreglo por medio de las funciones len() y cap() respectivamente. La cantidad de elementos es el número de elementos a los que se hace referencia con el slice, la capacidad es la cantidad total de espacios disponibles para almacenar elementos.

1. len(slice) //Regresa la cantidad de elementos en slice
2. cap(slice) //Regresa la cantidad de espacios total del slice

Puedes realizar una copia de un slice con la función copy() como se muestra a continuación:

copy(slice\_destino, silice\_original)

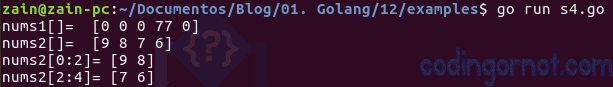
Una de las más importantes características de los slices es que permiten aumentar la capacidad máxima de elementos por medio de la función append() la cual añade uno o varios elementos nuevos desde el último índice del slice. Si el slice aún tiene sus últimos índices disponibles, simplemente se añade el nuevo elemento a la cola; en cambio, si el último índice ya está utilizado, se aumenta dinámicamente la capacidad del slice en 1 y se añade dicho elemento en la última posición.

1. slice = append(slice, “uno”, “dos”, “tres”) //Se añaden 3 elementos nuevos
2. slice = append(slice, “cuatro”) //Se pueden añadir N o 1 elemento nuevo

Ejemplo de acceso a índices y rangos:

1. package main
2. import "fmt"
3. func main(){
4. nums1:=make([]int, 5, 5) //Inicializa en 0 los 5 elementos del slice
5. nums1[3] = 77 //Modifica el índice 3 del slice nums1
6. fmt.Println("nums1[]= ", nums1)
7. nums2 := []int{9, 8, 7, 6} //Nuevo slice nums2 inicializado similar a un array
8. fmt.Println("nums2[]= ", nums2)
9. //Acceder a rangos (sub slices) de slices
10. fmt.Println("nums2[0:2]=", nums2[:2])
11. fmt.Println("nums2[2:4]=", nums2[2:])
12. }

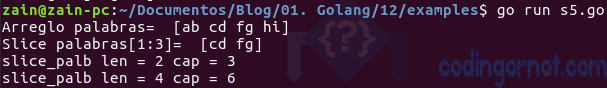
Salida:



Ejemplo de len(), cap() y append():

1. package main
2. import "fmt"
3. func main(){
4. var palabras = []string{"ab", "cd", "fg", "hi"}
5. //Inicializar un slice con un segmento del array
6. slice\_palab:=palabras[1:3]
7. fmt.Println("Arreglo palabras= ", palabras)
8. fmt.Println("Slice palabras[1:3]= ", slice\_palab)
9. //Funciones len() y cap() en acción
10. fmt.Printf("slice\_palab len = %d cap = %d\n", len(slice\_palab), cap(slice\_palab))
11. slice\_palab = append(slice\_palab, "jk", "lm") //Se añaden 2 elementos
12. /\*Capacidad de slice\_palab = 6 porque aumenta la capacidad
13. \* del array que referencia (palabras)\*/
14. fmt.Printf("slice\_palab len = %d cap = %d\n", len(slice\_palab), cap(slice\_palab))
15. }

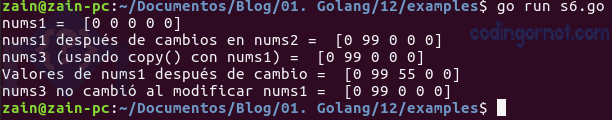
Salida:



Ejemplo de uso de copy():

1. package main
2. import "fmt"
3. func main(){
4. nums1:=make([]int, 5, 5) //Nuevo slice
5. nums2:=nums1[:] //Nuevo slice referenciando nums1
6. fmt.Println("nums1 = ", nums1)
7. nums2[1] = 99 //Modifica nums2
8. //Los cambios hecho en nums2 se reflejan en nums1
9. fmt.Println("nums1 después de cambios en nums2 = ",nums1)
10. nums3:=make([]int, 5, 5) //nuevo slice
11. //Ejemplo de copy()
12. copy(nums3, nums1)
13. fmt.Println("nums3 (usando copy() con nums1) = ", nums3)
14. nums1[2] = 55
15. //Usando copy(), nums3 no referencia a nums1
16. fmt.Println("Valores de nums1 después de cambio = ", nums1)
17. fmt.Println("nums3 no cambió al modificar nums1 = ", nums3)
18. }

Salida:



Finalizando…

Espero que todo te haya sido de utilidad, recuerda que para comprender mejor un concepto te recomiendo realizar muchas pruebas hasta que lo domines.

13. Maps en Golang

Continuamos con una de las últimas estructuras de datos que nos proporciona Golang nativamente, los **maps** (mapas o planos). Un **map** nos permite almacenar información de forma similar a como lo hacíamos con los arreglos o los *slices* pero con la particularidad de que cada elemento de un **map** tiene asociada una llave única.

Si alguna vez has trabajado con **hash table**/**hash map** en Java, *hashes* en Perl o cualquier otro lenguaje de programación, un **map** en Go sigue una filosofía de trabajo similar a estos, por tanto no te resultará complicado comprender su uso.

Conceptos generales

Los **maps** son unas de las estructuras más utilizadas en Golang gracias a que es muy sencillo utilizarlos y son poderosos métodos de gestión de elementos de conjuntos. A continuación se presenta la sintaxis de la declaración de un **map**:

1. Mapa = "map" "[" TipoLlave "]" TipoElemento
2. TipoLlave = Tipo

* Más información al respecto aquí.

En el siguiente ejemplo declaro un **map** que almacenará días de la semana donde la llave única será un número entero (correspondiente al orden de los días) y el elemento principal (el que no es la llave única) será una cadena de texto:

1. var diasSemana map[int]string

El **map** anterior ha sido declarado, pero no ha sido inicializado, si quisiéramos añadir algún elemento, ocurriría un error ya que el elemento sigue siendo nulo (apuntando a **nil**). Para inicializar el **map** utilizamos **make**:

1. diasSemana = make(map[int]string)

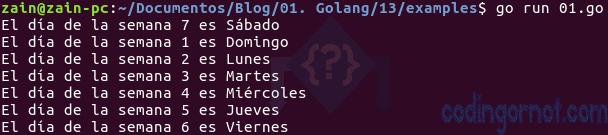
Para insertar elementos en el **map** anterior (uno a uno):

1. diasSemana[1] = "Domingo"
2. diasSemana[2] = "Lunes"
3. diasSemana[3] = "Martes"
4. diasSemana[4] = "Miércoles"
5. diasSemana[5] = "Jueves"
6. diasSemana[6] = "Viernes"
7. diasSemana[7] = "Sábado"

Si queremos recorrer todos los elementos, podemos hacer lo siguiente:

1. for dia := range diasSemana {
2. fmt.Println("El día", dia, "de la semana es", diasSemana[dia])
3. }

Salida:



Para buscar un elemento es necesario utilizar una asignación doble en donde la primera variable almacenará el resultado de la consulta y la segunda variable (generalmente llamada *ok*<) será inicializada con un valor *bool true* si el elemento existe:

1. string\_dia, ok := diasSemana[8]
2. if(ok){
3. fmt.Println("El elemento sí existe y almacena:", string\_dia)
4. }else{
5. fmt.Println("El elemento no existe")
6. }

Para consultar el tamaño de un map se utiliza la función **len()** como lo hacíamos con *slices*:

1. len(diasSemana)

La función **delete()** permite eliminar elementos de un **map** a través de la llave única de estos:

1. delete(diasSemana, 2) //Elimina el elemento con la llave 2

Ejemplo:

1. package main
2. import "fmt"
3. func main(){
4. var diasSemana map[int]string
5. diasSemana = make(map[int]string)
6. diasSemana[1] = "Domingo"
7. diasSemana[2] = "Lunes"
8. diasSemana[3] = "Martes"
9. diasSemana[4] = "Miércoles"
10. diasSemana[5] = "Jueves"
11. diasSemana[6] = "Viernes"
12. diasSemana[7] = "Sábado"
13. //Busca elemento con llave 8
14. string\_dia, encontrado := diasSemana[8]
15. if(encontrado){
16. fmt.Println("El día 2 es", string\_dia)
17. }else{
18. fmt.Println("El día 2 no fue encontrado")
19. }
20. //Eliminar el elemento con llave 2
21. delete(diasSemana, 2)
22. fmt.Println("Tamaño del map:", len(diasSemana))
23. }

Salida:



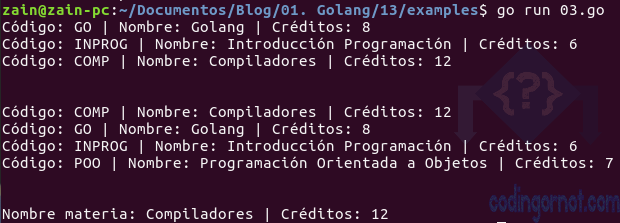
**Nota:** Golang no considera un error de compilación el tratar de añadir un elemento nuevo con una llave que ya existe en el **map**, en su lugar simplemente reemplaza el elemento antiguo con la información del nuevo.

maps y structs

Es posible declarar un **map** que almacene elementos de tipo **struct** y de ese modo guardar grandes cantidades de **struct** sin necesidad de programar estructuras dinámicas (las cuales serán tratadas en temas posteriores). El siguiente ejemplo muestra cómo podemos hacerlo:

1. package main
2. import "fmt"
3. //Registro de materias
4. type Materia struct{
5. nombre string
6. creditos int
7. }
8. func main(){
9. //Mapa con registros inicializado al declarar
10. var materias = map[string]Materia{
11. "COMP": Materia{
12. "Compiladores", 12,
13. },
14. "GO": Materia{
15. "Golang", 8,
16. },
17. "INPROG": Materia{
18. "Introducción Programación", 6,
19. },
20. }
21. //Imprime materias
22. for materia := range materias{
23. fmt.Println("Código:", materia, "| Nombre:", materias[materia].nombre,
24. "| Créditos:", materias[materia].creditos)
25. }
26. fmt.Printf("\n\n")
27. //Añade un elemento nuevo
28. materias["POO"] = Materia{"Programación Orientada a Objetos", 7,}
29. //Imprime materias
30. for materia := range materias{
31. fmt.Println("Código:", materia, "| Nombre:", materias[materia].nombre,
32. "| Créditos:", materias[materia].creditos)
33. }
34. fmt.Printf("\n\n")
35. //Busca materia con llave única "COMP"
36. mat, ok := materias["COMP"]
37. if(ok){
38. fmt.Println("Nombre materia:", mat.nombre, "| Créditos:", mat.creditos)
39. }else{
40. fmt.Println("La materia no existe")
41. }
42. }

Salida:



Finalizando…

En esta entrada aprendimos a utilizar **maps** en Go, recuerda siempre practicar para dominar los temas que estudiamos.

14. Recursión y conversión de tipos

Por [Uriel Ruelas](https://codingornot.com/author/zainiaz07)

En esta ocasión estudiaremos dos pequeños conceptos que nos servirán para complementar lo que hemos visto en las entradas pasadas: la recursión y la conversión de tipo (*type casting*).

¿Qué es la recursión?

La recursión es un método de resolución de problemas por medio de su división en pequeñas instancias que luego son unidas para generar una solución global.

La recursión se comporta como una pila en la que se ingresan valores hasta alcanzar uno o varios casos base a partir de los cuales se solucionará el problema.

Los algoritmos recursivos constan de dos elementos fundamentales:

1. **Casos base**: son los escenarios finales a partir de los cuales se comienzan a agrupar las pequeñas soluciones.
2. **Reglas**: las cuales se encargan de dividir el problema principal en diversos casos.

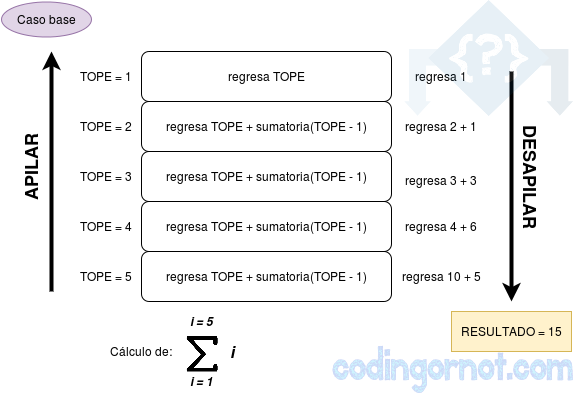
Imaginemos que deseamos obtener una sumatoria desde *i = 1* hasta *i = 5*. Una implementación recursiva sería la siguiente:

1. package main
2. import "fmt"
3. func sumatoria(tope int) int {
4. //Caso base
5. if tope == 1 {
6. return tope
7. }else{
8. //Suma tope a lo que regrese la recursión de tope - 1
9. return tope + sumatoria(tope - 1)
10. }
11. }
12. func main() {
13. tope := 10
14. fmt.Printf("La sumatoria hasta %d es: %d\n", tope, sumatoria(tope))
15. }

Salida:



El código anterior hace recursiones desde *tope = 5* hasta alcanzar el caso base *tope = 1*. Después se desapilan los llamados devolviendo los valores correspondientes a la recursión actual. La siguiente imagen ilustra esta breve descripción:



El siguiente es un ejemplo de una función recursiva que imprime la serie de [Fibonacci](https://es.wikipedia.org/wiki/Sucesión_de_Fibonacci) hasta la posición N:

1. package main
2. import "fmt"
3. func fibonacci(i int) int {
4. //Caso base
5. if(i == 0) {
6. return 0
7. }
8. //Otro caso base
9. if(i == 1) {
10. return 1
11. }
12. return fibonacci(i-1) + fibonacci(i-2)
13. }
14. func main() {
15. for n := 0; n < 15; n++ {
16. fmt.Printf("| %d ", fibonacci(n))
17. }
18. fmt.Printf("|\n")
19. }

Salida:



Conversión de tipo

La conversión de tipo (*Type Casting* es transformar momentáneamente una variable a un tipo diferente. En Golang esto se logra utilizando el nombre del tipo de dato objetivo, seguido de un par de paréntesis que encierren a la variable que deseamos convertir. Si tenemos una variable mi\_variable tipo int16 y lo queremos convertir en int32, la sintaxis de conversión sería la siguiente:

1. int32(mi\_variable)

**Nota**: recuerda que la conversión de tipo solamente funciona para realizar operaciones, no es posible cambiar permanentemente el tipo de ninguna variable. La sintaxis anterior es válida solamente si se usa para alguna operación.

El siguiente es un ejemplo de cómo se realiza una conversión para multiplicar variables de tipos distintos:

1. package main
2. import "fmt"
3. func main(){
4. var a uint16 = 120
5. var b uint32 = 3000
6. fmt.Println("Resultado axb: ", uint32(a)\*b)
7. fmt.Println("Este es un resultado distinto: ", a\*uint16(b))
8. }

Salida:



**Nota**: se puede apreciar que el segundo resultado es diferente al primero a pesar de que es la misma multiplicación. Esto sucede porque 120\*3,000 = 360,000 y este número es mayor al valor máximo que puede representar un uint16.

Finalizando

En esta entrada hemos aprendido dos conceptos nuevos que complementan todo lo que hemos visto hasta ahora.

15. Interfaces y métodos

Por [Uriel Ruelas](https://codingornot.com/author/zainiaz07)

Golang nos permite trabajar con métodos, la principal diferencia entre estos y las funciones es que, a diferencia de estas últimas, un método permite que se construyan diferentes algoritmos para tipos de registros diferentes bajo un mismo identificador; también se puede utilizar una función que decida cuál método ejecutar en base al tipo de registro que se le pase como argumento. Fuera de esas diferencias, un método en Go es una función que realiza operaciones referentes al comportamiento de un registro. Para más información referente a los métodos, consultar [aquí](https://es.wikipedia.org/wiki/Método_(informática)).

¿Qué es una interfaz?

Una interfaz es una colección de firmas de métodos, es decir, son declaraciones de métodos sin cuerpo (código interior o algoritmo), una firma de método es el equivalente al prototipo de una función en lenguajes como C. Las interfaces son muy útiles para visualizar fácilmente qué recibe de argumentos uno o varios métodos, para construirlos más adelante según se requiera, crear diferentes cuerpos con un mismo identificador, entre otras cosas.

La siguiente es la sintaxis de declaración de una interfaz en Go:

1. Tipo\_interfaz = "interface" "{" { Especificación\_método ";" } "}"
2. Especificación\_método = Nombre\_método Firma | Nombre\_tipo\_interfaz
3. Nombre\_método = identificador
4. Nombre\_tipo\_interfaz = Nombre\_tipo

**Nota**: más información al respecto [aquí](https://golang.org/ref/spec" \l "Interface_types).

¿Cómo utilizar interfaces?

El primer paso para utilizar interfaces es, aunque suene obvio, identificar la situación en que pueden ser de beneficio. Cualquier código pensado para utilizar interfaces es posible escribirlo sin necesidad de incluirlas, es por ello que saber identificar el caso adecuado es tan importante como saber usarlas. En este caso supongo que deseo almacenar información de figuras geométricas, las cuales, comparten métodos que se pueden consultar en todas ellas pero con algoritmos diferentes para cada una, como lo es el cálculo del área. Independientemente de que se trabaje con rectángulos, círculos, trapecios etc. todas las figuras geométricas permiten calcular su área aunque de forma distinta para todas ellas. La siguiente es una interfaz tipo **FiguraGeometrica** que contiene un método que devuelve el área de la figura:

1. type FiguraGeometrica interface {
2. area() float64
3. }

En seguida se declaran los miembros que trabajaran con la interfaz. Las figuras serán registros de tipo diferente. En mi caso he decidido que para ejemplificar el funcionamiento solamente usaré rectángulos y trapecios:

1. type Rectangulo struct {
2. /\* Información de rectángulos \*/
3. base, altura float64
4. }
5. type Trapecio struct {
6. /\* Información de trapecios \*/
7. base\_mayor, base\_menor, altura float64
8. }

**Nota**: en Golang no hay objetos como en Java o C++, por eso se trabaja con registros (*struct*). Una vez que se han declarado las figuras, construimos el método **area()** para cada una de ellas, este recibirá de argumento un registro del tipo de figura de la que deseemos calcular su área:

1. func(figura Rectangulo) area() float64 {
2. return figura.base \* figura.altura
3. }
4. func(figura Trapecio) area() float64 {
5. return (figura.base\_mayor + figura.base\_menor) \* figura.altura / 2
6. }

Se puede crear una función que acceda al método si así se desea. Para ejemplificar utilizaremos la función **dameArea()** que recibe de argumento un registro tipo **FiguraGeométrica** que en realidad será del tipo de figura que deseamos consultar su área pero enmascarado. Go determina a qué método **area()** deseamos acceder dependiendo del tipo real de ese argumento:

1. func dameArea(fig FiguraGeometrica) float64 {
2. return fig.area()
3. }

No es necesario crear la función **dameArea()** ya que podemos utilizar de forma directa el registro para llamar al método, suponiendo que deseamos imprimir lo que este devuelve podemos hacer lo siguiente:

1. fmt.Println(trapecio.area())

**Nota**: ambos modos de acceso a métodos (con función o con registro) son validos, su uso depende del programador y de las necesidades del proyecto. El siguiente es el ejemplo funcional completo incluyendo el llamado a cada método desde el ***main***:

1. package main
2. import "fmt"
3. /\* Declaración de una interfaz tipo FiguraGeométrica\*/
4. type FiguraGeometrica interface {
5. area() float64
6. //Aquí podrían ir más métodos
7. }
8. /\* Declaración de registro tipo Rectangulo \*/
9. type Rectangulo struct {
10. base, altura float64
11. }
12. /\* Declaración de registro tipo Trapecio \*/
13. type Trapecio struct {
14. base\_mayor, base\_menor, altura float64
15. }
16. /\* Método para los registros Rectangulo implementando la interfaz \*/
17. func(figura Rectangulo) area() float64 {
18. return figura.base \* figura.altura
19. }
20. /\* Método para los registros Trapecio implementando la interfaz \*/
21. func(figura Trapecio) area() float64 {
22. return (figura.base\_mayor + figura.base\_menor) \* figura.altura / 2
23. }
24. /\* Definición de un método para Figura \*/
25. func dameArea(fig FiguraGeometrica) float64 {
26. return fig.area()
27. }
28. func main() {
29. rectangulo := Rectangulo {base: 4, altura: 7.5}
30. trapecio := Trapecio {base\_mayor: 5, base\_menor: 2, altura: 3}
31. /\* Llamado a función dameArea() \*/
32. fmt.Printf("Area del rectángulo: %f\n", dameArea(rectangulo))
33. /\* Uso del registro para llamar a area() \*/
34. fmt.Printf("Area del trapecio: %f\n", trapecio.area())
35. }

Salida:



Finalizando…

Aunque Golang no es un lenguaje de programación orientado a objetos, nos permite simular métodos y sobrecarga para facilitarnos un poco la abstracción de problemas propios de los lenguajes orientados a objetos. Además, si utilizamos métodos será más sencillo trabajar con un único identificador para realizar cierta operación similares; en este caso se pudo volver poco práctico crear una función **dameAreaRectangulo()** y una **dameAreaTrapecio()** puesto que necesitaríamos crear funciones con diferentes identificadores para cada figura nueva que pudiéramos agregar a nuestro programas; en su lugar los métodos nos permiten utilizar una sola función **dameArea()** o simplemente utilizar el registro para llamar a **area()**.

Con esta entrada terminamos la primera parte de programación en Go básico, espero que todo te haya sido de utilidad.

func Ceil(x [float64](https://golang.org/pkg/builtin/" \l "float64)) [float64](https://golang.org/pkg/builtin/" \l "float64)

Ceil returns the least integer value greater than or equal to x.

func Floor(x [float64](https://golang.org/pkg/builtin/" \l "float64)) [float64](https://golang.org/pkg/builtin/" \l "float64)

Floor returns the greatest integer value less than or equal to x.