Otras operaciones de Slice

Las siguientes operaciones, de momento, no se pueden establecer con funciones propias de Golang:

- Remover elementos
- Insertar en una posicion dada un elemento.

Remover un elemento

Se hacen dos cortes en el slice a partir de la posicion dada.

Dado `[10, 20, 30, 40]`, al remover 20 queda algo similar a `10 + [30, 40]`

```
func remover([]int slice, int posicion) []int{
    return append(slice[:posicion], slice[posicion + 1:]...)
}
```

Insertar un elemento

Similar a la eliminacion dada una posicion se hace un corte, donde se inserta el nuevo valor.

Dado `[10, 20, 30]`, insertar 40 en la posicion 1 -> `10 + 40 + [20, 30]`

```
func insertar(slice []int, pos int, valor int) []int {
        return append(slice[:pos], valor, slice[pos:]...)
}
```

Structs

Generador de objetos

En Go, los structs se utilizan para crear tipos personalizados que pueden contener datos (similar a los objetos en POO).

```go
package main

import "fmt"

// Define a struct (like a class in other languages)
type Car struct {
    Brand string
    Model string
    Year  int
}

func main() {
    // Create an instance of Car
    myCar := Car{
        Brand: "Toyota",
        Model: "Corolla",
        Year:  2020,
    }

    fmt.Println("Car:", myCar.Brand, myCar.Model, myCar.Year)
}
```

Funciones

Go te permite definir métodos en structs, de manera similar a cómo definirías métodos en clases en otros lenguajes.

```
package main

import "fmt"

func (c Car) Details() string {
    return fmt.Sprintf("Brand: %s, Model: %s, Year: %d", c.Brand, c.Model, c.Year)
}

// Constructor function
func NewCar(brand, model string, year int) Car {
    return Car{Brand: brand, Model: model, Year: year}
}

func main() {
    myCar := Car{Brand: "Toyota", Model: "Corolla", Year: 2020}
    fmt.Println(myCar.Details())

    // usando el constructor
    myCar2 := NewCar("Toyota", "Corolla", 2020)
}
```

Es posible utilizar constructores para crear instancias de un struct, como se muestra en el siguiente ejemplo. Este constructor no es un método del struct, sino una función normal que devuelve una instancia del struct.

```
package main

import "fmt"

func NewCar(brand, model string, year int) Car {
    return Car{Brand: brand, Model: model, Year: year}
}

func main() {
    myCar := NewCar("Toyota", "Corolla", 2020)
    fmt.Println(myCar.Details())
}
```

Encapsulamiento

Go logra el encapsulamiento utilizando reglas de visibilidad. Los identificadores que comienzan con una **letra mayúscula** son exportados (públicos), mientras que los que comienzan con una **letra minúscula** son no exportados (privados).

```
type Car struct {
    Brand string // atributo public
    model string // atributo private
}

// metodo public
func (c Car) ShowModel() string {
    return c.model
}
```

Composición

Go no admite herencia como otros lenguajes orientados a objetos, pero se puede lograr una funcionalidad similar mediante la composición, incrustando un struct dentro de otro.

```go
package main

import "fmt"

type Engine struct {
    Horsepower int
}

type Car struct {
    Brand  string
    Model  string
    Engine // Embeber el struct Engine
}

func main() {
    myCar := Car{
        Brand:  "Toyota",
        Model:  "Corolla",
        Engine: Engine{Horsepower: 130},
    }
    fmt.Println("Potencia del motor:", myCar.Horsepower)
}
```

Interfaces

Interfaces

Go utiliza interfaces para implementar polimorfismo. Una interfaz es una colección de firmas de métodos, y cualquier tipo que implemente esos métodos se considera que satisface la interfaz.

```go
package main

import "fmt"

// Definir la interface
type Vehicle interface {
    Drive() string
}

// Implementa la interface para el struct Car
func (c Car) Drive() string {
    return fmt.Sprintf("Conduciendo %s %s", c.Brand, c.Model)
}

/* Esta funcion toma un objeto que implementa la interface Vehicle
o cualquier tipo que implemente el metodo Drive() */
func StartVehicle(v Vehicle) {
    fmt.Println(v.Drive())
}

func main() {
    myCar := Car{Brand: "Toyota", Model: "Corolla", Year: 2020}
    StartVehicle(myCar)
}
```

Punteros

Punteros

En Go, los punteros se utilizan para almacenar la dirección de memoria de una variable.
Los punteros se utilizan para pasar valores por referencia a funciones, lo que significa que la función puede modificar el valor original de la variable.
Cuando usamos Scan, por ejemplo, necesitamos pasar la dirección de la variable, no el valor.

Declaración y Uso

Un puntero se declara utilizando el símbolo * junto con el tipo de dato al que apunta.
Puedes obtener la dirección de memoria de una variable utilizando el operador & y acceder al valor al que apunta un puntero utilizando el operador *.

```go
package main

import "fmt"

func main() {
    x := 10
    p := &x // p es un puntero a x

    // forma mas convencional de declarar un puntero
    var q *int = &x

    fmt.Println("Valor de x:", x)
    fmt.Println("Direccion de x:", p)
    fmt.Println("Valor de x a traves del puntero:", *p)
}
```

Beneficios de los Pointers

- **Eficiencia:** Evitan copiar grandes estructuras de datos, ya que se trabaja directamente con su dirección.
- **Mutabilidad:** Permiten modificar el valor original desde una función, similar al paso por referencia.

Diferencia con los punteros de C

En Go, los punteros no admiten aritmética de punteros como en C. No puedes sumar o restar un número a un puntero.