مقدمه

فرض کن داریم یه برنامه ساده می‌نویسیم که قراره دمای هوا رو از سلسیوس به فارنهایت تبدیل کنه. رابطه‌اش رو که حتما دیدی:

```
F = (C × 1.8) + 32
```

حالا مثلا می‌خوای دمای 4 سوله از یک کارخونه رو که به ترتیب 10، 25، 37 و 40 درجه سلسیوس هستن به فارنهایت تبدیل کنی. چیکار می‌کنی؟ خیلی ساده، میای این رابطه رو برای هر دما جداگانه تو کدت می‌نویسی:

```
f1 := (10.0 * 1.8) + 32
f2 := (25.0 * 1.8) + 32
f3 := (37.0 * 1.8) + 32
f4 := (40.0 * 1.8) + 32
```

همه‌چیز فعلاً خوبه... ولی یه لحظه وایسا! واقعاً خوبه؟

مشکل کجاست؟

1. **تکرار بی‌مورد فرمول** :مجبور شدی 4 بار دقیقاً همون فرمول رو تکرار کنی. اگه بعداً بخوای دمای 10 شهر رو بزنی چی؟ یا 50 تا؟ باز باید همین فرمولو 50 بار تکرار کنی؟
2. **نگهداری سخت** :فرض کن یه روز تصمیم می‌گیری به‌جای عدد 1.8 از یه عدد دیگه استفاده کنی. حالا باید کل کدت رو بگردی دنبال این فرمول و دستی تغییرش بدی. وای اگه یه‌جا یادت بره یا اشتباهی بزنی!
3. **کپی‌پیستِ خسته‌کننده** :شاید با خودت بگی: "خب کپی پیست که چیز سختی نیست"... اما اینم در نظر بگیر: همه‌ی فرمول‌ها انقدر ساده نیستن. بعضی‌هاشون خودشون چند تا محاسبه‌ی ریز و درشت دارن. حالا اگه مجبور شی کل اون محاسبات رو هم چند بار کپی‌پیست کنی، تبدیل می‌شی به یه کپی‌پیست‌زن حرفه‌ای. بعدش یه تغییر کوچیک باعث می‌شه کلی جا خراب شه!
4. **خوانایی پایین** :اگه یکی دیگه بخواد کدت رو بخونه، احتمالاً با یه عالمه خط فرمول تکراری روبه‌رو می‌شه و هیچ ایده‌ای نداره که این همه فرمول چی‌ن و چرا اینطوری تکرار شدن.

راه‌حل چیه؟ تابع!

اینجاست که یه چیزی به‌اسم **تابع (Function)** میاد به کمکمون.
تابع یعنی: یه بار بنویس، هزار بار استفاده کن!

با تابع، می‌تونی اون فرمول تبدیل دما رو یه‌بار تو یه تابع بنویسی، و بعد هر وقت لازم داشتی، فقط اون تابع رو صدا بزنی:

```
fmt.Println(celsiusToFahrenheit(10.0))
fmt.Println(celsiusToFahrenheit(25.0))
```

هم کدت خیلی مرتب‌تر و کوتاه‌تر می‌شه، هم اگه یه روز خواستی فرمولو تغییر بدی، فقط کافیه بری توی یه خط، همون تابع رو تغییر بدی! بقیه‌ی جاها خودش درست کار می‌کنه!

حالا باید ببینیم که چجوری میشه یه تابع جدید برای انجام کاری تعریف کرد

ساختار کلی تعریف یک تابع

در زبان Go برای تعریف یک تابع از کلمه‌ی کلیدی func استفاده می‌کنیم. ساختار کلی یک تابع به شکل زیره:

```
func functionName(parameter1 type1, parameter2 type2, ...) returnType {
    // بدنه تابع
}
```

بیایم این ساختار رو دقیق‌تر بررسی کنیم:

func-1 (کلمه کلیدی)

همه چیز از اینجا شروع می‌شه! کلمه‌ی func مشخص می‌کنه که قراره یک تابع تعریف کنیم.

2- نام تابع

بعد از func نام تابع میاد. این نام باید واضح و توصیفی باشه، طوری که بشه از اسمش حدس مثلاً celsiusToFahrenheit زد چه کاری انجام میده.

3-پارامترها(Inputs)

توی پرانتز بعد از اسم تابع، لیست پارامترهایی که تابع بهشون نیاز داره رو می‌نویسیم. هر پارامتر شامل یه نام و نوع داده‌ست:

```
func greet(name string, age int)
```

اگه چند تا پارامتر پشت‌سر هم از یک نوع داشته باشیم، می‌تونیم نوع رو فقط یک بار بنویسیم:

```
func add(a, b int)
```

4-مقدار بازگشتی (Output)

بعد از پرانتز پارامترها، می‌تونیم نوع خروجی تابع رو بنویسیم:

```
func square(x int) int
```

ولی یک نکته مهم اینه که **می‌تونیم برای مقدار بازگشتی، اسم هم مشخص کنیم** (البته اختیاریه). این کار باعث می‌شه خروجی داخل بدنه تابع قابل استفاده باشه و حتی بدون return مقداردهی بشه:

```
func square(x int) (result int) {
    result = x * x
    return // اینجا نیازی به نوشتن return result نیست، چون خودش مشخصه
}
```

توابعی که چند تا مقدار برمی‌گردونن هم می‌تونن برای هر خروجی نام بذارن:

```
func getNameAndAge() (name string, age int) {
    name = "Arian"
    age = 13
    return
}
```

و اگه نخواهیم اسم‌گذاری کنیم، می‌تونیم به‌سادگی نوع‌ها رو بنویسیم:

```
func getCoordinates() (float64, float64)
```

5- بدنه تابع

بین } و { دستورات تابع قرار می‌گیرن. این قسمت جاییه که منطق تابع اجرا می‌شه. معمولاً توش از دستور return برای برگردوندن نتیجه استفاده می‌کنیم، مگر اینکه خروجی رو با نام مشخص کرده باشیم (که تو اون صورت می‌تونیم returnخالی بنویسیم).

مثال: تبدیل سانتی‌گراد به فارنهایت

بدون نام برای خروجی:

```
func celsiusToFahrenheit(c float64) float64 {
    return (c * 1.8) + 32
}
```

با نام برای خروجی:

```
func celsiusToFahrenheit(c float64) (f float64) {
    f = (c * 1.8) + 32
    return
}
```

جدول مقایسه استفاده کردن و استفاده نکردن از تابع

ویژگی/مزیت	با استفاده از تابع	بدون تابع
قابلیت استفاده مجدد	یک‌بار تعریف، هرچندبار که بخوای استفاده کن	باید بارها و بارها یک منطق رو تکرار کنیم
خوانایی کد	کد مرتب‌تر و قابل فهم‌تر	کد شلوغ، طولانی و گیج‌کننده
اشکال‌زدایی و تست کردن	راحت‌تر چون می‌تونیم تابع‌ها رو جداگانه تست کنیم	سخت‌تر چون همه چیز قاطیه
تفکیک وظایف(Modularity)	هر تابع یه کار مشخص انجام می‌ده	همه‌ی منطق در هم قاطیه
توسعه‌پذیری در آینده	راحت می‌تونیم تابع‌ها رو آپدیت کنیم	تغییرات دردسر داره
کاهش خطاهای انسانی	یه منطق واحد داریم، احتمال خطا کمتره	کپی-پیست زیاد = احتمال خطای بیشتر

جمع بندی

دیدیم که یک تابع می‌تونه پارامتر ورودی داشته باشه یا نداشته باشه، می‌تونه مقدار برگردونه یا نه، و حتی می‌تونیم برای خروجی‌هاش اسم بذاریم یا نه. همه‌ی این‌ها ابزارهایی هستن که باعث می‌شن کدمون تمیزتر، قابل‌استفاده‌تر و قابل‌فهم‌تر بشه.

فراخوانی تابع

برای استفاده از یک تابعی که تعریف کردیم، کافیه **اسم تابع رو بنویسیم و در ادامه، پرانتز باز و بسته بذاریم** .اگر تابع پارامتر ورودی داشته باشه، باید داخل پرانتز مقدار بدیم. اگر هم خروجی داشته باشه، می‌تونیم اون رو دریافت کنیم.

ساختار کلی فراخوانی تابع

```
result := functionName(arg1, arg2, ...)
```

functionName: اسم تابعی که می‌خوایم صدا بزنیم

arg1, arg2, ...: مقادیری که قراره به تابع پاس بدیم

result: متغیری که خروجی تابع داخلش ریخته می‌شه

پاس دادن پارامترها

اگر تابع یک پارامتر بگیره:

```
c := 25
f := celsiusToFahrenheit(c)
```

یا حتی مستقیم:

```
f := celsiusToFahrenheit(25)
```

اگر تابع چند پارامتر بگیره:

```
result := addThreeNumbers(10, 20, 30)
```

تعداد و نوع پارامترهایی که می‌دیم، باید دقیقاً با تعریف تابع هماهنگ باشه.

دریافت مقادیر بازگشتی

اگر فقط یک خروجی داشته باشیم:

```
f := celsiusToFahrenheit(30)
fmt.Println(f)
```

یا حتی بدون ذخیره کردن تو متغیر:

```
fmt.Println(celsiusToFahrenheit(30))
```

اگر چند خروجی داشته باشیم:

```
func calculateBMI(weight, height float64) (float64, string) {
    bmi := weight / (height * height)
    status := "Normal"
    if bmi < 18.5 {
        status = "Underweight"
    } else if bmi > 25 {
        status = "Overweight"
    }
    return bmi, status
}
```

فراخوانیش این‌طوری می‌شه:

```
bmi, status := calculateBMI(70, 1.75)
fmt.Println("BMI:", bmi, "Status:", status)
```

نادیده گرفتن مقادیر بازگشتی

گاهی اوقات همه‌ی خروجی‌ها برامون مهم نیست. تو این حالت می‌تونیم از **علامت** _ (Blank Identifier) استفاده کنیم.

نادیده گرفتن خروجی دوم:

```
bmi, _ := calculateBMI(70, 1.75)
fmt.Println("BMI فقط:", bmi)
```

نادیده گرفتن خروجی اول:

```
_, status := calculateBMI(70, 1.75)
fmt.Println("وضعیت:", status)
```

فراخوانی تابع celsiusToFahrenheit

```
package main
import “fmt”

func celsiusToFahrenheit(c float64) float64 {
    return (c * 1.8) + 32
}

func main() {
    var temperatureInCelsius float64 = 10.0
    temperatureInFahrenheit := celsiusToFahrenheit(temperatureInCelsius)
    fmt.Println(“Temperature in Fahrenheit:”, temperatureInFahrenheit)
}
```

انواع توابع

تا اینجای کار یاد گرفتیم که تابع چیه، چطور تعریفش کنیم و چطور صداش بزنیم.
حالا وقتشه که یک قدم جلوتر بریم و انواع مختلف توابع رو بررسی کنیم.
در دنیای واقعی، توابع می‌تونن ساده یا پیچیده باشن، با ورودی یا بدون ورودی، با خروجی یا بدون خروجی، و حتی چند تا خروجی داشته باشن!

توی این بخش، قراره 6 نوع متداول تابع رو با هم یاد بگیریم، برای هر کدوم یه توضیح مختصر بدیم و یه مثال کاربردی واقعی هم براش بزنیم که بهتر تو ذهن جا بیفته.

1- تابع بدون ورودی و بدون خروجی

گاهی اوقات نیازی نداریم تابع چیزی بگیره یا چیزی برگردونه؛ فقط می‌خوایم یه کاری انجام بده. اینجا همه‌چی داخل خود تابع انجام میشه. نه چیزی از بیرون می‌گیره، نه چیزی برمی‌گردونه:

```
func showBMI() {
    weight := 70.0
    height := 1.75
    bmi := weight / (height * height)
    fmt.Println("BMI:", bmi)
}
```

2-تابع با یک ورودی و بدون خروجی

بعضی وقت‌ها تابع نیاز داره اطلاعاتی از بیرون بگیره ولی خروجی نداره.
تو مثال زیر با وارد کردن ورودی mode (که یکی از مقادیر number یا text میتونه داشته باشه) مشخص میکنیم که نتیجه محاسبه BMI رو به صورت عددی بنویسه یا متنی.

```
func showBMI(mode string) {
    weight := 70.0
    height := 1.75
    bmi := weight / (height * height)

    if mode == "number" {
        fmt.Println("BMI:", bmi)
    } else if mode == "text" {
        var status string
        switch {
        case bmi < 18.5:
            status = "Underweight"
        case bmi < 25:
            status = "Normal"
        case bmi < 30:
            status = "Overweight"
        default:
            status = "Obese"
        }
        fmt.Println("Body Status:", status)
    } else {
        fmt.Println("Invalid mode. Please use 'number' or 'text'.")
    }
}
```

3-تابع با چند ورودی و بدون خروجی

بعضی توابع نیاز دارن که به جای تنها یک ورودی چند ورودی داشته باشن. به طور کلی توابع هر چه ورودی های بیشتری داشته باشه یعنی کنترل بیشتر و آزادی عملی بیشتری داریم.

در مثال زیر با تغییر مثال قبل و اضافه کردن weight و height به عنوان پارامتر برنامه انعطاف پذیر تر شد و میتونه برای هر وزن و قدی محاسبه BMI انجام بده و نتیجه رو به صورت عددی یا متنی چاپ کنه.

```
func showBMI(weight, height float64, mode string) {
    bmi := weight / (height * height)

    if mode == "number" {
        fmt.Println("BMI:", bmi)
    } else if mode == "text" {
        var status string
        switch {
        case bmi < 18.5:
            status = "Underweight"
        case bmi < 25:
            status = "Normal"
        case bmi < 30:
            status = "Overweight"
        default:
            status = "Obese"
        }
        fmt.Println("Body Status:", status)
    } else {
        fmt.Println("Invalid mode. Please use 'number' or 'text'.")
    }
}
```

4-تابع با چند ورودی و یک خروجی

وقتی تعداد پارامترهای ورودی تابع بیشتر می‌شه، انعطاف‌پذیری و آزادی عمل بیشتری پیدا می‌کنیم. ولی فقط داشتن ورودی کافی نیست؛ داشتن مقدار بازگشتی (خروجی) هم به همون اندازه مهمه.

تو مثال زیر تابع با دریافت وزن، قد و یک حالت (چاپ عدد یا متن) کار می‌کنه، اما در نهایت مقدار عددی BMI رو برمی‌گردونه. می‌تونیم نتیجه رو چاپ کنیم، یا با مقدار BMI در جای دیگه کد تصمیم‌گیری کنیم، یا حتی این مقدار رو به توابع دیگه بدیم برای محاسبات پیچیده‌تر.

```
func showBMI(weight, height float64, mode string) float64 {
    bmi := weight / (height * height)

    if mode == "number" {
        fmt.Println("BMI:", bmi)
    } else if mode == "text" {
        var status string
        switch {
        case bmi < 18.5:
            status = "Underweight"
        case bmi < 25:
            status = "Normal"
        case bmi < 30:
            status = "Overweight"
        default:
            status = "Obese"
        }
        fmt.Println("Body Status:", status)
    } else {
        fmt.Println("Invalid mode. Please use 'number' or 'text'.")
    }

    return bmi
}
```

5-تابع با چند ورودی و چند خروجی

داشتن چند خروجی باعث می‌شه آزادی عمل و انعطاف‌پذیری کدنویسی خیلی بیشتر بشه، چون می‌تونیم چند داده مختلف رو همزمان از تابع بگیریم و تصمیم بگیریم کجا و چطور ازشون استفاده کنیم.

این باعث می‌شه کد تمیزتر، خواناتر و قابل توسعه‌تر باشه، چون خود تابع فقط مسئول محاسبه است و وظیفه‌ی نمایش یا پردازش خروجی‌ها به جای دیگه (جایی که تابع رو فراخوانی می‌کنیم) واگذار می‌شه.

در مثال قبلی، تابع فقط یک خروجی داشت — مقدار عددی BMI — که خودش یا چاپ می‌شد یا در جایی دیگه استفاده می‌شد.
اما حالا تو این مثال، تفاوت اصلی اینه که تابع ما چند خروجی داره: هم مقدار عددی BMI و هم تفسیر وضعیت سلامت بدن به صورت متن.

```
func showBMI(weight, height float64) (float64, string) {
    bmi := weight / (height * height)

    var status string
    switch {
    case bmi < 18.5:
        status = "Underweight"
    case bmi < 25:
        status = "Normal"
    case bmi < 30:
        status = "Overweight"
    default:
        status = "Obese"
    }

    return bmi, status
}
```

تابع با خروجی‌های نام‌گذاری‌شده

علاوه بر اینکه می‌تونیم چند مقدار خروجی داشته باشیم، امکانش هست به هر خروجی یک اسم یا نام دلخواه بدیم.

این کار چند تا مزیت داره:

- کد خواناتر می‌شه چون وقتی خروجی‌ها اسم داشته باشن، مشخص‌تره که هر مقدار چی هست.
- درون بدنه تابع می‌تونیم بدون نوشتن صریح return همراه با مقادیر، فقط return ساده بزنیم چون خروجی‌ها از قبل مقداردهی شدن.
- مستندسازی بهتر می‌شه و فهم تابع برای برنامه‌نویس‌ها ساده‌تر می‌شه.

```
func showBMI(weight, height float64) (bmi float64, status string) {
    bmi = weight / (height * height)

    switch {
    case bmi < 18.5:
        status = "Underweight"
    case bmi < 25:
        status = "Normal"
    case bmi < 30:
        status = "Overweight"
    default:
        status = "Obese"
    }

    return  // ساده کافیه return چون خروجی‌ها نام دارن، فقط
}
```

همونطور که دیدیم، وقتی از خروجی‌های نامگذاری‌شده استفاده می‌کنیم، دیگه لازم نیست برای متغیر status داخل تابع جداگانه تعریف کنیم. در واقع، این کار باعث می‌شه تعداد خطوط کد داخل بدنه تابع کمتر بشه، ولی در عین حال خوانایی و نظم کد بهتر و واضح‌تر بمونه.

نکته درباره انواع توابع

در واقع، اگر دقیق نگاه کنیم، بر اساس ترکیب ورودی و خروجی‌ها توابع می‌تونن 9 حالت مختلف داشته باشن

حالت	ورودی	خروجی
1	0	0
2	1	0
3	چند	0
4	0	1
5	1	1
6	چند	1
7	0	چند
8	1	چند
9	چند	چند

اما ما برای اینکه طولانی نشه و گیج‌کننده نباشه، فقط روی حالت های 1، 2، 3، 6 و 9 تمرکز کردیم این 5 حالت، پایه و ستون اصلی کار با توابع هستن و وقتی این‌ها رو خوب یاد بگیری، بقیه حالت‌ها رو هم خیلی راحت می‌فهمی و می‌تونی تو برنامه‌هات استفاده کنی.

قبلاً یه تابع ساده نوشته بودیم برای تبدیل دما از سلسیوس به فارنهایت و دیدیم چطوری میشه ازش استفاده کرد

```
package main
import "fmt"

func celsiusToFahrenheit(c float64) float64 {
    return (c * 1.8) + 32
}

func main() {
    var temperatureInCelsius float64 = 10.0
    temperatureInFahrenheit := celsiusToFahrenheit(temperatureInCelsius)
    fmt.Println("Temperature in Fahrenheit:", temperatureInFahrenheit)
}
```

حالا فرض کن قراره دمای 4 تا سوله‌ی یه کارخونه رو محاسبه کنیم (دقیقاً همون موقعیتی که اول این درسنامه بهش اشاره کردیم).
ساده‌ترین راه اینه که برای هر دما، یه بار تابع celsiusToFahrenheit رو صدا بزنیم.

دقیقاً مثل کاری که اینجا کردیم:

```
package main
import "fmt"

func celsiusToFahrenheit(c float64) float64 {
    return (c * 1.8) + 32
}

func main() {
    var warehouseATempInCelsius float64 = 10.0
    var warehouseBTempInCelsius float64 = 25.0
    var warehouseCTempInCelsius float64 = 37.0
    var warehouseDTempInCelsius float64 = 40.0

    warehouseATempInFahrenheit := celsiusToFahrenheit(warehouseATempInCelsius)
    warehouseBTempInFahrenheit := celsiusToFahrenheit(warehouseBTempInCelsius)
    warehouseCTempInFahrenheit := celsiusToFahrenheit(warehouseCTempInCelsius)
    warehouseDTempInFahrenheit := celsiusToFahrenheit(warehouseDTempInCelsius)

    fmt.Printf("Warehouse A Temp: %5.2fF°\n", warehouseATempInFahrenheit)
    fmt.Printf("Warehouse B Temp: %5.2fF°\n", warehouseBTempInFahrenheit)
    fmt.Printf("Warehouse C Temp: %5.2fF°\n", warehouseCTempInFahrenheit)
    fmt.Printf("Warehouse D Temp: %5.2fF°\n", warehouseDTempInFahrenheit)
}
```

تو این روش برای هر دما یک متغیر ساختیم، بعدش هم برای دمای تبدیل‌شده یه متغیر دیگه، و در آخر برای هرکدوم یه fmt.Printf جدا نوشتیم! خب طبیعیه که وقتی داده‌ها زیاد بشن، این روش خیلی بهینه نیست.

اینجاست که می‌تونیم با کمک آرایه یا بهتر بگیم slice، هم کدمونو کوتاه‌تر کنیم، هم قابل‌مدیریت‌تر.

بیا همین مثال رو با استفاده از slice بازنویسی کنیم:

```
package main
import "fmt"

func celsiusToFahrenheit(c float64) float64 {
    return (c * 1.8) + 32
}

func main() {
    var tempsInCelsius []float64 = []float64{10.0, 25.0, 37.0, 40.0}
    var tempsInFahrenheit []float64 = make([]float64, len(tempsInCelsius))

    for index, temp := range tempsInCelsius {
        tempsInFahrenheit[index] = celsiusToFahrenheit(temp)
    }

    for index, temp := range tempsInFahrenheit {
        fmt.Printf("Warehouse %d Temp: %5.2fF°\n", index, temp)
    }
}
```

همون‌طور که دیدی، کد خیلی جمع‌وجورتر شد. مرحله‌به‌مرحله اینجوریه:

1. اول یه slice به اسم tempsInCelsius ساختیم و مقادیر دمای هر سوله رو داخلش ریختیم.
2. بعد یه slice خالی به اسم tempsInFahrenheit با همون طول ساختیم تا دماهای تبدیل‌شده توش ذخیره بشن.
3. با for range روی tempsInCelsius چرخیدیم، هر دما رو به فارنهایت تبدیل کردیم و همونجا توی tempsInFahrenheit ذخیره کردیم.
4. در آخر هم با یه حلقه دیگه روی tempsInFahrenheit چاپ کردیم.

نکته: حتی می‌تونستیم همون تبدیل و چاپ رو توی یه حلقه انجام بدیم. یعنی نیازی به ساختن tempsInFahrenheit جدا نبود.

برنامه‌مون خیلی خوب کار می‌کنه و همون نتیجه‌ای که می‌خوایم رو می‌ده.

ولی یه سوال مهم پیش میاد:

اگه فقط همین یه جا نباشه که قراره چندتا دما رو از سلسیوس به فارنهایت تبدیل کنیم چی؟ ممکنه تو جاهای دیگه برنامه هم همین نیاز پیش بیاد. خب منطقی‌تره که یه تابع بسازیم مخصوص این کار.

یه تابع بنویسیم که یه slice از دماهای سلسیوس بگیره و یه slice از دماهای تبدیل‌شده به فارنهایت برگردونه.

تو دل این تابع هم از همون تابع celsiusToFahrenheit استفاده می‌کنیم.

```
package main
import "fmt"

func celsiusToFahrenheit(c float64) float64 {
    return (c * 1.8) + 32
}

func allCelsiusToFahrenheit(tempsInCelsius []float64) []float64 {
    var tempsInFahrenheit []float64
    for _, temp := range tempsInCelsius {
        f := celsiusToFahrenheit(temp)
        tempsInFahrenheit = append(tempsInFahrenheit, f)
    }
    return tempsInFahrenheit
}

func main() {
    var tempsInCelsius []float64 = []float64{10.0, 25.0, 37.0, 40.0}
    var tempsInFahrenheit []float64 = allCelsiusToFahrenheit(tempsInCelsius)

    for index, temp := range tempsInFahrenheit {
        fmt.Printf("Warehouse %d Temp: %5.2fF°\n", index, temp)
    }
}
```

حالا هر وقت بخوایم یه لیست از دماها رو تبدیل کنیم، فقط کافیه
تابع allCelsiusToFahrenheit رو صدا بزنیم و یه slice بهش بدیم. اینجوری:

- از تکرار کد جلوگیری می‌شه
- برنامه تمیزتر می‌شه
- و خوندنش هم راحت‌تره

یه نکته خیلی مهم:

از اونجایی که ورودی تابع allCelsiusToFahrenheit یه slice هست، باید بدونیم که اگر داخل تابع روی اون ورودی تغییری بدیم، اون تغییر روی داده‌ی اصلی هم اعمال می‌شه!

بیا اینو با یه مثال ثابت کنیم:

```
package main
import "fmt"

func allCelsiusToFahrenheit(tempsInCelsius []float64) []float64 {
    var tempsInFahrenheit []float64
    for index, temp := range tempsInCelsius {
        tempsInCelsius[index] = temp * 0
        f := celsiusToFahrenheit(temp)
        tempsInFahrenheit = append(tempsInFahrenheit, f)
    }

    return tempsInFahrenheit
}

func main() {
    var tempsInCelsius []float64 = []float64{10.0, 25.0, 37.0, 40.0}
    var tempsInFahrenheit []float64 = allCelsiusToFahrenheit(tempsInCelsius)

    for index, temp := range tempsInFahrenheit {
        fmt.Printf("Warehouse %d Temp: %5.2fF°\n", index, temp)
    }

    for index, temp := range tempsInCelsius {
        fmt.Printf("Warehouse %d Temp: %5.2fC°\n", index, temp)
    }
}
```

تو کد بالا، همه‌ی دماهای tempsInCelsius داخل تابع صفر شدن!
و بعد که دوباره توی main چاپشون کردیم، دیدیم واقعاً صفر شدن. این یعنی تابع تونسته روی داده اصلی تاثیر بذاره.

به این حالت که تغییرات روی پارامتر داخل تابع، بیرون از تابع هم دیده می‌شه، می‌گن **Call by Reference**

حالا بیا یه مثال دیگه ببینیم که برعکس اینه — یعنی تغییری که داخل تابع انجام می‌شه، بیرون تأثیری نداره:

```
package main
import "fmt"

func celsiusToFahrenheit(c float64) float64 {
    c = 0.0
    return (c * 1.8) + 32
}

func main() {
    var temperatureInCelsius float64 = 10.0
    temperatureInFahrenheit := celsiusToFahrenheit(temperatureInCelsius)

    fmt.Printf("First Warehouse Temp: %5.2fF°\n", temperatureInFahrenheit)
    fmt.Printf("First Warehouse Temp: %5.2fC°\n", temperatureInCelsius)
}
```

تو این مثال، متغیر temperatureInCelsius داخل تابع صفر شد، ولی وقتی بعد از برگشت از تابع چاپش کردیم، دیدیم همچنان 10 مونده. یعنی اینجا تابع روی مقدار اصلی اثر نذاشته. این مدل رو می‌گن **Call by Value**.

انواع داده‌ها و نحوه ارسال اون‌ها به تابع

وقتی یه چیزی (مثلاً عدد، رشته، آرایه، ...) رو به یه تابع می‌فرستی، بسته به نوع اون چیز، یا یه کپی ازش ارسال می‌شه (Call by Value) یا مستقیماً فرستاده می‌شه (Call by Reference).

تو جدول زیر مشخص شده که هر نوع داده تو زبان Go به چه صورتی به تابع فرستاده می‌شه:

نوع داده	حالت ارسال به تابع	توضیح
int, float64, bool, string	Call by Value	فقط یه کپی از مقدار اصلی به تابع داده می‌شه.
array	Call by Value	کل آرایه کپی می‌شه (نه خیلی به‌صرفه!)
slice	Call by Reference	فقط آدرس حافظه‌اش ارسال می‌شه، پس تابع می‌تونه اصل داده رو تغییر بده.
map	Call by Reference	تغییرات داخل تابع روی map اصلی هم اثر داره.
struct	Call by Value	یک کپی کامل از struct به تابع داده می‌شه.
pointer	Call by Reference	آدرس حافظه مستقیماً ارسال می‌شه.
channel, func, interface	Call by Reference	اینا هم در واقع خودشون به شکل reference فرستاده می‌شن.

تفاوت Call by Value و Call by Reference

ویژگی	Call by Value	Call by Reference
چی به تابع می‌ره؟	کپی از داده	خودِ داده (یا آدرسش)
داده‌ی اصلی تغییر می‌کنه؟	نه	بله
امن‌تره؟	بله (چون اصلی رو نمی‌زنه!)	ممکنه خراب کنه
مصرف حافظه؟	بیشتر	کمتر
مثال	float64, int, array, struct	slice, map, pointer

نتیجه‌گیری

تو Go بیشتر نوع‌ها به صورت Call by Value به تابع داده می‌شن، یعنی یه کپی ازشون می‌ره و اگه داخل تابع تغییرش بدی، بیرون اثری نداره. اما بعضی نوع‌ها مثل slice و map به صورت Call by Reference فرستاده می‌شن، یعنی اگه روشون توی تابع دست ببری، اون تغییر بیرون از تابع هم دیده می‌شه.

پس موقعی که داری تابع می‌نویسی، حواست باشه:

- اگه نمی‌خوای داده‌ی اصلی تغییر کنه، نذار slice یا map مستقیماً تغییر کنن.
- اگه می‌خوای داده‌ها واقعاً تغییر کنن، از pointer یا نوع‌هایی استفاده کن که Reference هستن.
- و همیشه یه نیم‌نگاه به نوع داده داشته باش تا بدونی قراره کپی بشه یا آدرسش بره!

پارامتر تابع از نوع variadic

تو بخش قبل دیدیم چطوری می‌تونیم یه تابع تعریف کنیم که پارامتر ورودیش از نوع slice باشه. یعنی تابع می‌تونه یه لیست از داده‌ها رو به‌صورت پشت‌سرهم بگیره و باهاشون کار کنه.

اما یه نکته این وسط هست که شاید اذیتت کنه:

وقتی تابعی می‌نویسیم که ورودی‌ش slice هست، موقع فراخوانی‌ش **مجبوریم حتماً یه slice بهش بدیم**.

حالا اگه داده‌هایی که داریم توی slice نباشن چی؟

همون مثال تبدیل سلسیوس به فارنهایت رو در نظر بگیر — فرض کن دمای 4 تا سوله رو به‌جای اینکه تو یه slice داشته باشیم، به صورت 4 تا متغیر جدا جدا تعریف کردیم. اون وقت باید چی‌کار کنیم؟

خب باید دستی یه slice بسازیم، اون مقادیر رو توش بریزیم، بعد بدیم به تابع!
یه چیزی مثل این:

```
allCelsiusToFahrenheit([]float64{10.0, 25.0, 37.0, 40.0})
```

صادقانه بگم... یه کم زور داره دیگه!

بهتر نبود یه راهی بود که بشه هم تابع رو با slice صدا زد، هم بدون slice؟

خب خبر خوب اینه که تو Go یه امکان باحال داریم به اسم **پارامتر variadic**

پارامتر variadic دقیقاً برای همین ساخته شده!

این نوع پارامتر بهت اجازه می‌ده هر تعداد دلخواه مقدار رو به تابع پاس بدی — حتی بدون اینکه اون‌ها رو توی slice بریزی! ولی جالبیش اینجاست که **پشت‌صحنه Go خودش از اون مقادیر یه slice می‌سازه** و به تابع می‌ده. یعنی داخل تابع همچنان داری با یه slice کار می‌کنی.

بیا یه مثال ببینیم:

```
package main
import "fmt"

func allCelsiusToFahrenheit(temps ...float64) []float64 {
    var tempsInFahrenheit []float64

    for index, c := range temps {
        f := celsiusToFahrenheit(c)
        tempsInFahrenheit = append(tempsInFahrenheit, f)
    }

    return tempsInFahrenheit
}

func main() {
    tempsInFahrenheit := allCelsiusToFahrenheit(10.0, 25.0, 37.0, 40.0)
    for index, temp := range tempsInFahrenheit {
        fmt.Printf("Warehouse %d Temp: %5.2fF°\n", index, temp)
    }
}
```

تو نگاه اول انگار تابع allCelsiusToFahrenheit داره یه تعداد مقدار float64 می‌گیره، ولی در واقع پشت پرده Go اینا رو تبدیل می‌کنه به یه slice و به تابع می‌ده.

تابع هم دقیقا همونطوری باهاش رفتار می‌کنه که انگار یه slice گرفته.

حالا برعکسش چی؟ اگه از قبل یه slice داشته باشیم، چطوری بدیمش به تابعی که پارامترش variadic هست؟

اینجا می‌تونیم از اپراتور (ellipse) ... استفاده کنیم.
این اپراتور باعث می‌شه که Go عناصر اون slice رو از هم باز کنه و به‌صورت تکی به تابع بده
— انگار که ما جدا جدا اونارو نوشتیم!

مثالش:

```
package main
import "fmt"

func allCelsiusToFahrenheit(temps ...float64) []float64 {
    var tempsInFahrenheit []float64

    for index, c := range temps {
        f := celsiusToFahrenheit(c)
        tempsInFahrenheit = append(tempsInFahrenheit, f)
    }

    return tempsInFahrenheit
}

func main() {
    var tempsInCelsius = []float64{10.0, 25.0, 37.0, 40.0}
    tempsInFahrenheit := allCelsiusToFahrenheit(tempsInCelsius...)
    for index, temp := range tempsInFahrenheit {
        fmt.Printf("Warehouse %d Temp: %5.2fF°\n", index, temp)
    }
}
```

اما اینجاست که یه اتفاق جالب‌تر میفته!
Go باهوشه! می‌فهمه که این temps در واقع همون sliceـایه که قراره پشت صحنه ساخته بشه! پس می‌گه:

"چه کاریه؟ همین slice رو مستقیم بفرستیم دیگه"!

و این یعنی تابع باز هم همون slice رو می‌گیره و روی خودش کار می‌کنه.

حالا چون ورودی، **همون slice اصلی** هست، اگه داخل تابع تغییری توی اون انجام بدی، **روی خود tempsInCelsius اصلی هم تاثیر می‌ذاره** — یعنی همون داستان **call by reference**

جایگاه تعریف تابع

تا حالا شده از خودت بپرسی که آیا باید توابع رو **حتماً قبل از ()main** بنویسیم؟
یا می‌تونیم **بعدش** هم بنویسیم و همچنان کار کنه؟

جواب ساده‌اش اینه:

نه! تو زبان Go فرقی نمی‌کنه تابع رو قبل از main تعریف کنی یا بعدش.

یعنی این دوتا کد، از نظر کامپایلر دقیقاً یکی هستن:

```
package main
import "fmt"

func sayHi() {
    fmt.Println("Hi!")
}

func main() {
    sayHi()
}
```

```
package main
import "fmt"

func main() {
    sayHi()
}

func sayHi() {
    fmt.Println("Hi!")
}
```

تابع با اسم مشابه

تا اینجای کار، با دو تا تابع به اسم‌های celsiusToFahrenheit و allCelsiusToFahrenheit آشنا شدیم. اولی مخصوص تبدیل **یه دمای تکی** از سلسیوس به فارنهایت بود، دومی هم برای تبدیل **یه مجموعه از دماها**.

اما یه سؤال مهم اینجا پیش میاد:

وقتی قراره این دوتا تابع رو توی برنامه استفاده کنیم، باید **دو اسم مختلف** رو حفظ کنیم و به‌درستی صداشون بزنیم.

خب این یه کم اذیت‌کننده‌ست، نه؟

واقعاً چی میشد اگه **هر دوتا تابع یه اسم مشترک** داشتن؟ مثلاً هر دو اسمشون celsiusToFahrenheit بود و فقط **بر اساس نوع ورودی**، خودشون تشخیص می‌دادن کدوم کار رو انجام بدن.

یعنی مثلاً اگه ورودی‌مون float64 باشه، اون تابعی اجرا شه که یه دما رو تبدیل می‌کنه.
اگه ورودی‌مون []float64 باشه، همون اسم تابع اجرا شه، ولی این بار برای چند تا دما.

یعنی چیزی شبیه این:

```
package main
import "fmt"

func celsiusToFahrenheit(c float64) float64 {
    return (c * 1.8) + 32
}

func celsiusToFahrenheit(temps ...float64) []float64 {
    var tempsInFahrenheit []float64

    for index, c := range temps {
        f := celsiusToFahrenheit(c)
        tempsInFahrenheit = append(tempsInFahrenheit, f)
    }

    return tempsInFahrenheit
}

func main() {
    var temperatureInCelsius float64 = 10.0
    temperatureInFahrenheit := celsiusToFahrenheit(temperatureInCelsius)
    fmt.Printf("First Warehouse Temp: %5.2fF°\n", temperatureInFahrenheit)

    var tempsInCelsius = []float64{25.0, 37.0, 40.0}
    tempsInFahrenheit := celsiusToFahrenheit(tempsInCelsius...)
    for index, temp := range tempsInFahrenheit {
        fmt.Printf("Warehouse %d Temp: %5.2fF°\n", index, temp)
    }
}
```

خیلی تر و تمیز و راحت میشه، مگه نه؟

ولی یه مشکله کوچیک هست...

اگه سعی کنی **دو تا تابع با یه اسم** مثل celsiusToFahrenheit توی Go تعریف کنی، کامپایلر مستقیم بهت گیر می‌ده و می‌گه:

"تو قبلاً همچین تابعی ساختی! اسم تکراری مجاز نیست"!

یعنی چی؟ یعنی Go اصلاً همچین چیزی رو نمی‌پذیره و برنامه‌تو اجرا نمی‌کنه.
دلیلش هم واضحه:

Go از نظر طراحی، خیلی روی **سادگی**، **وضوح** و **جلوگیری از ابهام** حساسه.

برخلاف زبان‌هایی مثل C++ یا Java که **function overloading**دارن (یعنی می‌تونی چند تا تابع با اسم یکسان ولی پارامترهای متفاوت بسازی)، تو Go این امکان وجود نداره. یعنی هر تابع باید **یه اسم منحصر به‌فرد** داشته باشه. تموم!

پس چاره چیه؟

دو تا راه بیشتر نداریم:

1. **هر تابعی رو با یه اسم متفاوت تعریف کنیم** (مثل کاری که اولش کردیم).
2. **یه تابع بسازیم، ولی داخلش با شرط و منطق، دو حالت مختلف رو کنترل کنیم.**

بیا روش دوم رو با هم بررسی کنیم:

فرض کن یه تابع تعریف کردیم که بتونه هم با یه دمای تکی کار کنه، هم با یه لیست از دماها.
برای این کار مجبور می‌شیم یه **پارامتر کنترلی** به اسم مثلاً single بهش اضافه کنیم که مشخص کنه الان قراره کدوم حالت رو اجرا کنیم.

یه چیزی شبیه این:

```
package main
import "fmt"

func celsiusToFahrenheit(temps ...float64) []float64 {
    var tempsInFahrenheit []float64

    for _, c := range temps {
        f := (c * 1.8) + 32
        tempsInFahrenheit = append(tempsInFahrenheit, f)
    }

    return tempsInFahrenheit
}

func main() {
    var temperatureInCelsius float64 = 10.0
    temperatureInFahrenheit := celsiusToFahrenheit(temperatureInCelsius)
    fmt.Printf("First Warehouse Temp: %5.2fF°\n", temperatureInFahrenheit[0])

    var tempsInCelsius = []float64{25.0, 37.0, 40.0}
    tempsInFahrenheit := celsiusToFahrenheit(tempsInCelsius...)
    for index, temp := range tempsInFahrenheit {
        fmt.Printf("Warehouse %d Temp: %5.2fF°\n", index, temp)
    }
}
```

ولی واقعاً این روش خوبه؟

بذار رک بگم: نه!

چرا؟ چون این روش **تابع رو پیچیده‌تر می‌کنه**.

حالا دیگه حتی وقتی بخوایم یک دما رو تبدیل کنیم باز هم نتیجه رو به شکل slice دریافت می‌کنیم.

یعنی یه تابع داریم با **دو وظیفه** و **دو مسیر اجرا** که این خودش یه ایراد بزرگه.

حتی از نظر اصول طراحی نرم‌افزار هم این کار نقض اصول SOLID ـه!
به‌طور خاص اصل **Single Responsibility Principle (SRP)** یعنی هر تابع باید **فقط یه وظیفه** داشته باشه.

پس چیکار کنیم؟

راه‌حل پیشنهادی اینه که همون روش اول رو ادامه بدیم:
دو تا تابع با دو اسم متفاوت، هر کدوم با یه وظیفه مشخص. اینطوری هم ساده‌تره، هم خواناتر، هم قابل نگهداری‌تر.

توابع درجه یک (First-Class Function)

یادتِ تو بخش‌های قبل دوتا تابع داشتیم برای تبدیل دما از سلسیوس به فارنهایت؟
یه دونه celsiusToFahrenheit که فقط یه دما رو تبدیل می‌کرد، و یه دونه
allCelsiusToFahrenheit که یه مجموعه از دماها رو تبدیل می‌کرد.

حالا تو این بخش، می‌خوایم دو تا تابع مشابه بنویسیم، ولی این‌بار برای تبدیل از **سلسیوس به کلوین**.

```
func celsiusToKelvin(c float64) float64 {
    return c + 273.15
}

func allCelsiusToKelvin(temps ...float64) []float64 {
    var tempsInKelvin []float64

    for _, c := range temps {
        f := celsiusToKelvin(c)
        tempsInKelvin = append(tempsInKelvin, f)
    }

    return tempsInKelvin
}
```

دوتابع جدید برای تبدیل به کلوین

- celsiusToKelvin برای تبدیل یک دما
- allCelsiusToKelvin برای تبدیل یه لیست از دماها

در زیر میتونی ببینی که چطور از این توابع میتونیم استفاده کنیم

```
pckage main
import “fmt”

func main() {
    var temperatureInCelsius float64 = 10.0
    temperatureInKelvin := celsiusToKelvin(temperatureInCelsius)
    fmt.Printf("First Warehouse Temp: %5.2fK°\n", temperatureInKelvin)

    var tempsInCelsius = []float64{25.0, 37.0, 40.0}
    tempsInKelvin := allCelsiusToKelvin(tempsInCelsius...)
    for index, temp := range tempsInKelvin {
        fmt.Printf("Warehouse %d Temp: %5.2fK°\n", index, temp)
    }
}
```

تابع allCelsiusToKelvin تقریباً همون کاری رو می‌کنه که allCelsiusToFahrenheit هم قبلاً انجام می‌داد.

بیا هر دو رو کنار هم بررسی کنیم:

```
func allCelsiusToFahrenheit(temps ...float64) []float64 {
    var tempsInFahrenheit []float64

    for index, c := range temps {
        f := celsiusToFahrenheit(c)
        tempsInFahrenheit = append(tempsInFahrenheit, f)
    }

    return tempsInFahrenheit
}

func allCelsiusToKelvin(temps ...float64) []float64 {
    var tempsInKelvin []float64

    for index, c := range temps {
        f := celsiusToKelvin(c)
        tempsInKelvin = append(tempsInKelvin, f)
    }

    return tempsInKelvin
}
```

شباهت‌ها

- هر دو یه لیست از دماها می‌گیرن []float64
- یه لیست خالی برای دماهای تبدیل‌شده درست می‌کنن
- روی ورودی‌ها حلقه می‌زنن
- تو هر دور از حلقه، یه تابع برای تبدیل دما صدا می‌زنن
- نتیجه رو تو لیست ذخیره می‌کنن و در نهایت همون لیست رو برمی‌گردونن

تفاوت‌ها

فقط و فقط تو **تابع تبدیل دما**یه که صدا می‌زنن:

- یکی از celsiusToFahrenheit استفاده می‌کنه
- اون یکی از celsiusToKelvin

همین!

خب، حالا که فهمیدیم فقط اون تابع تبدیل فرق داره، یه سوال:

آیا میشه به‌جای اینکه دوتا تابع جداگونه برای تبدیل لیست دما بنویسیم، یه دونه تابع کلی‌تر داشته باشیم و اون تابع تبدیل دما رو بهش بدیم؟

یعنی به‌جای نوشتن:

```
allCelsiusToFahrenheit()
allCelsiusToKelvin()
```

فقط یه دونه تابع مثل allTempsConverter() داشته باشیم که:

- ورودی اولش لیست دماها باشه ([]float64)
- ورودی دومش تابع تبدیل دما باشه (func(float64) float64)

به توابع زیر توجه کن

```
func celsiusToFahrenheit(c float64) float64 {
    return (c * 1.8) + 32
}

func celsiusToKelvin(c float64) float64 {
    return c + 273.15
}

func allTempsConverter(converter func(float64)float64,temps ...float64) []float64
{
    var outputTemps []float64

    for _, c := range temps {
        f := converter(c)
        outputTemps = append(outputTemps, f)
    }

    return outputTemps
}
```

دو تابع برای تبدیل از **سلسیوس به فارنهایت** و از **سلسیوس به کلوین** داریم

یک تابع برای **تبدیل مجموعه ای از دماها توسط یک converter** داریم.

دوتابع celsiusToFahrenheit و celsiusToKelvin نقش converter رو در تابع allTempsConverter بازی میکنن. چون یک ورودی از نوع float64 و یک خروجی از نوع float64 دارن.

به این شکل میتونی از این تابع جدید استفاده کنی:

```
pckage main
import “fmt”

func main() {
    var tempsInCelsius = []float64{10.0, 25.0, 37.0, 40.0}

    tempsInFahrenheit := allTempsConverter(tempsInCelsius..., celsiusToFahrenheit)
    for index, temp := range tempsInFahrenheit {
        fmt.Printf("Warehouse %d Temp: %5.2fF°\n", index, temp)
    }

    tempsInKelvin := allTempsConverter(tempsInCelsius..., celsiusToKelvin)
    for index, temp := range tempsInKelvin {
        fmt.Printf("Warehouse %d Temp: %5.2fK°\n", index, temp)
    }
}
```

این مفهوم که میتونیم یک تابع رو به عنوان پارامتر به یک تابع دیگه پاس بدیم یعنی First-Class Function.

اینجا دقیقا مفهوم First-Class Function وارد میشه

تو Go توابع **first-class citizen** هستن، یعنی:

- می‌تونی یه تابع رو به عنوان **پارامتر** به یه تابع دیگه بدی
- می‌تونی یه تابع رو از یه تابع **برگردونی**
- می‌تونی یه تابع رو تو یه **متغیر ذخیره کنی** و هر وقت خواستی صداش بزنی

همین کارایی که بالا کردیم، یعنی دادن تابع تبدیل به تابع allTempsConverter، به خاطر همین ویژگی ممکن شده.

مزایای استفاده از First-Class Function

- تابع‌هات **قابل تنظیم** می‌شن و نیاز به کپی‌کاری نیست
- خیلی راحت می‌تونی **callback** بنویسی
- کدت تمیزتر و **قابل استفاده مجدد (reusable)** میشه
- می‌تونی راحت از الگوهای functional مثل map, filter, reduce استفاده کنی

نتیجه‌گیری

تو زبان Go

- توابع first-class هستن
- می‌تونی اونا رو تو متغیر بریزی
- می‌تونی به یه تابع بدی یا ازش بگیری
- همین باعث میشه Go علیرغم سادگی، کلی انعطاف تو نوشتن کدهای تمیز داشته باشه

توابع بازگشتی

بهترین مثال برای بررسی توابع بازگشتی دنباله فیبوناچیه! چون این دنباله خودش ذاتا یک مفهوم ریاضی بازگشتی داره و از طرف دیگه قبلا تو تمرینات همین تمرینو تو درسای قبل خودت با حلقه نوشته بودی.

برنامه ای که نوشته بودی احتمالا یه چیزی شبیه این بوده:

```
func fibLoop(n int) int {
    if n == 0 {
        return 0
    }
    a, b := 0, 1
    for i := 2; i <= n; i++ {
        a, b = b, a+b
    }
    return b
}
```

مزایای این روش:

- خیلی سریع اجرا میشه
- مصرف حافظه‌اش پایینه
- تعداد عملیات‌ها کنترل‌شده‌ست

معایب این روش:

- برای فهمیدن ساختار ریاضی فیبوناچی، کمتر شهودی هست
- قابل تعمیم به مسائل بازگشتی‌تر یا درختی نیست

روش دوم برای نوشتن این دنباله استفاده از توابع بازگشتیه

تابع بازگشتی چیه؟

تابع بازگشتی، یه تابعیه که **توی دل خودش، خودش رو دوباره صدا می‌زنه!**

یعنی انگار می‌گه:

بذار جواب این مسئله رو با یه نسخه کوچیک‌تر از خودش حل کنم.

اما یه نکته مهم داره:

برای اینکه این صدا زدن‌ها تموم بشه و تا ابد تکرار نشه، باید **یه شرط توقف** داشته باشه.

حالا همون محاسبه رو با **تابع بازگشتی** بنویسیم:

```
func fibRecursive(n int) int {
    if n == 0 {
        return 0
    } else if n == 1 {
        return 1
    }
    return fibRecursive(n-1) + fibRecursive(n-2)
}
```

این همونیه که تعریف ریاضی دنباله فیبوناچی رو دقیق پیاده کرده!

بازگشت یعنی چی؟

تابع بازگشتی، تابعیه که **خودش رو صدا میزنه** تا به جواب برسه.
اما برای اینکه این صدا زدن‌ها بی‌نهایت نشه، باید **شرط توقف (base case)** داشته باشه.
تو اینجا گفتیم:

- اگه n == 0 برگردون 0
- اگه n == 1 برگردون 1
- وگرنه: برو دوتا عدد قبلی رو حساب کن، جمعشون کن و برگردون

مزایای روش بازگشتی:

- کدش ساده‌تر و شهودی‌تره (مثل تعریف ریاضی)
- برای آموزش الگوریتم‌های بازگشتی عالیه
- تو مسائل درختی یا تقسیم‌وحل (divide & conquer) خیلی کاربردیه

معایب روش بازگشتی:

- ممکنه خیلی کند باشه (چون بارها و بارها مقادیر تکراری حساب میشه)
- مصرف حافظه‌اش زیاد میشه (به خاطر stack)
- اگه n بزرگ باشه ممکنه باعث **stack overflow** بشه

چطور کار می‌کنه؟ (ماجرای stack و مسیر رفت‌وبرگشت)

وقتی یه تابع بازگشتی اجرا میشه، هر بار که خودش رو صدا میزنه، سیستم باید **صبر کنه** تا نتیجه‌ی اون صدا زدن برگرده.

اینجا سیستم از یه ساختار به اسم **call stack** (پشته‌ی فراخوانی) استفاده می‌کنه:

- هر بار که تابع خودش رو صدا می‌زنه، یه "کپی" از اون تابع و متغیرهاش میره بالای stack
- وقتی به **پایین‌ترین نقطه** (base case) رسید، تازه شروع می‌کنه به برگشت
- تو هر برگشت، یه مقدار محاسبه میشه و stack کم‌کم خالی میشه

نکته مهم:

تابع بازگشتی اگه درست نوشته بشه، خیلی قوی و منعطفه، ولی اگه مراقب نباشیم، ممکنه با stack overflow یا performance پایین مواجه بشیم.

تمرین 1: تمام حالت های تبدیل دما کلوین، سلسیوس و فارنهایت

تو این درسنامه یاد گرفتیم چجوری با استفاده از توابع، دما رو بین واحدهای مختلف (سلسیوس، فارنهایت، کلوین) تبدیل کنیم. حالا وقتشه یه تمرین ترکیبی انجام بدی که هم توابع جدید بنویسی، هم از توابع قبلی استفاده کنی.

مرحله ۱: تعریف دو تابع جدید برای تبدیل به سلسیوس

- یه تابع بنویس برای تبدیل **کلوین به سلسیوس**
- یه تابع دیگه بنویس برای تبدیل **فارنهایت به سلسیوس**

مرحله ۲: استفاده از توابع قبلی با **فراخوانی ضمنی**

حالا می‌خوایم دو تابع جدید تعریف کنیم، ولی این‌بار محاسبات رو مستقیم انجام نمی‌دیم. فقط از توابع قبلی استفاده می‌کنیم:

تابع اول :**کلوین به فارنهایت**
(یعنی اول کلوین رو به سلسیوس تبدیل می‌کنی، بعد سلسیوس رو به فارنهایت)

تابع دوم :**فارنهایت به کلوین**
(یعنی اول فارنهایت رو به سلسیوس تبدیل می‌کنی، بعد سلسیوس رو به کلوین)

به این روش که یک تابع، تابع‌های دیگه رو داخل خودش صدا می‌زنه، می‌گن **فراخوانی ضمنی (Chained Calls / Composition)**.
اینکار باعث می‌شه کدت تکراری و شلوغ نشه و از توابع کوچیک‌تر به عنوان آجرهای ساخت تابع‌های پیچیده‌تر استفاده کنی

تمرین 2: نوشتن تابع فاکتوریل به‌صورت بازگشتی

تابعی بنویس که مقدار **فاکتوریل** یه عدد صحیح رو به‌صورت **بازگشتی** حساب کنه.

یادآوری کوچولو:

- فاکتوریل یه عدد مثل n یعنی حاصل‌ضرب همه‌ی عددهای طبیعی از 1 تا n
- نمادش این شکلیه n!

مثال‌ها:

- 5! = 5 × 4 × 3 × 2 × 1 = 120
- 0! = 1 (به‌صورت قراردادی)

نکات مهم برای این تمرین:

- یه تابع تعریف کن مثل func factorial(n int) int
- این تابع باید **خودش رو صدا بزنه** (تابع بازگشتی باشه)
- یه شرط پایه بذار که اگه n == 0 یا n == 1 بود، برگردونه 1
- وگرنه برگردونه n * factorial(n-1)

خروجی رو با چند عدد مختلف امتحان کن (مثلاً 5، 7، 0) تا مطمئن شی درست کار می‌کنه.

تمرین 3: ساخت قوطی‌های تصادفی و بررسی اعتبار حجم آن‌ها

هدف تمرین:

تو این تمرین قراره:

- چند مدل قوطی با شکل‌های مختلف بسازیم (استوانه مربعی، کروی، ...)
- ابعاد و حجم هر قوطی به صورت **تصادفی** تولید بشه.
- با استفاده از یه تابع variadic قوطی‌ها رو به صورت رندوم از بین مدل‌های مختلف تولید کنیم.
- با استفاده از یه تابع **کنترل‌کننده (controller)** بررسی کنیم که حجم قوطی "مناسب" هست یا نه.

مرحله 1: توابع سازنده قوطی‌ها

هر تابع، دو مقدار برمی‌گردونه:

- توضیح متنی ابعاد قوطی
- حجم قوطی

ابعاد به صورت تصادفی تولید می‌شن.

```
func squareCylinder() (string, float64) {


}


func circularCylinder() (string, float64) {


}


func sphere() (string, float64) {


}
```

مرحله 2: تابع variadic تولید قوطی‌های رندوم

این تابع یه عدد n می‌گیره که مشخص می‌کنه چندتا قوطی تولید بشه.
همچنین یه لیست از تابع‌های سازنده قوطی می‌گیره.
هر بار یکی از این تابع‌ها به صورت تصادفی انتخاب و اجرا می‌شه.

```
func generateCans(n int, makers ...func()(string, float64)) []map[string]float64 {


}
```

مرحله 3: تابع کنترل حجم

این تابع باید به عنوان پارامتر ورودی داده بشه
و تشخیص بده حجم قوطی معتبر هست یا نه.

```
func volumeController(volume float64) bool {


}
```

مرحله 4: تابع بررسی نهایی

این تابع، خروجی تابع قبلی رو می‌گیره و فقط قوطی‌های معتبر رو نگه می‌داره.

```
func filterValidCans(
    volume float64,
    controller func(float64) bool
) []map[string]float64 {


}
```

مرحله نهایی: اجرای تمرین برای 100 قوطی و نمایش اطلاعات زیر

- تعداد کلی قوطی ها
- تعداد قوطی ها با حجم معتبر
- توضیحات هر قوطی با حجم معتبر

تمرین 5: بازی Tic-Tac-Toe (XO)

این برنامه در واقع همون بازی معروف **دوز** هست — یه بازی دونفره که خیلی ساده و در عین حال سرگرم‌کننده‌ست. تو این بازی، دو بازیکن به نوبت علامت خودشون رو داخل خونه‌های جدول قرار می‌دن.

هر بازیکن یه علامت مخصوص داره:

- بازیکن اول X
- بازیکن دوم O

در هر نوبت، یکی از بازیکن‌ها باید یه خونه خالی رو انتخاب کنه و علامتش رو اونجا بذاره.
هدف اصلی اینه که یکی از بازیکن‌ها بتونه **3 تا علامت پشت سر هم** بچینه — فرقی نمی‌کنه که:

- توی یه **ردیف**
- یا یه **ستون**
- یا روی **قطر جدول** باشه

اگه هیچ‌کدوم از بازیکن‌ها موفق نشن سه‌تایی پشت سر هم قرار بدن، و همه‌ی خونه‌های جدول پر بشه، بازی **مساوی** میشه.

راهنمای نوشتن کد بازی

ساخت صفحه بازی

از یه آرایه دوبعدی با نوع [3][3]string استفاده کن

نمایش صفحه بازی

یک تابع برای رسم صفحه بازی بنویس و تو هر مرحله درست بعد از هر حرکت بازیکنان با فراخوانی این تابع صفحه رو مجدد رسم کن. برای رسم صفحه بازی از تابع fmt.Printf و format verb ها استفاده کن خصوصا از ویژگی padding هر format verb.

گرفتن حرکت بازیکن

- از بازیکن بپرس که علامتش رو در کدوم سطر و ستون می‌خواد قرار بده
- مطمئن شو که اون خونه هنوز خالیه
- اگه خالی نیست، دوباره ازش بخواه یه خونه‌ی دیگه انتخاب کنه

بعد از هر حرکت

1. بررسی کن آیا بازیکن فعلی بازی رو برده یا نه
2. اگه کسی برنده نشده، بررسی کن آیا بازی مساوی شده یا نه (یعنی همه خونه‌ها پر شده باشن ولی کسی برنده نشده باشه)
3. اگه نه، نوبت بازیکن بعدی می‌رسه

دفعات اجرای بازی

بازی باید توی یه حلقه بی‌نهایت (یا نامعلوم) اجرا بشه
تا زمانی که یکی از این دو حالت پیش بیاد:

- یکی از بازیکن‌ها برنده بشه
- یا بازی مساوی بشه

به محض اینکه یکی از این دو حالت اتفاق افتاد، از حلقه خارج میشی و بازی تموم میشه.