معرفی حافظه و آدرس‌ها

اول از همه باید بدونی که وقتی تو برنامه یه متغیر می‌سازی، اون متغیر تو حافظه کامپیوتر ذخیره میشه.

یعنی تو رم (RAM) یه جایی هست که اون مقدار رو نگه می‌داره.

هر داده یه آدرس داره

مثل اینکه یه خونه تو خیابون وجود داره که شماره داره و تو می‌تونی بری سراغ اون خونه.
تو حافظه هم هر داده یه آدرس منحصر به فرد داره که نشون میده دقیقاً کجاست.

ساختار کلی حافظه و روش آدرس دهی

Address	Value
0x00	10010100
0x01	10111010
0x02	10111110
0x03	10010000
0x04	10000100
0x05	10100000
0x06	11101000
0x07	11011110
0x08	10111011
...	...
0xFE	11011110
0xFF	10111011

مثال:

حالا فرض کن اطلاعات یکی از سوله های کارخونه رو به صورت متغیرهای جدا تعریف کردیم و میخوایم آدرسشو چاپ کنیم.

```
var warehouseName string = "North-A"
var warehouseTemperature float64 = 27.80

fmt.Println("warehouseName address:", &warehouseName)
fmt.Println("warehouseTemperature address:", &warehouseTemperature)
```

وقتی این برنامه رو اجرا کنی، آدرس‌های حافظه متغیرها رو می‌بینی، مثلاً چیزی شبیه:
— 0xc0000941c0 یعنی آدرسی که اون متغیر تو رم داره.

خروجی

```
warehouseName address: 0xc0000941c0
warehouseTemperature address: 0xc00008c0a8
```

این مقادیر آدرس‌های حافظه هستند که هر متغیر روی آن‌ها ذخیره شده.

با هر بار اجرای برنامه، آدرس ها عوض میشن، چون هر بار که برنامه اجرا میشه داده های تو مکان های جدیدی از حافظه ذخیره میشن و این قابل پیش بینی نیست.

چرا آدرس مهمه؟

چون وقتی می‌خوای یه متغیر رو به جای دیگه بفرستی (مثلاً به یه تابع)، می‌تونی به جای اینکه کل مقدارش رو کپی کنی، فقط آدرسش رو بفرستی. این کار باعث میشه برنامه‌ات سریع‌تر و بهینه‌تر باشه، چون به جای کپی کردن، مستقیم سراغ همون داده اصلی می‌ری.

همین آدرس‌ها هستن که بعداً به ما کمک می‌کنن مفهوم اشاره‌گر (pointer) رو یاد بگیریم.

معرفی اشاره‌گر (Pointer)

ساختار کلی تعریف اشاره‌گر

قبل از اینکه مثال بزنیم، بیایم ساختار کلی تعریف اشاره‌گر رو ببینیم:

```
var name *TYPE
```

var کلمه کلیدی برای تعریف متغیر جدید.

name اسم اشاره‌گر.

علامت * قبل از TYPE یعنی name اشاره‌گری به داده‌ای از نوع TYPE.

نکته: تا اینجا فقط اشاره‌گر رو تعریف کردیم ولی مقدار ندادیم.

تعریف اشاره‌گر برای مثال سوله

حالا بیایم همون متغیرهای قبلی رو داشته باشیم:

```
var warehouseName string = "North-A"

var warehouseTemperature float64 = 27.80
```

و دو اشاره‌گر تعریف کنیم، یکی برای اسم سوله و یکی برای دمای سوله:

```
var pName *string

var pTemp *float64
```

مقدار اولیه اشاره‌گرها

اگر همین الان مقدارشون رو چاپ کنیم:

```
fmt.Println("pName:", pName)
fmt.Println("pTemp:", pTemp)
```

خروجی

```
pName: <nil>
pTemp: <nil>
```

یعنی چی؟ یعنی این اشاره‌گرها هنوز به هیچ جا اشاره نمی‌کنن!

مقداردهی به اشاره‌گرها

برای اینکه اشاره‌گرها به جای درستی اشاره کنن، باید آدرس متغیرهای اصلی رو بهشون بدیم:

```
pName = &warehouseName
pTemp = &warehouseTemperature
```

اگر دوباره چاپ کنیم:

```
fmt.Println("pName:", pName)
fmt.Println("pTemp:", pTemp)
```

حالا خروجی مثل این میشه:

```
pName: 0xc0000941c0
pTemp: 0xc00008c0a8
```

حالا این اشاره‌گرها دارن به آدرس واقعی متغیرها اشاره می‌کنن.

چطوری مقدار اشاره شده رو بخونیم؟

اگه بخوایم بدونیم اشاره‌گر به چه مقداری اشاره می‌کنه، چیکار کنیم؟

باید از عملگر **dereference** یعنی * استفاده کنیم.

عملگر dereference یعنی چی؟

وقتی بنویسیم pName* یعنی بریم سراغ آدرسی که pName نگه داشته و مقدار اونجا رو بخونیم یا تغییر بدیم.

خواندن مقدار از طریق اشاره‌گر

```
fmt.Println("warehouse name:", *pName)
fmt.Println("warehouse temperature:", *pTemp)
```

خروجی

```
warehouse name: North-A
warehouse temperature: 27.8
```

تغییر مقدار اصلی از طریق اشاره‌گر

می‌تونیم مقدار اصلی رو اینطوری تغییر بدیم:

```
*pName = "South-B"
*pTemp = 22.5
```

حالا اگه متغیرهای اصلی رو چاپ کنیم:

```
fmt.Println("warehouse name:", warehouseName)

fmt.Println("warehouse temperature:", warehouseTemperature)
```

خروجی میشه:

```
warehouse name: South-B

warehouse temperature: 22.5
```

پس تغییرات واقعا روی متغیر اصلی اعمال شده.

dereference کردن اشاره‌گر مقداردهی نشده

اما اگه اشاره‌گر تعریف کردیم ولی مقداردهی نکردیم و بخوایم **pName*** رو بخونیم یا تغییر بدیم چی میشه؟

```
var anotherPointerToName *string

fmt.Println("anotherPointerToName:", *anotherPointerToName)
```

در حین اجرا برنامه متوقف میشه و خطای runtime میده.

```
panic: runtime error: invalid memory address or nil pointer dereference
```

چون اشاره‌گر به هیچ جایی اشاره نمی‌کنه (مقدارش nil هست) و نمی‌شه ازش مقدار گرفت.

تفاوت * در هنگام تعریف اشاره گر و dereference کردن

وقتی می‌خوایم یه **اشاره‌گر تعریف کنیم**، مثل این:

```
var p *int
```

اون * کنار نوع (int) یعنی:

p یک **اشاره‌گر به نوع int** هست پس در نتیجه **آدرس یه متغیر از نوع int** رو نگه می‌داره.

اما وقتی می‌خوایم مقدار واقعی‌ای که اشاره‌گر بهش اشاره می‌کنه رو بخونیم یا تغییر بدیم، باید از * به روش زیر استفاده کنیم.

```
fmt.Println(*p)
*p = 10
```

اینجا * یعنی:

برو سراغ مقداری که آدرسش تو p هست و به مقدارش دسترسی داشته باش (مقدارو بخون یا مقدارو تغییر بده)

پس:

- * تو تعریف اشاره‌گر یعنی «**نوع داده‌ای** که اشاره‌گر بهش اشاره می‌کنه»
- * تو استفاده یعنی «**دسترسی به مقدار** داخل آدرس اشاره‌گر»

خیلی‌ها اول گیج میشن چون یه علامت ولی دو معنی متفاوت داره! ولی وقتی اینو بدونی، راحت‌تر می‌تونی کدهای اشاره‌گر رو بفهمی و بنویسی.

جمع‌بندی

- ساختار تعریف اشاره‌گر var name *TYPE
- اشاره‌گرها اول nil هستن اگر مقدار ندهیم
- برای مقدار دادن باید آدرس یه متغیر هم‌نوع بهش بدیم، با &
- p* یعنی مقدار داده شده روی آدرسی که اشاره‌گر نگه داشته
- با p* می‌تونیم مقدار اصلی رو بخونیم و تغییر بدیم
- dereference اشاره‌گر nil باعث توقف برنامه و خطای runtime میشه

لرن پث

چرا و چه زمانی از اشاره‌گر استفاده می‌کنیم؟

هدف اصلی اشاره‌گر: سرعت بیشتر و صرفه‌جویی در حافظه

خیلی وقت‌ها دلیل اصلی استفاده از اشاره‌گر اینه که برنامه سریع‌تر اجرا بشه و حافظه کمتری مصرف کنه. یعنی به جای اینکه همیشه یه نسخه کپی از داده‌ها بسازیم، مستقیم روی نسخه اصلی کار کنیم.

فرض کن متغیری داریم:

```
var warehouseTemperature float64 = 27.80
```

برای تبدیل این دما به فارنهایت، نیاز به تابع celsiusToFahrenheit داریم.

حالا دو تا راه داریم که این کار رو انجام بدیم.

روش اول: ارسال مقدار (Call by Value)

وقتی متغیر رو به تابع می‌فرستیم، در واقع یه **کپی** از مقدار اصلی ساخته میشه و به تابع میره. این رفتار پیش‌فرض Go برای خیلی از انواع مثل float64 هست.

تو این حالت:

یه زمان کمی صرف کپی گرفتن میشه.

حافظه جداگانه‌ای برای اون کپی اختصاص داده میشه.

تغییرات داخل تابع روی همون کپی انجام میشه و متغیر اصلی تغییر نمی‌کنه.

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

```
warehouseTemperatureInFahrenheit := celsiusToFahrenheit(warehouseTemperature)
fmt.Printf(
    "The warehouse temperature is %.2f°F\n",
    warehouseTemperatureInFahrenheit,
)
```

خروجی

```
The warehouse temperature is 72.50°F
```

روش دوم: ارسال آدرس با اشاره‌گر (Call by Reference)

می‌تونیم پارامتر تابع رو به جای مقدار، از نوع **اشاره‌گر به float64** تعریف کنیم.
تو این حالت باید با & آدرس متغیر رو به تابع بفرستیم چون پارامتر اشاره‌گر هست.

تو این حالت:

کپی‌برداری انجام نمیشه.

حافظه اضافی مصرف نمیشه.

اگر داخل تابع مقدار تغییر کنه، روی متغیر اصلی تغییر می‌کنه.

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

```
celsiusToFahrenheit(&warehouseTemperature)
fmt.Printf("The warehouse temperature is %.2f°F\n", warehouseTemperature)
```

خروجی

```
The warehouse temperature is 72.50°F
```

نکته مهم

تو این روش چون تغییر مستقیماً روی متغیر اصلی انجام میشه، دیگه نیازی به برگشت مقدار از تابع نیست. پس تابع مقدار برگشتی نداره (void) و مقدار جدید داخل خود متغیر اصلی ذخیره میشه.

اما یه نکته منفی هم داره!

با این کار، مقدار اصلی سلسیوس از بین میره و با مقدار فارنهایت جایگزین میشه.
یعنی دیگه دمای سلسیوس رو نداری، فقط مقدار فارنهایت باقی میمونه.

جدول مقایسه روش call by value با call by reference

روش	توضیح	مزایا	معایب
ارسال مقدار (Value)	کپی مقدار به تابع فرستاده میشه	متغیر اصلی دست‌نخورده باقی میمونه	مصرف حافظه و زمان کپی گرفتن
ارسال آدرس (Pointer)	آدرس متغیر به تابع فرستاده میشه	سریع‌تر، حافظه کمتر مصرف میشه	مقدار اصلی تغییر میکنه و از بین میره

لرن پث

آرایه‌های با طول ثابت و اشاره‌گر

مدل پیش فرض پارامتر از نوع آرایه با طول ثابت

تو Go وقتی یه آرایه **با طول ثابت** (مثل [6]float64) رو به تابع می‌فرستی، اتفاقی که میفته اینه که **کل آرایه کپی میشه** و اون کپی به تابع میره، یعنی به شکل call by value به تابع ارسال میشه.

تو این حالت:

کل داده‌ها دوباره تو حافظه ساخته میشن.

هر تغییری تو تابع انجام بدی، روی نسخه اصلی اثر نداره.

برای آرایه‌های بزرگ، این کپی‌برداری می‌تونه سرعت رو بیاره پایین و حافظه بیشتری مصرف کنه.

اثبات call by value بودن پارامتر آرایه با طول ثابت

به مثال زیر توجه کن:

```
func printAndModifyCopy(temps [6]float64) {
    fmt.Printf("inside copy: &temps = %p\n", &temps)
    fmt.Printf("inside copy: &temps[0] = %p\n", &temps[0])

    for i, t := range temps {
        fmt.Printf("warehouse %d (inside): %5.2f°C\n", i+1, t)
    }

    temps[0] = 99.99
    fmt.Println("changed temps[0] inside function to 99.99")
}
```

```
warehouseTemps := [6]float64{27.80, 34.60, 22.40, 41.60, 20.95, 25.82}
fmt.Printf("main: &warehouseTemps = %p\n", &warehouseTemps)
fmt.Printf("main: &warehouseTemps[0] = %p\n", &warehouseTemps[0])

printAndModifyCopy(warehouseTemps)

fmt.Println("after function, main sees warehouseTemps = ", warehouseTemps)
```

خروجی

```
main: &warehouseTemps = 0xc0000140a0
main: &warehouseTemps[0] = 0xc0000140a0
inside copy: &temps = 0xc0000160b0
inside copy: &temps[0] = 0xc0000160b0
warehouse 1: 27.80°C
warehouse 2: 34.60°C
warehouse 3: 22.40°C
warehouse 4: 41.60°C
warehouse 5: 20.95°C
warehouse 6: 25.82°C
changed temps[0] inside function to 99.99
after function, main sees warehouseTemps =  [27.8 34.6 22.4 41.6 20.95 25.82]
```

این کد نشون میده وقتی آرایه رو با مقدار (نه آدرس) به تابع می‌فرستی، یک کپی مستقل ساخته میشه — آدرس آرایه داخل تابع با آدرس آرایه در main فرق داره — و تغییر داخل تابع روی کپی اثر داره، نه روی اصلی.

پس به همین خاطر با وجود اینکه **داخل تابع مقدار دمای اولین سوله رو عوض کردیم ولی بیرون تابع هیچ تغییری نکرد** .چون داشتیم با کپی آرایه کار می‌کردیم.

چرا این کپی مشکل‌سازه؟

هر عنصر float64 در حافظه معمولاً **8 بایت** فضا می‌گیره.
پس برای کپی کردن یک آرایه 6 عضوی دقیقا به 48 بایت فضا نیاز داریم.

درسته که 48 بایت فضای زیادی نیست! اما اگه بزرگ باشه هزینه سنگین میشه: مثلاً برای کپی کردن آرایه با 1,000,000 عنصر از نوع float64، حدود 8 مگابایت فضا نیازه.

پس در صورتی که آرایه بزرگ باشه (به عنوان مثال 1,000,000 عنصر) با هر بار فراخوانی تابع با چنین آرایه‌ای، حدود 8 مگابایت برای کپی شدن مصرف میشه — که هم زمان‌بره هم حافظه بیشتری اشغال می‌کنه. اینجاست که باید حواسمون جمع باشه و راه حل بهتری انتخاب کنیم.

لرن پث

استفاده از اشاره‌گر

حالا بیایم همین کار رو با اشاره‌گر به آرایه انجام بدیم؛ تابع آدرس آرایه می‌گیره و مستقیم روی داده اصلی کار می‌کنه:

```
func printAndModifyPointer(temps *[6]float64) {
    fmt.Printf("inside copy: &temps = %p\n", temps)
    fmt.Printf("inside copy: &temps[0] = %p\n", &temps[0])

    for i, t := range (*temps) {
        fmt.Printf("warehouse %d (inside): %5.2f°C\n", i+1, t)
    }

    (*temps)[0] = 99.99
    fmt.Println("changed temps[0] inside function to 99.99")
}
```

```
warehouseTemps := [6]float64{27.80, 34.60, 22.40, 41.60, 20.95, 25.82}
fmt.Printf("main: &warehouseTemps = %p\n", &warehouseTemps)
fmt.Printf("main: &warehouseTemps[0] = %p\n", &warehouseTemps[0])

printAndModifyPointer(&warehouseTemps)

fmt.Println("after function, main sees warehouseTemps = ", warehouseTemps)
```

- این‌بار آدرس داخل تابع همان آدرس main است — یعنی هیچ کپی ساخته نشد.
- تغییر روی temps[0] داخل تابع، دقیقاً مقدار اصلی در main را تغییر داد.

نکتهٔ: داخل تابع وقتی پارامتر temps *[6]float64 است، تو می‌تونی مستقیم بنویسی temps[i] — نیاز نیست بنویسی (*temps)[i] — کامپایلر این رو برای خوانایی تبدیل می‌کنه.

خروجی

```
main: &warehouseTemps = 0xc0000123f0
main: &warehouseTemps[0] = 0xc0000123f0
inside copy: &temps = 0xc0000123f0
inside copy: &temps[0] = 0xc0000123f0
warehouse 1: 27.80°C
warehouse 2: 34.60°C
warehouse 3: 22.40°C
warehouse 4: 41.60°C
warehouse 5: 20.95°C
warehouse 6: 25.82°C
changed temps[0] inside function to 99.99
after function, main sees warehouseTemps =  [99.99 34.6 22.4 41.6 20.95 25.82]
```

تو این حالت:

دیگه کل آرایه کپی نمیشه.

تغییرات روی نسخه اصلی اعمال میشه.

سرعت بیشتر، حافظه کمتر مصرف میشه.

مزایا و معایب هر روش

ارسال آرایه به شکل value (پیش‌فرض):

- مزایا:
  - امن‌تر: آرایه اصلی تغییر نمی‌کنه، ایزولیشن بهتر.
  - کد ساده‌تر وقتی می‌خوای فقط بخونی.
- معایب:
  - برای آرایه‌های بزرگ هزینهٔ زمان و حافظه داره (کپی کامل).
  - اگر بخوای تغییر بدی، باید مقدار را برگردونی یا از اشاره‌گر استفاده کنی.

ارسال آرایه به شکل pointer:

مزایا:

- هیچ کپی‌ای انجام نمیشه — سریع‌تر و حافظه کمتر مصرف میشه.
- تغییرات داخل تابع روی آرایه اصلی اعمال میشه — مناسب وقتی می‌خوای mutate کنی.

- معایب:
  - کد پیچیده‌تر میشه و باید مراقب aliasing باشی (چند جا به یک داده اشاره کنند).
  - خطر تغییر ناخواستهٔ دادهٔ اصلی وجود داره؛ لازم است مفهوم side-effect رو در نظر بگیری.
  - اگر توی توابع چند سطحی از همون آرایه استفاده بشه، فهمیدن اینکه چه جاهایی آرایه رو عوض می‌کنن سخت‌تر میشه.

عمل‌کرد پیشنهادی

اگر آرایه **خیلی کوچک** و غیرحساسه و فقط می‌خوای بخونی — ارسال به صورت value اشکال نداره. (مثلاً آرایه‌هایی با طول کمتر از 10 معمولاً مشکلی ندارن)

اگر آرایه **بزرگه** یا **می‌خوای داخل تابع تغییرش بدی**، یا می‌خوای از ساختارهای پیچیده استفاده کنی — یکی از این‌ها رو بکن:

- آدرس آرایه را بفرست
- به جای استفاده از آرایه با طول ثابت از slice استفاده کن (slice رفتار reference-like داره و معمولاً راه عملی‌تره)

نکته:

وقتی آدرس می‌فرستی، تابع می‌تونه مقدار را مستقیماً تغییر بده — پس قبل از تغییر، فکر کن آیا می‌خواهی مقدار اولیه حفظ بشه یا نه.

اگر واقعاً نیاز به یک نسخهٔ جدید داری ولی نمی‌خواهی آرایه اصلی تغییر کند، خودت یک کپی بساز (مثلاً ()copy یا با حلقه) و با آن کار کن.

برای نشان دادن «من دارم مقدار را تغییر می‌دهم» بهتره تابع اسمش را طوری بگذاری که گویا باشه: مثلاً mutateTemps یا increaseTemps تا خواننده بداند تابع side-effect دارد.

جدول مقایسه روش ها

روش	رفتار	مزایا	معایب
ارسال آرایه (Value)	کپی کامل آرایه	امنیت داده اصلی	کندتر، مصرف بیشتر حافظه
ارسال آدرس آرایه (Pointer)	کار روی داده اصلی	سریع‌تر، حافظه کمتر	احتمال تغییر ناخواسته داده اصلی

آرایه‌های با طول پویا (slice) و اشاره‌گر

Slice واقعاً چی هست؟

slice یک **struct سبکه** که خودش داده‌ها رو ذخیره نمی‌کنه.
این struct فقط سه بخش اصلی داره:

```
type sliceHeader struct {
    Data uintptr // آدرس اولین عنصر در آرایه اصلی (underlying array)
    Len  int     // تعداد عناصری که slice الان داره
    Cap  int     // ظرفیت slice تا انتهای آرایه اصلی
}
```

به زبان ساده:

Data: میگه از کجا شروع کن.

Len: تعداد خونه‌هایی که فعلاً داری.

Cap: حداکثر خونه‌هایی که می‌تونی بدون ساختن آرایه جدید استفاده کنی.

مثال — دمای سوله‌های یک کارخانه

```
temperatures := []float64{27.80, 34.60, 22.40, 41.60, 20.95, 25.82}
```

پشت صحنه:

```
Slice struct in memory:
Data → آدرس عنصر index 0 از آرایه اصلی
Len  = 6
Cap  = 6

Underlying Array in memory:
[27.80, 34.60, 22.40, 41.60, 20.95, 25.82]
```

slice فقط یه پنجره‌ست که به این آرایه نگاه می‌کنه.

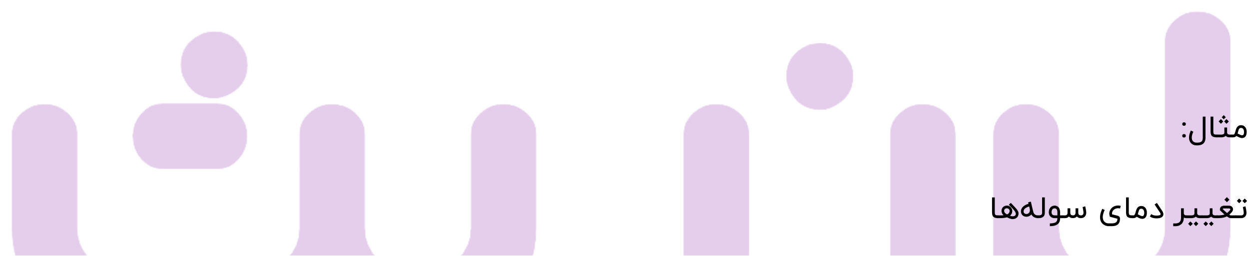
ارسال slice به تابع

وقتی slice رو به تابع میدی:

struct اون slice کپی میشه (این یعنی سه تا مقدار Data, Len, Cap کپی میشن)

ولی چون Data فقط یک pointer به آرایه اصلیه، همه کپی‌ها به همون داده وصل هستن.

یعنی تغییر محتویات slice توی تابع، داده اصلی رو تغییر میده. اما تغییر خود slice (مثل تغییر طول) فقط روی نسخه کپی اثر داره.



مثال:

تغییر دمای سوله‌ها

```
func increaseTemperatures(s []float64, delta float64) {
    for i := range s {
        s[i] += delta
    }
}

func printTemps(s []float64) {
    for i, t := range s {
        fmt.Printf("warehouse %d: %5.2f°C\n", i+1, t)
    }
}
```

```
warehouseTemps := []float64{27.80, 34.60, 22.40, 41.60, 20.95, 25.82}
increaseTemperatures(warehouseTemps, 2)
printTemps(warehouseTemps)
```

پشت صحنه:

کپی از struct اون slice ساخته شد.

هر دو struct به همون آرایه اصلی وصله.

تغییر s[i] باعث تغییر داده اصلی میشه.

خروجی

```
warehouse 1: 29.80°C
warehouse 2: 36.60°C
warehouse 3: 24.40°C
warehouse 4: 43.60°C
warehouse 5: 22.95°C
warehouse 6: 27.82°C
```

استفاده از pointer در slice

وقتی بخوای **خود slice** رو تغییر بدی، مثل:

- برش دادن(s = s[2:])
- اضافه کردن عناصر جدید (append) وقتی ظرفیت تموم شده
- جایگزینی کل slice با slice جدید

در این حالت، اگر pointer ندی، تغییرات روی کپی انجام میشه و بیرون تابع دیده نمیشه.

مثال:

تغییر ساختار (ظرفیت یا طول slice

```
func keepLastThree(s *[]float64) {
    *s = (*s)[len(*s)-3:]
}

func printTemps(s []float64) {
    for i, t := range s {
        fmt.Printf("warehouse %d: %5.2f°C\n", i+1, t)
    }
}
```

```
warehouseTemps := []float64{27.80, 34.60, 22.40, 41.60, 20.95, 25.82}
keepLastThree(&warehouseTemps)
printTemps(warehouseTemps)
```

خروجی

```
warehouse 1: 41.60°C
warehouse 2: 20.95°C
warehouse 3: 25.82°C
```

اینجا:

pointer به slice دادیم

طول و شروع slice عوض شد

بیرون تابع هم تغییر دیده میشه

جدول راهنمای استفاده از اشاره‌گر

نوع تغییر	نیاز به pointer؟	چرا؟
تغییر محتویات (s[i] = ...)	ندارد	Data یکسانه، تغییر روی داده اصلیه
تغییر طول (s = s[2:])	دارد	Len و Data عوض میشه، باید روی اصل struct اعمال شه
اضافه کردن (append)	دارد (وقتی cap پره)	ممکنه آرایه جدید ساخته بشه و Data عوض شه

Map ها و اشاره‌گر

در زبان Go وقتی یک **map** رو به تابع می‌فرستیم، این اتفاق میفته:

کپی کل داده‌ها ساخته نمی‌شه — فقط یک reference (آدرس) به map ارسال میشه.

تغییرات داخل تابع روی **همون map اصلی** اعمال میشه.

این کار باعث **سرعت بیشتر** و **مصرف حافظه کمتر** میشه.

ارسال map به تابع و تغییر مستقیم داده‌ها

فرض کنیم دمای سوله‌های مختلف کارخانه رو به شکل زیر داریم:

```
warehouseTemps := map[string]float64{
    "North-A": 27.80,
    "North-B": 34.60,
    "North-C": 22.40,
    "South-A": 41.60,
    "South-B": 20.95,
    "South-C": 25.82,
}
```

حالا میخوایم تابعی بنویسیم که همه دماها رو به فارنهایت تبدیل کنه.

```
func convertAllTempsToFahrenheit(temps map[string]float64) {
    for name, temp := range temps {
        temps[name] = (temp * 1.8) + 32
    }
}
```

وقتی این تابع رو اجرا کنیم:

```
convertAllTempsToFahrenheit(warehouseTemps)

fmt.Println("Updated warehouse temps (°F):", warehouseTemps)
```

خروجی میشه این:

```
Updated warehouse temps (°F): map[North-A:82.04 North-B:94.28 North-C:72.32 South-A:106.88 South-B:69.71 South-C:78.476]
```

همونطور که دیدی، تابع بدون برگردوندن مقدار، **مستقیم** map اصلی رو تغییر داد. دلیلش اینه که **reference type** هست.

مشکل این روش

این رفتار برای سرعت خوبه، ولی خطر داره!

اگه اشتباهی داخل تابع داده‌ها رو تغییر بدیم، داده اصلی رو از دست میدیم و ممکنه برنامه‌مون خراب بشه.

جلوگیری از تغییر ناخواسته با کپی گرفتن از map

برای این کار می‌تونیم یک تابع بنویسیم که کپی کامل map رو بسازه:

```
func copyWarehouseTempsMap(original map[string]float64) map[string]float64 {
    copied := make(map[string]float64)
    for name, temp := range original {
        copied[name] = temp
    }
    return copied
}
```

حالا به جای ارسال map اصلی، یک نسخه کپی رو می‌فرستیم:

```
warehouseTempsCopy := copyWarehouseTempsMap(warehouseTemps)

convertAllTempsToFahrenheit(warehouseTempsCopy)


fmt.Println("Original warehouse temps (°C):", warehouseTemps)

fmt.Println("Modified copy temps (°F):", warehouseTempsCopy)
```

خروجی

```
Original warehouse temps (°C): map[North-A:27.80 North-B:34.60 North-C:22.40 South-A:41.60 South-B:20.95 South-C:25.82]

Modified copy temps (°F): map[North-A:82.04 North-B:94.28 North-C:72.32 South-A:106.88 South-B:69.71 South-C:78.476]
```

این بار داده اصلی تغییر نکرد، چون با یک کپی کار کردیم.

آیا اشاره‌گر به map کاربرد داره؟

بیشتر وقت‌ها لازم نیست، چون خود map یک reference هست.

فقط وقتی نیاز داری **کل map رو جایگزین کنی** (نه فقط تغییر مقادیرش) ممکنه از اشاره‌گر به map استفاده کنی.

مثلا بخوای داخل تابع یک map جدید بسازی و آدرسش رو به بیرون بدی.

جدول راهنمای استفاده از map یا کپی map

وضعیت	روش پیشنهادی
تغییر مستقیم داده‌ها	ارسال map به تابع
جلوگیری از تغییر ناخواسته	ارسال کپی map
ساخت map جدید داخل تابع	استفاده از pointer به map
داده‌ها کوچک و تغییرات کم	مهم نیست، هردو روش ممکنه
داده‌ها بزرگ و تغییرات زیاد	ارسال مستقیم map (برای سرعت)

لرن پث

Struct ها و اشاره‌گر

رفتار پیش‌فرض ارسال Struct به تابع

در زبان Go ، وقتی یک struct رو به تابع می‌فرستی، **به‌صورت پیش‌فرض Call by Value** ارسال میشه.

تو این حالت:

یک **کپی کامل** از struct ساخته میشه و به تابع داده میشه.

تغییراتی که داخل تابع روی struct انجام میدی **تأثیری روی نسخه اصلی** خارج از تابع نداره.

دلیل این رفتار:

جلوگیری از تغییر ناخواسته داده‌ها.

ساده‌تر شدن درک و پیش‌بینی رفتار کد.

ساخت Struct سوله (Warehouse)

```
type Warehouse struct {
    Name        string
    Temperature float64
}
```

یه متغیر از این نوع تعریف میکنیم برای نگهداری اطلاعات سوله North-A

```
northAWarehouse := Warehouse {
    Name: “North-A”,
    Temperature: 27.80,
}
```

یه تابع هم برای تبدیل دما از سلسیوس به فارنهایت داریم:

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

ارسال Struct به صورت Value (بدون pointer)

یک تابع داریم که یه ورودی از نوع Warehouse میگیره و دماش رو از سلسیوس به فارنهایت تبدیل می‌کنه:

```
func convertToFahrenheit(w Warehouse) {
    w.Temperature = celsiusToFahrenheit(w.Temperature)
}
```

و اینطور استفاده می‌کنیم:

```
fmt.Println("Before:", northAWarehouse)
convertToFahrenheit(northAWarehouse)
fmt.Println("After:", northAWarehouse)
```

خروجی

```
Before: {North-A 27.8}
After: {North-A 27.8}
```

دما **هیچ تغییری نکرده**! چون w داخل تابع یک کپی از struct اصلیه. تغییرات فقط روی این کپی انجام شده و نسخه اصلی خارج از تابع دست‌نخورده مونده.

ارسال Struct به صورت Reference (با pointer)

حالا بیایم از pointer استفاده کنیم:

```
func convertToFahrenheitPtr(w *Warehouse) {
    w.Temperature = celsiusToFahrenheit(w.Temperature)
}
```

```
fmt.Println("Before:", northAWarehouse)
convertToFahrenheit(&northAWarehouse)
fmt.Println("After:", northAWarehouse)
```

خروجی

```
Before: {North-A 27.8}
After: {North-A 82.03999999999999}
```

دما تغییر کرد!

چون این بار به جای کپی struct **آدرس struct اصلی** رو دادیم به تابع و تابع مستقیماً روی داده اصلی تغییر انجام داد.

جدول مقایسه روش ها

روش	مزایا	معایب
Call by Value	تغییرات ناخواسته نداریم، کد ساده‌تر	اگه struct بزرگ باشه، کپی‌کردن سنگین میشه
Call by Reference (pointer)	تغییر مستقیم، سرعت بیشتر برای struct بزرگ	خطر تغییر ناخواسته داده اصلی، نیاز به دقت بیشتر

استراتژی تصمیم گیری

struct کوچک (چند فیلد ساده):

بهتره value بدی، مگر اینکه بخوای تغییرش بدی.

struct بزرگ (فیلدهای زیاد یا فیلدهایی که reference هستن مثل slice ، map و ...):

pointer بده تا هم حافظه کمتر مصرف بشه و هم سرعت بیشتر باشه.

تفاوت متد به روش value receiver با pointer receiver

تعریف متد روی struct

تو Go ما می‌تونیم متد تعریف کنیم که روی یک struct کار کنه. یعنی می‌تونیم عملکرد و رفتارهایی به داده‌های struct اضافه کنیم.

متدها باید مشخص کنن که **receiver** شون چطوری به struct اشاره می‌کنه:

یا با مقدار (value receiver)

یا با اشاره‌گر (pointer receiver)

ساختار کلی متد:

```
func (receiver Type) methodName() {
    // statement
}
```

یا

```
func (receiver *Type) methodName() {
    // statement
}
```

تفاوت receiver ها

Value receiver یعنی اینکه وقتی متد رو صدا می‌زنی، struct به صورت کپی کامل به متد ارسال میشه و متد روی همون کپی کار می‌کنه.

Pointer receiver یعنی به جای کپی، آدرس struct به متد فرستاده میشه و متد می‌تونه مستقیم روی struct اصلی تغییر اعمال کنه.

```
func (w Warehouse) convertToFahrenheit() {
    w.Temperature = celsiusToFahrenheit(w.Temperature)
}

func (w *Warehouse) convertToFahrenheitPtr() {
    w.Temperature = celsiusToFahrenheit(w.Temperature)
}
```

هر دو متد تقریبا عملکردشون شبیه به همدیگست تنها تفاوتشون تو نحوه دریافت w هست، اولی به صورت value و دومی به صورت pointer دریافت میکنه.

حالا بیایم تست کنیم

```
northAWarehouse := Warehouse{"North-A", 27.80}


fmt.Printf("Before value receiver: %.2f°C\n", northAWarehouse.Temperature)
northAWarehouse.convertToFahrenheit()
fmt.Printf("After value receiver: %.2f°C\n\n", northAWarehouse.Temperature)


fmt.Printf("Before pointer receiver: %.2f°C\n", northAWarehouse.Temperature)
northAWarehouse.convertToFahrenheitPtr()
fmt.Printf("After pointer receiver: %.2f°F\n", northAWarehouse.Temperature)
```

```
Before value receiver: 27.80°C
After value receiver: 27.80°C


Before pointer receiver: 27.80°C
After pointer receiver: 82.04°F
```

وقتی متد convertToFahrenheit با **value receiver** فراخوانی شد، در واقع یک کپی از northAWarehouse ساخته شد. تغییر دما روی این کپی انجام شد.

بنابراین در بیرون از تابع، مقدار اصلی northAWarehouse.Temperature **بدون تغییر باقی موند**.

اما وقتی متد convertToFahrenheitPtr با **pointer receiver** فراخوانی شد،
آدرس northAWarehouse به متد فرستاده شد و تغییر دما مستقیماً روی داده اصلی اعمال شد.

پس بعد از اجرای این متد، دمای northAWarehouse واقعاً تبدیل به فارنهایت شد.

رفتار خاص کامپایلر Go در اجرای متدها

حالا نکته جالب ماجرا اینه که Go خیلی انعطاف‌پذیر رفتار می‌کنه:

حالت 1: متغیر معمولی (T)

متدهای Value receiver به صورت مستقیم اجرا میشن.

متدهای Pointer receiver انتظار آدرس دارن، پس کامپایلر خودش آدرس متغیر رو می‌گیره و به متد می‌ده (بدون اینکه خودت & بزنی)

حالت 2: متغیر اشاره‌گر (T*)

متدهای Pointer receiver به صورت مستقیم اجرا میشن.

متدهای Value receiver انتظار یک کپی از struct رو دارن، پس کامپایلر یک کپی از struct رو می‌سازه و به متد می‌ده.

یعنی از نظر «امکان اجرا» هر دو مدل روی هر دو نوع متغیر کار می‌کنن، چون کامپایلر تبدیل لازم رو خودش انجام می‌ده.

نکته مهم

در عمل، اگه حتی یکی از متدهای یک struct رو به صورت pointer receiver بنویسی، بهتره بقیه متدها رو هم pointer receiver کنی تا طراحی یکدست و بدون دردسر داشته باشی.

چرا توصیه می‌کنن همه متدها رو یکدست pointer receiver کنیم؟

این دیگه بحث فنی صرف نیست، بحث طراحی تمیز و آینده‌نگریه.

جلوگیری از سردرگمی

وقتی بعضی متدها pointer باشن و بعضی value، هر کسی کد رو می‌خونه باید مدام حواسش باشه:

"آیا این متد تغییرات رو روی نسخه اصلی اعمال می‌کنه یا فقط روی یک کپی؟"

کارایی و هزینه کپی

اگر struct بزرگ باشه (مثلاً چند ده فیلد یا داده‌های سنگین مثل slice، map و...) باعث میشه هر بار یک کپی کامل ساخته بشه. این هم حافظه می‌خوره، هم سرعت رو پایین میاره. اما Pointer receiver فقط یک آدرس کوچک رد و بدل می‌کنه.

سازگاری باInterface ها

در Go مجموعه متدها تعیین می‌کنه که یک نوع می‌تونه یک interface رو implement کنه یا نه. ترکیب pointer و value ممکنه باعث بشه ناخواسته یک interface رو از دست بدی، مخصوصاً موقع refactor.

راحتی ذهنی

وقتی همه متدها pointer باشن:

همیشه تغییرات روی داده اصلی انجام میشه.

هزینه کپی وجود نداره.

لازم نیست فکر کنی "این متد pointer بود یا value؟"

جمع‌بندی

از نظر فنی:

Pointer receiver روی متغیر معمولی هم کار می‌کنه (با گرفتن آدرس خودکار).

Value receiver روی متغیر اشاره‌گر هم کار می‌کنه (با گرفتن کپی خودکار).

اما برای **وضوح، کارایی، پیش‌بینی‌پذیری و راحتی طراحی**، اگر حتی یک متد pointer receiver داری، بهتره بقیه رو هم pointer receiver کنی.

لرن پث

تمرین 1: محاسبه فاکتوریل

فرض کن می‌خوایم فاکتوریل یک عدد رو حساب کنیم. اما نه اینکه نتیجه رو برگردونیم و بیرون ذخیره کنیم، اینجا می‌خوایم مستقیم روی همون متغیر ورودی تغییر بدیم.

توضیح سناریو:

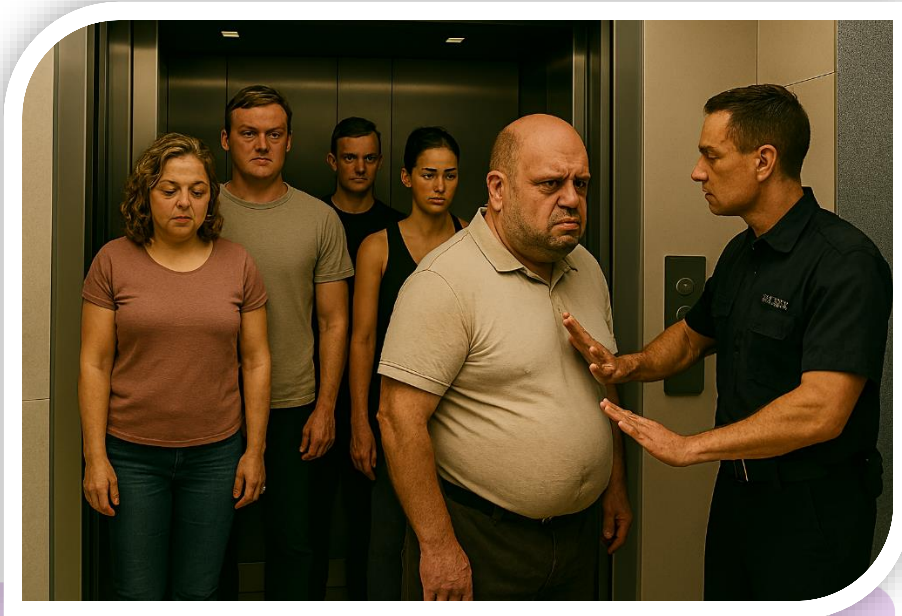
ورودی یک عدد int هست.

تابع هیچ خروجی نداره.

نتیجه‌ی فاکتوریل باید روی همون ورودی نوشته بشه.

لرن پث

تمرین 2: بررسی وزن پیش از استفاده از آسانسور



فرض کن در ورودی آسانسور یک ترازو گذاشتیم که وزن هر فرد رو **به واحد پوند** اندازه می‌گیره. آسانسور ظرفیتش 5 نفره، پس نهایتاً 5 تا وزن رو نگه می‌داریم.

کاری که باید بکنی:

وزن‌ها رو در یک آرایه ذخیره کن.

یک تابع بنویس که این آرایه رو بگیره و همه‌ی وزن‌ها رو با کمک تابع poundToKg به **کیلوگرم** تبدیل کنه.

از **pointer به آرایه** استفاده کن تا تغییرات مستقیم روی نسخه اصلی اعمال بشه.

در آخر هم تو خروجی برنامه توضیح بده که استفاده از pointer در این حالت چه **مزایا** و چه **معایبی** داشت.

تمرین 3: فروش ویژه محصولات

سناریو:

برنامه از ورودی قیمت کالاها (float64) رو می‌گیره.

وقتی کاربر حرف s رو زد، گرفتن ورودی متوقف میشه.

تمام قیمت‌ها رو در یک slice ذخیره کن.

بخش اول:

یک تابع بنویس که قیمت و میزان تخفیف (به درصد) رو می‌گیره و قیمت جدید با تخفیف رو برمی‌گردونه.

بخش دوم:

یک تابع جدید بنویس که قیمت‌های بالای **20 دلار** رو از **همون slice اصلی** حذف کنه.
این تابع باید با **pointer به slice** کار کنه تا تغییرات روی نسخه اصلی اعمال بشه.

نکته:

قبل و بعد از هر مرحله، لیست قیمت‌ها رو چاپ کن که تغییرات دیده بشه.

تمرین 4: پیش‌بینی جمعیت

فرض کن یک بیماری مسری در چند شهر شیوع پیدا کرده.

```
citiesPopulation := map[string]int{
    "London":    9000000,
    "Paris":     2140000,
    "Dubai":     3300000,
    "NewYork":   8400000,
    "Tehran":    9000000,
    "Muscat":    1300000,
    "Antalya":   1200000,
    "Rome":      2800000,
    "Berlin":    3700000,
}
```

جمعیت هر شهر رو در یک map[string]int ذخیره کردیم.

یک تابع کمکی داریم که با گرفتن جمعیت اولیه، تعداد تقریبی قربانیان رو برمی‌گردونه.

```
func calculateVictims(pop int) int {
    return pop / 10
}
```

کاری که باید بکنی:

یک تابع بنویس که با استفاده از این تابع کمکی، جمعیت هر شهر رو بعد از بیماری محاسبه کنه.

اول این کار رو طوری انجام بده که **روی map اصلی تغییر کنه**.

بعد برنامه رو جوری بازنویسی کن که **نسخه اصلی map تغییر نکنه** و فقط روی کپی محاسبه انجام بشه.

لرن پث

تمرین 5: عملکرد کلاس درس

یک struct به اسم Course داریم که این شکلیه:

```go
type Course struct {
    Name      string
    Teacher   string
    Year      int
    Grades    []float64
}
```

مراحل کار:

یک متد بنویس (به روش value receiver) که تعداد مردودی‌ها (نمره < 10) رو برگردونه.

یک متد دیگه (به روش value receiver) که میانگین نمرات رو حساب کنه.

فرض کن همین معلم، همین درس رو در همین سال تحصیلی دوباره ارائه داده و دو تا Course داریم.
حالا یک متد بنویس که این دو درس رو **ادغام** کنه، طوری که لیست نمرات‌شون با هم ترکیب بشه.

بگو این متد ادغام رو به روش value receiver می‌نویسی یا pointer receiver ؟ چرا؟

لرن پث