عملگرهای ترکیبی در زبان Go

فرض کن می‌خوای وزن یک نفر رو ذخیره کنی و بعد چند کیلو بهش اضافه کنی:

```
weight := 70
weight = weight + 5
```

این یعنی:

وزن قبلی رو بگیر، 5 کیلو بهش اضافه کن، و دوباره بریزش تو همون متغیر weight

چون این کار خیلی رایجه، زبان Go یه روش کوتاه‌تر برای انجامش داره:

```
weight := 70
weight += 5
```

این همون کار بالا رو می‌کنه، فقط کوتاه‌تر و تمیزتره. به این می‌گیم **عملگر ترکیبی**.

تعریف عملگر ترکیبی

عملگر ترکیبی یعنی یه عملیات (مثل جمع یا ضرب) رو **با مقداردهی به خود متغیر** ترکیب می‌کنیم.

چند مثال ساده

کد	توضیح
x += 1	x = x + 1
x -= 2	x = x – 2
x *= 3	x = x * 3
x /= 2	x = x / 2

نکته:

- این عملگرها فقط روی **متغیرها** کار می‌کنن، نه روی ثابت‌ها یا مقادیر مستقیم.
- مقدار سمت راست باید **هم‌نوع** با متغیر باشه، مثلاً نمی‌تونی int += float64 انجام بدی

جدول عملگرهای ترکیبی

عملگر ترکیبی	معادل	توضیح	نوع داده قابل استفاده
+=	x = x + y	جمع مقدار جدید با مقدار قبلی	int, float, string
-=	x = x – y	کم‌کردن مقدار جدید	int, float
*=	x = x * y	ضرب	int, float
/=	x = x / y	تقسیم	int , float
%=	x = x % y	باقیمانده تقسیم	int

مثال

```
package main

import "fmt"

func main() {
    score := 10
    score += 5 // score 15 حالا میشه
    fmt.Println(“score is: ”, score)

    greeting := "Hello"
    greeting += " World" // greeting = "Hello World"
    fmt.Println(“greeting is: ”, greeting)

    health := 100
    health -= 30 // حالا health 70 میشه
    fmt.Println(“health is: ”, health)

    price := 2000
    price *= 3 // حالا price 6000 میشه
    fmt.Println(“price is: ”, price)

    total := 100
    total /= 4 // total = 25
    fmt.Println(“total is: ”, total)

    x := 17
    x %= 5 // x = 2
    fmt.Println(“x is: ”, x)
}
```

عملگرهای افزایشی و کاهشی

گاهی نیاز هست که از یک متغیر از نوع عدد صحیح یکی کم یا یکی زیاد کنیم، اگه بخوایم کمی هوشمندانه عمل کنیم احتمالا از عملگرهای ترکیبی به شکل زیر استفاده میکنیم

```
package main

import "fmt"

func main() {
    candles := 10
    candles += 1
    fmt.Println(“The number of candles are: ”, candles)
}
```

در زبان GO دو عملگر افزایشی و کاهشی برای این موضوع تعریف شده که حتی نوشتن این شکل از کدهارو برامون ساده تر هم کرده. مثلا در همین مثال چون فقط قصد داشتیم مقدار متغیر candleOnTheCake رو فقط یکی افزایش بدیم میتونستیم از عملگر افزایشی به شکل زیر استفاده کنیم

```
package main

import "fmt"

func main() {
    candles := 10
    candles++ // معادلcandles += 1
    fmt.Println(“The number of candles are: ”, candles)
}
```

نکته: عملگر افزایشی و کاهشی تنها بر روی متغیر های نوع صحیح قابل استفادست. یعنی انواع زیر

int8 uint8 int16 uint16 int32 uint32 int64 uint64 int uint byte rune

انواع داده‌ها و انواع زیرمجموعه آنها

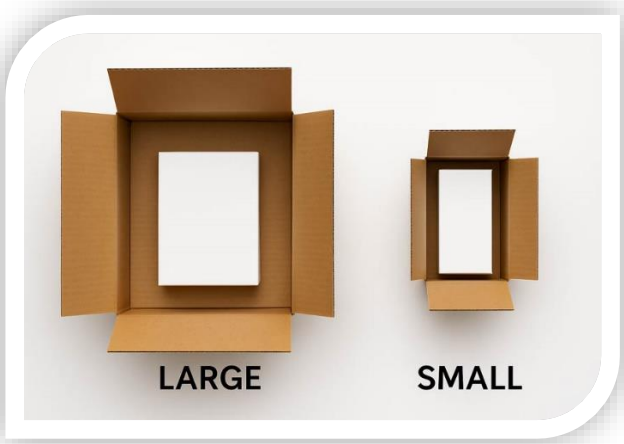
زبانGo ، علاوه بر انواع داده‌ی اصلی مثل int، float و string برای بعضی از این انواع داده مثل عدد صحیح (int) و عدد اعشاری (float)، انواع **زیرمجموعه** (subtypes) با اندازه‌های مختلف و ویژگی‌های خاص داریم.

این کار به ما اجازه می‌دهد تا بتوانیم متناسب با نیاز خود، بهترین نوع داده را انتخاب کنیم و برنامه‌ای بهینه‌تر از نظر مصرف حافظه و عملکرد بنویسیم.

یک مثال ساده

فرض کن یه انبار داری که فقط یه مقدار مشخص جا داره. حالا می‌خوای توی این انبار، یه سری جعبه (متغیر) بذاری.

- اگه هر جعبه خیلی بزرگ باشه ولی فقط یه چیز کوچیک توشه، کلی جا تلف می‌شه!
- اگه هر جعبه درست به اندازه محتواش باشه، انبارت کلی جا برای بقیه چیزها داره و بهینه‌ست.



چرا انواع زیرمجموعه (مثل int8، uint16، float32) داریم؟

هر متغیر در برنامه‌نویسی یک **هزینه** دارد، یعنی:

- **فضایی که در حافظه‌ی RAM اشغال می‌کند**

از آنجا که حافظه کامپیوتر ظرفیت محدودی دارد، مهم است که:

- **نوع داده‌ی متغیر را طوری انتخاب کنیم که متناسب با مقداری باشد که قرار است در آن ذخیره کنیم.**

این انتخاب مناسب باعث می‌شود:

- برنامه فضای کمتری اشغال کند
- سرعت اجرای برنامه بهبود پیدا کند
- امکان اجرای برنامه روی سخت‌افزارهای ضعیف‌تر فراهم شود

انواع داده‌ی عددی در Go و تفاوت‌هایشان

انواع صحیح (Integer Types)

در Go انواع مختلفی از اعداد صحیح وجود دارد که از نظر **اندازه حافظه** و **محدوده‌ی اعداد قابل ذخیره** با هم تفاوت دارند:

- int8 عدد صحیح 8 بیتی
- uint8 عدد صحیح 8 بیتی بدون علامت
- int16 عدد صحیح 16 بیتی
- uint16 عدد صحیح 16 بیتی بدون علامت
- int32 عدد صحیح 32 بیتی
- uint32 عدد صحیح 32 بیتی بدون علامت
- int64 عدد صحیح 64 بیتی
- uint64 عدد صحیح 64 بیتی بدون علامت

انواع اعشاری (Floating Point Types)

برای اعداد اعشاری (اعدادی که ممکن است قسمت کسری داشته باشند) Go دو نوع اصلی دارد:

- float32 عدد اعشاری 32 بیتی، دقت تقریباً 7 رقم اعشار
- float64 عدد اعشاری 64 بیتی، دقت تقریباً 15 رقم اعشار

جدول مقایسه‌ای انواع عددی درGo

نوع داده	علامت دار؟	حافظه اشغالی	بازه عددی قابل نگهداری
int8	بله	1 بایت	-128 -> 127
uint8	نه	1 بایت	0 -> 255
int16	بله	2 بایت	-32,768 -> 32,767
uint16	نه	2 بایت	0 -> 65,535
int32	بله	4 بایت	-2,147,483,648 -> 2,147,483,647
uint32	نه	4 بایت	0 -> 4,294,967,295
int64	بله	8 بایت	-9,223,372,036,854,775,808 -> 9,223,372,036,854,775,807
uint64	نه	8 بایت	0 -> 18,446,744,073,709,551,615
float32	بله	4 بایت	$\pm 3.4028235 * 10^{38}$
float64	بله	8 بایت	$\pm 1.7976931348623157 * 10^{308}$

مثال‌های کاربردی و ساده

```
var age uint8 = 25        // uint8 سن، عدد کوچیک و مثبت، مناسب
var temperature int16 = -5  // int16 دما، عدد منفی و مثبت، نیاز به
var population int64 = 85000000 //  نیاز داره int64جمعیت کشور، عدد خیلی بزرگ،
var pi float32 = 3.14      // عدد اعشاری با دقت متوسط
```

جمع‌بندی

- هر متغیر فضای مشخصی در حافظه اشغال می‌کند.
- برای استفاده بهینه از حافظه، باید اندازه‌ی متغیر را متناسب با مقداری که قرار است ذخیره کند انتخاب کنیم.
- اگر مقدار متغیر کوچک است، از نوع داده‌ی کوچک‌تر استفاده کن تا فضای کمتری اشغال شود.
- اگر مقدار می‌تواند منفی باشد، باید نوع داده علامت‌دار (int) استفاده شود.
- اگر مقدار فقط مثبت است، نوع داده بدون علامت (uint) بهینه‌تر است.
- برای اعداد اعشاری، بین دقت و حجم حافظه باید تعادل برقرار کرد.

دلیل وجود type های int و uint

از اونجایی که هنوز هم دستگاه‌هایی با معماری سخت‌افزاری 32 بیت وجود دارند، در نتیجه سیستم‌عامل‌هایی هم هست که مخصوص همان سخت‌افزارها ساخته شده‌اند. یعنی وقتی برنامه می‌نویسیم، باید به این فکر کنیم که نرم‌افزار ما ممکن است روی سخت‌افزار 32 بیت یا 64 بیت اجرا شود.

اما ما می‌خواهیم هنگام برنامه‌نویسی، فارغ از اینکه نرم‌افزار روی چه معماری‌ای (32 یا 64 بیت) اجرا می‌شود، بتوانیم به سادگی کد بنویسیم. به همین دلیل نوع داده‌ای مثل intوجود دارد که یعنی یک عدد صحیح، ولی اندازه دقیق آن (32 بیت یا 64 بیت) در زمان کامپایل مشخص می‌شود.

دلیل وجود int بدون پسوند چیست؟

وقتی ما در زبان برنامه‌نویسی از نوع int استفاده می‌کنیم، یعنی عدد صحیح با اندازه‌ای که متناسب با معماری هدف است. یعنی:

- اگر برنامه روی معماری 32 بیت کامپایل شود، intبه صورت int32(عدد صحیح 32 بیتی) در می‌آید.
- اگر روی معماری 64 بیت کامپایل شود، intبه صورت int64 (عدد صحیح 64 بیتی) تبدیل می‌شود.

این باعث می‌شود کد ما قابل انتقال بین معماری‌ها باشد و نیاز نباشد در هر جایی مشخص کنیم که حتماً عدد 32 یا 64 بیتی است.

حالا اگر خودمان همه جا از int64 استفاده کنیم چه می‌شود؟

فرض کنیم به جای int، همه متغیرهای عدد صحیح را int64 تعریف کنیم:

- **مزایا:**
  - شما همیشه مطمئن هستید که اندازه عدد 64 بیت است، فرقی نمی‌کند برنامه روی چه معماری‌ای اجرا شود.
  - برای برنامه‌هایی که نیاز به عددهای خیلی بزرگ یا حافظه زیاد دارند، int64 ضروری است.
- **معایب:**
  - اگر برنامه روی معماری 32 بیت اجرا شود، استفاده از int64 باعث می‌شود عملیات روی این اعداد کندتر شود، چون پردازنده 32 بیت باید عدد 64 بیتی را شبیه‌سازی کند و این بهینه نیست.
  - مصرف حافظه بیشتر می‌شود، مخصوصاً وقتی تعداد زیادی عدد کوچک دارید که می‌توانست با int32 بهینه‌تر ذخیره شود.

نکته: همین دلیل مجدد وجود uint را توجیح میکند

جدول راهنما

ویژگی	int یا uint	int64 یا uint64
سایز	به معماری CPU بستگی داره	64 بیت
پلتفرم 32 بیتی	32 بیت	64 بیت
پلتفرم 64 بیتی	64 بیت	64 بیت
قابل پیش‌بینی؟	نه	بله

مثال

```
package main

import "fmt"

func main() {
	// این عدد بسته به معماری سیستم می‌تونه باشه int64 یا int32
	var a int = 100

	// این عدد همیشه دقیقاً int64 هست
	var b int64 = 200

	fmt.Println("a:", a)
	fmt.Println("b:", b)

	// اگر بخوایم a و b رو با هم جمع کنیم، باید تبدیل انجام بدیم
	sum := int64(a) + b

	fmt.Println("sum:", sum)
}
```

a از نوع int هست و ممکنه روی سیستم 32 بیت به int32 تبدیل بشه یا روی سیستم 64 بیت به int64.

b به‌طور مشخص int64 تعریف شده.

چون Go روی عملیات بین انواع ناسازگار حساسه، برای جمع زدن a و b مجبوریم a رو به int64 تبدیل کنیم int64(a)

این نشون می‌ده که int نوعی انعطاف‌پذیر برای معماریه ولی اگه دقت یا اندازه دقیق نیاز داریم، int64 رو مشخص می‌کنیم.

دلیل وجود نداشتن تایپ float

برخلاف int که معماری سخت‌افزار اندازه آن را مشخص می‌کند و می‌تواند 32 یا 64 بیت باشد، در مورد اعداد اعشاری معماری سخت‌افزار معمولاً تعیین‌کننده اندازه دقیق نیست .استاندارد IEEE 754 که برای نمایش اعداد اعشاری استفاده می‌شود، دقیقاً تعریف می‌کند که چطور است و float64 چطور است و بیشتر زبان‌ها به صورت صریح این دو نوع رو معرفی می‌کنند.

نتیجه گیری

وقتی می‌خواهیم عدد اعشاری تعریف کنیم، باید خودمان مشخص کنیم که می‌خواهیم دقت 32 بیت یا 64 بیت باشد. یعنی:

- اگر نیاز به دقت کمتر و مصرف حافظه کمتر داریم، از float32 استفاده می‌کنیم.
- اگر دقت بالا و دامنه بزرگتری می‌خواهیم، از float64 استفاده می‌کنیم.

سرریز شدن متغیر (Overflow) یعنی چی؟

سرریز یعنی یه مقدار عددی از ظرفیت نوع داده خارج بشه .مثلاً وقتی متغیری از نوع uint8 داریم که فقط می‌تونه عددی بین 0تا 255 رو نگه داره، اگه بهش مقدار 256 بدیم، دیگه نمی‌تونه نگهش داره و یا خطا می‌گیریم یا عدد می‌پره به اول بازه (wrap-around)

دو حالت داریم

1-هنگام تعریف و مقداردهی اولیه

در این حالت، کامپایلر Go خطا می‌ده و برنامه حتی اجرا هم نمی‌شه:

```
var x uint8 = 300 // ✕ خطای کامپایل
```

خطا

```
constant 300 overflows uint8
```

یعنی Go از همون اول جلوی اشتباه رو می‌گیره.

2-هنگام انجام **محاسبه در زمان اجرا**

در این حالت، اگر محاسبه‌ای انجام بدی که نتیجه‌اش از بازه‌ی نوع داده خارج بشه، GO برنامه رو اجرا می‌کنه ولی مقدار نهایی اشتباه میشه (سرریز اتفاق می‌افته).

مثال 1

```
var x uint8 = 255
x += 1       // سرریز میشه
fmt.Println(x) // خروجی: 0
```

uint8 فقط تا 255 جا داره. وقتی 1 بهش اضافه کنی، می‌پره به 0 (wrap-around)

مثال 2

```
var a int8 = 127
a += 1
fmt.Println(a) // خروجی: -128 (سرریز شده)
```

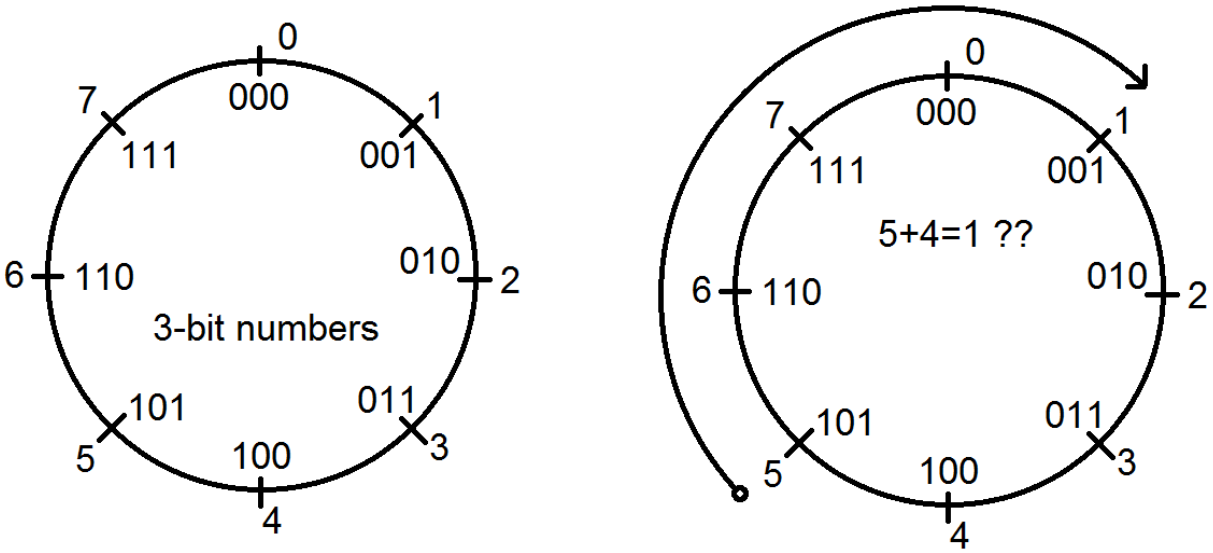
int8 بازه‌ای بین -128 تا +127 داره. عدد 128 رو نمی‌تونه نگه داره، پس می‌پره به -128

چطور جلوی سرریز رو بگیریم؟

نوع داده‌ی بزرگ‌تر انتخاب کن

```
var x uint16 = 255
x += 1
fmt.Println(x) // خروجی: 256
```

مفهوم Wrap-Around



در این تصویر، ما با **3بیت** کار می‌کنیم، یعنی می‌تونیم اعدادی بین 0 تا 7 داشته باشیم.

- **حداکثر مقدار** 7: یعنی 111 در مبنای باینری
- **حداقل مقدار** 0: یعنی 000 در مبنای باینری

تصویر یک دایره رو نشون می‌ده که:

- روی اون اعداد 7 → ... → 2 → 1 → 0 به صورت ساعت‌گرد چیده شدن
- وقتی از 7 یک واحد جلوتر می‌ریم به 0 برمی‌گردیم
- اگه از 0 یک واحد کم کنیم به 7 برمی‌گردیم

این یعنی مقدار عددی وقتی از انتهای بازه رد می‌شه wrap می‌شه (می‌پیچه) و به ابتدا یا انتهای بازه برمی‌گرده.

تبدیل از نوع بزرگ‌تر به کوچک‌تر

وقتی می‌خوای یه عدد رو از یک نوع داده با اندازه بزرگ‌تر (مثلاً uint16) به یک نوع کوچکتر (مثل uint8) تبدیل کنی، باید حواست باشه که ممکنه بخشی از داده‌ات از بین بره یا نتیجه اشتباهی بگیری. به این اتفاق می‌گن **overflow** یا **data loss**

چرا این اتفاق می‌افته؟

چون uint8 فقط می‌تونه عددهایی بین 0 تا 255 رو نگه داره (8 بیت)
ولی uint16 می‌تونه عددهایی بین 0 تا 65535 رو نگه داره (16 بیت)

پس اگر عددی بزرگ‌تر از 255 توی uint16 داشته باشی و اون رو به uint8 تبدیل کنی، فقط 8 بیت پایینی عدد نگه داشته می‌شن و بقیه بریده می‌شن .این یعنی عددت عوض می‌شه!

```
package main

import "fmt"

func main() {
	var bigNumber uint16 = 1000

	// تبدیل به uint8 — فقط 8 بیت پایین عدد باقی می‌مونه
	var smallNumber uint8 = uint8(bigNumber)

	fmt.Println("original (uint16):", bigNumber)   // 1000
	fmt.Println("converted (uint8):", smallNumber) // 232 ☜ اشتباه!
}
```

قبل از اینکه متغیری رو به نوع کوچکتر تبدیل کنی:

- بررسی کن که آیا عدد داخل محدوده نوع جدید هست یا نه.
- مثلاً برای تبدیل به uint8 مطمئن شو عددت کوچکتر یا مساوی 255 باشه.

```
if bigNumber <= 255 {
    smallNumber := uint8(bigNumber)
    fmt.Println("Safe conversion:", smallNumber)
} else {
    fmt.Println("Warning: value too big for uint8!")
}
```

نکته: ساختار های شرطی مثل if را به زودی بررسی میکنیم.

تبدیل انواع داده به یکدیگر

گاهی اوقات نیاز هست عملیات محاسباتی روی دو متغیر با دو نوع مختلف انجام بدیم (مثلا float64 و int) در این صورت در زبان GO مجبور هستیم که هر دو متغیر را به یک نوع مشترک تبدیل کنیم تا بتوانیم محاسبات را انجام بدیم.

حالا که با انواع داده‌های عددی (مثل int, int64, uint8, float64) آشنا شدیم، وقتشه مبحث کامل و ساده‌ای از "Type Casting" یا تبدیل نوع رو داشته باشیم.

Cast کردن یعنی چی؟

Type Castingیعنی تبدیل مقدار یک نوع داده به نوع دیگه
مثلاً تبدیل یک عدد int به float64 یا تبدیل uint16 به uint8

در زبان Go این تبدیل با نوشتن نوع جدید به شکل تابع انجام می‌شه

```
var x int = 10
var y float64 = float64(x)
```

فرمول کلی cast کردن از نوع A به نوع B

```
var y A = B(x)
```

انواع Cast های مجاز (که منطقی‌ان)

از نوع	به نوع	نکته
int	float64	بدون مشکل
float64	int	اعشار حذف میشه
int	int64/int32	بدون مشکل (تا وقتی اندازه کافیه)
int64	int	فقط اگه عدد در محدوده intباشه
uint8	int	مشکلی نیست
int16	uint16	فقط اگه منفی نباشه
int	string	نه مستقیماً! باید strconv.Itoa یا fmt.Sprintf استفاده بشه
byte	rune	مجاز و رایج

Cast های خطرناک یا غیرمجاز

از نوع	به نوع	چرا مشکل داره؟
int64	int32	ممکنه داده از بین بره (overflow)
uint16	uint8	اگر عدد بزرگتر از 255 باشه overflow می‌شه
float64	uint	اگر عدد منفی یا خیلی بزرگ باشه، مشکل ایجاد می‌کنه
string	int	به صورت غیرممکن (string باید parse بشه)
bool	float64/float32/int	در Go مجاز نیست
float64	int	اعشار حذف میشه، دقت از بین می‌ره

مثلاً در Go این غیرقانونیه:

```
var x string = "123"
var y int = int(x)  // ✕ کامپایل نمی‌شه
```

باید از strconv.Atoi(x) یا fmt.Sscanf() استفاده کنی.

نکات مهم در Cast کردن

1. تبدیل به نوع بزرگ‌تر امن‌تره (مثلاً int به int64)
2. تبدیل به نوع کوچکتر فقط وقتی امنه که مطمئن باشی داده داخل محدوده نوع مقصد قرار داره
3. برای تبدیل رشته به عدد یا برعکس از توابع کتابخونه‌ای استفاده بشه (strconv, fmt)
4. در GO bool به int یا float تبدیل نمیشه

جدول خلاصه تبدیل‌ها

	int	uint	int8	uint8	int16	uint16	int32	uint32	int64	uint64	float32	float64	string	bool	rune	byte
int	✓	⚠	✓	✓	✓	✓	✓	✓	⚠	⚠	⚠	⚠	✗	✗	✓	✓
uint	⚠	✓	⚠	✓	⚠	✓	⚠	✓	⚠	⚠	⚠	⚠	✗	✗	✓	✓
int8	⚠	⚠	✓	⚠	⚠	⚠	⚠	⚠	⚠	⚠	⚠	⚠	✗	✗	⚠	⚠
uint8	⚠	⚠	⚠	✓	⚠	✓	⚠	⚠	⚠	⚠	⚠	⚠	✗	✗	✓	✓
int16	⚠	⚠	✓	✓	✓	⚠	✓	⚠	⚠	⚠	⚠	⚠	✗	✗	✓	✓
uint16	⚠	⚠	✓	✓	⚠	✓	⚠	⚠	⚠	⚠	⚠	⚠	✗	✗	✓	✓
int32	⚠	⚠	✓	✓	✓	✓	✓	⚠	⚠	⚠	⚠	⚠	✗	✗	✓	✓
uint32	⚠	⚠	✓	✓	✓	✓	⚠	✓	⚠	⚠	⚠	⚠	✗	✗	✓	✓
int64	⚠	⚠	✓	✓	✓	✓	✓	✓	✓	⚠	⚠	⚠	✗	✗	✓	✓
uint64	⚠	⚠	⚠	✓	⚠	✓	⚠	✓	⚠	✓	⚠	⚠	✗	✗	✓	✓
float32	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	⚠	✗	✗	✓	✓
float64	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✗	✗	✓	✓
string	⚠	⚠	⚠	⚠	⚠	⚠	⚠	⚠	⚠	⚠	✗	✗	✓	✗	⚠	⚠
bool	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✓	✗	✗
rune	✓	⚠	✓	✓	✓	✓	✓	✓	⚠	⚠	✗	✗	✗	✗	✓	⚠
byte	⚠	⚠	⚠	✓	⚠	⚠	⚠	⚠	⚠	⚠	✗	✗	✗	✗	⚠	✓

✓ بدون مشکل (امن)

⚠ نیاز به احتیاط: ممکنه overflow ، underflow یا precision loss اتفاق بیفته

✗ مستقیم مجاز نیست؛ باید از توابع یا کتابخانه استفاده بشه

مفهوم مقدار اولیه (zero value)

یادته گفتیم متغیر در برنامه‌نویسی مثل یه جعبه‌ست؟
توش چیزی می‌ریزیم و با یه اسم صداش می‌زنیم.

حالا فرض کن یه جعبه می‌سازی:

```
var x int
```

تو اینجا گفتی:

من یه جعبه‌ی عددی می‌خوام به اسم x ولی هنوز نمی‌دونم چی توش بذارم

زبان Go نمی‌ذاره جعبه‌ت خالی بمونه.
می‌گه:
من برات یه چیز پیش‌فرض می‌ذارم تا بعداً خواستی عوضش کنی

اون چیز پیش‌فرض همون Zero Value هست.

الان اگه بیایم مقدار این متغیر x رو چاپ کنیم عدد 0 رو دریافت میکنیم. در واقع زبان GO مقدار اولیه متغیر x که در اینجا از نوع int هست رو 0 در نظر گرفته

```
fmt.Println(“x is:”, x) //مقدار 0 چاپ میشه
```

جدول مقدار اولیه (zero value) برای انواع مختلف

نوع داده (type)	مقدار اولیه (zero value)
int8 uint8 int16 uint16 int32 uint32 int64 uint64 int uint byte rune	0
float32 float64	0
bool	false
string	“”
pointer slice map interface chan func	nil
struct	همه فیلدها با zero value
array	همه خانه ها با zero value

مثال

```
package main

import "fmt"

func main() {
    var age int       // صفر
    fmt.Println(age)  // 0
    var name string   // رشته‌ی خالی ""
    fmt.Println(name) // ""
}
```

تعریف و مقداردهی همزمان چند متغیر از یک نوع

گاهی تو برنامه‌نویسی، لازم داریم چند تا متغیر مختلف رو با مقادیر مختلف به‌صورت همزمان تعریف کنیم
اگه Go این قابلیت رو نداشت، مجبور بودیم برای هر متغیر یه خط جداگونه بنویسیم.

مثلاً:

```
var a uint = 1
var b uint = 2
var c uint = 3
```

این روش وقت‌گیرتره، کد رو شلوغ‌تر می‌کنه و خوانایی رو کم می‌کنه.

با استفاده از قابلیت مقداردهی همزمان، می‌تونیم بنویسیم:

```
var a, b, c uint = 1, 2, 3
```

همزمان سه متغیر تعریف و مقداردهی شدن، بدون تکرار نوع داده یا چند خط اضافه.

چی میشه اگه این ویژگی وجود نداشت؟

- باید برای هر متغیر خط جدا بنویسیم
- کد طولانی‌تر می‌شد
- احتمال اشتباه (مثل تایپ اشتباه نوع یا اسم متغیر) بیشتر می‌شد
- خوندن کد سخت‌تر می‌شد

مثال:

```
package main

import "fmt"

func main() {
    var length, width uint8 = 4, 3
    fmt.Println(“Area is: ”, length * width)
}
```

در اینجا اضلاع مستطیل رو به سادگی با یک دستور تعریف و مقداردهی کردیم

نکته:

استفاده با := هم ممکنه

```
length, width := 4, 3
```

Go به طور خودکار نوع داده رو از مقدارها تشخیص می‌ده.

خطای تعریف متغیر بدون استفاده

فرض کن یه متغیر تعریف کردی، ولی هیچ‌وقت نرفتی سراغش یا کاری باهاش نکردی. تو زبان Go این موضوع قبول نمی‌شه و کامپایلر می‌گه: «تو این متغیر رو ساختی ولی اصلاً استفاده نکردی! این کار اشتباهه

مثال

```
package main

import "fmt"

func main() {
    var age uint8 = 10  // تعریف متغیر
    // اما استفاده‌ای از age نشده
}
```

این برنامه کامپایل نمیشه چون متغیر age تعریف شده ولی هیچوقت استفاده نشده

دلایل وجود این محدودیت

کد تمیزتر و مرتب‌تر

فرض کن کلی متغیر تعریف کنی ولی هیچ‌وقت ازشون استفاده نکنی. این باعث می‌شه کدت نامرتب و گیج‌کننده بشه.
کامپایلر می‌خواد برنامه‌نویس‌ها کد تمیز و بدون بخش‌های اضافی بنویسن.

اشتباهات احتمالی رو زودتر پیدا کنه

گاهی وقت‌ها تعریف یک متغیر بدون استفاده یعنی یه خطای منطقی تو برنامه هست. مثلاً شاید فراموش کردی از متغیر استفاده کنی یا کدی که باید نوشته می‌شد رو ننوشتی.
کامپایلر با خطا دادن بهت هشدار می‌ده که احتمالاً یه جای کار اشتباهه.

کاهش حجم برنامه و بهینه‌تر شدن کد

متغیرهای بدون استفاده باعث می‌شن کد بزرگ‌تر و سنگین‌تر بشه، که توی برنامه‌های بزرگ این می‌تونه باعث کاهش کارایی یا افزایش حجم فایل اجرایی بشه.

تمرین 1: بررسی عملگرهای ترکیبی، افزایشی و کاهشی

ابتدا این برنامه رو با استفاده از مفاهیمی که یادگرفتی تحلیل کن و پیش بینی کن بعد از اجرای هر دستور مقدار score چقدر میشه، بعدش برنامه رو بازنویسی و اجرا کن تا پیش بینی هاتو صحت سنجی کنی

```
package main

import "fmt"

func main() {
    var score int = 10
    score += 5
    score *= 2
    score -= 4
    score /= 2
    score++
    score--

    fmt.Println("Final score:", score)
}
```

تمرین 2: بررسی مفهوم wrap around و overflow

برنامه رو بازنویسی و اجرا کن و به این سوالا پاسخ بده:

1-چه عددی چاپ میشه؟

2-این رفتار چه خطرهایی داره؟

3-با تغییر نوع متغیر به int8 و uint16 مجدد برنامه رو اجرا کن، نتیجه چه تغییری کرد؟

```
package main

import "fmt"

func main() {
    var small uint8 = 250
    small += 10
    fmt.Println("small =", small)
}
```

تمرین 3: تبدیل نوع و خطاهای مربوطه

برنامه رو بازنویسی و اجرا کن و به این سوالا پاسخ بده:

1-چه تفاوتی بین x و y وجود داره؟

2-نوع متغیر x رو به string تغییر بده و مقدارش هم از 3.9 به "3.9" تغییر بده. تو این حالت چه اتفاقی میوفته؟ میتونی یه توضیح منطقی درباره این رفتار بدی؟

```
package main

import "fmt"

func main() {
    var x float64 = 3.9
    var y int = int(x)
    fmt.Println("x =", x)
    fmt.Println("y =", y)
}
```

تمرین 4: محاسبه حجم جعبه

برنامه ای بنویس که حجم یک جعبه رو محاسبه کنه.

برای محاسبه حجم جعبه به 3 متغیر برای طول، عرض و ارتفاع نیاز داریم، این 3 متغیر رو به یکباره تعریف و مقداردهی کن، یک متغیر برای نتیجه حجم به اسم volume تعریف کن، قبل از اینکه محاسبه رو انجام بدی volume چه مقداری داره؟ بعد از انجام محاسبه چطور؟

فرمول محاسبه حجم جعبه:

```
volume = length * width * height
```