مفهوم آرایه با طول متغیر

قراره یه برنامه بنویسیم که **میانگین نمرات دو دانش‌آموز** رو حساب کنه.
اما یه نکته مهم داریم: این دو دانش‌آموز ممکنه تو مقاطع مختلفی باشن و **تعداد درس‌هاشون با هم فرق داشته باشه**.
مثلاً یکی از اونا ممکنه 7 درس داشته باشه و اون یکی 9 تا.
پس برنامه باید اون‌قدر **انعطاف‌پذیر** باشه که بتونه بدون مشکل، میانگین نمرات هر دو نفر رو جداگانه حساب کنه.

برای مثال، می‌خوایم با همین برنامه **میانگین نمرات ندا و امید** رو به‌دست بیاریم، حتی اگه تعداد درس‌هاشون یکی نباشه.

نمرات دروس ندا

درس	نمره
فارسی	19.50
دین و زندگی	11.00
زبان خارجی	20.00
تربیت بدنی	16.50
جبر	17.50
ریاضیات گسسته	18.00
حساب دیفرانسیل	16.25
فیزیک	18.50
هندسه	19.00

نمرات دروس امید

درس	نمره
فارسی	20.00
ریاضی	6.00
علوم تجربی	17.75
مطالعات اجتماعی	20.00
دین و زندگی / قرآن	14.00
هنر	19.50
تربیت بدنی	19.00

از اونجایی که **ندا تو پایه یازدهمه** و **امید تو پایه چهارم**، طبیعیه که **تعداد و نوع درس‌هاشون با هم فرق داشته باشه**.

مثلاً ندا ممکنه زیست و فیزیک و ریاضی داشته باشه، ولی امید فقط فارسی و ریاضی.

پس وقتی می‌خوایم برنامه‌ای بنویسیم که نمرات این دو نفر رو بگیره و میانگین حساب کنه، باید حواسمون باشه که **ما از قبل نمی‌دونیم قراره چند تا نمره وارد بشه**.

اینجاست که یه سؤال مهم پیش میاد:

ما می‌خوایم چند تا عدد (نمره) از ورودی بگیریم و ذخیره کنیم.

چون این نمره‌ها همه عدد هستن (و هم‌نوع)، باید اونا رو توی یه آرایه بریزیم.

ولی طول آرایه چقدر باشه؟ ما که از قبل تعداد درس‌ها رو نمی‌دونیم!

اینجاست که **آرایه‌های با طول ثابت** به کارمون نمی‌آن، چون اونا موقع تعریف باید طولشون مشخص باشه.

ما نیاز داریم به یه ساختار داده که بتونه با شرایط ما کنار بیاد.

در واقع چون ما از قبل نمی‌دونیم هر دانش‌آموز چند تا درس داره، نمی‌تونیم طول آرایه رو از قبل مشخص کنیم. پس باید یه کاری کنیم که **خود کاربر (یا همون اجراکننده‌ی برنامه)** بهمون بگه قراره میانگین نمرات چند تا درس رو حساب کنیم.

یعنی چی؟ یعنی وقتی برنامه اجرا می‌شه، اول از کاربر بپرسیم:

چند تا درس داری که می‌خوای نمره‌هاش رو وارد کنی؟

بعد براساس همون عددی که کاربر وارد می‌کنه، بیایم یه آرایه بسازیم که **طولش متناسب با همون تعداد درس باشه**.

به این صورت، برنامه‌مون هم **انعطاف‌پذیر می‌شه** و هم می‌تونه با تعداد نمرات مختلف بدون مشکل کار کنه.

روش تعریف آرایه با طول متغیر

```
var <name> []<type>
```

var

کلمه‌ی کلیدی برای تعریف متغیر

name

نام متغیر برای دسترسی به اون

یعنی قراره با این اسم به slice دسترسی داشته باشیم. (به آرایه با طول متغیر میگن slice)

هر اسمی می‌تونی بذاری (مثل grades)

[]

این دو براکت خالی یعنی این یه آرایه به طول متغیر(یا slice) هست، نه آرایه ثابت.

برخلاف آرایه‌های ثابت که داخل براکت باید طول رو مشخص کنیم، اینجا هیچ عددی نمی‌نویسیم.

یعنی:

- طولش می‌تونه بعداً بزرگ یا کوچیک بشه
- متغیره (دینامیکه)

type

نوع داده‌ی عناصری که می‌خوای داخل slice قرار بدی

همه‌ی عناصر باید از همین نوع باشن.

مثلاً int[] یعنی این slice قراره فقط عدد صحیح داخلش باشه، یا string[] یعنی فقط متن داخلشه.

مثال

```
var grades []float32
```

- grades اسم متغیره
- float32[] یعنی این یه slice از عددهای اعشاریه (float32)
- هنوز هیچ مقداری نداره، و نیاز هست قبل از استفاده یک آرایه ساخته بشه

روش عملکرد آرایه با طول متغیر

قبل از اینکه کاربرد آرایه با طول متغیر یا همون **آرایه پویا** رو ببینیم، بهتره اول از نظر مفهومی بدونیم این نوع آرایه چطوری کار می‌کنه.

چرا آرایه پویا انعطاف‌پذیره؟

وقتی ما یه آرایه با طول متغیر تعریف می‌کنیم، مثلاً اینطوری:

```
var grades []float32
```

این grades **خودش آرایه نیست!**

در واقع یک **اشاره‌گر (pointer)** به یک آرایه پشت صحنه (که بهش می‌گیم underlying array) هست.

نکته مهم:

- در آرایه با طول متغیر **طول را موقع تعریف مشخص نمی‌کنیم.**
- یعنی طول آرایه پشت صحنه می‌تونه تغییر کنه و slice ممکنه بارها در حین اجرای برنامه به آرایه‌های مختلف اشاره کنه.

چطور طول آرایه با طول متغیر در حین اجرا تغییر میکنه؟

فرض کن:

- در ابتدا، slice به آرایه‌ای با طول 10 اشاره می‌کنه
- این آرایه تو حافظه از آدرس 9000 شروع شده و شامل 10 مکان حافظه برای ذخیره‌ی int64 هست.
- هر عدد int64، 8 بایت جا می‌گیره (هر مکان حافظه 1 بایت ظرفیت داره پس هر عدد تو 8 مکان حافظه ذخیره میشه) پس آدرس‌ها به شکل زیر هستن:

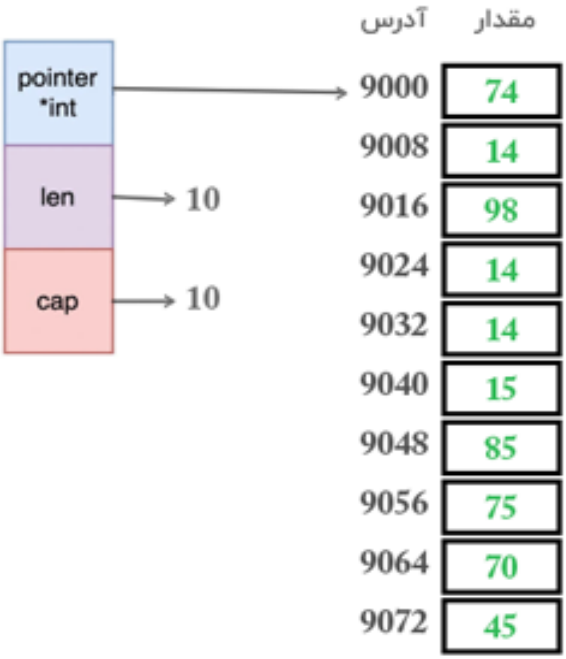
شماره مکان حافظه	آدرس حافظه
0	9000
1	9008
...	...
9	9072

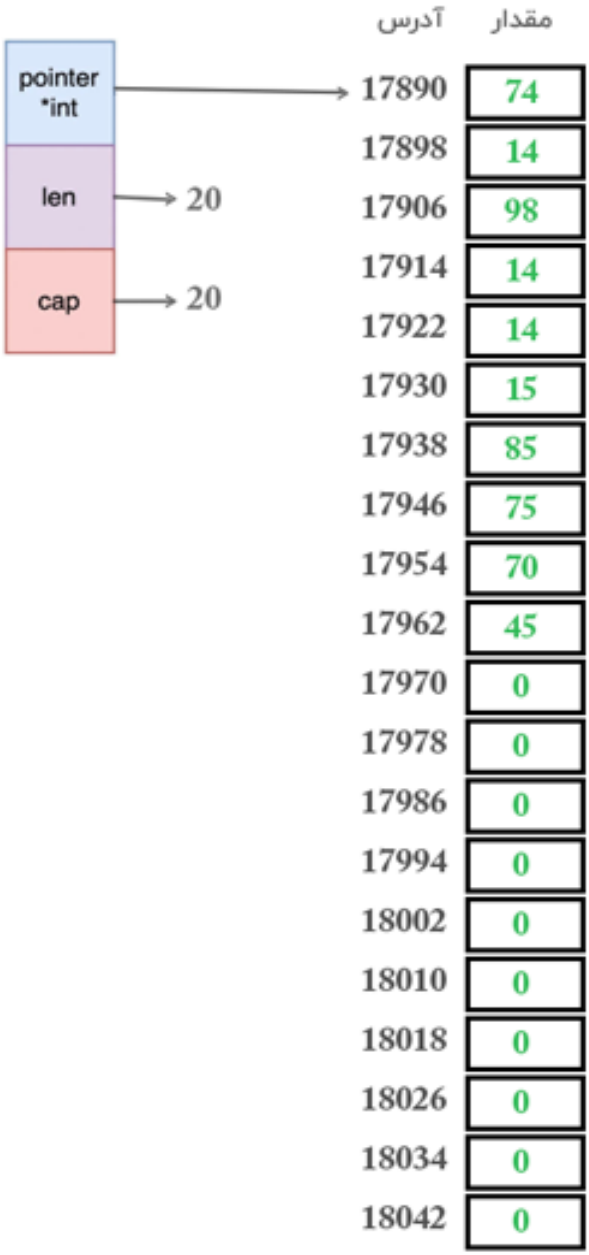
حالا فرض کنیم می‌خواهیم فضای ذخیره‌سازی را به 20 عدد افزایش بدیم

- هر عدد تو 8 مکان حافظه ذخیره میشه پس برای ذخیره کردن 20 عدد int64 به 160 مکان حافظه نیاز داریم
- مثلاً این فضای جدید از آدرس 17890 شروع می‌شود.
- تمام 10 عدد قدیمی که از آدرس 9000 تا 9072 قرار داشتند، به این فضای جدید (از آدرس 17890 به بعد) کپی می‌شوند.
- باقی فضای جدید (80 مکان حافظه اضافه) آماده برای ذخیره داده‌های جدید است.

خلاصه:

- **آدرس حافظه نشان‌دهنده‌ی شروع فضای اختصاص داده شده است.**
- هر عنصر در فاصله 8 بایت از عنصر قبلی ذخیره می‌شود (چون int64 هشت بایت است)
- وقتی حجم ذخیره‌سازی افزایش پیدا می‌کند، فضای جدیدی در حافظه اختصاص داده شده و داده‌ها به آنجا منتقل می‌شوند.
- **slice اشاره‌گرش را به این فضای جدید به‌روزرسانی می‌کند.**


آدرس
مقدار
pointer *int
len
10
cap
10
9000
74
9008
14
9016
98
9024
14
9032
14
9040
15
9048
85
9056
75
9064
70
9072
45


آدرس
مقدار
pointer *int
len
20
cap
20
17890
74
17898
14
17906
98
17914
14
17922
14
17930
15
17938
85
17946
75
17954
70
17962
45
17970
0
17978
0
17986
0
17994
0
18002
0
18010
0
18018
0
18026
0
18034
0
18042
0

نگاهی دقیق تر به slice

slice یه **ساختار داده** (data structure) هست که برای کار با آرایه‌ها به شکل **انعطاف‌پذیر** طراحی شده.
برخلاف آرایه که طولش ثابته slice می‌تونه طولش تو زمان اجرا تغییر کنه.

slice خودش آرایه نیست! بلکه یه **اشاره‌گر به یه آرایه پشت صحنه** هست (بهش می‌گیم underlying array)

slice دقیقاً از 3 تا چیز تشکیل شده:

1- pointer (اشاره‌گر)

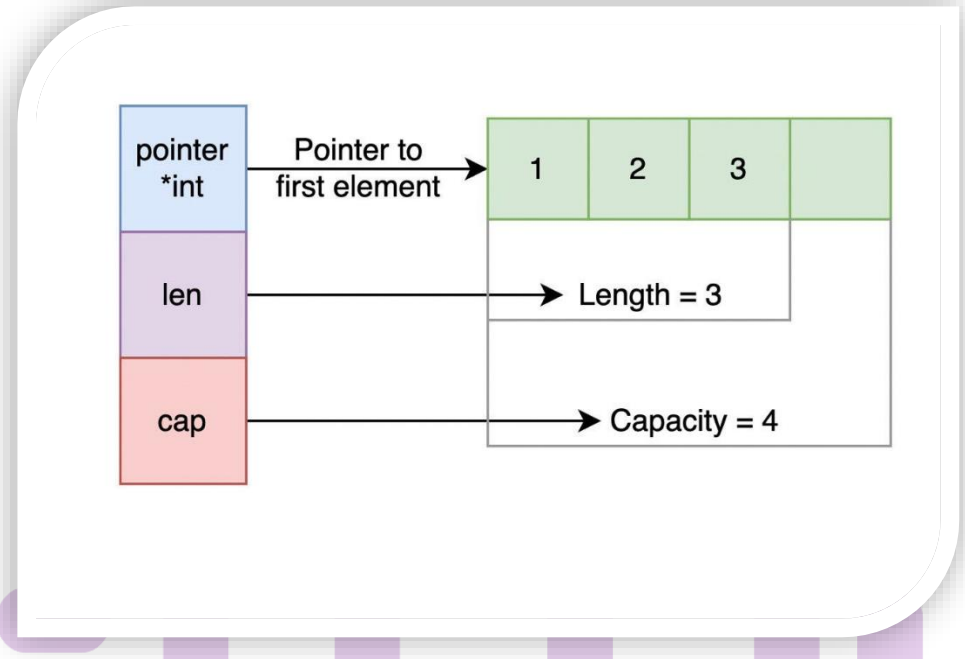
نشون می‌ده که داده‌ها از کجای حافظه (آدرس آرایه اصلی) شروع می‌شن.

2- length (طول)

می‌گه چند تا عنصر در حال حاضر داخل آرایه ای که slice بهش اشاره داره ذخیره شده.

3- capacity (ظرفیت)
یعنی از جایی که pointer اشاره می‌کنه، حداکثر چند عنصر می‌تونیم استفاده کنیم (ممکنه از length بیشتر باشه)

بررسی ساختار slice با مثال



بخش آبی (pointer) نشون‌دهنده آدرس شروع داده‌ها هست

بخش بنفش (len) تعداد عنصرهای قابل دسترسی توسط slice رو نگه می‌داره

بخش قرمز (cap) ظرفیت واقعی slice از آدرس شروع تا پایان underlying array رو نمایش می‌ده

فلش نشون می‌ده که slice به آرایه پشتیبان اشاره می‌کنه

در این تصویر، ما یک slice به اسم numbers داریم.

این slice خودش داده‌ای نگه نمی‌داره، بلکه فقط یه اشاره‌گره — یعنی داره به یه آرایه (underlying array) توی حافظه اشاره می‌کنه.

برای اینکه مفهوم length و capacity رو درست متوجه بشیم بهتره قبلش با دستور make و کاربردش آشنا بشیم

دستور make

```
make([]type, length, capacity)
```

یادت هست گفتیم که یک slice فقط یه اشاره‌گر (pointer) به یک آرایه درون فضای RAM هست؟
خب، پس طبیعتاً ما باید اون آرایه‌ای که slice قراره بهش اشاره کنه رو بسازیم.

دستور make دقیقاً همین کار رو برای ما انجام می‌ده.

در واقع با make به Go می‌گیم:

لطفاً یه آرایه از نوع type برای من بساز که به تعداد length بتونم عنصر داخلش قرار بدم

اما capacity یعنی چی؟

یادت هست گفتیم اگه بخوایم آرایه‌ای رو بزرگ‌تر کنیم GO یه آرایه‌ی جدید توی رم می‌سازه، مقادیر قبلی رو توش کپی می‌کنه و سپس به اون اشاره می‌کنه؟

خب حالا فرض کن موقعی که داریم اون آرایه‌ی اول رو می‌سازیم، از همون اول **پیش‌بینی کنیم که ممکنه بعداً نیاز به بزرگ‌تر شدن داشته باشه.**

مثلاً:

تو آدرس حافظه 9000 ما می‌خوایم الان فقط 10 تا عدد int64 ذخیره کنیم

ولی احتمال می‌دیم که در آینده بخوایم 10 تا عدد دیگه هم بهش اضافه کنیم.

پس به جای اینکه فقط یه آرایه با طول 10 بسازیم، می‌گیم:

فعلاً تو آرایه فقط 10 تا مقدار میخوایم قرار بدیم، ولی از همین حالا جا برای 20 تا آماده کن که بعداً راحت بتونیم مقدارهای جدید رو بدون دردسر اضافه کنیم

یعنی:

- len = 10 یعنی 10 عدد اول واقعاً در حال استفاده هستن

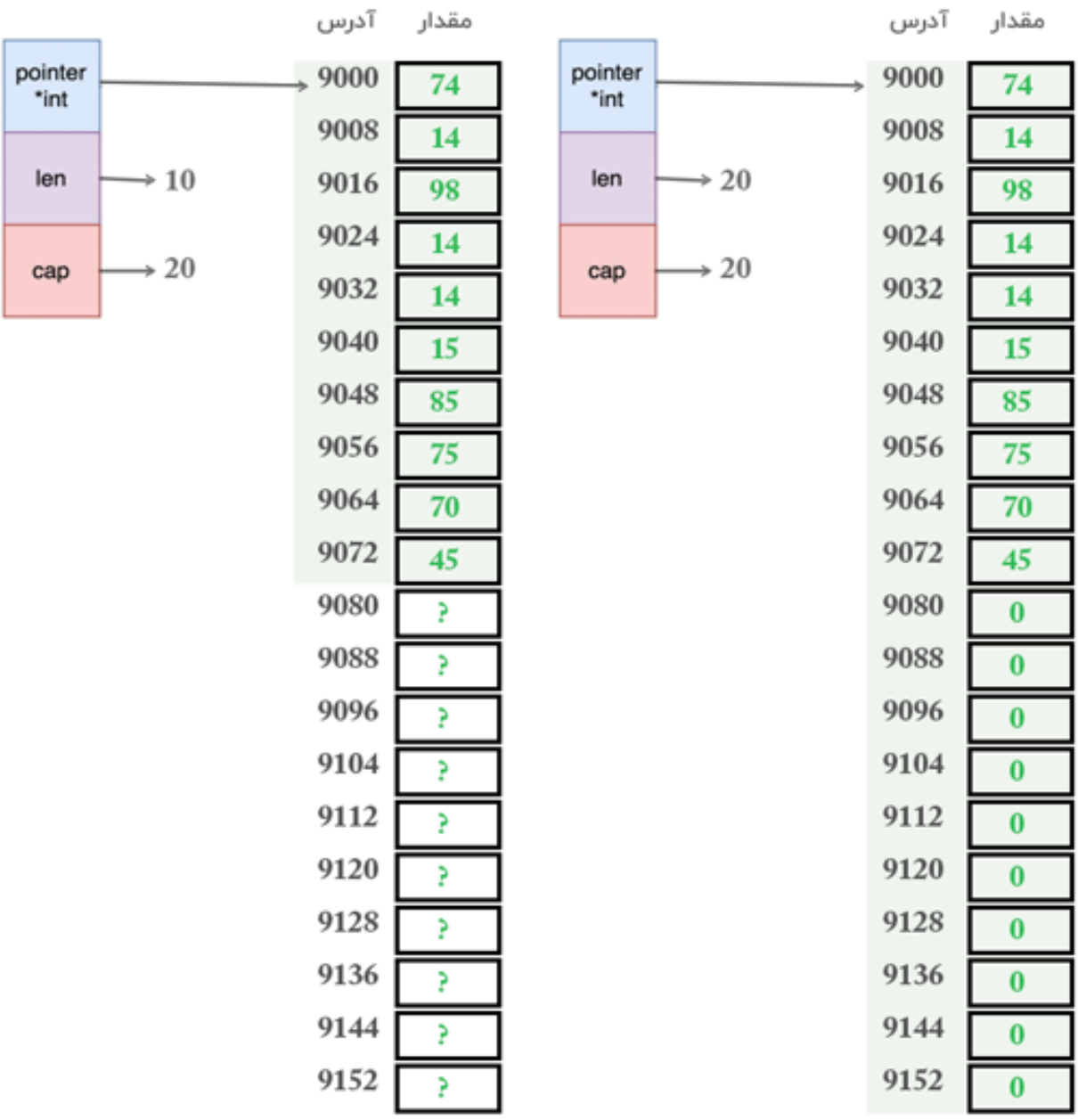
- cap = 20 یعنی 10 مقدار دیگه هم از قبل برامون رزرو کن

حالا اگه بخوایم عددهای جدید به slice اضافه کنیم و هنوز جا خالی وجود داشته باشه، Go نیازی به ساختن آرایه‌ی جدید و کپی‌کردن داده‌ها نداره، چون از اول پیش‌بینی کردیم و جا گذاشتیم.

فرض کن:

- آدرس شروع حافظه 9000
- هر عدد int64، هشت بایت حافظه لازم داره
- پس برای 10 عدد 80 بایت حافظه نیاز هست، پس تا آدرس 9080 نیاز هست
- و برای 10 عدد بعدی (که جایگاه براش رزرو میکنیم) 80 بایت دیگه نیاز هست، پس از آدرس 9080 تا آدرس 9160 رزرو میشه برای استفاده احتمالی در آینده

اگر ما از اول این 160 بایت رو رزرو کنیم، Go می‌تونه بعداً بدون دردسر از ادامه‌ی همون فضای حافظه استفاده کنه.


pointer
*int
len
10
cap
20
آدرس
مقدار
9000 74
9008 14
9016 98
9024 14
9032 14
9040 15
9048 85
9056 75
9064 70
9072 45
9080 ?
9088 ?
9096 ?
9104 ?
9112 ?
9120 ?
9128 ?
9136 ?
9144 ?
9152 ?
pointer
*int
len
20
cap
20
آدرس
مقدار
9000 74
9008 14
9016 98
9024 14
9032 14
9040 15
9048 85
9056 75
9064 70
9072 45
9080 0
9088 0
9096 0
9104 0
9112 0
9120 0
9128 0
9136 0
9144 0
9152 0

در این حالت، اول اومدیم با استفاده از دستور make یک **آرایه از نوع int64** ساختیم که:

- **طولش 10** هست (یعنی قراره فعلاً فقط 10 عدد داخلش قرار بدیم)
- و **ظرفیتش 20** تعیین شده (یعنی در صورت نیاز، می‌تونیم تا 20 عدد در اون ذخیره کنیم)

در واقع این یعنی Go از همون اول توی حافظه فضایی در نظر می‌گیره که جا برای **20 عدد int64** داشته باشه.
اما فقط 10 تای اولش فعلاً فعال هستن (یعنی len = 10)

اون 10 فضای دیگه همون‌جا پشت سر این آرایه، رزرو می‌مونه تا اگه لازم شد در آینده تعداد عنصرها رو زیاد کنیم،

نیازی به ساخت آرایه‌ی جدید و کپی‌کردن داده‌ها نباشه — و این باعث می‌شه برنامه سریع‌تر و بهینه‌تر اجرا بشه.

برنامه میانگین نمرات

حالا که با مفهوم **آرایه‌های پویا (slice)** آشنا شدیم، می‌تونیم برنامه‌ی محاسبه‌ی **میانگین نمرات** رو کامل کنیم.

این برنامه قراره تعدادی نمره از ورودی دریافت کنه، اون‌ها رو در یک آرایه‌ی پویا ذخیره کنه و در نهایت **میانگین‌شون** رو محاسبه و چاپ کنه.

در ابتدای اجرا، برنامه از کاربر (یعنی اجراکننده‌ی برنامه) می‌پرسه:

می‌خوای میانگین چند تا نمره رو حساب کنی؟

سپس با توجه به عددی که کاربر وارد می‌کنه، یه آرایه‌ی پویا (slice) به همون اندازه ایجاد می‌کنیم و نمره‌ها رو به ترتیب از ورودی می‌گیریم و داخل اون slice ذخیره می‌کنیم.

از اونجایی که این آرایه قرار نیست در طول اجرای برنامه تغییر اندازه بده (یعنی فقط همون یه بار نمره‌ها رو می‌گیریم)، نیازی به تعیین ظرفیت بیشتر نداریم — همون length که کاربر گفته برامون کافیه.

بقیه‌ی مراحل مثل برنامه‌های قبلیه: مجموع نمره‌ها رو حساب می‌کنیم و بعد میانگین می‌گیریم.

```
package main

import "fmt"

func main() {
    var numberOfCourses uint8
    fmt.Println("میانگین چند تا نمره رو میخوای حساب کنی؟")
    fmt.Scanln(&numberOfCourses)

    var grades []float32 = make([]float32, int(numberOfCourses))
    for i := 0; i < int(numberOfCourses); i++ {
        fmt.Println("نمره رو وارد کن:")
        fmt.Scanln(&grades[i])
    }

    var sum float32
    for i := 0; i < int(numberOfCourses); i++ {
        sum = sum + grades[i]
    }

    var average float32 = sum / float32(numberOfCourses)
    fmt.Println("میانگین نمرات: ", average)
}
```

ما توی این برنامه قراره از **کاربر نمره‌ها رو دریافت کنیم**. و چون از قبل مشخص کردیم قراره چند تا نمره وارد بشه، باید به همون تعداد:

1. پیام «نمره رو وارد کن» نشون بدیم
2. و بعد نمره رو از کاربر بگیریم و توی آرایه ذخیره کنیم.

از اونجایی که این کار چند بار پشت سر هم تکرار می‌شه (چند بار پیام نشون بدیم و ورودی بگیریم)، از **حلقه‌ی for** استفاده کردیم تا این بخش از برنامه مرتب و کوتاه‌تر نوشته بشه.

بعد از اینکه همه‌ی نمره‌ها وارد شدن، باید **جمع کل نمرات** رو حساب کنیم.

برای این کار هم دوباره از یه حلقه استفاده کردیم، چون جمع زدن هم **یه عملیات تکراری** روی همه‌ی نمره‌هاست.

در آخر، برای محاسبه‌ی میانگین فقط کافیه جمع نمرات رو تقسیم کنیم به تعدادشون. و نتیجه رو چاپ کنیم.

نکته:
از اونجایی که **طول و ظرفیت** آرایه‌ی پویا در این برنامه با هم برابر بودن، دیگه نیازی نبود که **پارامتر سوم** تابع makeرو بنویسیم. وقتی پارامتر سوم رو ننویسیم، Go به طور خودکار همون مقدار length رو به عنوان capacity هم در نظر می‌گیره.

روش های دیگر مقدار دهی آرایه متغیر

1-ایجاد slice از یک آرایه پویا یا آرایه با طول ثابت

تا اینجا فهمیدیم که برای استفاده از آرایه‌های پویا (slice)، معمولاً با دستور make شروع می‌کنیم.

```
s := make([]int, 3)
```

این یه راه استاندارد برای ساختن slice هست. اما تنها راه نیست!

باید بدونیم که وقتی یه متغیر از نوع []type تعریف می‌کنیم، داریم به Go می‌گیم:

"قراره یه slice از این نوع داده داشته باشیم".

حالا مهم نیست این slice چطور ساخته می‌شه — با make یا یه روش دیگه، فقط کافیه یه slice از همون نوع باشه.

قبل از اینکه روش ایجاد slice از یک آرایه پویا یا آرایه با طول ثابتو توضیح بدیم نیاز هست با عملگر [:] آشنا بشیم.

عملگر [:]

این عملگر مخصوص **slice** هست، و بهش می‌گن **slice operator**
باهاش می‌تونی از یه slice یا آرایه، یه زیرمجموعه (زیر slice یا sub-slice) بسازی

ساختار کلی

```
slice[start:end]
```

start ایندکس شروع (شامل میشه)

end ایندکس پایان (شامل نمی‌شه!)

[:] بدون start و end یعنی (کل محتوا)

ایجاد slice از آرایه با طول ثابت

```
var a [4]int = [4]int{1, 2, 3, 4}
s := a[:] // ساخت یک slice از کل آرایه a
```

ایجاد slice از آرایه پویا

```
s := make([]int, 5)
s[0] = 10
s[1] = 20
s[2] = 30
s[3] = 40
s[4] = 50
fmt.Println(s[1:4])  // → [20 30 40]
fmt.Println(s[:3])   // → [10 20 30]
fmt.Println(s[2:])   // → [30 40 50]
fmt.Println(s[:])    // → [10 20 30 40 50] (کل slice)
```

نکته: این عملگر فقط روی slice و array جواب می‌ده

ساخت slice از آرایه با طول ثابت

میتونیم از یک آرایه با طول ثابت یک slice ایجاد کنیم

به مثال زیر دقت کن:

```
var familyMembers [3]string = [3]string{“arian”, “neda”, “omid”}
var siblings []string = familyMembers[:]
fmt.Println(“خواهران/برادران:”, siblings)
```

اینجا:

- یه آرایه‌ی با طول ثابت به اسم familyMembers تعریف کردیم
- با استفاده از [:] از کل آرایه، یه slice به اسم siblings ساختیم

نکته:این (siblings) slice دقیقاً به همون آرایه (familyMembers) **اشاره می‌کنه**.

یعنی اگه یکی از اون‌ها تغییر کنه، روی اون یکی هم تأثیر می‌ذاره!

```
package main

import "fmt"

func main() {
    var familyMembers [3]string = [3]string{“arian”, “neda”, “omid”}
    var siblings []string = familyMembers[:]
    familyMembers[1] = “NEDA”

    fmt.Println(“خواهران/برادران:”, siblings)
}
```

ساخت slice از آرایه پویا

میتونیم از یک آرایه با طول پویا یک slice ایجاد کنیم

به مثال زیر دقت کن:

```
var familyMembers []string = make([]string, 3)
familyMembers[0] = “arian”
familyMembers[1] = “neda”
familyMembers[2] = “omid”
var siblings []string = familyMembers[:]
fmt.Println(“خواهران/برادران:”, siblings)
```

اینجا:

- یه آرایه‌ی پویا (slice) به اسم familyMembers تعریف کردیم
- با استفاده از [:] از کل آرایه، یه slice به اسم siblings ساختیم

نکته:این (siblings) slice دقیقاً به همون آرایه (familyMembers) **اشاره می‌کنه**.

یعنی اگه یکی از اون‌ها تغییر کنه، روی اون یکی هم تأثیر می‌ذاره!

```
package main

import "fmt"

func main() {
    var familyMembers []string = make([]string, 3)
    familyMembers[0] = “arian”
    familyMembers[1] = “neda”
    familyMembers[2] = “omid”
    var siblings []string = familyMembers[:]
    familyMembers[1] = “NEDA”

    fmt.Println(“خواهران/برادران:”, siblings)
}
```

2-مقدار دهی مستقیم

وقتی یه slice رو همون لحظه‌ی تعریفش، **هم تعریف می‌کنی هم براش مقدار مشخص می‌کنی**، می‌گیم "مقداردهی مستقیم" یا "استفاده از literal"

به زبان ساده، یعنی: همزمان با تعریف slice، مقادیرش رو هم بنویسی

شکل کلی

```
var <name> []<type> = []<type>{<value1>, <value2>, ...}
```

var

کلمه‌ی کلیدی برای تعریف متغیر

name

نام متغیر برای دسترسی به اون

یعنی قراره با این اسم به slice دسترسی داشته باشیم. (به آرایه با طول متغیر میگن slice)

هر اسمی می‌تونی بذاری (مثل grades)

[]

این دو براکت خالی یعنی این یه آرایه به طول متغیر(یا slice) هست، نه آرایه ثابت.

برخلاف آرایه‌های ثابت که داخل براکت باید طول رو مشخص کنیم، اینجا هیچ عددی نمی‌نویسیم.

یعنی:

- طولش می‌تونه بعداً بزرگ یا کوچیک بشه
- متغیره (دینامیکه)

type

نوع داده‌ی عناصری که می‌خوای داخل slice قرار بدی

همه‌ی عناصر باید از همین نوع باشن.

مثلاً []int یعنی این slice قراره فقط عدد صحیح داخلش باشه، یا []string یعنی فقط متن داخلشه.

[]<type>{<value1>, <value2>, ...}

این قسمت به Go میگه:

- یه آرایه‌ی پویا بساز که عناصرش مقدارهای داده‌شده باشن
- طول و ظرفیت برابر تعداد عناصر مشخص شده است

مثال

```
var siblings []string = []string{“arian”, “neda”, “omid”}
```

3-مقدار دهی توسط تابع append

در زبان Go، یکی از روش‌های خیلی رایج برای مقداردهی و بزرگ کردن slice، استفاده از تابع append هست.

تابع append در واقع دو کار مهم انجام می‌ده:

1. **مقداردهی اولیه به متغیر slice**، در صورتی که هنوز به هیچ آرایه‌ای در حافظه اشاره نمی‌کنه.
2. **بزرگ‌کردن slice**، زمانی که بخوایم عنصر جدیدی بهش اضافه کنیم اما ظرفیتش پر شده باشه.

در اینجا ما به هر دو کاربرد اشاره می‌کنیم، اما تمرکز اصلی‌مون روی کاربرد اول هست؛ یعنی اینکه چطور با استفاده از append می‌تونیم مقدار اولیه رو داخل یک slice قرار بدیم، حتی اگر هنوز به هیچ آرایه‌ای اشاره نکنه.

به عبارت ساده‌تر:
وقتی یه slice ساختی ولی بهش مقدار ندادی، می‌تونی با append همون موقع هم مقدار بهش بدی، هم کاری کنی که به یه آرایه در حافظه وصل بشه. این روش ساده، امن و خیلی پرکاربرده.

هر بار که از تابع append استفاده کنیم چند کار رو به ترتیب انجام میده

مرحله اول-بررسی اینکه آیا slice واقعاً وجود داره یا نه

وقتی می‌نویسیم:

```
var grades []float64 // هنوز هیچ آرایه‌ای پشتش نیست!
grades = append(grades, 18.5)
```

Go اول چک می‌کنه که این grades واقعاً به یه **آرایه (underlying array)** اشاره می‌کنه یا نه.
اگه نه، خودش یه آرایه جدید تو حافظه می‌سازه، و بعد مقدار رو داخلش قرار می‌ده. پس نگران اینکه متغیر به هیچ آرایه ای اشاره نکنه نباش!

مرحله دوم-اضافه‌کردن مقدار به انتهای slice

این ساده‌ترین بخش کاره، append میاد مقدار جدید رو به آخرین عنصر slice اضافه می‌کنه.

بررسی اینکه جا داره یا نه (بررسی capacity)

اگر فضای خالی توی slice وجود داشته باشه (یعنی ظرفیتش از طولش بیشتر باشه)، همون آرایه قبلی رو استفاده می‌کنه.

ولی اگه ظرفیت پر شده باشه چی؟

مرحله سوم-ایجاد یه آرایه جدید و کپی‌کردن داده‌ها

اگر دیگه جا نداشته باشه capacity) پر باشه(، Goاین کارو می‌کنه:

- یه **آرایه بزرگ‌تر** درست می‌کنه (معمولاً دو برابر یا بیشتر)
- همه‌ی داده‌های قبلی رو توی این آرایه جدید **کپی** می‌کنه
- مقدار جدید رو به تهش **اضافه** می‌کنه
- و در نهایت، sliceجدیدی می‌ده که به این آرایه جدید اشاره می‌کنه

نکته: حتماً خروجی append رو به همون متغیر **برگردون**، چون ممکنه یه slice جدید باشه:

```
s := []int{1, 2, 3}
s2 := append(s, 4)
// اگر ظرفیت s پر بود، s2 و s هنوز همون قبلیه به یه آرایه جدید اشاره می‌کنه
```

خلاصه کاری که append انجام می‌ده:

1. بررسی می‌کنه slice به آرایه‌ای اشاره می‌کنه یا نه
2. اگه نه، یه آرایه جدید درست می‌کنه
3. اگه جا داره، مقدار جدید رو اضافه می‌کنه
4. اگه جا نداره، آرایه جدید می‌سازه، داده‌های قبلی رو کپی می‌کنه و مقدار جدید رو اضافه می‌کنه
5. slice جدید (با آرایه جدید) رو برمی‌گردونه

الگوریتم رشد append چطوریه؟

وقتی با append عنصر جدید اضافه می‌کنی، Go اول بررسی می‌کنه:

```
if len(slice) < cap(slice) {
    // جا هست: عنصر رو به همون آرایه اضافه کن
} else {
    // جا نیست: آرایه جدید بساز
}
```

اگر جا نباشه Go به شکل هوشمند (و بسته به نسخه کامپایلر و سایز آرایه) یه آرایه جدید با ظرفیت بیشتر می‌سازه. الگوریتم معمول اینه:

ظرفیت جدید = ظرفیت قبلی × 2

مثال

```
s := make([]int, 2, 2) // len = 2, cap = 2
fmt.Println(len(s), cap(s)) // 2 2
```

حالا اگر بخوایم یه عنصر جدید اضافه کنیم:

```
s = append(s, 100) // ظرفیت پر شده! پس Go یه آرایه جدید می‌سازه
fmt.Println(len(s), cap(s)) // 3 4
```

در این لحظه:

- Go یه **آرایه جدید با ظرفیت 4** می‌سازه (2 × 2)
- مقادیر [0] و [1] رو از آرایه قبلی به جدید کپی می‌کنه
- مقدار جدید 100 رو در [2] قرار می‌ده
- متغیر s حالا به این آرایه جدید اشاره می‌کنه

نکته: اینکه ظرفیت چند برابر می‌شه، بسته به شرایط ممکنه دقیقاً دو برابر نباشه. ولی در اکثر موارد، تا زمانی که cap کوچک‌تر از 1024 باشه، **تقریباً دو برابر** می‌شه.

خلاصه ساده

- append اول بررسی می‌کنه جا هست یا نه.
- اگر جا نباشه Go خودش آرایه‌ی جدید با ظرفیت بیشتر می‌سازه.
- داده‌های قبلی رو کپی می‌کنه.
- داده‌ی جدید رو اضافه می‌کنه.
- slice به آرایه‌ی جدید اشاره می‌کنه.
- معمولاً ظرفیت رو **دو برابر** می‌کنه برای افزایش راندمان.

4-مقدار دهی توسط تابع copy

قبل از اینکه روش مقداردهی به یک متغیر از نوع slice با استفاده از تابع copy رو توضیح بدیم، باید اول بدونیم که این تابع دقیقاً چیه و چرا اصلاً وجود داره.

تابع copy

وقتی که از عملگر = برای دو slice استفاده می‌کنیم، این کار فقط باعث می‌شه که هر دو متغیر به **یک آرایه مشترک در حافظه** اشاره کنن. در واقع هیچ "کپی واقعی" اتفاق نمی‌افته؛ فقط آدرس مشترک داده‌ها به اشتراک گذاشته می‌شه.

مثلا

```
a := []int{1, 2, 3}
b := a
b[0] = 99
fmt.Println(a) // خروجی: [99 2 3] ← چون هر دو به یک آرایه اشاره می‌کنن
```

پس برای مواقعی که بخوایم **کپی واقعی داده‌ها** رو داشته باشیم، یعنی یه slice جدید با داده‌های مستقل بسازیم، از تابع copy استفاده می‌کنیم.

روش کپی آرایه پویا به آرایه پویا

```
a := []int{1, 2, 3}
b := make([]int, len(a))
copy(b, a)

b[0] = 99

fmt.Println(a) // خروجی: [1 2 3] هیچ تغییری نکرد!
fmt.Println(b) // خروجی: [99 2 3] مستقل عمل کرد!
```

اول یه آرایه پویا جدید ساختیم، دقیقاً به همون اندازه‌ی آرایه‌ی a بعد با کمک تابع copy، همه‌ی داده‌ها رو از a ریختیم تو b

حالا دیگه این دو تا کاملاً جدا شدن، یعنی هر تغییری توی b بدی، اصلاً روی a تاثیری نمی‌ذاره و بالعکس.

نکته: دقت کن که قبل از اینکه از تابع copy استفاده کنی قبلش باید یه آرایه پویا برای مقصد کپی، ایجاد کنی

نکته:تابع copy فقط روی **slice**ها کار می‌کنه، نه روی آرایه‌های با طول ثابت یا سایر نوع‌ها.

به طور خلاصه:

1-اگه بخوایم **یک آرایه با طول ثابت** رو داخل **یه آرایه با طول ثابت دیگه** کپی کنیم، از همون **عملگر** = استفاده می‌کنیم. چون آرایه‌های ثابت در Go خودشون مقدار کامل (value) هستن و = باعث می‌شه یه کپی واقعی انجام بشه.

2-اما اگه بخوایم **یک آرایه پویا (slice)** رو داخل یه **sliceدیگه** کپی کنیم، دیگه = کافی نیست! چون = فقط آدرس رو کپی می‌کنه و باعث می‌شه هر دو slice به یه جای مشترک اشاره کنن. برای همین باید از **تابع copy**استفاده کنیم تا یه کپی واقعی از داده‌ها داشته باشیم.

البته دو حالت خاص دیگه هم داریم:

3-می‌خوایم یه **آرایه پویا بسازیم بر اساس یک آرایه با طول ثابت** اما نمی‌خوایم این آرایه پویا به همون آرایه با طول ثابت اشاره کنه! واقعاً می‌خوایم کپی انجام بدیم؛ جوری که **تغییر در آرایه پویا اثری روی آرایه ثابت نذاره و برعکس**.

4-می‌خوایم یه **آرایه با طول ثابت بسازیم بر اساس یک آرایه پویا** اما باز هم نمی‌خوایم این آرایه جدید به همون آرایه‌ای اشاره کنه که slice بهش اشاره می‌کنه! می‌خوایم کپی کامل انجام بشه؛ جوری که **تغییر در آرایه ثابت اثری روی slice نذاره و برعکس**.

جدول زیر این 4 روش رو به طور مختصر بازنویسی کرده

شماره	کپی از	کپی به	روش درست
1	آرایه با طول ثابت	آرایه با طول ثابت	عملگر =
2	آرایه پویا (slice)	آرایه پویا (slice)	تابع copy(dest, src)
3	آرایه با طول ثابت	آرایه پویا (slice)	عملگر [:]
4	آرایه پویا (slice)	آرایه با طول ثابت	ترکیب عملگر [:] و تابع copy(dest, src)

تا اینجا حالت‌های ۱ تا ۳ رو کامل توضیح دادیم و دیدیم، حالا بریم سراغ حالت 4

کپی از آرایه پویا (slice) به آرایه با طول ثابت

وقتی بخوایم **از یه آرایه پویا (slice) یه آرایه با طول ثابت** بسازیم، جوری که این دوتا کاملاً مستقل باشن و تغییر یکی روی اون یکی اثر نذاره. در این حالت، باید از **ترکیب [:] و تابع copy** استفاده کنیم.

```
slice := []int{1, 2, 3}
var arr [3]int


copy(arr[:], slice)
```

1. [:] یعنی کل آرایه‌ی arr رو به شکل یک slice ببین. چون تابع copy فقط روی slice کار می‌کنه.
2. copy(arr[:], slice) یعنی از slice کپی کن و بریز داخل آرایه arr

در نتیجه، حالا arr و slice دو ساختار کاملاً مستقل دارن. تغییر در یکی، هیچ اثری روی اون یکی نمی‌ذاره.

در واقع وقتی از عملگر [:] روی متغیر arr استفاده کردیم، یه slice ساختیم که **به همون آرایه‌ای اشاره می‌کنه که خود arr بهش اشاره می‌کرد.**

یعنی arr[:] فقط یه نمای پویا (slice) از آرایه‌ی arr ساخت، نه یه نسخه جدا از اون.

حالا وقتی با تابع copy، داده‌های یه slice دیگه رو ریختیم داخل این نمای arr[:]، در اصل محتوا رو مستقیماً داخل **آرایه‌ی اصلی arr** ریختیم. چون اون slice موقتی (arr[:]) به همون حافظه‌ای اشاره می‌کرد که arr اشاره می‌کرد.

خب حالا که با تابع copyآشنا شدیم، می‌دونیم چطور می‌تونیم یه **آرایه پویا جدید** بسازیم و با استفاده از copy، اون رو **مستقل از آرایه قبلی مقداردهی کنیم**.

برای اینکه موضوع بهتر جا بیفته، بیایم **همون مثال قبلی** رو دوباره اجرا کنیم،

```
package main

import "fmt"

func main() {
    var familyMembers []string =[]string{“arian”, “neda”, “omid”}
    var siblings []string = make([]string, len(familyMembers))
    copy(siblings, familyMembers)
    familyMembers[1] = “NEDA”
    fmt.Println(“اعضای خانواده:”, familyMembers)
    fmt.Println(“خواهران/برادران:”, siblings)
}
```

عملگرهای مجاز در آرایه با طول متغیر

عملگر =

وقتی دو تا slice رو با = به هم نسبت می‌دیم، Goنمیاد مقادیر رو کپی کنه؛ فقط کاری می‌کنه که هر دو به **یه جای حافظه** اشاره کنن.

```
package main


import "fmt"


func main() {
    var familyMembers []string = []string{“arian”, “neda”, “omid”}
    var siblings []string = familyMembers
    familyMembers[1] = “NEDA”


   fmt.Println(“خواهران/برادران:”, siblings)
}
```

پیش از اینکه عملگر == و =! بررسی کنیم نیاز هست با مقدار nil آشنا بشیم

مقدار nil

وقتی یه متغیر تعریف می‌کنیم ولی هنوز بهش مقدار ندیم، زبان Go خودش یه **مقدار اولیه** براش در نظر می‌گیره.

مثلاً:

- برای عدد صحیح (مثل int) مقدار پیش‌فرضش 0 هست.
- برای bool مقدار پیش‌فرض false هست.
- برای string مقدار پیش‌فرض "" (رشته‌ی خالی) هست.
- و برای بعضی نوع‌ها مثل slice، map، pointer، channel یا interface مقدار پیش‌فرضشون nil هست.

nil یعنی «هیچی»، یعنی **هیچ ارجاعی وجود نداره** .اینجوری تصور کن که یه آدرس خونه داریم، ولی هنوز به هیچ خونه‌ای اشاره نمی‌کنه. یه اشاره‌گر خالیه.

```
package main

import "fmt"

func main() {
    var s []int // هنوز به هیچ آرایه‌ای وصل نیست
    fmt.Println(s == nil) // true ✅ چون خالیه
}
```

عملگر ==

عملگر == در آرایه پویا تنها برای تشخیص nil بودن یا نبودن آن کاربرد دارد. به هیچ عنوان نمیشه دو تا آرایه پویا را با این عملگر با هم مقایسه کرد. باید به این نکته توجه ویژه داشته باشیم.

```
package main


import "fmt"


func main() {
    var arianGrades []float32
    fmt.Println(s == nil) // true ✅ چون خالیه
}
```

عملگر =!

عملگر =! در آرایه پویا تنها برای تشخیص nil بودن یا نبودن آن کاربرد دارد. به هیچ عنوان نمیشه دو تا آرایه پویا را با این عملگر با هم مقایسه کرد. باید به این نکته توجه ویژه داشته باشیم.

```
package main


import "fmt"


func main() {
    var arianGrades []float32 = []float32 {14.0, 18.0, 17.0}
    fmt.Println(s != nil) // true ✅ چون خالی نیست
}
```

عملگر ... (ellipsis)

تو زبانGo ، این عملگر ... بسته به جایی که استفاده می‌شه **دو معنی متفاوت** داره.

اینجا فقط با یکی‌ش کار داریم:

وقتی این عملگر رو درست بعد از یک آرایه با طول ثابت یا آرایه پویا قرار بدیم باعث میشه **مقادیر اون آرایه باز بشه!**

اما این **باز شدن** یعنی چی؟

فرض کن یک آرایه پویا داری و میخوای مقادیرشو دونه دونه پرینت کنی

خب ساده ترین روش اینه به شکل زیر مقادیرشو پرینت کنی

```
var arianGrades []any = []any {14.0, 18.0, 17.0}
fmt.Println(arianGrades[0], arianGrades[1], arianGrades[2])
```

خروجی

```
14 18 17
```

یه روش ساده تر اینه که نام اون slice رو بنویسی و درست بعد از اون، عملگر ... قرار بدی

```
var arianGrades []any = []any {14.0, 18.0, 17.0}
fmt.Println(arianGrades…)
```

خروجی

```
14 18 17
```

همونطور که دیدیم خروجی در هر دو حالت برابر هست.

اما استفاده از این عملگر راحت تره. چون دیگه نیاز نیست دونه دونه هر کدوم از عنصر های آرایه رو با عملگر ایندکس بخونیم و چاپ کنیم

این عملگر علاوه بر آرایه پویا رو آرایه با طول ثابتم کاربرد داره

ویژگی های آرایه پویا (slice)

تا اینجا یاد گرفتیم که slice یه چیزی شبیه آرایه‌ست، ولی منعطف‌تر و هوشمندتر. حالا بیایم ببینیم دقیقاً چه ویژگی‌هایی داره که باعث می‌شه با آرایه فرق کنه:

اندازه‌ی متغیر داره

بر خلاف آرایه که طولش ثابته، slice می‌تونه رشد کنه یا کوچیک بشه.

به یک آرایه در پشت‌صحنه اشاره می‌کنه

خود slice فقط یه نمایی از یه آرایه‌ی واقعی تو حافظه‌ست.

اگه چندتا slice به یه آرایه اشاره کنن، تغییر تو یکی روی بقیه هم تاثیر می‌ذاره!

از سه قسمت تشکیل شده

اشاره‌گر، طول، ظرفیت

مقدار اولیه‌اش می‌تونه nil باشه

یعنی اگه فقط تعریفش کنی ولی مقداری ندی، مقدارش nilمی‌شه.

می‌تونی از یه آرایه، Slice بسازی

با استفاده از عملگر [:]می‌تونی از کل یا بخشی از یه آرایه یا حتی slice دیگه، یه slice جدید بسازی.

با تابع copy می‌تونی کپی واقعی بسازی

اگه نخوای چند slice به یه جا اشاره کنن، می‌تونی از copyاستفاده کنی تا محتوا واقعاً جدا شه.

تفاوت آرایه با طول ثابت با آرایه پویا

ویژگی	آرایه با طول ثابت	آرایه پویا (slice)
نحوه تعریف	[3]int{1, 2, 3}	[]int{1, 2, 3}
اندازه	ثابت (غیرقابل تغییر)	متغیر (قابل رشد با append)
نوع داده	value type مقدار	reference type ارجاعی
رفتار هنگام انتساب با =	کپی کامل داده	فقط ارجاع (هر دو به یک داده اشاره می‌کنن)
مقدار اولیه (zero value)	تمام عناصر zero value	nil
تابع len()	تعداد عناصر آرایه	تعداد عناصر درslice
تابع cap()	ظرفیت = طول	ظرفیت از نقطه شروع تا انتهای آرایه پشت‌صحنه
قابلیت استفاده با append	ندارد	دارد
حافظه	کل داده داخلش ذخیره می‌شود	فقط یک اشاره‌گر به آرایه دارد
ساختار پشت صحنه	فقط داده	اشاره‌گر + طول + ظرفیت
مناسب برای	داده‌های با اندازه مشخص	داده‌های پویا و لیست‌های متغیر

حذف یک عنصر از آرایه پویا

حذف عنصر از آرایه پویا امکان پذیره اما با ترفند! در واقع در زبان GO به صورت پیشفرض دستوری برای حذف یک عنصر از آرایه پویا تعریف نشده.

با مفاهیمی که تا به الان یاد گرفتیم میتونیم این کارو به سادگی انجام بدیم.

فرض کنید یک آرایه پویا برای نگهداری نمرات دروس فارسی، دین و زندگی، زبان خارجی، تربیت بدنی، جبر، ریاضیات گسسته، حساب دیفرانسیل، فیزیک و هندسه که متعلق به ندا هست به شکل زیر تعریف کردیم

```
var nedaGrades []float32 = []float32 {
    19.50, 11.00, 20.00, 16.50, 17.50, 18.00, 16.25, 18.50, 19.00,
}
```

بنابه دلایلی نیاز داریم که یک آرایه جدید داشته باشیم که تمام نمرات به جز نمره تربیت بدنی رو در اون داشته باشیم.

خب شاید یکی از راحت ترین راه ها این باشه که بیایم یه آرایه جدید بسازیم و با عملگر ایندکس اون نمره هایی که نیاز داریم بمونن در آرایه جدید مقداردهی کنیم. یعنی به شکل زیر

```
var nedaGrades []float32 = []float32 {
    19.50, 11.00, 20.00, 16.50, 17.50, 18.00, 16.25, 18.50, 19.00,
}

var excluded []float32 = []float32 {
    nedaGrades[0], // فارسی
    nedaGrades[1], // دین و زندگی
    nedaGrades[2], // زبان خارجی
    nedaGrades[4], // جبر
    nedaGrades[5], // ریاضیات گسسته
    nedaGrades[6], // حساب دیفرانسیل
    nedaGrades[7], // فیزیک
    nedaGrades[8], // هندسه
}
```

اما این روش ایرادات زیادی داره

1-روش دستیه و وابسته به برنامه نویسه

توی این روش، خودت باید دونه‌دونه اون ایندکس‌هایی رو بنویسی که می‌خوای تو آرایه جدید بریزی.
برای یه لیست کوتاه مثل همین نمرات ندا شاید قابل تحمله،
ولی اگه لیست 100 تا نمره باشه چی؟ یا اگه بخوای چندتا نمره حذف کنی؟

روش دستی تو مقیاس کوچیک جواب می‌ده، ولی تو پروژه‌های واقعی اصلاً به‌صرفه نیست.

2-کد تکراری و شکننده

مثلاً اگه بعداً یکی بیاد ترتیب نمره‌ها رو عوض کنه، یا بینشون یه درس جدید اضافه کنه، کد تو می‌تونه نمره اشتباه رو نگه داره یا حذف کنه — چون داری با ایندکس‌ها به صورت **هاردکُد شده** کار می‌کنی.

3- قابل تعمیم نیست

اگه بعداً خواستی به‌جای حذف تربیت بدنی، نمره «زبان خارجی» رو حذف کنی، باید بری دوباره ایندکس‌ها رو یکی‌یکی اصلاح کنی.

راه بهتر چیه؟

ترکیب تابع append و عملگر [:]

```
var nedaGrades []float32 = []float32 {
    19.50, 11.00, 20.00, 16.50, 17.50, 18.00, 16.25, 18.50, 19.00,
}

excluded := append(nedaGrades[:3], nedaGrades[4:]...)
```

nedaGrades[:3] یعنی چی؟

این یعنی **از ابتدای slice تا قبل از اندیس 3** رو بردار.
در واقع این قسمت برمی‌گردونه:

```
[]float32{19.50, 11.00, 20.00}
```

یعنی نمرات فارسی، دین و زندگی، و زبان خارجی — درست قبل از تربیت بدنی.

nedaGrades[4:] یعنی چی؟

این یعنی **از اندیس 4 به بعد تا آخر slice** رو بردار.
در واقع این قسمت برمی‌گردونه:

```
[]float32{17.50, 18.00, 16.25, 18.50, 19.00}
```

پس چی شد؟ حالا دو تکه داریم:

1. nedaGrades[:3] تکه‌ی قبل از تربیت بدنی
2. nedaGrades[4:] تکه‌ی بعد از تربیت بدنی

حالا append(nedaGrades[:3], nedaGrades[4:]...) داره چیکار می‌کنه؟

```
append(before, after...)
```

حالا اینجا یه نکته‌ی خیلی مهم وجود داره:

چرا از ... استفاده کردیم؟

- چون append انتظار داره **چند تا مقدار جدا جدا** بگیره.
- ولی nedaGrades[4:] خودش یه **slice** هست.
- برای اینکه Go بتونه اعضای اون slice رو **یکی‌یکی** به append بده، باید از ... استفاده کنیم تا اون رو باز کنه.

بدون ... ارور می‌گیریم:

```
append(nedaGrades[:3], nedaGrades[4:]) // ✕ غلط! چون یه slice داریم نه چند مقدار جدا
```

با ... درست کار می‌کنه:

```
append(nedaGrades[:3], nedaGrades[4:]...) // ✓ درسته
```

این یعنی همه عناصر nedaGrades[4:] رو باز کن و به صورت جدا جدا به append بده

نکته مهم درباره تابع append

تابع append در Go می‌تونه **بیش از یک مقدار** بگیره:

```
append(slice, 1, 2, 3) // ☜ سه مقدار جدید اضافه می‌کنه
```

و اگه بخوای یه slice کامل رو بهش اضافه کنی، باید ... بذاری:

```
append(slice, otherSlice...) // ☜ رو باز می‌کنه و اضافه می‌کنه otherSlice اعضای
```

نتیجه نهایی

```
excluded := append(nedaGrades[:3], nedaGrades[4:]...)
```

این خط کد، یه slice جدید درست می‌کنه که توش **همه‌ی نمره‌ها هستن به جز تربیت بدنی**.

خروجی slice جدید:

```
fmt.Println(excluded) // [19.5 11 20 17.5 18 16.25 18.5 19]
```

خطر مهم: اشتراک حافظه بین Slice ها

وقتی این کد رو می‌نویسی:

```
excluded := append(nedaGrades[:3], nedaGrades[4:]...)
```

ممکنه excluded و nedaGrades **هر دو به یک آرایه (underlying array)** اشاره کنن!

یعنی اگه یکی از مقادیر excluded رو تغییر بدی، ممکنه مقدار nedaGrades هم تغییر کنه — و برعکس!

چرا گفتیم ممکنه excluded و nedaGrades هر دو به یک آرایه اشاره کنن؟

چون تابع append **در برخی شرایط از آرایه‌ی قبلی استفاده می‌کنه و در برخی شرایط آرایه‌ی جدید می‌سازه**.
همه‌چی بستگی داره به اینکه **ظرفیت (capacity)** اون slice اولیه چقدره.

فرض کن slice زیر رو داریم:

```
grades := []int{10, 20, 30, 40, 50}
```

حالا اینو می‌نویسیم:

```
cut := append(grades[:2], grades[3:]...)
```

در اینجا Go می‌گه:

- آیا فضای کافی (capacity) در grades[:2] وجود داره که بتونم عناصر grades[3:] رو هم بریزم اون تو؟
- اگه **ظرفیت کافی باشه** append همون آرایه‌ی قبلی رو استفاده می‌کنه
- اگه **ظرفیت کافی نباشه** append یه آرایه‌ی جدید می‌سازه و داده‌ها رو می‌بره اونجا

پس در نتیجه:

شرایط	اتفاقی که میوفته
ظرفیت کافی باشه	حافظه‌ی مشترک باقی می‌مونه -> خطر تغییر دوطرفه
ظرفیت کافی نباشه	حافظه‌ی جداگانه ساخته می‌شه -> تغییرات مستقل

چرا آرایه پویا ساختارش با آرایه ثابت فرق داره؟

ممکنه برات سوال پیش بیاد که چرا از همون ساختار آرایه با طول ثابت برای آرایه پویا استفاده نکردیم؟

وقتی آرایه با طول ثابت می‌سازی، طولش رو همون لحظه مشخص می‌کنی. اما اگر بخوای طول آرایه رو با یک متغیر تعیین کنی، مثلاً اینطوری:

```
var numberOfItems int
fmt.Scanln(&numberOfItems)
var grades [numberOfItems]float32
```

متأسفانه تو زبان Go این امکان وجود نداره. آرایه با طول متغیر رو نمی‌شه اینطوری تعریف کرد و باید از همون روش آرایه پویا استفاده کنی که قبلاً بررسی کردیم.

حالا چرا این محدودیت وجود داره؟
چون سیاست‌های زبان Go به این شکل است:

- **شفافیت کد:**
  Go می‌خواد دقیقاً بدونی داری چی می‌نویسی T[n] یعنی آرایه‌ای با اندازه ثابت که تو زمان کامپایل مشخصه.
  اگر اجازه داده می‌شد که T[numberOfItems] باشه و numberOfItems متغیر باشه، کد خیلی مبهم و گیج‌کننده می‌شد.
- **پیشگیری از باگ و رفتار غیرمنتظره:**
  اگر Go خودش به صورت خودکار تشخیص بده که آرایه باید ثابت باشه یا پویا، ممکنه کد رفتاری نشون بده که فهمیدنش سخت باشه و خطاهای پنهانی ایجاد کنه.
- **سادگی کامپایلر و ابزارها:**
  کامپایلر و ابزارهای تحلیل کد راحت‌تر و سریع‌تر کار می‌کنن وقتی معنی سینتکس‌ها واضح باشه و دو نوع آرایه با دو ساختار جداگانه تعریف بشن.

آرایه ثابت یا آرایه پویا؟ کدوم رو استفاده کنیم؟

یه نکته خیلی مهم که باید بدونی اینه که اگر جایی می‌تونی از **آرایه با طول ثابت** استفاده کنی، بهتره همون رو انتخاب کنی و سمت آرایه پویا نری!

چرا؟ دلیل اصلی این جدا شدن ساختار آرایه ثابت و آرایه پویا **بهینه بودن اجرای برنامه** هست.

وقتی آرایه با طول ثابت می‌سازی

کامپایلر دقیقاً در زمان کامپایل می‌دونه که آرایه چقدر فضا نیاز داره. به همین خاطر، اون فضا رو **همون موقع کنار میذاره**، یعنی حافظه‌ای روی **stack** رزرو می‌کنه.

این باعث میشه وقتی برنامه اجرا میشه، حافظه برای اون آرایه به‌صورت کاملاً سریع و یکباره اختصاص پیدا کنه. پس برنامه در حالت آرایه ثابت سریع‌تر اجرا میشه.

وقتی آرایه پویا می‌سازی

چون طول آرایه تو زمان اجرا مشخص میشه و معلوم نیست چقدر باید حافظه کنار بذاره، کامپایلر نمی‌تونه در زمان کامپایل حافظه رزرو کنه. در نتیجه حافظه تو **heap** اختصاص پیدا می‌کنه که کمی کندتره، چون باید در زمان اجرا این حساب و کتاب‌ها انجام بشه و حافظه به صورت پویا تخصیص داده بشه.

پس اگه هر دو نوع آرایه (ثابت و پویا) رو با یک روش واحد می‌ساختن، اون وقت آرایه ثابت هم کندتر می‌شد. اما جدا کردن این دو ساختار باعث شده برنامه‌هایی که آرایه ثابت دارن سریع‌تر اجرا بشن و در کل Go زبانی بهینه‌تر و سریع‌تر باشه.

پس اینو همیشه یادت باشه:

وقتی می‌تونی آرایه با طول ثابت استفاده کنی، حتما ازش استفاده کن؛ چون هم سریع‌تره و هم کاراتر. آرایه پویا وقتی لازمه که طولش رو دقیق ندونی و بخوای در زمان اجرا تغییرش بدی.

آرایه های تو در تو پویا

فرض کن می‌خوای اطلاعات مربوط به نمرات چند تا دانش‌آموز رو توی برنامه ذخیره کنی. ولی یه مشکل هست:
نه تعداد دانش‌آموزا رو از قبل می‌دونی، نه تعداد درس‌هاشون رو!
می‌خوای برنامه‌ت اونقدر انعطاف‌پذیر باشه که کاربر (کسی که برنامه رو اجرا می‌کنه) **خودش مشخص کنه** که:

- قراره نمرات چند دانش‌آموز رو وارد کنه؟
- و برای هر دانش‌آموز، چندتا نمره قراره ذخیره بشه؟

خب توی چنین شرایطی، **بهترین انتخاب استفاده از آرایه‌های تو در تو هست**.
البته نه هر آرایه‌ای! چون اینجا تعداد سطرها (دانش‌آموزها) و ستون‌ها (نمرات هر نفر) **از قبل معلوم نیست** و فقط **در زمان اجرا** مشخص میشه، پس باید از آرایه‌های **پویا** استفاده کنیم — هم برای خود آرایه‌ی اصلی، هم برای آرایه‌های داخلی.

روش تعریف آرایه تو در تو پویا

همون‌طور که تو درس‌های قبلی دیدیم، فقط تعریف یه متغیر از نوع slice کافی نیست.
ما باید:

1. **یه متغیر تعریف کنیم** برای نگه‌داشتن اطلاعات
2. با استفاده از make (یا روش های جایگزین)، یه آرایه‌ی پویا بسازیم و **اون متغیر بهش اشاره کنه**

برای آرایه‌های تو در تو (مثلاً برای نگه داشتن نمرات چند دانش‌آموز)، باید یه متغیر از نوع slice دو بعدی بسازیم، یعنی:

```
var name [][]type
```

اما این فقط یه متغیره، هنوز حافظه‌ای واسش ایجاد نشده!
پس باید با make بیایم و برای سطرهای این آرایه حافظه رزرو کنیم:

```
var name [][]type = make([][]type, rows_length)
```

بخش‌های مختلف این تعریف:

- var کلمه کلیدی برای تعریف متغیر
- name اسم متغیرمون؛ هر اسمی می‌تونی بذاری (مثلاً grades)
- type[][] یعنی این یه آرایه‌ی تو در توئه، هر عنصرش یه slice دیگه‌ست
- type نوع داده‌ست (مثلاً int یا float32) باید برای همه‌ی عناصر یکی باشه
- rows_length تعداد سطرها (یعنی چندتا دانش‌آموز داریم) که باید حتماً یه عدد صحیح مثبت باشه

مثال:
اگه rows_length رو بذاریم 4 یعنی قراره نمرات 4 دانش‌آموز رو ذخیره کنیم.
هر دانش‌آموز خودش یه لیست از نمرات داره، که اون‌ها هم قراره آرایه باشن.

مرحله‌ی بعد: ساخت آرایه‌های داخلی

تا اینجا فقط یه آرایه‌ی پویا داریم که قراره نمرات هر دانش‌آموز توش باشه.
حالا باید بیایم و **برای هر دانش‌آموز (یعنی هر سطر)** یه آرایه‌ی جدا بسازیم (برای نمرات اون دانش‌آموز).
اینو با یه حلقه for انجام می‌دیم:

```
for i := 0; i < rows_length; i++ {
    name[i] = make([]type, columns_length)
}
```

اینجا:

- columns_length یعنی تعداد نمراتی که هر دانش‌آموز قراره داشته باشه
- همه‌ی آرایه‌های داخلی **هم‌اندازه هستن**، مثلاً هر کدوم 5 نمره دارن

این ساختار دقیقاً همون چیزیه که نیاز داریم برای ساخت آرایه‌های تو در تو با اندازه‌های پویا.
نه فقط برای نمرات دانش‌آموزا، بلکه برای خیلی از موقعیت‌های مشابه دیگه تو دنیای برنامه‌نویسی Go هم کاربرد داره.

برنامه محاسبه میانگین نمرات

فرض کن می‌خوای برنامه‌ای بنویسی که نمرات چند دانش‌آموز رو بگیره و میانگین نمرات هرکدوم رو حساب کنه. اما از قبل نمی‌دونی چندتا دانش‌آموز داریم یا هر دانش‌آموز چندتا نمره داره!

می‌خوای برنامه‌ت انقدر منعطف باشه که کاربر موقع اجرا مشخص کنه:

- چند دانش‌آموز داریم
- برای هر دانش‌آموز چند نمره باید وارد بشه

هدف برنامه:

- تعداد دانش‌آموزها رو از ورودی بگیره
- تعداد نمرات هر دانش‌آموز رو هم از ورودی بگیره (و برای همه برابر در نظر بگیره)
- نمرات رو از ورودی بگیره
- میانگین نمرات هر دانش‌آموز رو محاسبه کنه
- در پایان میانگین‌ها رو چاپ کنه

مرحله اول: گرفتن ورودی‌ها و ساخت آرایه تو در تو

برای نگهداری چنین اطلاعاتی، آرایه تو در تو پویا بهترین انتخابه. اما نه یه آرایه‌ای که تعداد سطر و ستونش از قبل معلوم باشه، بلکه آرایه‌ای که این ابعادش موقع اجرای برنامه از کاربر گرفته بشه.

خب، همون‌طور که قبلاً یاد گرفتیم، برای ساخت یک آرایه تو در تو پویا باید:

1. اول یک متغیر از نوع [][]type تعریف کنیم
2. بعد با استفاده از make، آرایه بیرونی (سطرها) رو بسازیم
3. و در نهایت با استفاده از یک حلقه for، برای هر سطر (هر دانش‌آموز)، آرایه داخلی (ستون‌ها، یعنی نمرات) رو ایجاد کنیم

مثلاً تا اینجای برنامه این‌جوری میشه:

```
var numberOfStudents int
var numberOfGrades int

fmt.Print("چند دانش‌آموز داری؟ ")
fmt.Scanln(&numberOfStudents)

fmt.Print("برای هر دانش‌آموز چند نمره وارد می‌کنی؟ ")
fmt.Scanln(&numberOfGrades)

var grades [][]float32 = make([][]float32, numberOfStudents)

for i := 0; i < numberOfStudents; i++ {
    grades[i] = make([]float32, numberOfGrades)
}
```

مرحله دوم: دریافت نمرات

حالا که آرایه‌ی دوبعدی (grades) رو ساختیم با استفاده از make وقتشه که نمره‌ها رو از کاربر بگیریم.

برای این کار از دو حلقه‌ی تو در تو استفاده می‌کنیم:

- حلقه‌ی بیرونی (i) به تعداد دانش‌آموزها (numberOfStudents) اجرا میشه.
- حلقه‌ی داخلی (j) برای گرفتن نمرات هر دانش‌آموز (numberOfGrades) اجرا میشه.

با استفاده از این حلقه‌ها، نمره‌ها رو یکی‌یکی از کاربر می‌گیریم و داخل grades[i][j] قرار می‌دیم:

```
for i := 0; i < numberOfStudents; i++ {
    for j := 0; j < numberOfGrades; j++ {
        fmt.Printf("نمره %d از دانش‌آموز %d رو وارد کن: ", j+1, i+1)
        fmt.Scanln(&grades[i][j])
    }
}
```

مرحله سوم: محاسبه‌ی میانگین نمرات

بعد از اینکه همه‌ی نمره‌ها رو دریافت کردیم، حالا نوبت محاسبه‌ی میانگین نمرات هر دانش‌آموزه.

برای این کار یک آرایه‌ی جدید به اسم averages درست می‌کنیم که توش میانگین هر دانش‌آموز رو ذخیره کنیم. از اونجایی که به ازای هر دانش‌آموز فقط یک عدد میانگین داریم، پس این آرایه تک‌بعدی به طول numberOfStudents خواهد بود.

```
averages := make([]float32, numberOfStudents)
```

حالا با یه حلقه‌ی ساده روی دانش‌آموزها، میانگین هر کدوم رو حساب می‌کنیم:

```
for i := 0; i < numberOfStudents; i++ {
    var sum float32 = 0
    for j := 0; j < numberOfGrades; j++ {
        sum += grades[i][j]
    }
    averages[i] = sum / float32(numberOfGrades)
}
```

در این حلقه:

- قبل از شروع جمع زدن نمرات، متغیر sumرو صفر می‌کنیم.
- با حلقه‌ی داخلی jروی نمرات دانش‌آموز iحرکت می‌کنیم و هر نمره رو به sumاضافه می‌کنیم.
- در نهایت، جمع نمرات رو بر تعداد نمرات تقسیم می‌کنیم و نتیجه رو در averages[i] می‌ذاریم.

مرحله چهارم: نمایش میانگین نمرات

تا اینجای کار، همه‌ی نمره‌ها رو گرفتیم و میانگین هر دانش‌آموز رو داخل آرایه‌یaverages ذخیره کردیم. حالا وقتشه که این میانگین‌ها رو برای کاربر نمایش بدیم.

برای این کار فقط به یک حلقه‌ی ساده نیاز داریم که روی آرایه‌ی averages حرکت کنه و مقدار میانگین هر دانش‌آموز رو چاپ کنه:

```
for i := 0; i < numberOfStudents; i++ {
    fmt.Println("میانگین نمرات دانش‌آموز", i+1, ": ", averages[i])
}
```

- این حلقه از 0 تا numberOfStudents تکرار میشه.
- چون شماره دانش‌آموز رو میخوایم از 1 شروع کنیم i+1 رو چاپ می‌کنیم

```
var numberOfStudents, numberOfGrades int

fmt.Print("چند دانش‌آموز داری؟ ")
fmt.Scanln(&numberOfStudents)
fmt.Print("برای هر دانش‌آموز چند نمره وارد می‌کنی؟ ")
fmt.Scanln(&numberOfGrades)

var grades [][]float32 = make([][]float32, numberOfStudents)
for i := 0; i < numberOfStudents; i++ {
    grades[i] = make([]float32, numberOfGrades)
}

for i := 0; i < numberOfStudents; i++ {
    for j := 0; j < numberOfGrades; j++ {
        fmt.Printf("نمره %d از دانش‌آموز %d رو وارد کن: ", j+1, i+1)
        fmt.Scanln(&grades[i][j])
    }
}

averages := make([]float32, numberOfStudents)
for i := 0; i < numberOfStudents; i++ {
    var sum float32 = 0
    for j := 0; j < numberOfGrades; j++ {
        sum += grades[i][j]
    }
    averages[i] = sum / float32(numberOfGrades)
}

for i := 0; i < numberOfStudents; i++ {
    fmt.Println("میانگین نمرات دانش‌آموز", i+1, ": ", averages[i])
}
```

تمرین 1: 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.

انتخاب اندازه صفحه بازی

در ابتدای بازی، از کاربر خواسته میشه که اندازه جدول بازی رو وارد کنه.
این عدد مشخص می‌کنه که صفحه‌ی بازی چند در چند باشه.
کاربر فقط اجازه داره یکی از این عددها رو وارد کنه:

- گزینه 1: صفحه 3*3 (مجموعاً 9 خونه)
- گزینه 2: صفحه 4*4 (مجموعاً 16 خونه)
- گزینه 3: صفحه 5*5 (مجموعاً 25 خونه)

برای مثال، اگه کاربر عدد 2 رو وارد کنه، بازی با یه جدول 4 در 4 شروع میشه.
در این حالت شرط برنده شدن :**اولین بازیکنی که 4 تا علامت پشت سر هم بچینه، برنده‌ست**.

راهنمای نوشتن کد بازی

ساخت صفحه بازی

- از یه آرایه دوبعدی پویا با نوع string[][] استفاده کن
- اندازه‌ی این آرایه رو از ورودی کاربر بگیر و بر اساس اون آرایه رو بساز (مثلاً اگه کاربر عدد 3 داد، باید یه آرایه 3*3 بسازی)

نمایش صفحه بازی

نیازی نیست که کل صفحه بازی رو رسم کنی یا شکل خاصی نشون بدی.
کافیه در هر نوبت، از بازیکن بپرسی که قصد داره علامتش رو توی کدوم «سطر» و «ستون» قرار بده.
قبل از قرار دادن علامت، چک کن که اون خونه خالیه یا نه:

- اگه خونه خالی بود، علامت بازیکن رو اونجا بذار و نوبت رو به بازیکن بعدی بده
- اگه خونه قبلاً پر شده بود، باید یه پیام خطا نشون بدی و از همون بازیکن بخوای که یه جای دیگه انتخاب کنه

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.

تمرین 2: مدیریت لیست دوستان

برنامه‌ای بنویس که کارهای زیر رو انجام بده:

1. اول از کاربر بپرسه که چند تا دوست داره (مثلاً ۵ نفر)
2. بعد اسم همه‌ی اون دوست‌ها رو از ورودی بگیره و در یک آرایه‌ی پویا (slice) ذخیره کنه
3. حالا از کاربر بپرسه:
   با کدوم دوستت قهر کردی؟
   اسم اون دوست رو بگیره و از slice حذفش کنه
4. بعد از اون بپرسه:
   کدوم دوستت صمیمی‌ترین رفیقته؟
5. اسم اون دوست رو بگیره و اسمش رو **با حروف بزرگ** (capital) در همون لیست تغییر بده
6. در نهایت، **هر دو لیست** رو چاپ کن:
   - یکی: لیست اصلی دوستان (بعد از تغییرات)
   - یکی: لیستی که فقط اسم دوستی که قهر کردی ازش حذف شده

آیا اسم دوست صمیمی، توی هر دو لیستی که چاپ می‌شن **با حروف بزرگ** دیده میشه یا نه؟ تحلیل خودتو بنویس