آموزش زبان برنامه نویسی RUST

با تاکید بر کاربردها

زمستان 403
دانشکده مهندسی کامپیوتر
دانشگاه اصفهان

مقدمه‌ای بر زبان برنامه‌نویسی Rust 0
مقایسه با زبان‌های دیگر 0
چرا Rust؟ 0
نصب محیط توسعه Rust 0
ابزارهای مورد نیاز برای توسعه در Rust 0
اولین برنامه‌ی Rust 0
کدهای اولیه Rust 0
تعریف یک متغیر و چاپ آن 0
استفاده از متغیرهای ثابت 0
تعریف انواع داده‌ها در Rust 0
عدد صحیح (Integer) 0
اعداد اعشاری (Floating Point) 0
کاراکترها (Character) 0
بولین (Boolean) 0
Tuple 0
آرایه‌ها (Arrays) 0
تعریف متغیرها در Rust 0
عملیات روی انواع داده 0
تعریف و استفاده از متغیرهای ثابت 0
استفاده از متغیرهای نوع ثابت (Static Variables) 0
عملکردهای ورودی و خروجی در Rust 0
عملکرد توابع در Rust 0
تعریف یک تابع 0
پارامترها و نوع داده‌ها 0
نوع بازگشتی تابع 0
توابع بدون نیاز به بازگشت (Functions without return) 0

استفاده از کلمه return در توابع 0
استفاده از return برای بازگشت مقدار 0
پیش‌فرض بازگشت بدون return 0
استفاده از return در توابع بدون بازگشت مقدار 0
توابع با مقادیر پیش‌فرض 0
توابع درون‌خطی (Inline Functions) 0
توابع در Rust در مقابل C 0
مقداردهی متغیرها در توابع 0
مقداردهی مستقیم به متغیرها 0
مقداردهی از نوع‌های مختلف 0
انتقال و واگذاری مالکیت 0
داده‌های اصلی (Copy types) 0
داده‌های غیر اصلی (Non-Copy types) 0
استفاده از ارجاع‌ها (References) 0
ارجاع‌های قابل تغییر (Mutable References) 0
ساختارهای کنترلی در Rust 0
دستور if 0
دستور if-else 0
دستور if به عنوان عبارت 0
توجه به نوع‌های بازگشتی در if 0
دستور match 0
دستور match به عنوان عبارت 0
توجه به نوع بازگشتی در match 0
استفاده از match برای مقادیر پیچیده‌تر 0
الگوهای پیچیده‌تر در match 0
الگوهای ترکیبی و چندگانه (Multiple Patterns) 0
گارد شرطی در (Match Guards) match 0
حلقه‌ها در زبان راست (Rust) 0
حلقه loop 0
بازگشت مقدار از حلقه loop 0
حلقه while 0
حلقه for 0
پیمایش محدوده اعداد 0

پیمایش آرایه‌ها 0
پیمایش با enumerate 0
دستورات continue و break 0
دستور continue 0
دستور break 0
حلقه‌های تو در تو و برچسب‌گذاری 0
مثال عملی: جستجو در آرایه 0
مثال عملی: محاسبه فیبوناچی 0
رشته‌ها در Rust 0
تعریف و ایجاد رشته‌ها 0
ایجاد رشته‌های :String 0
تغییر رشته‌ها 0
اضافه کردن به رشته: 0
دسترسی به رشته‌ها 0
حلقه زدن روی کاراکترها: 0
حلقه زدن روی بایت‌ها: 0
String Slices (&str) : 0
String Slice چیست؟ 0
تفاوت با :String 0
ایجاد :String Slice 0
از :String literals 0
از String (با استفاده از slicing): 0
انواع Slice ها: 0
کاربردهای String Slice 0
متدهای رایج رشته‌ها و تبدیل نوع در Rust 0
تبدیل نوع به رشته و برعکس 0
Structs (ساختارها) : 0
تعریف ساختار (Defining Structs) 0
ایجاد نمونه از ساختار (Instantiating Structs) 0
دسترسی به فیلدهای ساختار (Accessing Struct Fields) 0
Enum (شمارشگرها) 0

تعریف شمارشگر (Defining Enums) 0

گونه‌های Enum با داده (Enum Variants with Data) 0

استفاده از Enum (Using Enums) 0

Enum Option 0

استفاده از match با Enum 0

match غیر جامع (Non-Exhaustive Match): 0

پردازش خطا و مقادیر اختیاری در Rust 0

۱. Option<T>: مقادیر اختیاری 0

مفهوم T (جنریک ساده): 0

تعریف Option<T>: 0

هشدار: unwrap و expect: 0

مفاهیم T و E (جنریک ساده): 0

تعریف Result<T, E>: 0

unwrap و expect برای Result: 0

عملگر ? (The Question Mark Operator): 0

مجموعه Vector 0

Vector (Vec<T>) 0

ایجاد یک Vector 0

اضافه کردن عناصر به Vector 0

دسترسی به عناصر Vector 0

حذف عناصر از Vector 0

پیمایش (Iteration) روی Vector 0

سایر متدهای مفید Vector 0

قوانین مالکیت و قرض گرفتن با Vector (یادآوری) 0

ذخیره انواع مختلف در Vector با استفاده از Enum 0

مثال‌های کاربردی از Vector 0

مثال 1: محاسبه مجموع و میانگین اعداد در یک Vector 0

مثال 2: ذخیره و پردازش لیستی از نام‌ها 0

مثال 3: فیلتر کردن عناصر Vector و ساخت Vector جدید 0

کلمه کلیدی impl در Rust 0

تعریف متدها برای ساختارها 0

توابع مرتبط (Associated Functions) 0
تعریف متدها برای شمارشگرها 0
کلوژرها (Closures) در Rust 0
سینتکس پایه کلوژر: 0
کلوژرها و محیط اطراف: 0
ایتریتورها (Iterators) در Rust 0
متد .:()iter 0
متدهای ایتریتور: filter 0
استفاده از :filter 0
متدهای ایتریتور: collect 0
3. take(n) 0
4. skip(n) 0
5. enumerate() 0
Traitها در Rust 0
تعریف یک Trait 0
پیاده‌سازی یک Trait برای یک نوع داده 0
استفاده از Traitها به عنوان پارامتر 0
مثال کامل: Trait برای اشکال هندسی (مساحت و محیط) 0
کار با فایل‌ها در Rust 0
۳. نوشتن در فایل‌ها (Writing to Files) 0
مثالی از یک پایگاه داده ساده با هش مپ 0
مثالی از یک پایگاه داده cli 0
طول عمر (Lifetimes) 0
مثال ۱: طول عمر حدس زده شده توسط کامپایلر (Compiler Inferred Lifetime) 0
مثال ۲: نیاز به مشخص کردن صریح طول عمر (Explicit Lifetime Annotation) 0
طول عمر 'static (The 'static Lifetime) 0
برنامه‌نویسی ناهمزمان (Asynchronous Programming) در Rust 0
1. Future (آینده) 0

2. async/await 0
3. Executor (اجراکننده) 0
استفاده از spawn برای اجرای مستقل 0
ماژول‌ها و کریت‌ها در Rust 0
کریت چیست؟ 0
انواع کریت 0
ماژول چیست؟ 0
تعریف ماژول‌ها (با کلمه کلیدی mod) 0
قابلیت مشاهده (Visibility با کلمه کلیدی pub) 0
مسیرها (Paths) 0
وارد کردن مسیرها (با کلمه کلیدی use) 0
استفاده از کریت‌های خارجی 0
خلاصه: سازماندهی فایل‌ها 0

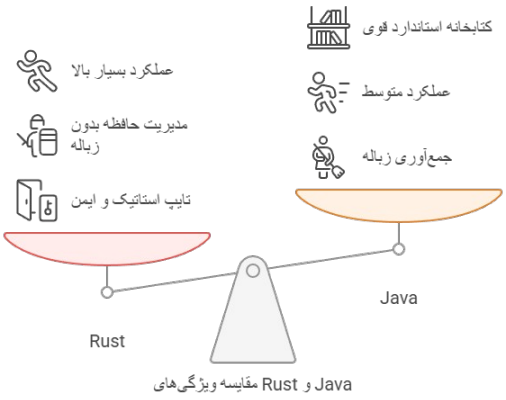
مقدمه‌ای بر زبان برنامه‌نویسی Rust

Rust یک زبان برنامه‌نویسی مدرن و ایمن است که برای برنامه‌نویسی سیستم‌ها طراحی شده است. این زبان به دلیل مدیریت حافظه‌ای امن و کارایی بالا، در سال‌های اخیر محبوبیت زیادی پیدا کرده است.

مقایسه با زبان‌های دیگر

ویژگی	Rust	Java	Python	C++	Go
نوع سیستم تایپ	استاتیک، ایمن	استاتیک، ایمن	داینامیک	استاتیک	استاتیک
مدیریت حافظه	بدون جمع‌آوری زباله، مالکیت و قرض گرفتن	جمع‌آوری زباله	جمع‌آوری زباله	بدون جمع‌آوری زباله، اشاره‌گرها	جمع‌آوری زباله
عملکرد	بسیار بالا	متوسط	پایین	بسیار بالا	بالا
حمایت از اشاره‌گرها	بله (با مدیریت ایمن)	خیر	خیر	بله	خیر
کتابخانه استاندارد	بسیار قوی	بسیار قوی	بسیار قوی	بسیار قوی	متوسط
سینتکس	مشابه C++	مشابه C	مشابه	مشابه C	مشابه C

ویژگی	Rust	Java	Python	++C	Go
			Python		



مقایسه ویژگی‌های Rust و Java

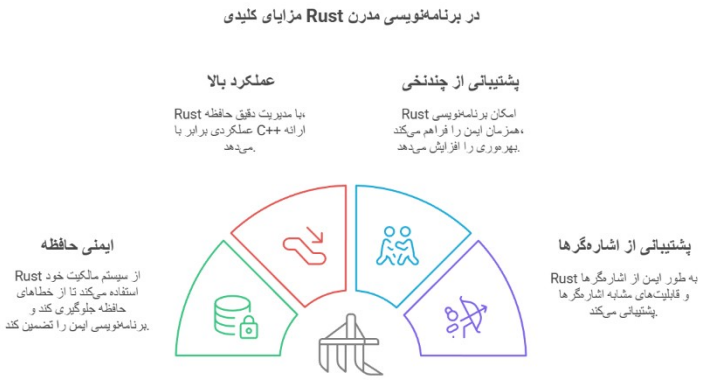
چرا Rust؟

۱- ایمنی حافظه: Rust به شما اجازه می‌دهد که بدون نگرانی از خطاهای مربوط به حافظه، برنامه بنویسید. با استفاده از سیستم مالکیت، اشاره‌گرهای خطرناک از بین می‌روند.

۲- عملکرد بالا: Rust به اندازه C++ سریع است، زیرا بدون جمع‌آوری زباله، مدیریت حافظه به صورت دقیق انجام می‌شود.

۳- پشتیبانی از چندنخی: Rust به شما امکان نوشتن برنامه‌های ایمن و همزمان را می‌دهد.

۴- پشتیبانی از اشاره‌گرها: Rust به طور ایمن از اشاره‌گرها و قابلیت‌های مشابه اشاره‌گرها پشتیبانی می‌کند.



نصب محیط توسعه Rust

برای نصب Rust، باید مراحل زیر را دنبال کنید:

۵- به وب‌سایت رسمی Rust بروید: https://www.rust-lang.org/tools/install
۶- برنامه نصب rustup را دانلود و اجرا کنید. rustup مدیر نسخه‌های Rust است.
۷- پس از نصب، برای اطمینان از نصب صحیح، دستور زیر را در ترمینال وارد کنید:
۸- rustc --version

این دستور باید نسخه‌ای از Rust را نمایش دهد.

ابزارهای مورد نیاز برای توسعه در Rust

- VSCode: برای توسعه برنامه‌های Rust، بهترین و پرکاربردترین ویرایشگر کد، VSCode است. برای این کار افزونه‌ی Rust (rls) را نصب کنید تا امکانات تکمیلی مانند تکمیل خودکار، هشدارهای خطا و قابلیت‌های دیگر را داشته باشید.
- Cargo: ابزار مدیریت بسته‌ها و ساخت در Rust است که همراه با نصب Rust نصب می‌شود. با استفاده از Cargo می‌توانید پروژه‌های Rust را ایجاد و مدیریت کنید.

اولین برنامه‌ی Rust

برای نوشتن اولین برنامه در Rust، مراحل زیر را دنبال کنید:

۱- در دایرکتوری پروژه خود یک فایل با نام main.rs ایجاد کنید.
۲- کد زیر را در آن بنویسید:

```
fn main() {
    println!("Hello, world!");
}
```

۳- برای کامپایل و اجرای برنامه، از دستور زیر استفاده کنید:

```
cargo run
```

این دستور برنامه را کامپایل کرده و خروجی آن را نمایش می‌دهد.

کدهای اولیه Rust

تعریف یک متغیر و چاپ آن

```
fn main() {
    let x = 5;
    println!("The value of x is: {}", x);
}
```

استفاده از متغیرهای ثابت

```
fn main() {
    const MAX_POINTS: u32 = 100_000;
    println!("The maximum points are: {}", MAX_POINTS);
```

```
}
```

تعریف انواع داده‌ها در Rust

در زبان Rust، انواع داده‌ها به دو دسته اصلی تقسیم می‌شوند:
۱. انواع داده اصلی: شامل اعداد صحیح، اعداد اعشاری، کاراکترها و بولین‌ها.
۲. انواع داده پیچیده: شامل **Tuple** و **Array**.

عدد صحیح (Integer)

Rust انواع مختلفی از اعداد صحیح را پشتیبانی می‌کند که به طول‌های مختلف وجود دارند. مثلا:

- i8, i16, i32, i64, i128 برای اعداد منفی و مثبت
- u8, u16, u32, u64, u128 برای اعداد مثبت فقط

مثال:

```
let x: i32 = 10;
let y: u64 = 20;
```

اعداد اعشاری (Floating Point)

Rust از دو نوع عدد اعشاری پشتیبانی می‌کند:

- f32 برای اعداد اعشاری با دقت کمتر
- f64 برای اعداد اعشاری با دقت بیشتر

مثال:

```
let a: f32 = 3.14;
let b: f64 = 2.71828;
```

کاراکترها (Character)

کاراکترها در Rust با استفاده از علامت ' تعریف می‌شوند و هر کاراکتر می‌تواند یک **Unicode** باشد.

مثال:

```
let ch: char = 'A';
```

بولین (Boolean)

نوع داده بولین فقط دو مقدار **true** یا **false** را می‌پذیرد.

مثال:

```
let is_rust_fun: bool = true;
```

Tuple

یک **Tuple** مجموعه‌ای از مقادیر از انواع مختلف است. طول یک Tuple ثابت است.

مثال:

```
;let tup: (i32, f64, char) = (500, 6.4, 'a')
```

برای دسترسی به مقادیر در Tuple از اندیس گذاری استفاده می کنیم:

```
let x = tup.0; // 500
```

آرایه‌ها (Arrays)

آرایه‌ها در Rust مجموعه‌ای از مقادیر با نوع داده یکسان هستند و طول ثابت دارند.

مثال:

```
;let arr: [i32; 5] = [1, 2, 3, 4, 5]
```

برای دسترسی به مقادیر آرایه از اندیس گذاری استفاده می کنیم:

```
let first = arr[0]; // 1
```

تعریف متغیرها در Rust

در Rust، متغیرها به صورت پیش‌فرض ثابت هستند و پس از مقداردهی اولیه نمی‌توانند تغییر کنند، مگر اینکه از کلمه کلیدی **mut** استفاده کنید.

مثال:

```
ثابت // ;let x = 5
متغیر قابل تغییر // ;let mut y = 10
;y = 20
```

عملیات روی انواع داده

در Rust می توان عملیات مختلفی مانند جمع، تفریق، ضرب و تقسیم را انجام داد. برای انجام این عملیات باید از نوع داده مناسب استفاده کرد.

مثال:

```
;let sum = 5 + 10
;let diff = 10 - 5
;let product = 5 * 2
;let quotient = 10 / 2
```

تعریف و استفاده از متغیرهای ثابت

در Rust می توان متغیرهایی را تعریف کرد که غیرقابل تغییر باشند. این کار با استفاده از کلمه کلیدی **const** انجام می‌شود.

مثال:

```
;const MAX_POINTS: u32 = 100_000
```

استفاده از متغیرهای نوع ثابت (Static Variables)

در Rust می توان از متغیرهای ثابت سراسری استفاده کرد که در تمام طول اجرای برنامه قابل دسترسی هستند. این کار با استفاده از کلمه کلیدی **static** انجام می‌شود.

مثال:

```
;"!static HELLO_WORLD: &str = "Hello, World
```

عملکردهای ورودی و خروجی در Rust

در Rust برای تعامل با کاربر و چاپ پیام‌ها از ماژول **std::io** و ماکرو **println!** استفاده می‌کنیم.

برای چاپ یک رشته در کنسول از ماکرو **println!** استفاده می‌کنیم:

```
;println!("Hello, world!")
```

برای گرفتن ورودی از کاربر از ماژول **std::io::stdin** استفاده می‌کنیم:

```
;use std::io


;()let mut name = String::new
;println!("Enter your name:")
;io::stdin().read_line(&mut name).expect("Failed to read line")
;println!("Hello, {}", name)
```

عملکرد توابع در Rust

در Rust، عملکردها به توابع یا **functions** شناخته می‌شوند. توابع بخش‌هایی از برنامه هستند که یک کار خاص را انجام می‌دهند و می‌توانند ورودی بگیرند و خروجی برگردانند.

تعریف یک تابع

برای تعریف یک تابع در Rust، از کلمه کلیدی **fn** استفاده می‌کنیم.

یک تابع می‌تواند ورودی‌ها (پارامترها) داشته باشد و می‌تواند خروجی (return) برگرداند.

مثال:

```
} ()fn greet
;println!("Hello, world!")
{
```

پارامترها و نوع داده‌ها

توابع در Rust می‌توانند پارامترهایی از نوع‌های مختلف بپذیرند. نوع هر پارامتر باید مشخص باشد.

مثال:

```
} fn add(a: i32, b: i32) → i32
    a + b
{
```

در این مثال، تابع **add** دو پارامتر از نوع **i32** می‌گیرد و یک **i32** را برمی‌گرداند.

نوع بازگشتی تابع

در Rust، اگر تابع خروجی نداشته باشد، از نوع () (که به آن **unit type** گفته می‌شود) استفاده می‌کنیم. این نوع به معنای "هیچ چیزی" است.

مثال:

```
fn say_hello() {
    println!("Hello!");
}
```

اگر تابع خروجی داشته باشد، نوع بازگشتی باید مشخص شود. در مثال **add** نوع بازگشتی **i32** است.

توابع بدون نیاز به بازگشت (Functions without return)

اگر یک تابع نیازی به بازگشت مقدار نداشته باشد، Rust به صورت خودکار نوع بازگشتی آن را () در نظر می‌گیرد.

مثال:

```
fn print_message() -> () {
    println!("This is a message");
}
```

استفاده از کلمه return در توابع

در Rust، برای برگرداندن مقدار از یک تابع، از کلمه کلیدی **return** استفاده می‌کنیم. استفاده از return به تابع این امکان را می‌دهد که مقداری را به فراخوانی کننده برگرداند. اگر در یک تابع نیازی به برگرداندن مقدار نباشد، می‌توانیم آن را به صورت خودکار به نوع () (که به آن **unit type** گفته می‌شود) در نظر بگیریم.

استفاده از return برای بازگشت مقدار

در صورتی که تابع مقداری را برگرداند، باید نوع بازگشتی را مشخص کنیم و از **return** برای برگرداندن مقدار استفاده کنیم.

مثال:

```
fn add(a: i32, b: i32) -> i32 {
    return a + b;
}

let result = add(5, 3);
println!("The result is: {}", result);
```

در اینجا، تابع **add** دو پارامتر **i32** می‌گیرد و مقدار **i32** را برمی‌گرداند. از **return** برای برگرداندن مقدار **a + b** استفاده می‌کنیم.

پیش‌فرض بازگشت بدون return

در Rust، اگر آخرین دستور یک تابع مقداردهی باشد، می‌توانیم بدون استفاده از return نیز مقدار را بازگردانیم. به عبارت دیگر، اگر در آخر تابع فقط یک مقدار وجود داشته باشد، Rust به صورت خودکار آن را بازمی‌گرداند.

مثال:

```
fn add(a: i32, b: i32) -> i32 {
```

```
    a + b  // بدون استفاده از return
}
let result = add(5, 3);
println!("The result is: {}", result);
```

در اینجا، هیچ نیازی به استفاده از return نیست. مقدار **a + b** به طور خودکار بازگشت داده می‌شود.

استفاده از return در توابع بدون بازگشت مقدار

اگر تابعی نیازی به بازگشت مقدار نداشته باشد، می‌توانیم از **return** برای خروج زودهنگام از تابع استفاده کنیم.

مثال:

```
fn print_message() {
    println!("This is a message.");
    return; // این خط اختیاری است
}
```

در اینجا، از **return** استفاده کرده‌ایم که به طور اختیاری از تابع خارج می‌شود. اگر return را حذف کنیم، تابع همچنان به درستی اجرا خواهد شد، زیرا تابع مقدار خاصی بازنمی‌گرداند.

- اگر تابع مقداری را برمی‌گرداند، می‌توانیم از return برای بازگرداندن مقدار استفاده کنیم.
- در Rust، اگر آخرین دستور تابع یک مقدار باشد، نیازی به استفاده از return نیست و آن مقدار به طور خودکار بازگشت داده می‌شود.
- در توابع بدون بازگشت مقدار، می‌توانیم از return برای خروج زودهنگام استفاده کنیم، اما این کار اختیاری است.

توابع با مقادیر پیش‌فرض

در Rust، برخلاف برخی زبان‌های دیگر، نمی‌توانیم به طور مستقیم مقادیر پیش‌فرض برای پارامترها تعیین کنیم.

توابع درون‌خطی (Inline Functions)

Rust به طور کلی از توابع درون‌خطی پشتیبانی می‌کند که به بهینه‌سازی سرعت کمک می‌کند.

برای تعریف یک تابع درون‌خطی از ویژگی #[**inline**] استفاده می‌کنیم.

مثال:

```
#[inline]
fn multiply(a: i32, b: i32) -> i32 {
    a * b
}
```

توابع در Rust در مقابل C

در C، توابع می‌توانند با استفاده از **return** داده‌های مختلفی را برگردانند. در Rust نیز همین عملکرد وجود دارد، اما با تفاوت‌های خاص مانند **ownership** و **borrowing**.

مقداردهی متغیرها در توابع

در Rust، هنگام تعریف توابع، می‌توانیم از مقادیر مختلف استفاده کنیم و آنها را به متغیرها اختصاص دهیم.

مقداردهی مستقیم به متغیرها

مقداردهی به متغیرها در Rust به سادگی و مشابه زبان‌های دیگر صورت می‌گیرد. به این صورت که می‌توانیم در هنگام فراخوانی یک تابع مقادیر مختلفی را ارسال کنیم.

مثال:

```
} fn add(a: i32, b: i32) → i32
                        a + b
{


;let result = add(5, 10)
;println!("Result: {}", result)
```

در اینجا، مقادیر ۵ و ۱۰ به تابع add ارسال شده و جمع آنها محاسبه و چاپ می‌شود.

مقداردهی از نوع‌های مختلف

Rust امکان استفاده از انواع مختلف در توابع را فراهم می‌کند. می‌توانیم از انواع پیچیده‌تری مانند **Tuples** یا **Structs** استفاده کنیم.

مثال:

```
} fn create_tuple() → (i32, f64)
                    (3.14 ,42)
{


;()let tup = create_tuple
;println!("First: {}, Second: {}", tup.0, tup.1)
```

در اینجا، تابع create_tuple یک **Tuple** با دو مقدار مختلف برمی‌گرداند.

انتقال و واگذاری مالکیت

در Rust، مقادیر در هنگام انتقال به توابع ممکن است مالکیت خود را به تابع منتقل کنند. این ویژگی به دلیل سیستم **ownership** و **borrowing** است.

مثال:

```
} fn take_ownership(s: String)
        ;println!("{}", s)
{


;let my_string = String::from("Hello")
;take_ownership(my_string)
// پس از اینجا، my_string دیگر قابل استفاده نیست
```

در اینجا، متغیر my_string پس از انتقال به تابع take_ownership دیگر قابل استفاده نیست.

انتقال و واگذاری مالکیت در Rust

در Rust، یکی از مفاهیم مهم که باید در هنگام کار با داده‌ها به آن توجه کنیم، **ownership** (مالکیت) است. وقتی داده‌ای در Rust ایجاد می‌شود، مالک آن مشخص می‌شود و فقط مالک داده می‌تواند آن را تغییر دهد یا از آن استفاده کند. در برخی موارد، مالکیت داده‌ها می‌تواند منتقل شود.

در Rust، تفاوت‌هایی بین داده‌های اصلی (مانند اعداد صحیح، اعداد اعشاری و بولین‌ها) و داده‌های غیر اصلی (مانند **String** و **Vec** که دارای حافظه heap هستند) وجود دارد.

داده‌های اصلی (Copy types)

داده‌هایی که به صورت پیش‌فرض از نوع **Copy** هستند، مانند اعداد صحیح (i32) یا اعداد اعشاری (f64)، زمانی که به یک متغیر دیگر منتقل می‌شوند، کپی می‌شوند. به این معنی که دو متغیر می‌توانند به صورت همزمان مقدار مشابهی از داده را داشته باشند.

مثال:

```
let x = 5; // داده اصلی
let y = x; // کپی مقدار
println!("x: {}, y: {}", x, y); // x و y هر دو قابل استفاده هستند
```

در اینجا، مقدار x به متغیر y کپی شده و هر دو متغیر مستقل از هم عمل می‌کنند. هیچ مشکلی در دسترسی به x و y وجود ندارد.

داده‌های غیر اصلی (Non-Copy types)

داده‌هایی که از نوع **non-Copy** هستند، مانند **String** و **Vec**، در هنگام انتقال، مالکیت خود را به متغیر دیگری منتقل می‌کنند. به این معنی که پس از انتقال مالکیت، متغیر قبلی دیگر قابل استفاده نیست.

مثال:

```
let s1 = String::from("Hello"); // داده غیر اصلی
let s2 = s1; // انتقال مالکیت
println!("s2: {}", s2); // s2 قابل استفاده است
// println!("s1: {}", s1); // این خط خطا میدهد چون مالکیت s1 به s2 منتقل شده است
```

در اینجا، پس از انتقال مالکیت s1 به s2، متغیر s1 دیگر نمی‌تواند به داده‌ها دسترسی پیدا کند و تلاش برای استفاده از آن باعث بروز خطا خواهد شد.
داده‌های اصلی (مانند اعداد و بولین‌ها) در هنگام انتقال کپی می‌شوند و هیچ مشکلی در استفاده از متغیرهای اصلی پس از انتقال وجود ندارد. در حالی که داده‌های غیر اصلی (مانند **String** و **Vec**) هنگام انتقال مالکیت، مالکیت داده‌ها را به متغیر جدید منتقل می‌کنند و پس از آن متغیر اولیه دیگر قابل استفاده نیست.

استفاده از ارجاع‌ها (References)

برای جلوگیری از انتقال مالکیت، می‌توانیم از ارجاع‌ها استفاده کنیم. ارجاع‌ها به توابع امکان می‌دهند که بدون تغییر مالکیت داده‌ها، به آن‌ها دسترسی داشته باشیم.

مثال:

```
fn print_string(s: &String) {
```

```
    println!("{}", s);
}

let my_string = String::from("Hello");
print_string(&my_string); // استفاده از ارجاع
```

در اینجا، ارجاع به my_string به تابع print_string ارسال شده است و همچنان مالکیت آن حفظ می‌شود.

ارجاع‌های قابل تغییر (Mutable References)

در صورت نیاز به تغییر مقادیر داخل یک تابع، می‌توانیم از ارجاع‌های قابل تغییر استفاده کنیم. برای این کار از کلمه کلیدی **mut** استفاده می‌شود.

مثال:

```
fn change_string(s: &mut String) {
    s.push_str(", World!");
}

let mut my_string = String::from("Hello");
change_string(&mut my_string);
println!("{}", my_string); // "Hello, World!"
```

در اینجا، از ارجاع قابل تغییر به my_string استفاده کرده‌ایم تا مقدار آن را درون تابع تغییر دهیم.

ساختارهای کنترلی در Rust

در Rust، ساختارهای کنترلی مانند حلقه‌ها و دستورات شرطی مشابه دیگر زبان‌ها وجود دارند، اما با تفاوت‌هایی در نحوه استفاده از آنها.

دستور if

دستور **if** در Rust برای انجام عملیات شرطی استفاده می‌شود. این دستور می‌تواند یک یا چند بلاک کد را بر اساس یک شرط اجرا کند.

مثال:

```
let x = 5;
if x > 3 {
    println!("x is greater than 3");
} else {
    println!("x is less than or equal to 3");
}
```

دستور if-else

می‌توانیم چندین شرط را با استفاده از دستور **else if** بررسی کنیم.

مثال:

```
;let x = 10
 } if x < 5
;println!("x is less than 5")
 } else if x == 10 {
;println!("x is 10")
 } else {
;println!("x is greater than 5 and not 10")
{
```

نکته مهمی است که در مورد دستور **if** در Rust باید به آن توجه کنیم، این است که دستور **if** می‌تواند مقدار بازگشتی داشته باشد. در Rust، برخلاف بسیاری از زبان‌های دیگر، **if** به عنوان یک عبارت (**expression**) عمل می‌کند و نه فقط یک دستور (**statement**). این بدین معنی است که نتیجه یک **if** می‌تواند به یک متغیر اختصاص داده شود و در صورت نیاز می‌تواند به عنوان یک مقدار بازگشتی استفاده شود.

دستور if به عنوان عبارت

در Rust، دستور **if** می‌تواند یک مقدار را بازگرداند و در نتیجه می‌توان از آن برای مقداردهی به متغیرها یا حتی بازگرداندن مقدار از توابع استفاده کرد.

مثال:

```
} fn max(a: i32, b: i32) → i32
 } if a > b
  a  //  اگر شرط برقرار بود، مقدار a بازگشت داده میشود
 } else {
  b  //  اگر شرط برقرار نبود، مقدار b بازگشت داده میشود
 {
{
;let result = max(5, 10)
;println!("The maximum is: {}", result)
```

در اینجا، دستور **if** مقدار **a** یا **b** را بر اساس مقایسه‌ای که انجام می‌دهد، باز می‌گرداند. پس **if** در اینجا به عنوان یک عبارت عمل کرده است و نه صرفاً یک دستور برای اجرا.

توجه به نوع‌های بازگشتی در if

در این مثال، چون **a** و **b** از نوع **i32** هستند، دستور **if** نیز باید مقدار مشابهی از نوع **i32** بازگرداند. این ویژگی به Rust اجازه می‌دهد تا دستور **if** را به طور ایمن و بدون نیاز به تایید نوع داده‌های برگشتی، مورد استفاده قرار دهد.

دستور **if** در Rust می‌تواند مقداری را بازگشت دهد، که این ویژگی را به یک عبارت تبدیل می‌کند. به همین دلیل می‌توان از **if** برای مقداردهی به متغیرها یا حتی بازگشت مقدار از توابع استفاده کرد.

دستور match

دستور **match** در Rust مانند یک **switch** در زبان‌های دیگر عمل می‌کند و می‌تواند بر اساس مقادیر مختلف عمل کند.

مثال:

```
    let x = 2;
    match x {
        1 => println!("One"),
        2 => println!("Two"),
        3 => println!("Three"),
        _ => println!("Other"),
    }
```

در اینجا، دستور **match** مقدار x را با مقادیر مختلف مقایسه می‌کند و یکی از بلاک‌های مناسب را اجرا می‌کند. علامت _ به معنای "هر مقدار دیگر" است.

دستور match به عنوان عبارت

دستور **match** در Rust می‌تواند یک مقدار را بازگرداند. این ویژگی آن را به ابزاری قدرتمند برای تصمیم‌گیری و انجام عملیات مختلف بر اساس مقادیر ورودی تبدیل می‌کند. مانند **if**، در **match** نیز هر شاخه (arm) می‌تواند مقداری بازگشتی داشته باشد، و Rust به طور خودکار نوع بازگشتی مناسب را بررسی می‌کند.

مثال:

```
fn classify_number(n: i32) -> &'static str {
    match n {
        1 => "One",
        2 => "Two",
        3 => "Three",
        _ => "Other",  // شاخه پیشفرض برای مقادیر دیگر
    }
}

let result = classify_number(2);
println!("The number is: {}", result);
```

در اینجا، دستور **match** مقادیر مختلف را بررسی می‌کند و یکی از رشته‌ها را باز می‌گرداند. مقدار بازگشتی بسته به مقدار ورودی متغیر n متفاوت خواهد بود.

توجه به نوع بازگشتی در match

همانطور که در مثال بالا مشاهده می‌شود، تمام شاخه‌های دستور **match** باید نوع مشابهی را بازگردانند. در اینجا، تمام شاخه‌ها یک **str&** (رشته‌ای) را باز می‌گردانند. Rust به طور خودکار بررسی می‌کند که تمام شاخه‌ها یک نوع بازگشتی یکسان دارند و از بروز خطا جلوگیری می‌کند.

استفاده از match برای مقادیر پیچیده‌تر

دستور **match** همچنین می‌تواند با انواع پیچیده‌تری مانند **tuples** یا **structs** استفاده شود. در این حالت، **match** مقادیر داخل ساختارها را بررسی کرده و مطابق با شرایط مختلف عمل می‌کند.

مثال با **Tuple**:

```
} fn get_point(quadrant: (i32, i32)) → &'static str
                                    } match quadrant
                                ,"Origin" ≤ (0 ,0)
                          ,"if x == y ⇒ "Diagonal (x, y)
                                ,"Other point" ≤ _
                                                {
                                                  {


                                    ;let point = (3, 3)
                                ;let result = get_point(point)
                                ;println!("The point is: {}", result)
```

در این مثال، دستور **match** یک **tuple** را بررسی می‌کند و مطابق با شرایط مختلف، یکی از مقادیر را باز می‌گرداند.

دستور **match** در Rust نیز مانند **if** به عنوان یک عبارت عمل می‌کند و می‌تواند مقدار بازگشتی داشته باشد. این ویژگی آن را به ابزاری مفید برای انجام تصمیم‌گیری‌های پیچیده و بازگشت مقادیر مختلف تبدیل می‌کند.

الگوهای پیچیده‌تر در match

در **Rust**، می‌توانیم از الگوها (patterns) برای بررسی مقادیر مختلف استفاده کنیم. این الگوها می‌توانند بسیار پیچیده باشند و انواع مختلف داده‌ها را پوشش دهند. می‌توانیم از تکرار شونده‌ها (مثل رنج‌ها)، ترکیب‌های **tuple** و حتی اختیارات (**enums**) استفاده کنیم.

الگوهای ترکیبی و چندگانه (Multiple Patterns)

در **match** می‌توانیم چندین الگو را با استفاده از | برای بررسی چندین حالت مختلف ترکیب کنیم.

مثال:

```
} fn is_even_or_odd(n: i32) → &'static str
                                } match n
                          ,"Even" ≤ 8 | 6 | 4 | 2
                                ,"Odd" ≤ _
                                                {
                                                  {


                                ;let result = is_even_or_odd(4)
                                ;println!("The number is: {}", result)
```

در اینجا، الگوهای ۲ | ۴ | ۶ | ۸ مشخص می‌کنند که اگر n یکی از این مقادیر باشد، مقدار **Even** برگشت داده می‌شود.

گارد شرطی در match (Match Guards)

شما می‌توانید از گاردهای شرطی (که همانطور که در دستور if استفاده می‌کنید، شرط اضافی برای هر الگو اضافه می‌کنند) در دستور **match** استفاده کنید. این گاردها با استفاده از کلمه کلیدی **if** به الگوهای مختلف اعمال می‌شوند.

مثال:

```
} fn check_number(n: i32) → &'static str
                        } match n
                ,"x if x < 0 ⇒ "Negative
                ,"x if x > 0 ⇒ "Positive
                            ,"Zero" ≤ _
                                    {
                                {

                ;let result = check_number(-5)
;println!("The number is: {}", result)
```

در اینجا، از گاردهای شرطی برای تشخیص مقادیر مثبت، منفی و صفر استفاده کرده‌ایم.

حلقه‌ها در زبان راست (Rust)

حلقه loop

ساده‌ترین نوع حلقه در راست، حلقه loop است. این حلقه بدون هیچ شرطی، کد داخل خود را تا بی‌نهایت تکرار می‌کند. برای خروج از این حلقه باید از دستور break استفاده کنیم.

```
                                } ()fn main
                            ;let mut counter = 0
                                    } loop
                ;println!("Counter: {}", counter)
                                    ;counter += 1
                            } if counter == 5
                                    ;break
                                            {
                                                {
                ;println!("Loop finished!")
                                        {
```

در مثال بالا، حلقه loop پنج بار اجرا می‌شود و سپس به کمک دستور break از حلقه خارج می‌شویم.

بازگشت مقدار از حلقه loop

یکی از ویژگی‌های جالب حلقه loop در راست، امکان بازگشت مقدار هنگام خروج از حلقه است:

```
                                } ()fn main
                            ;let mut counter = 0

                            } let result = loop
                                ;counter += 1
```

```
            } if counter == 10
        ;break counter * 2
                        {
                          ;{

    println!("Result: {}", result);  // 20 چاپ میشود
{
```

در این مثال، عبارت counter * 2 پس از دستور break نوشته شده است. این مقدار (۲۰) در متغیر result ذخیره می‌شود.

سوالات بخش حلقه **loop**:

۱. تفاوت حلقه loop با سایر حلقه‌ها چیست؟

۲. چگونه می‌توان از یک حلقه loop خارج شد؟

۳. آیا می‌توان از یک حلقه loop مقداری را برگرداند؟ اگر بله، چگونه؟

حلقه while

حلقه while تا زمانی که شرط آن برقرار باشد (مقدار true داشته باشد)، بدنه حلقه را اجرا می‌کند.

```
                    } ()fn main
            ;let mut number = 1

                     } while number < 5
        ;println!("Number: {}", number)
                        ;number += 1
                                {

    ;println!("While loop finished!")
{
```

در این مثال، حلقه while تا زمانی که number کمتر از ۵ باشد ادامه می‌یابد.

سوالات بخش حلقه **while**:

۱. تفاوت حلقه while با حلقه loop چیست؟

۲. اگر شرط حلقه while از ابتدا false باشد، آیا بدنه حلقه حتی یک‌بار هم اجرا می‌شود؟

۳. برای خروج زودهنگام از یک حلقه while، از چه دستوری استفاده می‌کنیم؟

حلقه for

حلقه for در راست برای پیمایش عناصر یک مجموعه مانند آرایه‌ها، رشته‌ها یا محدوده‌های عددی استفاده می‌شود. ساختار آن از سایر زبان‌ها متفاوت است و بیشتر شبیه به حلقه foreach در زبان‌های دیگر می‌باشد.

پیمایش محدوده اعداد

برای پیمایش یک محدوده عددی، از عملگر .. یا ..= استفاده می کنیم:

```
fn main() {
    // محدوده 0 تا 4 (بدون شامل شدن 5)
    for i in 0..5 {
        println!("Number: {}", i);
    }

    // محدوده 1 تا 5 (با شامل شدن 5)
    for i in 1..=5 {
        println!("Inclusive number: {}", i);
    }
}
```

در مثال اول، اعداد ۰ تا ۴ چاپ می‌شوند. در مثال دوم، اعداد ۱ تا ۵ چاپ می‌شوند.

پیمایش آرایه‌ها

```
fn main() {
    let numbers = [10, 20, 30, 40, 50];

    for number in numbers {
        println!("Array value: {}", number);
    }

    // پیمایش با استفاده از اندیس
    for i in 0..numbers.len() {
        println!("Index: {}, Value: {}", i, numbers[i]);
    }
}
```

پیمایش با enumerate

اگر به شماره عنصر (اندیس) و مقدار عنصر همزمان نیاز داشته باشید، می‌توانید از متد enumerate استفاده کنید:

```
fn main() {
    let fruits = ["apple", "banana", "cherry"];
```

```
} ()for (index, fruit) in fruits.iter().enumerate
    ;println!("Fruit {} is {}", index, fruit)
                                                {
{
```

سوالات بخش حلقه **for**:

۱. تفاوت بین .. و ..= در حلقه‌های for چیست؟

۲. برای پیمایش مقادیر یک آرایه بدون نیاز به شاخص (index) چگونه عمل می‌کنیم؟

۳. اگر بخواهیم همزمان به شاخص و مقدار هر عنصر آرایه دسترسی داشته باشیم، از چه متدی استفاده می‌کنیم؟

دستورات continue و break

دستور continue

دستور continue باعث می‌شود که اجرای بدنه حلقه متوقف شده و به ابتدای حلقه برگردیم (تکرار بعدی حلقه):

```
                                                } ()fn main
                                              } for i in 0..10
                                            } if i % 2 == 0
                                  ;continue  // اعداد زوج را رد میکند
                                                        {
        ;println!("Odd number: {}", i)  // فقط اعداد فرد چاپ میشوند
                                                    {
                                                {
```

در این مثال، فقط اعداد فرد چاپ می‌شوند زیرا برای اعداد زوج، دستور continue اجرا می‌شود.

دستور break

دستور break باعث خروج کامل از حلقه می‌شود:

```
                                                } ()fn main
                                          ;let mut sum = 0

                                        } for i in 1..100
                                            ;sum += i

                                        } if sum > 50
                ;println!("Sum exceeded 50! Current sum: {}", sum)
                                              ;break
                                                    {
                                                        {
```

```
{
```

در این مثال، به محض اینکه مجموع اعداد از ۵۰ بیشتر شود، از حلقه خارج می‌شویم.

حلقه‌های تو در تو و برچسب‌گذاری

در حلقه‌های تو در تو، می‌توانیم با استفاده از برچسب‌ها (label) مشخص کنیم که دستور break یا continue مربوط به کدام حلقه است:

```
} ()fn main
    } outer: for i in 1..6'
        ;println!("Outer loop: {}", i)

            } inner: for j in 1..4'
                ;println!("  Inner loop: {}", j)

                    } if i == 3 && j == 2
                        ;break 'outer  // خروج از حلقه بیرونی
                                    {
                                        {
                                            {

    ;println!("Loops finished")
{
```

در این مثال، وقتی i برابر ۳ و j برابر ۲ باشد، از حلقه بیرونی خارج می‌شویم.

سوالات بخش دستورات **continue** و **break**:

۱. تفاوت بین break و continue چیست؟

۲. در حلقه‌های تو در تو، اگر بخواهیم از حلقه بیرونی خارج شویم، چگونه عمل می‌کنیم؟

۳. برچسب‌گذاری (labeling) در حلقه‌ها چه کاربردی دارد؟

مثال عملی: جستجو در آرایه

برای درک بهتر کاربرد حلقه‌ها، یک برنامه ساده برای جستجوی یک عدد در آرایه می‌نویسیم:

```
} ()fn main
    ;let numbers = [4, 8, 15, 16, 23, 42]
        ;let search_for = 16
    ;let mut found = false
        ;let mut position = 0

} ()for (index, &number) in numbers.iter().enumerate
        } if number == search_for
```

```
                found = true;
            position = index;
                    break;
                }
            {

        } if found
println!("Found {} at position {}", search_for, position);
        } else {
    println!("{} not found in the array", search_for);
            {
    {
```

در این مثال:

۱. آرایه‌ای از اعداد داریم و می‌خواهیم عدد ۱۶ را پیدا کنیم.

۲. از حلقه for و متد enumerate برای پیمایش آرایه استفاده می‌کنیم.

۳. اگر عدد مورد نظر پیدا شد، از حلقه خارج می‌شویم.

۴. در پایان، نتیجه جستجو را چاپ می‌کنیم.

مثال عملی: محاسبه فیبوناچی

مثال دیگری از کاربرد حلقه‌ها، محاسبه اعداد فیبوناچی است:

```
} ()fn main
let n = 10;  // تعداد اعداد فیبوناچی که میخواهیم محاسبه کنیم
    let mut a = 0;
    let mut b = 1;

println!("Fibonacci sequence up to {}:", n);

print!("{} ", a);  // چاپ اولین عدد فیبوناچی

    } if n > 1
print!("{} ", b);  // چاپ دومین عدد فیبوناچی
    {

} for _ in 2..n
    let temp = a + b;
print!("{} ", temp);
    a = b;
b = temp;
```

```
{

    println!(); // رفتن به خط جدید
}
```

در این مثال، ۱۰ عدد اول دنباله فیبوناچی را محاسبه و چاپ می‌کنیم. توجه کنید که از متغیر _ برای متغیرهای شمارنده استفاده کرده‌ایم چون به مقدار شمارنده نیازی نداریم.

سوالات پایانی:

۱. برای چه نوع مسائلی استفاده از حلقه‌ها مناسب است؟

۲. در چه مواردی باید از حلقه for به جای while استفاده کنیم؟

۳. در مثال جستجو در آرایه، اگر عدد مورد نظر در آرایه نباشد، چه اتفاقی می‌افتد؟

۴. اگر بخواهیم تمام تکرارهای یک عدد در آرایه را پیدا کنیم (نه فقط اولین تکرار)، چگونه باید کد را تغییر دهیم؟

رشته‌ها در Rust

در درس‌های قبلی، ما با انواع داده‌های اولیه‌ای مانند اعداد و بولین‌ها آشنا شدیم. اما برای کار با متن و جملات، به نوع داده‌ی دیگری نیاز داریم که به آن رشته (String) می‌گویند. در این درس، با رشته‌ها در Rust آشنا می‌شویم و یاد می‌گیریم که چگونه آن‌ها را تعریف کنیم، تغییر دهیم و از آن‌ها استفاده کنیم. رشته‌ها یکی از مهم‌ترین انواع داده‌ها در برنامه‌نویسی هستند و در بسیاری از برنامه‌ها کاربرد دارند.

تعریف و ایجاد رشته‌ها

در Rust، دو نوع اصلی برای کار با رشته‌ها وجود دارد:

- String: این نوع داده، رشته‌ای قابل تغییر و قابل گسترش است که در حافظه heap ذخیره می‌شود.
- &str: این نوع داده، یک "slice" از رشته است که به بخشی از داده‌های رشته‌ای اشاره می‌کند و غیر قابل تغییر است. به آن "string slice" هم می‌گویند

برای شروع، بیشتر با نوع String کار خواهیم کرد.

ایجاد رشته‌های String:

۱. رشته خالی:

```
fn main() {
    let s = String::new();
}
```

با استفاده از String::new() می‌توان یک رشته‌ی خالی ایجاد کرد.

۲. رشته از لیترال رشته‌ای:

```
fn main() {
```

```
    ;let s = String::from("hello")
{
```

با استفاده از String::from() می‌توان یک رشته از یک مقدار رشته‌ای ثابت (string literal) ایجاد کرد. راه دیگر این است که مستقیماً متد to_string() را روی یک لیترال رشته‌ای صدا بزنیم:

```
} ()fn main
    ;()let s = "hello".to_string
{
```

هر دو روش بالا رشته‌ای با مقدار اولیه "hello" ایجاد می‌کنند.

سوالات کوتاه:

۱. تفاوت اصلی بین String و &str چیست؟

۲. چگونه می‌توان یک رشته‌ی خالی از نوع String ایجاد کرد؟

۳. دو روش برای ایجاد رشته‌ی String از یک لیترال رشته‌ای را نام ببرید.

تغییر رشته‌ها

رشته‌های String در Rust قابل تغییر هستند، به این معنی که می‌توان محتوای آن‌ها را بعد از ایجاد تغییر داد.

اضافه کردن به رشته:

۱. اضافه کردن یک رشته دیگر:

```
} ()fn main
    ;let mut s1 = String::from("foo")
    ;let s2 = String::from("bar")
    ;s1.push_str(&s2)
s1 is foobar :خروجی // ;println!("s1 is {}", s1)
{
```

متد push_str() یک string slice (&str) را به انتهای رشته‌ی String اضافه می‌کند. توجه کنید که s2 را با علامت & به push_str پاس دادیم، زیرا push_str نیاز به یک ارجاع به رشته دارد نه مالکیت آن.

```
} ()fn main
    ;let mut s1 = String::from("foo")
("bar")s1.push_str; // مستقیماً یک string literal به push_str پاس دادهایم
s1 is foobar :خروجی // ;println!("s1 is {}", s1)
{
```

۲. اضافه کردن یک کاراکتر:

```
} ()fn main
    ;let mut s = String::from("lo")
    ;s.push('l')
```

```
    s is lol :خروجی // ;println!("s is {}", s)
}
```

متد push() یک کاراکتر (char) را به انتهای رشته‌ی String اضافه می‌کند.

سوالات کوتاه:

۱. آیا رشته‌های String در Rust قابل تغییر هستند؟

۲. از چه متدی برای اضافه کردن یک string slice به انتهای یک String استفاده می‌کنیم؟

۳. از چه متدی برای اضافه کردن یک کاراکتر به انتهای یک String استفاده می‌کنیم؟

دسترسی به رشته‌ها

دسترسی مستقیم به کاراکترهای یک رشته در Rust کمی پیچیده‌تر از زبان‌های دیگر است، زیرا Rust از UTF-8 برای نمایش کاراکترها استفاده می‌کند. هر کاراکتر UTF-8 می‌تواند از یک تا چهار بایت فضا اشغال کند. به همین دلیل، اندیس‌گذاری مستقیم رشته‌ها با براکت [] ممکن نیست، زیرا این کار ممکن است باعث شود یک کاراکتر به درستی جدا نشود.

برای دسترسی به کاراکترها، معمولاً از روش‌های زیر استفاده می‌شود:

حلقه زدن روی کاراکترها:

```
Rust
} ()fn main
;let s = String::from("Hello")
    } ()for c in s.chars
;println!("{}", c)
        {
{
```

متد chars() یک iterator بر روی کاراکترهای رشته برمی‌گرداند که می‌توان با حلقه for روی آن پیمایش کرد.

حلقه زدن روی بایت‌ها:

```
Rust
} ()fn main
;let s = String::from("Hello")
    } ()for b in s.bytes
;println!("{}", b)
        {
{
```

متد bytes() یک iterator بر روی بایت‌های رشته برمی‌گرداند.

سوالات کوتاه:

۱. چرا اندیس‌گذاری مستقیم رشته‌ها با براکت [] در Rust توصیه نمی‌شود؟

۲. چه متدی برای حلقه زدن روی کاراکترهای یک رشته استفاده می‌شود؟

۳. چه متدی برای حلقه زدن روی بایت‌های یک رشته استفاده می‌شود؟

نتیجه‌گیری (تکمیلی)

در این درس، با نوع داده‌ی String در Rust آشنا شدیم و یاد گرفتیم که چگونه رشته‌ها را ایجاد کنیم، تغییر دهیم و به کاراکترهای آن‌ها دسترسی پیدا کنیم. رشته‌ها ابزار قدرتمندی برای کار با متن در Rust هستند و در درس‌های بعدی، بیشتر با کاربردهای آن‌ها آشنا خواهیم شد. در درس‌های بعدی، به مباحث پیشرفته‌تری مانند string slices (&str) و مالکیت رشته‌ها خواهیم پرداخت.

String Slices (&str) :

مقدمه

در درس قبل، با نوع داده‌ی String آشنا شدیم. به یاد داریم که String برای رشته‌های قابل تغییر و مالکیت‌دار استفاده می‌شود. اما نوع دیگری هم برای کار با رشته‌ها در Rust وجود دارد به نام **String Slice** یا &str. در این درس، یاد می‌گیریم که string slice چیست، چه تفاوتی با String دارد و چه کاربردهایی دارد. درک string slice برای فهم بهتر مفهوم مالکیت و قرض گرفتن در Rust بسیار مهم است.

String Slice چیست؟

یک string slice (&str) ارجاعی به بخشی از داده‌های رشته‌ای است که در حافظه ذخیره شده است. به عبارت دیگر، string slice مالک داده‌های رشته‌ای نیست، بلکه فقط به بخشی از آن اشاره می‌کند. می‌توان string slice را به عنوان یک "نما" یا "برش" از یک رشته در نظر گرفت.

تفاوت با String:

- مالکیت: String مالک داده‌های رشته‌ای خود است، در حالی که &str مالک نیست.
- قابلیت تغییر: String قابل تغییر است، اما &str غیرقابل تغییر است. شما نمی‌توانید محتوای یک string slice را تغییر دهید.
- محل ذخیره‌سازی: String داده‌ها را در heap ذخیره می‌کند، در حالی که &str می‌تواند به داده‌هایی اشاره کند که در مکان‌های مختلف حافظه باشند، از جمله:
  - string literals (مقادیر رشته‌ای ثابت که در کد نوشته می‌شوند)
  - بخشی از یک String
  - حتی داده‌های استاتیک

ایجاد String Slice:

از String literals:

```
Rust
```

```
} ()fn main
    "!let s: &str = "Hello, world!"; // "Hello, world یک string literal و از نوع &str
است
{
```

وقتی یک string literal را مستقیماً در کد می‌نویسید، نوع آن به طور پیش‌فرض &str است.

از String (با استفاده از slicing):

```
Rust
} ()fn main
    ;let s1 = String::from("hello world")

    (5 غیر شامل) 5 تا 0 از اندیس let hello = &s1[0..5]; // slice
    (11 غیر شامل) 11 تا 6 از اندیس let world = &s1[6..11]; // slice

    hello, world :خروجی // ;println!("{}, {}", hello, world)
{
```

در اینجا، &s1[0..5] و &s1[6..11] عمل slicing را روی s1 انجام می‌دهند و string slice هایی از s1 ایجاد می‌کنند. توجه کنید که از عملگر & قبل از s1 استفاده می‌کنیم تا یک ارجاع به s1 بگیریم، زیرا slicing روی ارجاع کار می‌کند. بازه [۰..۵] به معنی اندیس‌های ۰، ۱، ۲، ۳ و ۴ است (اندیس ۵ شامل نمی‌شود).

انواع Slice ها:

- [start..end]: slice از اندیس start تا end (غیر شامل end)
- [start..]: slice از اندیس start تا انتهای رشته
- [..end]: slice از ابتدای رشته تا اندیس end (غیر شامل end)
- [..]: slice از کل رشته

```
} ()fn main
    ;let s = String::from("Rust is powerful")
        ;println!("slice1: {}", &s[0..4])
        ;println!("slice2: {}", &s[5..])
        ;println!("slice3: {}", &s[..4])
        ;println!("slice4: {}", &s[..])
    ;println!("Original String: {}", s)
{
```

سوالات کوتاه:

۱. String slice (&str) چیست؟

۲. تفاوت اصلی بین String و &str در چیست؟ (مالکیت و قابلیت تغییر)

۳. چگونه می‌توان یک string slice از یک String ایجاد کرد؟

۴. وقتی یک string literal در Rust می‌نویسیم، نوع آن چیست؟

کاربردهای String Slice

String slices بسیار پرکاربرد هستند، به خصوص زمانی که نمی‌خواهیم مالکیت رشته را منتقل کنیم یا نیاز به تغییر رشته نداریم.

- به عنوان آرگومان تابع: بسیاری از توابع Rust string slice (&str) را به عنوان ورودی می‌پذیرند، زیرا این کار انعطاف‌پذیری بیشتری ایجاد می‌کند. تابع می‌تواند هم با String و هم با string literals کار کند بدون اینکه مالکیت رشته را بگیرد.

```
Rust
fn first_word(s: &str) -> &str { // تابع یک string slice به عنوان ورودی و خروجی میگیرد
    let bytes = s.as_bytes();
    for (i, &item) in bytes.iter().enumerate() {
        if item == b' ' {
            return &s[0..i];
        }
    }
    &s[..]
}

fn main() {
    let my_string = String::from("hello world");

    // first_word بر روی String کار میکند
    let word = first_word(&my_string[..]);

    let my_string_literal = "hello world literal";

    // first_word بر روی string literal هم کار میکند
    let word2 = first_word(my_string_literal);

    println!("{}, {}", word, word2); // خروجی: hello, hello
}
```

در تابع first_word، ورودی و خروجی هر دو از نوع &str هستند. این تابع اولین کلمه از یک رشته را پیدا می‌کند و یک slice به آن برمی‌گرداند. همانطور که می‌بینید، تابع first_word هم با String (با استفاده از &my_string[..]) و هم با string literal (my_string_literal) به خوبی کار می‌کند.

- جلوگیری از کپی غیر ضروری: وقتی از string slice استفاده می‌کنید، از کپی کردن داده‌های رشته‌ای جلوگیری می‌کنید. فقط یک ارجاع به بخشی از داده‌ها منتقل می‌شود که باعث افزایش کارایی می‌شود.

نکات مهم:

- String slices به داده‌های اصلی که به آن‌ها اشاره می‌کنند وابسته هستند. اگر داده‌های اصلی از بین بروند، string slice دیگر معتبر نخواهد بود (dangling reference). Rust از طریق سیستم مالکیت و قرض گرفتن از این مشکل جلوگیری می‌کند.

- غیرقابل تغییر بودن string slices به این معنی است که شما نمی‌توانید محتوای آن‌ها را تغییر دهید، اما می‌توانید string slices جدیدی از بخش‌های مختلف یک رشته ایجاد کنید.

سوالات کوتاه:

۱. چرا string slice به عنوان آرگومان تابع پرکاربرد است؟

۲. فایده‌ی استفاده از string slice از نظر کارایی چیست؟

۳. آیا می‌توان محتوای یک string slice را تغییر داد؟

۴. اگر داده‌های اصلی که یک string slice به آن اشاره می‌کند از بین برود، چه اتفاقی می‌افتد؟

نتیجه‌گیری

در این درس، با مفهوم string slice (&str) در Rust آشنا شدیم و تفاوت آن را با String بررسی کردیم. یاد گرفتیم که string slice یک ارجاع به بخشی از داده‌های رشته‌ای است و برای کار با رشته‌ها به صورت کارآمد و بدون انتقال مالکیت بسیار مفید است. درک string slices برای فهم مکانیسم مالکیت و قرض گرفتن در Rust اساسی است و در درس‌های بعدی بیشتر از آن استفاده خواهیم کرد.

متدهای رایج رشته‌ها و تبدیل نوع در Rust

در درس‌های قبل، با انواع داده‌ی String و &str آشنا شدیم و نحوه‌ی ایجاد و برش رشته‌ها را یاد گرفتیم. اما برای کارآمدتر شدن در برنامه‌نویسی، نیاز داریم تا عملیات‌های رایج‌تری را روی رشته‌ها انجام دهیم. در این درس، با تعدادی از متدهای رایج رشته‌ها در Rust آشنا می‌شویم که به ما کمک می‌کنند تا رشته‌ها را دستکاری، جستجو و تغییر فرمت دهیم. همچنین، مبحث مهم تبدیل نوع به رشته و برعکس را نیز بررسی خواهیم کرد.

متدهای رایج رشته‌ها

Rust متدهای متنوعی برای کار با رشته‌ها ارائه می‌دهد. در اینجا به برخی از پرکاربردترین آن‌ها اشاره می‌کنیم:

۱. **replace(target, replacement)**: جایگزینی زیررشته

این متد، تمامی نمونه‌های زیررشته target را با زیررشته replacement در رشته اصلی جایگزین می‌کند و یک رشته‌ی جدید برمی‌گرداند. رشته‌ی اصلی تغییر نمی‌کند.

Rust

```
fn main() {
    let s = String::from("Hello World");
    let new_s = s.replace("World", "Rust");
    println!("New string: {}", new_s); // خروجی: New string: Hello Rust
    println!("Original string: {}", s); // خروجی: Original string: Hello World (رشته اصلی تغییر نکرده)
}
```

۲. **trim**(): حذف فاصله‌های خالی ابتدا و انتها

این متد، فاصله‌های خالی (whitespace) را از ابتدا و انتهای رشته حذف می‌کند و یک **string slice** به داده‌های trimmed شده برمی‌گرداند.

```
Rust
fn main() {
    let s = String::from("   hello   ");
    let trimmed_s = s.trim();
    println!("Trimmed string: {}", trimmed_s); // خروجی: Trimmed string: hello
    println!("Original string: {}", s); // خروجی: Original string:    hello   (رشته اصلی تغییر نکرده)
}
```

۳. **split(pattern)**: تبدیل رشته به بردار (Vector) از string slice ها

این متد، رشته را بر اساس یک الگو (pattern) که می‌تواند یک کاراکتر یا یک string slice باشد، به چند بخش تقسیم می‌کند و یک iterator بر روی string slice های به دست آمده برمی‌گرداند.

```
Rust
fn main() {
    let s = String::from("foo bar baz");
    for part in s.split(" ") {
        println!("Part: {}", part);
    }
    // خروجی:
    // Part: foo
    // Part: bar
    // Part: baz
}
```

۴. **contains(substring)**: بررسی وجود زیررشته

این متد بررسی می‌کند که آیا رشته شامل زیررشته‌ی مشخص شده (substring) است یا خیر و مقدار true یا false برمی‌گرداند.

```
Rust
fn main() {
```

```
    let s = String::from("Rust programming");
    let contains_rust = s.contains("Rust");
    let contains_java = s.contains("Java");
    println!("Contains 'Rust': {}", contains_rust); // خروجی: Contains 'Rust': true
    println!("Contains 'Java': {}", contains_java); // خروجی: Contains 'Java': false
}
```

۵. **starts_with(prefix)**: بررسی شروع رشته با پیشوند

این متد بررسی می‌کند که آیا رشته با پیشوند مشخص شده (prefix) شروع می‌شود یا خیر و مقدار true یا false برمی‌گرداند.

```
fn main() {
    let s = String::from("Rust language");
    let starts_with_rust = s.starts_with("Rust");
    let starts_with_java = s.starts_with("Java");
    println!("Starts with 'Rust': {}", starts_with_rust); // خروجی: Starts with 'Rust': true
    println!("Starts with 'Java': {}", starts_with_java); // خروجی: Starts with 'Java': false
}
```

۶. **ends_with(suffix)**: بررسی پایان رشته با پسوند

این متد بررسی می‌کند که آیا رشته با پسوند مشخص شده (suffix) پایان می‌یابد یا خیر و مقدار true یا false برمی‌گرداند.

```
fn main() {
    let s = String::from("programming in Rust");
    let ends_with_rust = s.ends_with("Rust");
    let ends_with_java = s.ends_with("Java");
    println!("Ends with 'Rust': {}", ends_with_rust); // خروجی: Ends with 'Rust': true
    println!("Ends with 'Java': {}", ends_with_java); // خروجی: Ends with 'Java': false
}
```

۷. **to_lowercase**(): تبدیل به حروف کوچک

این متد تمامی حروف بزرگ رشته را به حروف کوچک تبدیل می‌کند و یک رشته‌ی جدید برمی‌گرداند.

```
fn main() {
    let s = String::from("HELLO");
    let lower_s = s.to_lowercase();
    println!("Lowercase: {}", lower_s); // خروجی: Lowercase: hello
    println!("Original string: {}", s); // خروجی: Original string: HELLO (رشته اصلی تغییر نکرده)
}
```

```
{
```

۸. **to_uppercase**(): تبدیل به حروف بزرگ

این متد تمامی حروف کوچک رشته را به حروف بزرگ تبدیل می‌کند و یک رشته‌ی جدید برمی‌گرداند.

```
Rust
} ()fn main {
    ;let s = String::from("hello")
    ;()let upper_s = s.to_uppercase
    ;println!("Uppercase: {}", upper_s) // خروجی: Uppercase: HELLO
    ;println!("Original string: {}", s) // خروجی: Original string: hello (رشته اصلی تغییر نکرده)
{
```

سوالات کوتاه:

۱. متد replace() چه کاری انجام می‌دهد و آیا رشته اصلی را تغییر می‌دهد؟

۲. متد trim() برای چه منظوری استفاده می‌شود و چه نوع داده‌ای را برمی‌گرداند؟

۳. متد split() رشته را به چه چیزی تبدیل می‌کند؟

۴. متدهای contains(), starts_with() و ends_with() چه نوع مقداری را برمی‌گردانند؟

۵. متدهای to_lowercase() و to_uppercase() چه تغییری در رشته ایجاد می‌کنند؟

تبدیل نوع به رشته و برعکس

در Rust، اغلب نیاز داریم تا انواع داده‌های دیگر را به رشته تبدیل کنیم و برعکس.

تبدیل به رشته:

برای تبدیل هر نوع داده‌ای به رشته، می‌توان از متد to_string() استفاده کرد. این متد برای بسیاری از انواع داده‌ها در Rust پیاده‌سازی شده است.

```
Rust
} ()fn main {
    ;let number = 123
    ;()let number_str = number.to_string
    ;println!("Number as string: {}", number_str) // خروجی: Number as string: 123

    ;let boolean = true
    ;()let boolean_str = boolean.to_string
    ;println!("Boolean as string: {}", boolean_str) // خروجی: Boolean as string: true
{
```

تبدیل از رشته به نوع دیگر:

برای تبدیل یک رشته به نوع دیگر (مانند عدد صحیح، عدد اعشاری، بولین و غیره)، می‌توان از متد parse() استفاده کرد. متد parse() می‌تواند با خطا مواجه شود (مثلاً اگر رشته فرمت درستی نداشته باشد)، بنابراین باید نتیجه‌ی آن را با استفاده از Result مدیریت کرد.

```
Rust
} ()fn main
    ;"let num_str = "42
    ;()let num: i32 = num_str.parse().unwrap // در صورت موفقیت، مقدار را برمیگرداند، در صورت خطا panic میکند (فعلا از unwrap استفاده میکنیم)
    Parsed number: 42 :خروجی // ;println!("Parsed number: {}", num)

    ;"let float_str = "3.14
    ;()let float_num: f64 = float_str.parse().unwrap
    Parsed float: 3.14 :خروجی // ;println!("Parsed float: {}", float_num)

    // مثال خطا:
    ;"let invalid_num_str = "abc
    // ;()let invalid_num: i32 = invalid_num_str.parse().unwrap // این خط باعث panic میشود!
    // ;println!("Parsed invalid number: {}", invalid_num)
{
```

توضیح **unwrap**(): در مثال بالا، از unwrap() برای دریافت مقدار موفقیت‌آمیز از Result استفاده کردیم. فعلاً برای سادگی از آن استفاده می‌کنیم، اما در درس‌های بعدی به طور مفصل با مدیریت خطا و Result آشنا خواهیم شد.

سوالات کوتاه:

۱. چه متدی برای تبدیل انواع داده‌های دیگر به String استفاده می‌شود؟

۲. متد parse() برای چه منظوری استفاده می‌شود؟

۳. چرا نتیجه‌ی متد parse() باید مدیریت شود؟ (به دلیل احتمال خطا)

۴. unwrap() در مثال‌های تبدیل نوع چه کاری انجام می‌دهد؟ (دریافت مقدار موفقیت‌آمیز یا panic در صورت خطا)

نتیجه‌گیری (تکمیلی)

در این درس، با تعدادی از متدهای رایج رشته‌ها در Rust آشنا شدیم و یاد گرفتیم که چگونه با استفاده از آن‌ها رشته‌ها را دستکاری کنیم. همچنین، روش‌های تبدیل انواع داده به رشته و برعکس را بررسی کردیم. این مهارت‌ها برای پردازش متن و داده‌های ورودی در برنامه‌های Rust بسیار حیاتی هستند. در درس‌های بعدی، به مباحث پیشرفته‌تری مانند مدیریت خطا و ساختمان داده‌های دیگر خواهیم پرداخت.

Structs : (ساختارها)

ساختارها (Structs) در Rust برای گروه‌بندی و سازماندهی داده‌های مرتبط استفاده می‌شوند. آن‌ها به شما اجازه می‌دهند تا نوع داده‌ای سفارشی خود را ایجاد کنید که شامل چندین مقدار با انواع داده‌های مختلف است.

تعریف ساختار (Defining Structs)

برای تعریف یک ساختار، از کلمه کلیدی struct استفاده می‌کنیم. سپس نام ساختار و در داخل آکولاد {} فیلدهای آن را مشخص می‌کنیم. هر فیلد دارای یک نام و یک نوع داده است.

```
} struct User
    ,username: String
    ,email: String
    ,sign_in_count: u64
    ,active: bool
{
```

در مثال بالا، یک ساختار به نام User تعریف کرده‌ایم که شامل چهار فیلد است: username با نوع String، email با نوع String، sign_in_count با نوع u64 (عدد صحیح ۶۴ بیتی بدون علامت) و active با نوع bool (مقدار منطقی).

ایجاد نمونه از ساختار (Instantiating Structs)

برای استفاده از یک ساختار، باید یک نمونه (instance) از آن ایجاد کنیم. این کار با نوشتن نام ساختار و سپس مقادیر فیلدها در داخل آکولاد {} انجام می‌شود. ترتیب فیلدها در هنگام ایجاد نمونه مهم نیست، اما نام فیلدها باید مطابقت داشته باشد.

```
} ()fn main
    } let user1 = User
        ,email: String::from("someone@example.com")
        ,username: String::from("someusername123")
        ,active: true
        ,sign_in_count: 1
    ;{
    ;(user1.username ,"{} :نام کاربری"(println!
    ;(user1.email ,"{} :ایمیل"(println!
{
```

در اینجا، یک نمونه از ساختار User به نام user1 ایجاد کرده‌ایم و مقادیر اولیه‌ای را به فیلدهای آن اختصاص داده‌ایم. سپس با استفاده از عملگر نقطه (.) به فیلدهای username و email دسترسی پیدا کرده و آن‌ها را چاپ کرده‌ایم.

دسترسی به فیلدهای ساختار (Accessing Struct Fields)

همانطور که در مثال بالا دیدید، برای دسترسی به فیلدهای یک نمونه از ساختار، از عملگر نقطه (.) استفاده می‌کنیم. به این صورت که ابتدا نام متغیر نمونه و سپس نام فیلد را بعد از نقطه می‌نویسیم.

```
;let username = user1.username
```

```
;(username ,"{}" ,"نام کاربری: "(println!
```
مثال توصیفی بیشتر

فرض کنید می‌خواهیم اطلاعات مربوط به یک مستطیل را ذخیره کنیم. می‌توانیم یک ساختار به نام Rectangle تعریف کنیم که طول و عرض آن را نگهداری کند.

```
} struct Rectangle
    ,width: u32
    ,height: u32
{


} ()fn main
} let rect1 = Rectangle
    ,width: 30
    ,height: 50
    ;{

    ;(rect1.width ,"{}" ,"عرض مستطیل: "(println!
    ;(rect1.height ,"{}" ,"ارتفاع مستطیل: "(println!
{
```
در این مثال، ساختار Rectangle دارای دو فیلد width و height با نوع u32 است. سپس یک نمونه به نام rect1 ایجاد کرده و به فیلدهای آن مقادیر ۳۰ و ۵۰ را اختصاص داده‌ایم.

سوالات برای کلاس:

۱. ساختار (Struct) در Rust چیست و چه کاربردی دارد؟
   - پاسخ: ساختار برای گروه‌بندی داده‌های مرتبط با انواع مختلف در یک واحد به نام نوع داده‌ای سفارشی استفاده می‌شود.
۲. چگونه یک ساختار در Rust تعریف می‌شود؟
   - پاسخ: با استفاده از کلمه کلیدی struct، نام ساختار و سپس فیلدها با نام و نوع داده در داخل آکولاد {}.
۳. چگونه می‌توان یک نمونه از یک ساختار ایجاد کرد؟
   - پاسخ: با نوشتن نام ساختار و سپس مقادیر فیلدها در داخل آکولاد {} به صورت name: value.
۴. چگونه می‌توان به فیلدهای یک نمونه از ساختار دسترسی پیدا کرد؟
   - پاسخ: با استفاده از عملگر نقطه (.) به این صورت: instance_name.field_name.

Enum (شمارشگرها)

شمارشگرها (Enums) در Rust به شما اجازه می‌دهند تا نوعی داده‌ای را تعریف کنید که می‌تواند یکی از چندین مقدار ممکن باشد. این مقادیر به عنوان "variants" (گونه‌ها) شناخته می‌شوند. Enumها برای نمایش حالات مختلف یا مجموعه‌ای از انتخاب‌های محدود بسیار مفید هستند.

تعریف شمارشگر (Defining Enums)

برای تعریف یک enum، از کلمه کلیدی enum استفاده می‌کنیم. سپس نام enum و در داخل آکولاد {} گونه‌های آن را مشخص می‌کنیم. هر گونه می‌تواند بدون هیچ داده‌ای، یا همراه با داده‌هایی با انواع مختلف باشد.

```
} enum IpAddrKind
    ,V4
    ,V6
{
```

در مثال بالا، یک enum به نام IpAddrKind تعریف کرده‌ایم که دو گونه دارد: V4 و V6. این enum می‌تواند برای نشان دادن نوع یک آدرس IP استفاده شود.

گونه‌های Enum با داده (Enum Variants with Data)

گونه‌های enum می‌توانند داده‌هایی را در خود نگه دارند. این کار به شما اجازه می‌دهد تا اطلاعات بیشتری را در کنار نوع گونه ذخیره کنید.

```
} enum IpAddr
    ,V4(u8, u8, u8, u8)
    ,V6(String)
{
```

در این مثال، enum IpAddr دو گونه دارد:

- V4 که چهار مقدار u8 (عدد صحیح ۸ بیتی بدون علامت) را برای نشان دادن یک آدرس IPv4 نگه می‌دارد.
- V6 که یک مقدار String را برای نشان دادن یک آدرس IPv6 نگه می‌دارد.

استفاده از Enum (Using Enums)

برای استفاده از یک enum، می‌توانیم یک متغیر با نوع آن تعریف کرده و یکی از گونه‌های آن را به آن اختصاص دهیم.

```
} ()fn main
    ;let ipv4_addr = IpAddr::V4(127, 0, 0, 1)
    ;let ipv6_addr = IpAddr::V6(String::from("::1"))

    } match ipv4_addr
,(IPv4: {}.{}.{}.{}", a, b, c, d آدرس")!IpAddr::V4(a, b, c, d) ⇒ println
```

```
        ,(IPv6: {}", address آدرس")!IpAddr::V6(address) ⇒ println
                                                                {

                                                    } match ipv6_addr
    ,(IPv4: {}.{}.{}.{}", a, b, c, d آدرس")!IpAddr::V4(a, b, c, d) ⇒ println
        ,(IPv6: {}", address آدرس")!IpAddr::V6(address) ⇒ println
                                                                {
                                                                {
```

در این مثال، دو متغیر ipv4_addr و ipv6_addr با نوع IpAddr ایجاد کرده‌ایم و گونه‌های مختلف را به آن‌ها اختصاص داده‌ایم. سپس از دستور match برای بررسی نوع گونه و استخراج داده‌های آن استفاده کرده‌ایم.

Enum Option

یکی از پرکاربردترین enumها در Rust، Option است که در کتابخانه استاندارد تعریف شده است. Option برای مقابله با حالتی استفاده می‌شود که یک مقدار ممکن است وجود داشته باشد یا وجود نداشته باشد.

```
                                                } <enum Option<T
                                                    ,Some(T)
                                                      ,None
                                                             {

                                                       } enum sn
                                                     , some
                                                     , none
                                                            ;{
                                            < template < class T
                                                 } struct Option
                                                ; sn type
                                                } union z
                                             ; T a
                                                      ;{
                                                             {
                                                ; Option<int> x
                                                ; x.type = some
                                                      ; x.a = 5
```

Option دارای دو گونه است:

- Some(T): نشان می‌دهد که یک مقدار از نوع T وجود دارد.

- None: نشان می‌دهد که هیچ مقداری وجود ندارد.

استفاده از Option به جلوگیری از خطاهای مربوط به مقادیر null کمک می‌کند.

```
fn main() {
    let some_number: Option<int> = Some(5);
    let some_string = Some("a string");
    let absent_number: Option<i32> = None;

    match some_number {
        Some(value) => println!("مقدار وجود دارد: {}", value),
        None => println!("مقدار وجود ندارد"),
    }
}
```

استفاده از match با Enum

همانطور که در مثال‌های قبلی دیدید، دستور match معمولاً برای کار با enumها استفاده می‌شود. match به شما اجازه می‌دهد تا بر اساس گونه‌های مختلف یک enum، اقدامات متفاوتی انجام دهید. هر شاخه از match باید تمام گونه‌های ممکن enum را پوشش دهد.

مثال توصیفی بیشتر

فرض کنید می‌خواهیم وضعیت یک پیام را نشان دهیم. می‌توانیم یک enum به نام Message تعریف کنیم که حالات مختلف یک پیام را نشان دهد.

```
enum Message {
    Quit,
    Move { x: i32, y: i32 },
    Write(String),
    ChangeColor(i32, i32, i32),
}

fn process_message(msg: Message) {
    match msg {
        Message::Quit => println!("درخواست خروج"),
        Message::Move { x, y } => println!("حرکت به x: {}, y: {}", x, y),
        Message::Write(text) => println!("نوشتن متن: {}", text),
        Message::ChangeColor(r, g, b) => println!("تغییر رنگ به قرمز: {}, سبز: {}, آبی: {}", r, g, b),
    }
}
```

```
} ()fn main
    ;let quit_message = Message::Quit
    ;let move_message = Message::Move { x: 10, y: 20 }
;let write_message = Message::Write(String::from("Hello, world!"))
    ;let color_message = Message::ChangeColor(255, 0, 0)

    ;process_message(quit_message)
    ;process_message(move_message)
;process_message(write_message)
;process_message(color_message)
{
```

در این مثال، enum Message چهار گونه مختلف دارد که هر کدام نشان‌دهنده یک نوع پیام است. گونه Move دارای داده‌های نام‌دار (مانند یک struct)، گونه Write دارای یک String و گونه ChangeColor دارای سه مقدار i32 است. تابع process_message با استفاده از match نوع پیام را بررسی کرده و عمل مناسب را انجام می‌دهد.

سوالات برای کلاس:

۱. Enum (شمارشگر) در Rust چیست و چه کاربردی دارد؟

- پاسخ: Enum نوعی داده است که می‌تواند یکی از چندین مقدار ممکن را داشته باشد و برای نمایش حالات مختلف یا مجموعه‌ای از انتخاب‌های محدود استفاده می‌شود.

۲. چگونه یک enum در Rust تعریف می‌شود؟

- پاسخ: با استفاده از کلمه کلیدی enum، نام enum و سپس گونه‌های آن در داخل آکولاد {}.

۳. آیا گونه‌های enum می‌توانند داده‌ای را در خود نگه دارند؟ اگر بله، چگونه؟

- پاسخ: بله، گونه‌های enum می‌توانند داده‌هایی با انواع مختلف را در خود نگه دارند. این کار با مشخص کردن نوع داده‌ها بعد از نام گونه در هنگام تعریف انجام می‌شود.

۴. enum Option در Rust چه کاربردی دارد؟

- پاسخ: Option برای نشان دادن این که یک مقدار ممکن است وجود داشته باشد (Some) یا وجود نداشته باشد (None) استفاده می‌شود و به جلوگیری از خطاهای مربوط به مقادیر null کمک می‌کند.

۵. چرا دستور match معمولاً با enumها استفاده می‌شود؟

- پاسخ: match به شما اجازه می‌دهد تا بر اساس گونه‌های مختلف یک enum، اقدامات متفاوتی انجام دهید و تضمین می‌کند که تمام حالات ممکن پوشش داده شده‌اند.

match غیر جامع (Non-Exhaustive Match):

چه اتفاقی می‌افتد اگر یک یا چند حالت enum را در match پوشش ندهیم؟

```
*/
// THIS CODE WILL NOT COMPILE!
enum Message {
    Quit,
    Write(String),
    Move { x: i32, y: i32 },
    ChangeColor(i32, i32, i32),
}

fn process_message_incomplete(msg: Message) {
    match msg {
        // Missing Message::Quit
        Message::Write(text) => {
            println!("Text message: {}", text);
        }
        Message::Move { x, y } => {
            println!("Move to: x={}, y={}", x, y);
        }
        // Missing Message::ChangeColor
        // Error: non-exhaustive patterns: `Quit`, `ChangeColor(_, _, _)` not covered
    }
}

fn main() {
    let msg = Message::Quit;
    process_message_incomplete(msg); // This call would cause a compile error
}
/*
```

- اگر این کد را کامپایل کنید، Rust خطا می‌دهد: non-exhaustive patterns.
- کامپایلر به شما می‌گوید که کدام حالت‌ها (Quit و ChangeColor) پوشش داده نشده‌اند.
- این یک ویژگی ایمنی مهم در Rust است. از خطاهای زمان اجرا جلوگیری می‌کند.

راه حل: استفاده از الگوی _ (**Wildcard**):

گاهی اوقات نمی‌خواهیم برای هر حالت خاص کد جداگانه‌ای بنویسیم. یا می‌خواهیم یک حالت پیش‌فرض داشته باشیم.

می‌توانیم از الگوی _ استفاده کنیم. _ با هر مقداری که توسط الگوهای قبلی پوشش داده نشده است، مطابقت دارد.

```
enum Message {
```

```
    ,Quit
    ,Write(String)
    ,Move { x: i32, y: i32 }
    ,ChangeColor(i32, i32, i32)
{

} fn process_message_with_wildcard(msg: Message)
    } match msg
        } ≤ Message::Write(text)
            ;println!("Handling Write specifically: {}", text)
            {
        Use '_' to match all other variants (Quit, Move, ChangeColor) //
            } ≤ _
    ;println!("Handling other message types (Quit, Move, or ChangeColor)")
            {
        {
    {
} ()fn main
    ;let msg1 = Message::Write(String::from("important text"))
    ;let msg2 = Message::Quit
    ;let msg3 = Message::Move{ x: 5, y: -5 }

    ;process_message_with_wildcard(msg1)
    ;process_message_with_wildcard(msg2)
    ;process_message_with_wildcard(msg3)
{
```

- در این مثال، حالت Write به طور خاص مدیریت می‌شود.
- تمام حالت‌های دیگر (Quit, Move, ChangeColor) با الگوی _ مطابقت داده شده و یک پیام عمومی چاپ می‌شود.
- الگوی _ باید آخرین الگو در match باشد، چون همه چیز را می‌گیرد.

نکته مهم:

- استفاده از _ راحت است، اما ممکن است باعث شود فراموش کنید حالت‌های جدیدی که بعداً به enum اضافه می‌شوند را مدیریت کنید.
- اگر می‌خواهید با اضافه شدن حالت جدید به enum کامپایلر به شما خطا دهد، بهتر است تمام حالت‌ها را صریحاً بنویسید (یا حداقل آنهایی که رفتار متفاوتی دارند).

پرسش‌های کلاسی:

۱. دستور match در Rust چه کاری انجام می‌دهد؟

- پاسخ کوتاه: یک مقدار را با الگوهای مختلف مقایسه کرده و کد مربوط به اولین الگوی مطابق را اجرا می‌کند.

۲. چرا match در Rust باید جامع (exhaustive) باشد؟

- پاسخ کوتاه: برای اطمینان از اینکه تمام حالت‌های ممکن پوشش داده شده‌اند و از خطاهای زمان اجرا جلوگیری شود (ویژگی ایمنی).

۳. الگوی _ در match چیست و چه زمانی استفاده می‌شود؟

- پاسخ کوتاه: یک الگوی wildcard (catch-all) است که با هر مقداری که الگوهای قبلی پوشش نداده‌اند مطابقت دارد. برای مدیریت حالت‌های باقی‌مانده یا ایجاد یک حالت پیش‌فرض استفاده می‌شود.

۴. چه مشکلی ممکن است در استفاده بیش از حد از _ پیش بیاید؟

- پاسخ کوتاه: اگر حالت جدیدی به enum اضافه شود، _ آن را پوشش می‌دهد و کامپایلر خطا نمی‌دهد، که ممکن است مطلوب نباشد.

پردازش خطا و مقادیر اختیاری در Rust

در بسیاری از زبان‌های برنامه‌نویسی، خطاها با مکانیزم‌هایی مانند exceptions مدیریت می‌شوند. Rust رویکرد متفاوتی دارد.

Rust ما را تشویق می‌کند تا احتمال خطا یا نبودن مقدار را به عنوان بخشی از نوع داده (data type) در نظر بگیریم.

دو ابزار اصلی Rust برای این کار عبارتند از:

۱. **Option<T>**: برای مقادیری که ممکن است وجود داشته باشند یا نداشته باشند (مقادیر اختیاری).

۲. **Result<T, E>**: برای عملیاتی که ممکن است موفقیت‌آمیز باشند (و مقداری برگردانند) یا با خطا مواجه شوند.

---

۱. Option<T>: مقادیر اختیاری

گاهی اوقات یک تابع یا عملیات ممکن است مقداری برای برگرداندن داشته باشد، و گاهی هم نه.

- مثال: جستجوی یک آیتم در لیست. ممکن است آیتم پیدا شود، یا نشود.

برای این موارد، Rust از enum ی به نام Option<T> استفاده می‌کند.

مفهوم T (جنریک ساده):

- T در Option<T> یک جانگهدار نوع (Type Placeholder) است.

- یعنی Option به خودی خود نوع داده‌ی کاملی نیست. باید مشخص کنیم Option برای چه نوع داده‌ای است.

- T می‌تواند هر نوع داده‌ای باشد: i32, f64, String, bool, یا حتی struct ها و enum های خودمان.
- Option<i32> یعنی یک مقدار اختیاری از نوع i32.
- Option<String> یعنی یک مقدار اختیاری از نوع String.

تعریف Option<T>:

Option<T> یک enum با دو حالت است:

```
} <enum Option<T
    (Some(T), // نشان میدهد که مقداری از نوع T وجود دارد
    None,     // نشان میدهد که هیچ مقداری وجود ندارد
{
```

- Some(value): مقدار value از نوع T است.
- None: هیچ مقداری وجود ندارد.

مثال استفاده با **match** (با استفاده از **String**):

بیایید تابعی بنویسیم که اندیس شروع یک کلمه (یا زیررشته) را در یک متن پیدا کند.
منطق تابع:

۱. اگر زیررشته خالی باشد، طبق رفتار استاندارد، آن را در ابتدای رشته (اندیس ۰) در نظر می‌گیریم.

۲. اگر زیررشته طولانی‌تر از رشته اصلی باشد، قطعاً نمی‌تواند در آن پیدا شود.

۳. در غیر این صورت، از ابتدای رشته اصلی شروع می‌کنیم.

۴. در هر موقعیت، یک بخش از رشته اصلی به طول زیررشته را جدا می‌کنیم (به این بخش "پنجره" یا window می‌گوییم).

۵. این پنجره را با زیررشته مقایسه می‌کنیم.

۶. اگر برابر بودند، اندیس شروع پنجره همان جواب ماست و آن را داخل Some برمی‌گردانیم.

۷. اگر تا انتهای رشته گشتیم و پیدا نشد، None برمی‌گردانیم.

```
Rust
.Our own implementation to find a substring //
.Returns Option<usize>: Some(index) if found, None otherwise //
} <fn find_substring_index(text: &str, substring: &str) → Option<usize
    Handle edge case: empty substring is found at index 0 //
    } ()if substring.is_empty
        ;return Some(0)
    {
```

```
                            // Handle edge case: substring longer than text
                                    if substring.len() > text.len() {
                                                        return None;
                                                                    }

        // Iterate through possible starting positions for the substring.
  // The last possible starting position is text.len() - substring.len().()
                            // The range needs to be inclusive, hence '..='.
                               for i in 0..=(text.len() - substring.len()) {
                  // Get the slice ('window') of 'text' starting at 'i'
                                  // with the same length as 'substring'.
                          let window = &text[i .. i + substring.len()];;

                                // Compare the window with the substring.
                                              if window == substring {
           // Found it! Return the starting index wrapped in Some.
                                                        return Some(i);
                                                                    }
                                                                }

    // If the loop finishes without returning, the substring was not found.
                                                                    None
                                                                }

           // The main function remains the same as it just calls our function.
                                                                fn main() {
    let main_text = String::from("hello world, welcome to rust programming");;
                                                    let word_to_find = "rust";;
                                                let non_existent_word = "java";;

                                            // --- Case 1: Substring exists ---
                let result1 = find_substring_index(&main_text, word_to_find);;

           println!("Searching for '{}' in '{}'", word_to_find, main_text);;
                                                                match result1 {
                                                            Some(index) => {
          println!("Substring found starting at index: {}", index);;
                                                                    }
                                                              None => {
```

```
                                    ;println!("Substring not found.")
                                                                {
                                                                    {

                                                println!("---"); // Separator

                                        --- Case 2: Substring does not exist --- //
                                ;let result2 = find_substring_index(&main_text, non_existent_word)

                            ;println!("Searching for '{}' in '{}'", non_existent_word, main_text)
                                                                } match result2
                                                            } ≤ Some(index)
                                ;println!("Substring found starting at index: {}", index)
                                                                {
                                                              } ≤ None
                                    ;println!("Substring not found.")
                                                                {
                                                                    {

                                                println!("---"); // Separator

                                                --- Case 3: Empty substring --- //
                                                        ;"" = let empty_substring
                                ;let result3 = find_substring_index(&main_text, empty_substring)
                            ;println!("Searching for '{}' in '{}'", empty_substring, main_text)
                                                              } match result3
                                                            } ≤ Some(index)
    println!("Substring found starting at index: {}", index); // Expected: 0
                                                                {
                                                              } ≤ None
                                    ;println!("Substring not found.")
                                                                {
                                                                    {
                                                                        {
```

- باز هم خروجی تابع Option<usize> است، که با match به زیبایی می‌توان هر دو حالتِ پیدا شدن (Some) و پیدا نشدن (None) را مدیریت کرد.

- ما از match برای مدیریت هر دو حالت ممکن (Some و None) استفاده می‌کنیم.

هشدار: unwrap و expect:

Option متدهایی مانند .unwrap() و .expect() دارد که سعی می‌کنند مقدار داخل Some را استخراج کنند.

- unwrap(): اگر مقدار Some(v) باشد، v را برمی‌گرداند. اگر None باشد، برنامه **panic** می‌کند!
- expect("message"): مانند unwrap() عمل می‌کند، اما در صورت panic، پیام داده شده را نمایش می‌دهد.

panic چیست؟ panic در Rust به معنی یک خطای غیرقابل بازیابی است. وقتی panic رخ می‌دهد، برنامه معمولاً متوقف شده و خطا نمایش داده می‌شود. باید تا حد امکان از کدی که می‌تواند **panic** کند اجتناب کرد. استفاده از match بسیار امن‌تر است.

```
Rust
Example of potential panic (use with caution!) //
;let result: Option<i32> = Some(10) //
let value = result.unwrap(); // This is okay, value is 10 //

;let result_none: Option<i32> = None //
!let value_panic = result_none.unwrap(); // This will PANIC //
let value_panic_expect = result_none.expect("Value was expected here!"); // This will //
!PANIC with a message
```

---

۲. Result<T, E>: عملیات موفق یا ناموفق

گاهی اوقات یک عملیات یا تابع نه تنها ممکن است مقداری برنگرداند، بلکه ممکن است با خطا مواجه شود.

- مثال: تقسیم دو عدد. اگر مقسوم‌علیه صفر باشد، خطا رخ می‌دهد.
- مثال: خواندن یک فایل. فایل ممکن است وجود نداشته باشد یا دسترسی به آن ممکن نباشد (خطا).
- مثال: تبدیل یک رشته به عدد. رشته ممکن است قالب عددی معتبری نداشته باشد (خطا).

برای این موارد، Rust از enum ی به نام Result<T, E> استفاده می‌کند.

مفاهیم T و E (جنریک ساده):

- T: جانگهدار نوع برای مقدار موفقیت‌آمیز (Success Value).
- E: جانگهدار نوع برای مقدار خطا (Error Value).

- Result<f64, String>: نتیجه عملیاتی که در صورت موفقیت یک f64 و در صورت خطا یک String (مثلاً پیام خطا) برمی‌گرداند.
- Result<i32, io::Error>: نتیجه عملیاتی که در صورت موفقیت یک i32 و در صورت خطا، یک خطای ورودی/خروجی (io::Error) برمی‌گرداند.

تعریف Result<T, E>:

Result<T, E> یک enum با دو حالت است:

```
Rust
} <enum Result<T, E
    Ok(T),  // نشان میدهد عملیات موفق بود و مقدار T را در خود دارد
    Err(E), // نشان میدهد عملیات ناموفق بود و خطای E را در خود دارد
{
```

- Ok(value): عملیات موفق بود. value مقدار نتیجه از نوع T است.
- Err(error): عملیات ناموفق بود. error مقدار خطا از نوع E است.

مثال استفاده با **match**:

تابعی برای تقسیم که خطای تقسیم بر صفر را مدیریت می‌کند.

```
Rust
Function that divides two numbers, returning a Result //
} <fn divide(numerator: f64, denominator: f64) → Result<f64, String
    } if denominator == 0.0
        Operation failed: Return an error message inside Err //
            Err(String::from("Cannot divide by zero!"))
                                                } else {
            Operation succeeded: Return the result inside Ok //
                                    Ok(numerator / denominator)
                                                        {
                                                    {

                                                } ()fn main
                        let result1 = divide(10.0, 2.0); // Should be Ok(5.0)
let result2 = divide(5.0, 0.0);  // Should be Err("Cannot divide by zero!")

                                                } match result1
```

```
                                                    } ⇐ Ok(value)
                                        ;println!("Division successful: {}", value)
                                                                {
                                                    } ⇐ Err(error_message)
                                    ;println!("Division failed: {}", error_message)
                                                                {
                                                                    {


                                                    } match result2
                                                    } ⇐ Ok(value)
                        println!("Division successful: {}", value); // This won't run
                                                                {
                                                    } ⇐ Err(error_message)
                    println!("Division failed: {}", error_message); // This will run
                                                                {
                                                                    {
                                                                {
```

- تابع divide یک Result<f64, String> برمی‌گرداند.
- با match هر دو حالت Ok و Err را مدیریت می‌کنیم.

unwrap و expect برای Result:

Result هم متدهای .unwrap() و .expect() دارد.

- unwrap(): اگر Ok(v) باشد، v را برمی‌گرداند. اگر Err(e) باشد، **panic** می‌کند!
- expect("message"): مانند unwrap() عمل می‌کند، اما در صورت panic پیام مشخص شده را نشان می‌دهد.

باز هم تأکید می‌شود: از **unwrap** و **expect** با احتیاط فراوان استفاده کنید! **match** بسیار امن‌تر است.

عملگر ? (The Question Mark Operator):

کار با match برای Result گاهی تکراری می‌شود، مخصوصاً وقتی چندین عملیات پشت سر هم انجام می‌دهیم که همگی ممکن است خطا بدهند.

عملگر ? یک راه کوتاه‌تر برای مدیریت Result در توابعی است که خودشان Result برمی‌گردانند.

- اگر مقدار result از نوع Ok(value) باشد، value از آن استخراج می‌شود و اجرای تابع ادامه می‌یابد.

- اگر مقدار result از نوع Err(error) باشد، کل تابع فوراً همان **Err(error)** را برمی‌گرداند. (خطا را به بالا "پروپاگیت" می‌کند).

مهم: عملگر ? فقط در توابعی قابل استفاده است که نوع بازگشتی آن‌ها Result (یا Option) باشد و نوع خطای آن با نوع خطای Result ی که ? روی آن اعمال می‌شود، سازگار باشد.

مثال با ?:

فرض کنید می‌خواهیم دو تقسیم انجام دهیم و نتیجه‌ها را با هم جمع کنیم.

```Rust
} <fn divide(numerator: f64, denominator: f64) → Result<f64, String
    } if denominator == 0.0
        Err(String::from("Cannot divide by zero!"))
    } else {
        Ok(numerator / denominator)
    {
{

// Function that performs two divisions and adds the results
// It returns a Result because the divisions might fail.
} <fn perform_complex_division(a: f64, b: f64, c: f64) → Result<f64, String
    // If divide(a, b) returns Err, this function immediately returns that Err.
    // Otherwise, it extracts the Ok value into div1
    ;?let div1 = divide(a, b)

    // If divide(div1, c) returns Err, this function immediately returns that Err.
    // Otherwise, it extracts the Ok value into div2
    ;?let div2 = divide(div1, c)

    // If both divisions were successful, return the final result in Ok.
    Ok(div1 + div2) // Note: Just an example calculation
{

} ()fn main
let calculation1 = perform_complex_division(100.0, 10.0, 2.0); // 100/10=10, 10/2=5.
    // Let's make it Ok(10.0 + 5.0) → Ok(15.0) ? No, Ok(5.0) sorry this should be Ok(div2) for example
    // So: Ok(10.0 + 5.0) → Ok(15.0)
    Ok(div1 + div2) for a more complex example
} match calculation1
```

```
        ,Ok(v) ⇒ println!("Complex calculation successful: {}", v)
          ,Err(e) ⇒ println!("Complex calculation failed: {}", e)
                                                                {


let calculation2 = perform_complex_division(100.0, 0.0, 2.0); // First division
                                                fails. Err("Cannot divide by zero!")
                                                        } match calculation2
        ,Ok(v) ⇒ println!("Complex calculation successful: {}", v)
          ,Err(e) ⇒ println!("Complex calculation failed: {}", e)
                                                                {
                                                                {
```

- در perform_complex_division، اگر هر کدام از divide ها Err برگردانند، ? باعث می‌شود کل تابع perform_complex_division فوراً همان Err را برگرداند.
- این کد بسیار کوتاه‌تر و خواناتر از نوشتن match های تودرتو است.

پرسش‌های کلاسی:

۱. Option<T> برای چه مواردی استفاده می‌شود؟
   - پاسخ کوتاه: برای نمایش مقادیری که ممکن است وجود داشته باشند (Some(T)) یا نداشته باشند (None).
۲. Result<T, E> برای چه مواردی استفاده می‌شود؟
   - پاسخ کوتاه: برای نمایش نتیجه عملیاتی که ممکن است موفقیت‌آمیز باشد (Ok(T)) یا با خطا مواجه شود (Err(E)).
۳. T در Option<T> و Result<T, E> به چه معناست؟
   - پاسخ کوتاه: یک جانگهدار (placeholder) برای نوع داده‌ای که در صورت وجود (Some یا Ok) نگهداری می‌شود.
۴. امن‌ترین راه برای کار با مقادیر Option و Result چیست؟ چرا؟
   - پاسخ کوتاه: استفاده از match (یا if let / while let). زیرا ما را مجبور می‌کند هر دو حالت ممکن (مثلاً Some/None یا Ok/Err) را در نظر بگیریم و از panic جلوگیری می‌کند.
۵. متدهای unwrap() و expect() چه می‌کنند و چه خطری دارند؟

- پاسخ کوتاه: سعی می‌کنند مقدار داخل Some یا Ok را استخراج کنند، اما اگر مقدار None یا Err باشد، باعث panic (توقف برنامه) می‌شوند.

۶. عملگر ? چه کاری انجام می‌دهد و در چه توابعی می‌توان از آن استفاده کرد؟

- پاسخ کوتاه: اگر Result (یا Option) برابر Err (یا None) باشد، فوراً آن Err (یا None) را از تابع فعلی برمی‌گرداند. در غیر این صورت، مقدار داخل Ok (یا Some) را استخراج می‌کند. فقط در توابعی که خودشان Result (یا Option) برمی‌گردانند قابل استفاده است.

مجموعه Vector

در برنامه‌نویسی، اغلب نیاز به ذخیره مجموعه‌ای از داده‌ها داریم.

تا الان با آرایه‌ها (Arrays) آشنا شدید.

آرایه‌ها طول ثابتی دارند که در زمان کامپایل مشخص است.

اما در بسیاری از سناریوها، تعداد داده‌ها در زمان اجرا مشخص نیست یا تغییر می‌کند.

برای این موارد، به ساختارهای داده‌ای با طول متغیر نیاز داریم.

یکی از پرکاربردترین این ساختارها در Rust، مجموعه **Vector** است.

Vector (Vec<T>)

- Vector یک آرایه قابل تغییر اندازه (Resizable Array) است.
- داده‌ها در Vector به صورت پشت سر هم در حافظه ذخیره می‌شوند.
- این ویژگی دسترسی به عناصر را با استفاده از اندیس بسیار سریع می‌کند.
- T در Vec<T> نشان‌دهنده نوع داده‌هایی است که Vector ذخیره می‌کند.
- تمام عناصر ذخیره شده در یک Vector باید از یک نوع باشند (مگر اینکه از روش‌های خاصی مانند Enum استفاده کنیم).

ایجاد یک Vector

- می‌توانید یک Vector خالی با استفاده از تابع Vec::new() بسازید.
- نوع عناصر در این حالت توسط Rust در ادامه و از روی عناصری که اضافه می‌کنید، استنباط می‌شود.
- اگر Vector قرار است تغییر کند (مثلاً با اضافه کردن عنصر)، باید آن را به صورت mut تعریف کنید.

```Rust
```

```
} ()fn main
    Rust can infer the type if you push elements later //
    ;()let mut words = Vec::new
    <words.push("first"); // Now words is Vec<&str
    ;println!("{:?}", words)
{
```

- می‌توانید یک Vector را با مقادیر اولیه با استفاده از ماکروی vec! ایجاد کنید. این روش بسیار رایج و راحت است.

```
Rust
} ()fn main
    Create a vector with initial values //
    ;let initial_numbers = vec![1, 2, 3, 4, 5]

    Create a vector of strings //
    let fruits = vec![String::from("apple"), String::from("banana"),
                                                ;String::from("cherry")]

    ;println!("{:?}", initial_numbers)
    ;println!("{:?}", fruits)
{
```

اضافه کردن عناصر به Vector

- برای اضافه کردن یک عنصر به انتهای Vector، از متد push() استفاده می‌کنیم.
- این متد Vector را تغییر می‌دهد، پس Vector باید mut تعریف شده باشد.

```
Rust
} ()fn main
    let mut my_vector = Vec::new(); // Type will be inferred as i32

    ;my_vector.push(10)
    ;my_vector.push(20)
    ;my_vector.push(30)

    println!("{:?}", my_vector); // Output: [10, 20, 30]
{
```

- اگر ظرفیت فعلی Vector برای اضافه کردن عنصر جدید کافی نباشد، Rust به صورت خودکار حافظه بیشتری را درخواست کرده، عناصر موجود را به مکان جدید کپی کرده و سپس عنصر جدید را اضافه می‌کند. این عملیات ممکن است زمان‌بر باشد.

دسترسی به عناصر Vector

- می‌توانید با استفاده از اندیس و علامت [] به عناصر Vector دسترسی پیدا کنید. اندیس‌ها از ۰ شروع می‌شوند.
- دسترسی با [] یک قرض تغییرناپذیر (&) به عنصر مورد نظر برمی‌گرداند.
- توجه: اگر اندیسی که استفاده می‌کنید خارج از محدوده Vector باشد، برنامه با خطا (panic) متوقف می‌شود.

```Rust
fn main() {
    let numbers = vec![10, 20, 30, 40, 50];
    let third_element = &numbers[2]; // Get a reference to the third element
    println!("The third element is: {}", third_element);

    let invalid_access = &numbers[100]; // This line would cause a panic
    println!("This won't be printed: {}", invalid_access); //
}
```

- روش امن‌تر برای دسترسی به عناصر، استفاده از متد get() است.
- get() اندیس را به عنوان ورودی گرفته و یک Option<&T> برمی‌گرداند.
- اگر اندیس معتبر باشد، get() مقدار عنصر را داخل Some() برمی‌گرداند (Some(&value)).
- اگر اندیس نامعتبر باشد، get() مقدار None را برمی‌گرداند.
- با استفاده از match یا if let می‌توانید مقدار برگشتی از get() را مدیریت کنید.

```Rust
fn main() {
    let numbers = vec![10, 20, 30, 40, 50];

    // Accessing a valid element
    match numbers.get(2) {
        Some(element) => println!("The third element is: {}", element),
        None => println!("There is no third element."),
    }

    // Accessing an invalid element
    match numbers.get(100) {
        Some(element) => println!("The hundredth element is: {}", element),
```

```
    ,None ⇒ println!("There is no hundredth element.")
                                                    {
                                                {
```

حذف عناصر از Vector

- متد pop(): آخرین عنصر Vector را حذف کرده و آن را برمی‌گرداند. اگر Vector خالی باشد، None برمی‌گرداند (زیرا خروجی آن Option<T> است). این عملیات بسیار سریع است.

```
Rust
                                                } ()fn main
                                    ;let mut numbers = vec![1, 2, 3, 4, 5]

                                  let last = numbers.pop(); // last will be Some(5)
                                                    ;println!("{:?}", last)
                                println!("{:?}", numbers); // Output: [1, 2, 3, 4]

                                let last2 = numbers.pop(); // last2 will be Some(4)
                                                   ;println!("{:?}", last2)
                                   println!("{:?}", numbers); // Output: [1, 2, 3]
                                                {
```

- متد remove(index): عنصری با اندیس مشخص را حذف می‌کند و آن را برمی‌گرداند. این عملیات می‌تواند کند باشد، چون تمام عناصر بعد از اندیس حذف شده باید جابجا شوند. اگر اندیس خارج از محدوده باشد panic می‌کند.

```
Rust
                                                } ()fn main
                                        ;let mut numbers = vec![10, 20, 30, 40, 50]

let removed_element = numbers.remove(1); // Removes element at index 1 (which is 20)
                        println!("Removed: {}", removed_element); // Output: Removed: 20
                                    println!("{:?}", numbers); // Output: [10, 30, 40, 50]

                                    numbers.remove(100); // This would cause a panic //
                                                {
```

پیمایش (Iteration) روی Vector

- می‌توانید با استفاده از حلقه for روی عناصر Vector پیمایش کنید.

- پیمایش با قرض تغییرناپذیر (v&): این رایج‌ترین روش است. به شما اجازه می‌دهد عناصر را بخوانید اما آن‌ها را تغییر ندهید. این روش از Vector شما یک قرض تغییرناپذیر می‌گیرد.

```Rust
fn main() {
    let numbers = vec![10, 20, 30];

    println!("Iterating over numbers:");
    for num in &numbers { // num is of type &i32
        println!("{}", num);
    }
}
```

- پیمایش با قرض تغییرپذیر (**&mut v**): به شما اجازه می‌دهد عناصر را بخوانید و تغییر دهید. حلقه یک قرض تغییرپذیر از Vector می‌گیرد.

```Rust
fn main() {
    let mut numbers = vec![1, 2, 3];

    println!("Iterating and modifying numbers:");
    for num in &mut numbers { // num is of type &mut i32
        *num += 10; // Dereference the mutable reference and add 10
        println!("{}", num);
    }
    println!("Modified vector: {:?}", numbers); // Output: Modified vector: [11, 12, 13]
}
```

- پیمایش با گرفتن مالکیت (**v**): این روش مالکیت Vector را به حلقه for منتقل می‌کند. در هر تکرار، مالکیت یک عنصر به متغیر حلقه منتقل می‌شود. بعد از پایان حلقه، Vector دیگر قابل استفاده نیست (چون مالکیت آن منتقل شده است). این روش زمانی مفید است که می‌خواهید عناصر را پردازش کرده و دیگر نیازی به Vector اصلی ندارید.

```Rust
fn main() {
    let numbers = vec![100, 200, 300];

    println!("Iterating and consuming vector:");
    for num in numbers { // num is of type i32 (value itself, not reference)
        println!("{}", num);
    }

    println!("{:?}", numbers); // ERROR: value borrowed here after move - Cannot use
                               // numbers anymore
}
```

سایر متدهای مفید Vector

- len(): تعداد عناصر موجود در Vector را برمی‌گرداند (usize).

```
Rust
} ()fn main
;let my_vector = vec![1, 2, 3, 4, 5]
println!("Vector length: {}", my_vector.len()); // Output: Vector length: 5
{
```

- capacity(): تعداد عناصری که Vector می‌تواند قبل از نیاز به تخصیص حافظه جدید در خود جای دهد را برمی‌گرداند (usize).

```
Rust
} ()fn main
let mut my_vector: Vec<i32> = Vec::with_capacity(10); // Allocate space for 10
elements initially
println!("Initial capacity: {}", my_vector.capacity()); // Output: Initial capacity:
10 (or more)

;my_vector.push(1)
println!("Capacity after push: {}", my_vector.capacity()); // Capacity is still 10
(or initial)
{
```

قوانین مالکیت و قرض گرفتن با Vector (یادآوری)

- قوانین مالکیت و قرض گرفتن Rust برای Vectorها نیز اعمال می‌شود.
- مهمترین نکته این است که نمی‌توانید همزمان قرض تغییرناپذیر (&) و قرض تغییرپذیر (&mut) از یک Vector یا بخش‌هایی از آن داشته باشید.
- مثال: نمی‌توانید همزمان به اولین عنصر یک Vector قرض تغییرناپذیر داشته باشید و یک عنصر جدید به انتهای آن اضافه کنید. اضافه کردن ممکن است باعث تخصیص حافظه جدید و جابجایی Vector در حافظه شود که قرض‌های موجود را نامعتبر می‌کند.

```
Rust
} ()fn main
;let mut v = vec![1, 2, 3]

let first = &v[0]; // Immutable borrow starts here

v.push(6); // ERROR: cannot borrow `v` as mutable because it is also borrowed as //
immutable
```

```
    // The push operation might reallocate and move the vector data,
    // which would invalidate the 'first' reference.

    println!("The first element is: {}", first);
    println!("The first element is: {}", first); // Immutable borrow ends here

    // Now we can safely push because the previous borrow is out of scope because of
    // Non-Lexical Lifetimes (NLL)
    v.push(6);
    println!("{:?}", v);
}
```

ذخیره انواع مختلف در Vector با استفاده از Enum

- همانطور که قبلاً اشاره شد، Vectorها به صورت پیش‌فرض همگن هستند (فقط یک نوع داده را ذخیره می‌کنند).
- برای ذخیره انواع مختلف در یک Vector، می‌توانید از یک Enum استفاده کنید که هر Variant آن یکی از انواع مورد نظر شما باشد.

```
Rust
enum Cell {
    Integer(i32),
    Float(f64),
    Text(String),
}

fn main() {
    let row_of_data = vec![
        Cell::Integer(99),
        Cell::Text(String::from("example data")),
        Cell::Float(15.5),
    ];

    println!("Processing data row:");
    for cell in &row_of_data { // Iterate using immutable references
        match cell {
            Cell::Integer(i) => println!("Found an integer: {}", i),
            Cell::Float(f) => println!("Found a float: {}", f),
            Cell::Text(s) => println!("Found text: {}", s),
        }
    }
}
```

```
{
```

مثال‌های کاربردی از Vector

مثال ۱: محاسبه مجموع و میانگین اعداد در یک Vector

```
Rust
} ()fn main
    ;let numbers = vec![10, 20, 30, 40, 50]
    ;let mut sum = 0

    Summing up elements using immutable references //
    } for num in &numbers
        sum += num; // num is &i32, Rust automatically dereferences here
    {

    ;println!("Vector: {:?}", numbers)
    ;println!("Sum: {}", sum)

    Calculate average (be careful with integer division) //
    } if numbers.len() > 0
        let average = sum as f64 / numbers.len() as f64; // Cast to f64 for float
division
        ;println!("Average: {}", average)
    } else {
        ;println!("Cannot calculate average of an empty vector.")
    {
{
```

مثال ۲: ذخیره و پردازش لیستی از نام‌ها

```
Rust
} ()fn main
    ]!let mut names = vec
        ,String::from("Alice")
        ,String::from("Bob")
        ,String::from("Charlie")
    ;[

    ;println!("Original names: {:?}", names)

    Add a new name //
    ;names.push(String::from("David"))
```

```
    println!("Names after adding: {:?}", names);

    // Remove a name by index
    if names.len() > 1 {
        let removed_name = names.remove(1); // Remove "Bob"
        println!("Removed name: {}", removed_name);
    }
    println!("Names after removing: {:?}", names);

    // Iterate and print names
    println!("Final list of names:");
    for name in &names {
        println!("- {}", name);
    }
}
```

مثال ۳: فیلتر کردن عناصر Vector و ساخت Vector جدید

فرض کنید می‌خواهیم فقط اعداد زوج را از یک Vector جدا کرده و در Vector جدیدی ذخیره کنیم.

```
Rust
fn main() {
    let numbers = vec![1, 2, 3, 4, 5, 6, 7, 8, 9, 10];
    let mut even_numbers = Vec::new();

    for num in &numbers { // Iterate over references
        if num % 2 == 0 {
            even_numbers.push(*num); // Dereference num to push the value
        }
    }

    println!("Original numbers: {:?}", numbers);
    println!("Even numbers: {:?}", even_numbers);
}
```

سوالات:

۱. Vector چیست و چه تفاوتی با Array دارد؟

- پاسخ: Vector آرایه‌ای قابل تغییر اندازه است، Array طول ثابت دارد.

۲. چگونه یک Vector خالی ایجاد می‌کنید که بعداً عناصر String را در آن قرار دهید؟

- پاسخ: let mut my_strings: Vec<String> = Vec::new(); یا let mut my_strings = Vec::new(); و سپس push کردن String.

۳. برای اضافه کردن عنصر به انتهای Vector از چه متدی استفاده می‌شود؟

- پاسخ: push().

۴. تفاوت استفاده از [] و get() برای دسترسی به عنصر چیست؟

- پاسخ: [] اگر اندیس اشتباه باشد panic می‌کند، get() یک Option برمی‌گرداند.

۵. چه متدی آخرین عنصر Vector را حذف کرده و برمی‌گرداند؟

- پاسخ: pop().

۶. اگر در یک حلقه for از &mut my_vector استفاده کنیم، چه کاری می‌توانیم با عناصر انجام دهیم؟

- پاسخ: می‌توانیم آن‌ها را بخوانیم و تغییر دهیم.

۷. اگر در یک حلقه for فقط از my_vector استفاده کنیم (بدون & یا &mut)، بعد از حلقه چه اتفاقی برای my_vector می‌افتد؟

- پاسخ: مالکیت Vector به حلقه منتقل شده و بعد از حلقه دیگر قابل استفاده نیست.

۸. چگونه می‌توان تعداد عناصر Vector را به دست آورد؟

- پاسخ: با متد len().

۹. چرا نمی‌توان همزمان به یک عنصر Vector دسترسی داشت (قرض تغییرناپذیر) و یک عنصر جدید به Vector اضافه کرد؟

- پاسخ: اضافه کردن ممکن است باعث تغییر مکان Vector در حافظه شود که قرض‌های موجود را نامعتبر می‌کند.

کلمه کلیدی impl در Rust

کلمه کلیدی impl برای تعریف متدها (methods) و توابع مرتبط (associated functions) برای ساختارها (structs) و شمارشگرها (enums) استفاده می‌شود. این امکان را به شما می‌دهد تا رفتار خاصی را به انواع داده‌ای که تعریف کرده‌اید، اضافه کنید.

تعریف متدها برای ساختارها

شما می‌توانید با استفاده از impl برای یک ساختار، متدهایی تعریف کنید که بر روی نمونه‌های آن ساختار عمل می‌کنند. متدها همیشه یک پارامتر به نام self دارند که به نمونه‌ای از ساختار که متد بر روی آن فراخوانی شده است، اشاره می‌کند.

```
Rust
} struct Rectangle {
    width: u32,
    height: u32,
}

} impl Rectangle {
    } fn area(&self) -> u32 {
        self.width * self.height
    }

    } fn can_hold(&self, other: &Rectangle) -> bool {
        self.width > other.width && self.height > other.height
    }
}

} ()fn main {
    let rect1 = Rectangle { width: 30, height: 50 };
    let rect2 = Rectangle { width: 10, height: 40 };

    println!("The area of the rectangle is {}", rect1.area());
    println!("Can rect1 hold rect2? {}", rect1.can_hold(&rect2));
{
```

در مثال بالا، دو متد area و can_hold برای ساختار Rectangle تعریف شده‌اند. متد area مساحت مستطیل را محاسبه می‌کند و متد can_hold بررسی می‌کند که آیا یک مستطیل می‌تواند مستطیل دیگری را در خود جای دهد یا خیر.

سوالات:

۱. کلمه کلیدی impl در Rust برای چه منظوری استفاده می‌شود؟
   - o تعریف متدها و توابع مرتبط برای ساختارها و شمارشگرها.
۲. پارامتر self در متدهای یک ساختار به چه چیزی اشاره می‌کند؟
   - o به نمونه‌ای از ساختار که متد بر روی آن فراخوانی شده است.
۳. در مثال بالا، متد area چه کاری انجام می‌دهد؟

- مساحت مستطیل را محاسبه می‌کند.

توابع مرتبط (Associated Functions)

علاوه بر متدها، شما می‌توانید توابع مرتبط را با استفاده از impl برای یک ساختار یا شمارشگر تعریف کنید. این توابع هیچ پارامتر self ندارند و معمولاً برای ایجاد نمونه‌های جدید از ساختار یا انجام کارهای مرتبط با آن نوع داده استفاده می‌شوند.

```
Rust
} struct Rectangle
   ,width: u32
  ,height: u32
                {

} impl Rectangle
    } fn square(size: u32) -> Self
Rectangle { width: size, height: size }
                    {
                {

} ()fn main
    ;let sq = Rectangle::square(20)
;println!("The width of the square is {}", sq.width)
                {
```

در این مثال، تابع مرتبط square برای ساختار Rectangle تعریف شده است. این تابع یک اندازه را به عنوان ورودی می‌گیرد و یک نمونه جدید از Rectangle با عرض و ارتفاع برابر با آن اندازه برمی‌گرداند. توجه کنید که برای فراخوانی توابع مرتبط از عملگر :: استفاده می‌شود.

سوالات:

۱. تفاوت اصلی بین متدها و توابع مرتبط در چیست؟
   - متدها پارامتر self دارند، در حالی که توابع مرتبط ندارند.
۲. از چه عملگری برای فراخوانی توابع مرتبط استفاده می‌شود؟
   - عملگر ::.
۳. در مثال بالا، تابع مرتبط square چه کاری انجام می‌دهد؟
   - یک نمونه جدید از Rectangle با عرض و ارتفاع برابر با اندازه داده شده ایجاد می‌کند.

تعریف متدها برای شمارشگرها

شما همچنین می‌توانید با استفاده از impl برای یک شمارشگر، متدهایی تعریف کنید که بر روی نمونه‌های آن شمارشگر عمل می‌کنند.

```Rust
} enum Message
    ,Quit
    ,Move { x: i32, y: i32 }
    ,Write(String)
    ,ChangeColor(i32, i32, i32)
{

} impl Message
    } fn process(&self)
        } match self
            ,Message::Quit ⇒ println!("Quit")
            ,Message::Move { x, y } ⇒ println!("Move to x={}, y={}", x, y)
            ,Message::Write(text) ⇒ println!("Write: {}", text)
            Message::ChangeColor(r, g, b) ⇒ println!("Change color to r={}, g={},
                ,b={}", r, g, b)
                {
                    {
                    {

} ()fn main
    ;let msg1 = Message::Quit
    ;let msg2 = Message::Move { x: 10, y: 20 }
    ;let msg3 = Message::Write(String::from("Hello"))

    ;()msg1.process
    ;()msg2.process
    ;()msg3.process
{
```

در این مثال، متد process برای شمارشگر Message تعریف شده است. این متد با توجه به نوع پیام، عمل متفاوتی را انجام می‌دهد.

سوالات:

۱. آیا می‌توان برای شمارشگرها متد تعریف کرد؟ اگر بله، چگونه؟
   - بله، با استفاده از کلمه کلیدی impl برای شمارشگر.
۲. در مثال بالا، متد process چه کاری انجام می‌دهد؟
   - با توجه به نوع پیام در شمارشگر Message، عمل متفاوتی را انجام می‌دهد.

چند نکته مهم

- شما می‌توانید چندین بلوک impl برای یک نوع داده داشته باشید. این کار می‌تواند کد شما را بهتر سازماندهی کند.
- متدها می‌توانند مقادیر را به صورت self (مالکیت منتقل می‌شود)، &self (قرض گرفته می‌شود) یا &mut self (قرض گرفته می‌شود و قابل تغییر است) بگیرند. این موضوع به قوانین مالکیت و قرض گرفتن در Rust مربوط می‌شود که قبلاً با آن آشنا شده‌اید.

سوالات:

۱. آیا می‌توان چندین بلوک impl برای یک نوع داده تعریف کرد؟
   - o بله.

۲. سه روش برای گرفتن self در متدهای Rust را نام ببرید.
   - o self, &self, &mut self.

کلوژرها (Closures) در Rust

کلوژر چیست؟

- کلوژرها توابع بی‌نامی هستند که می‌توانید آن‌ها را در متغیر ذخیره کنید.
- آن‌ها می‌توانند مقادیر محیطی که در آن تعریف شده‌اند را "capture" یا به خود جذب کنند.
- کلوژرها در Rust بسیار قدرتمند و پرکاربرد هستند، مخصوصاً با متدهای مجموعه‌ها.

سینتکس پایه کلوژر:

```Rust
} |let my_closure = |parameter1, parameter2
        code inside closure //
parameter1 + parameter2 // example return value
;{
```

- پارامترها بین | قرار می‌گیرند.
- بدنه کلوژر در {} نوشته می‌شود (اگر تک خطی باشد، {} اختیاری است).
- Rust معمولاً نوع پارامترها و مقدار بازگشتی را استنتاج می‌کند.

مثال ساده:

```Rust
} ()fn main
} let add_one = |x: i32| → i32
```

```
    x + 1
};

let five = 5;
let six = add_one(five);
println!("Result: {}", six); // Output: Result: 6

// Closure without type annotation (inferred)
let multiply = |a, b| a * b;
let product = multiply(3, 4);
println!("Product: {}", product); // Output: Product: 12
}
```

کلوژرها و محیط اطراف:

کلوژرها می‌توانند متغیرهای محیطی که در آن تعریف می‌شوند را استفاده کنند.

```Rust
fn main() {
let factor = 10;
let multiply_by_factor = |num| num * factor; // 'factor' is captured

let value = 5;
let result = multiply_by_factor(value);
println!("{} times {} is {}", value, factor, result); // Output: 5 times 10 is 50
}
```

سوالات برای این بخش:

۱. کلوژر (Closure) چیست؟

۲. کلوژر چگونه پارامترها را دریافت می‌کند؟

۳. منظور از "capture کردن محیط" توسط کلوژر چیست؟

ایتریتورها (Iterators) در Rust

ایتریتور چیست؟

- ایتریتور راهی برای پیمایش دنباله‌ای از آیتم‌ها است.
- در Rust، ایتریتورها "تنبل" (lazy) هستند؛ یعنی تا زمانی که مقادیر آن‌ها را مصرف نکنید، کاری انجام نمی‌دهند.
- بسیاری از انواع داده در Rust، مانند Vec، متدهایی برای تولید ایتریتور دارند.

متد .iter():

- متد .iter() یک ایتریتور روی ارجاع‌های غیرقابل تغییر (&T) به آیتم‌های مجموعه ایجاد می‌کند.

```
Rust
} ()fn main
    ;let numbers = vec![10, 20, 30]

                    Create an iterator //
    ;()let numbers_iter = numbers.iter

Use the iterator in a for loop (implicitly calls .next()) //
                        ;println!("Using for loop:")
                    } for num_ref in numbers_iter
        ;println!("Got number reference: {}", num_ref)
                                                {

Note: The original vector 'numbers' is still available //
                    ;println!("Original vector: {:?}", numbers)
}
```

حلقه **for** و ایتریتورها:

- حلقه for در Rust در واقع یک راه ساده برای کار با ایتریتورها است.
- وقتی روی یک مجموعه for می‌زنید، Rust به طور خودکار .into_iter() را فراخوانی می‌کند.

سوالات برای این بخش:

۱. ایتریتور چه کاری انجام می‌دهد؟

۲. منظور از "تنبل" بودن ایتریتورها چیست؟

۳. متد .iter() چه نوع ایتریتوری تولید می‌کند؟

متدهای ایتریتور: filter

آداپتورهای ایتریتور (**Iterator Adapters**):

- آداپتورها متدهایی هستند که روی یک ایتریتور فراخوانی می‌شوند و ایتریتور دیگری تولید می‌کنند.
- این متدها عملیات مختلفی مانند فیلتر کردن، نگاشت (mapping) و ... انجام می‌دهند.

- filter یکی از این آداپتورها است.

استفاده از filter:

- filter یک کلوژر به عنوان آرگومان می‌گیرد.
- این کلوژر برای هر آیتم اجرا می‌شود و باید true یا false برگرداند.
- ایتریتور جدید فقط شامل آیتم‌هایی است که کلوژر برای آن‌ها true برگردانده است.

مثال: فیلتر کردن اعداد زوج با **filter**:

```
Rust
} ()fn main
    ;let numbers = vec![1, 2, 3, 4, 5, 6]

                    Create an iterator, filter even numbers //
    ;let even_numbers_iter = numbers.iter().filter(|&&num| num % 2 == 0)
Note: |&&num| is used because iter() gives &i32, filter passes &(&i32) to the //
                                                                    .closure
                    .We dereference twice to get the i32 value //

                    ;println!("Even numbers (iterator version):")
        ()We need to consume the iterator, e.g., with a for loop or collect //
                    } for even_num in even_numbers_iter
                        ;println!("{}", even_num)
                                                {
                                                    {
```

همان کار با حلقه **for**:

```
Rust
} ()fn main
    ;let numbers = vec![1, 2, 3, 4, 5, 6]
    ;println!("\nEven numbers (for loop version):")
for &num in numbers.iter() { // Iterate over references, dereference with &num
                                                } if num % 2 == 0
                        ;println!("{}", num)
                                                    {
                                                        {
                                                    {
```

سوالات برای این بخش:

۱. آداپتور ایتریتور چیست؟

۲. متد filter چه کاری انجام می‌دهد؟

۳. آرگومان متد filter چیست؟

۴. چرا در کلوژر مثال filter از num&& استفاده شد؟

متدهای ایتریتور: collect

مصرف‌کننده‌های ایتریتور (**Iterator Consumers**):

- آداپتورها ایتریتورهای جدید می‌سازند، اما کاری انجام نمی‌دهند تا زمانی که مصرف شوند.
- مصرف‌کننده‌ها متدهایی هستند که ایتریتور را پیمایش کرده و نتیجه‌ای تولید می‌کنند.
- collect یکی از رایج‌ترین مصرف‌کننده‌ها است.

استفاده از **collect**:

- collect آیتم‌های یک ایتریتور را جمع‌آوری کرده و به یک مجموعه جدید تبدیل می‌کند.
- نوع مجموعه مقصد باید مشخص باشد (Rust معمولاً می‌تواند آن را استنتاج کند، اما گاهی نیاز به تعیین صریح دارد).

مثال: فیلتر کردن و جمع‌آوری با **filter** و **collect**:

```
Rust
} ()fn main
    ;let numbers = vec![1, 2, 3, 4, 5, 6]

                    Filter even numbers and collect them into a new Vec //
let even_numbers: Vec<_> = numbers // The type Vec<_> asks Rust to infer the element
                                                                type (&i32)
                        iter()                      // Get iterator of &i32.
                filter(|&&num| num % 2 == 0) // Keep only even numbers (&i32).
        <collect();                  // Collect the &i32 into a new Vec<&i32.

            ;println!("Collected even numbers (references): {:?}", even_numbers)

                    If we want a Vec<i32> instead of Vec<&i32>, we can use map //
                                let even_numbers_values: Vec<i32> = numbers
                                                                ()iter.
```

```
                                        filter(|&&num| num % 2 == 0).
                                map(|&num| num) // Dereference the &i32 to get i32.
                                                                ;()collect.

        ;println!("Collected even numbers (values): {:?}", even_numbers_values)
                                                                                {
```

همان کار با حلقه **for**:

```
                                                                        Rust
                                                                } ()fn main {
                                                    ;let numbers = vec![1, 2, 3, 4, 5, 6]
  let mut even_numbers_for = Vec::new(); // Create an empty Vec to store results

                                                            } ()for &num in numbers.iter
                                                                    } if num % 2 == 0
                                        even_numbers_for.push(num); // Push the value (i32)
                                                                                {
                                                                                    {

;println!("\nCollected even numbers (for loop version): {:?}", even_numbers_for)
                                                                                {
```

سوالات برای این بخش:

۱. مصرف‌کننده ایتریتور چیست؟

۲. متد collect چه کاری انجام می‌دهد؟

۳. چرا گاهی نیاز است نوع مجموعه حاصل از collect را مشخص کنیم؟

۴. تفاوت Vec<&i32> و Vec<i32> در مثال چیست؟

ترکیب کلوژرها و متدهای ایتریتور

می‌توانید چندین آداپتور ایتریتور را با هم زنجیر کنید تا عملیات پیچیده‌تری انجام دهید.

مثال: پیدا کردن اعداد زوج، دو برابر کردن آن‌ها و جمع‌آوری نتایج:

```
                                                                        Rust
                                                                } ()fn main {
                                                        ;let data = vec![1, 2, 3, 4, 5]

                                                    Using iterators, filter, map, and collect //
                                                            let processed_data: Vec<i32> = data
```

```
                                iter()                  // Iterator over &i32.
                            filter(|&&x| x % 2 == 0) // Keep even numbers (&i32).
                    map(|&x| x * 2)         // Double the even numbers (produces i32).
                        <collect();             // Collect i32 values into a Vec<i32.

println!("Processed data (iterator version): {:?}", processed_data); // Output: [4,
                                                                                8]
                                                                                {
```

همان کار با حلقه **for**:

```
                                                                        Rust
                                                                } ()fn main
                                                    ;let data = vec![1, 2, 3, 4, 5]
                                                ;()let mut processed_data_for = Vec::new

                                                            } ()for &x in data.iter
                                            if x % 2 == 0 {       // Filter condition
                                            let doubled = x * 2; // Map operation
                                    processed_data_for.push(doubled); // Collect step
                                                                                {
                                                                                    {

        println!("\nProcessed data (for loop version): {:?}", processed_data_for); //
                                                                        Output: [4, 8]
                                                                                {
```

مزایای استفاده از ایتریتورها:

- خوانایی: کد اغلب کوتاه‌تر و بیانگرتر است.
- کارایی: به دلیل تنبلی (laziness) و بهینه‌سازی‌های کامپایلر، می‌تواند بسیار کارآمد باشد (اغلب بهینه تر از حلقه‌های دستی).
- ترکیب‌پذیری: به راحتی می‌توان عملیات مختلف را زنجیر کرد.

سوالات برای این بخش:

۱. متد map روی ایتریتور چه کاری انجام می‌دهد؟
۲. چگونه می‌توان چندین آداپتور ایتریتور را با هم استفاده کرد؟
۳. یک مزیت استفاده از ایتریتورها نسبت به حلقه for چیست؟

Map:

- یک کلوژر می‌گیرد و آن را روی هر عنصر ایتریتور اعمال می‌کند.
- نتیجه، یک ایتریتور جدید با عناصر تبدیل شده است.

```Rust
fn main() {
    let numbers = vec![1, 2, 3];

    // Iterator version: Square each number
    let squares: Vec<i32> = numbers.iter().map(|&x| x * x).collect();
    println!("Squares (iterator): {:?}", squares); // Output: [1, 4, 9]

    // For loop version
    let mut squares_for = Vec::new();
    for &x in numbers.iter() {
        squares_for.push(x * x);
    }
    println!("Squares (for loop): {:?}", squares_for); // Output: [1, 4, 9]
}
```

۳. take(n)

- فقط n عنصر اول ایتریتور را برمی‌گرداند.
- بقیه عناصر نادیده گرفته می‌شوند.

```Rust
fn main() {
    let numbers = vec![10, 20, 30, 40, 50];

    // Take the first 3 numbers
    let first_three: Vec<&i32> = numbers.iter().take(3).collect();
    println!("First three: {:?}", first_three); // Output: [10, 20, 30]
}
```

۴. skip(n)

- n عنصر اول ایتریتور را نادیده می‌گیرد.
- بقیه عناصر را برمی‌گرداند.

```
Rust
} ()fn main
    ;let numbers = vec![10, 20, 30, 40, 50]

    Skip the first 2 numbers //
    ;()let skip_two: Vec<&i32> = numbers.iter().skip(2).collect
println!("Skipping first two: {:?}", skip_two); // Output: [30, 40, 50]
{
```

۵. enumerate()

- ایتریتوری از زوج‌های (index, element) تولید می‌کند.
- اندیس‌ها از ۰ شروع می‌شوند.

```
Rust
} ()fn main
;let names = vec!["Alice", "Bob", "Charlie"]

Enumerate names //
;println!("Enumerated names:")
} ()for (i, name) in names.iter().enumerate
    ;println!("Index {}: {}", i, name)
{
:Output //
Index 0: Alice //
Index 1: Bob //
Index 2: Charlie //
{
```

Traitها در Rust

Traitها در Rust راهی برای تعریف رفتار مشترک بین انواع مختلف داده هستند. آن‌ها به شما اجازه می‌دهند تا مجموعه‌ای از متدها را مشخص کنید که یک نوع داده باید پیاده‌سازی کند. Traitها مشابه رابط‌ها (interfaces) در زبان‌های دیگر هستند.

تعریف یک Trait

برای تعریف یک Trait، از کلمه کلیدی trait استفاده می‌کنید و سپس نام Trait و بدنه آن را مشخص می‌کنید. در بدنه Trait، شما امضای متدهایی را تعریف می‌کنید که هر نوع داده‌ای که این Trait را پیاده‌سازی می‌کند، باید آن‌ها را داشته باشد.

```
Rust
} trait Summary
```

```
    ;fn summarize(&self) → String
{
```

در این مثال، یک Trait به نام Summary تعریف شده است که یک متد به نام summarize دارد. این متد هیچ پارامتری نمی‌گیرد (به جز self) و یک مقدار از نوع String برمی‌گرداند.

سوالات:

۱. Trait در Rust چیست؟
   - o راهی برای تعریف رفتار مشترک بین انواع داده.
۲. از چه کلمه کلیدی برای تعریف یک Trait استفاده می‌شود؟
   - o trait.
۳. در مثال بالا، Trait Summary چه متدی را تعریف می‌کند؟
   - o متد summarize.

پیاده‌سازی یک Trait برای یک نوع داده

برای اینکه یک نوع داده (مانند ساختار یا شمارشگر) از یک Trait استفاده کند، باید آن Trait را برای آن نوع داده پیاده‌سازی کنید. برای این کار، از کلمه کلیدی impl استفاده می‌کنید و سپس نام Trait و کلمه کلیدی for و نام نوع داده را می‌آورید.

```
Rust
 } struct NewsArticle
,headline: String
,location: String
  ,author: String
  ,content: String
{


    } impl Summary for NewsArticle
 } fn summarize(&self) → String
format!("{}, by {} ({})", self.headline, self.author, self.location)
                                                                {
                                                                {


          } struct Tweet
,username: String
  ,content: String
    ,retweets: u32
{
```

```
} impl Summary for Tweet
    } fn summarize(&self) → String
        format!("{}: {}", self.username, self.content)
                                                    {
                                                        {

} ()fn main
    } let article = NewsArticle
        ,headline: String::from("Penguins win the Stanley Cup Championship!")
                ,location: String::from("Pittsburgh, PA, USA")
                        ,author: String::from("Iceburgh")
content: String::from("The Pittsburgh Penguins defeated the Detroit Red Wings 4-
    ,3 in Game 7 of the Stanley Cup Finals to win their fourth Stanley Cup championship!")
                                                    ;{

    } let tweet = Tweet
            ,username: String::from("horse_ebooks")
        ,content: String::from("today was a good day")
                ,retweets: 100
                                                    ;{

    ;println!("New article available! {}", article.summarize())
        ;println!("New tweet! {}", tweet.summarize())
{
```

در این مثال، Trait Summary برای دو ساختار NewsArticle و Tweet پیاده‌سازی شده است. هر کدام از این ساختارها، متد summarize را به روش خاص خود پیاده‌سازی می‌کنند.

سوالات:

۱. چگونه یک Trait برای یک ساختار پیاده‌سازی می‌شود؟
   - با استفاده از کلمه کلیدی impl، نام Trait، کلمه کلیدی for و نام ساختار.
۲. آیا دو ساختار می‌توانند یک Trait یکسان را به روش‌های مختلف پیاده‌سازی کنند؟
   - بله.
۳. در مثال بالا، متد summarize برای NewsArticle چه چیزی را برمی‌گرداند؟
   - یک خلاصه از عنوان، نویسنده و مکان خبر.

متدهای پیش‌فرض در **Trait**ها

شما می‌توانید متدهای پیش‌فرض را در یک Trait تعریف کنید. این متدها یک پیاده‌سازی پیش‌فرض دارند که انواع داده‌ای که Trait را پیاده‌سازی می‌کنند، می‌توانند از آن استفاده کنند یا آن را با پیاده‌سازی خاص خودشان جایگزین کنند.

```
Rust
} trait Summary
    ;fn summarize_author(&self) → String

    } fn summarize(&self) → String
        format!("(Read more from {} ...)", self.summarize_author())
    {
{

} struct Tweet
    ,username: String
    ,content: String
    ,retweets: u32
{

} impl Summary for Tweet
    } fn summarize_author(&self) → String
        format!("@{}", self.username)
    {
{

} ()fn main
    } let tweet = Tweet
        ,username: String::from("horse_ebooks")
        ,content: String::from("today was a good day")
        ,retweets: 100
    ;{

    ;println!("New tweet! {}", tweet.summarize())
{
```

در این مثال، Trait Summary دارای یک متد پیش‌فرض به نام summarize است که از متد summarize_author استفاده می‌کند. ساختار Tweet فقط متد summarize_author را پیاده‌سازی می‌کند و از پیاده‌سازی پیش‌فرض summarize استفاده می‌کند.

سوالات:

۱. آیا می‌توان متدهای پیش‌فرض در یک Trait تعریف کرد؟

- بله.

۲. اگر یک نوع داده یک متد پیش‌فرض را در Trait پیاده‌سازی نکند، چه اتفاقی می‌افتد؟

- از پیاده‌سازی پیش‌فرض آن متد استفاده می‌شود.

۳. در مثال بالا، ساختار Tweet کدام متد از Trait Summary را به صورت مستقیم پیاده‌سازی می‌کند؟

- متد summarize_author.

استفاده از Traitها به عنوان پارامتر

شما می‌توانید از Traitها به عنوان نوع پارامتر در توابع استفاده کنید. این کار به شما اجازه می‌دهد تا توابعی بنویسید که می‌توانند بر روی هر نوع داده‌ای که یک Trait خاص را پیاده‌سازی می‌کند، عمل کنند.

```Rust
} fn notify(item: &impl Summary)
;println!("Breaking news! {}", item.summarize())
{

} struct NewsArticle
,headline: String
,location: String
,author: String
,content: String
{

} impl Summary for NewsArticle
} fn summarize(&self) → String
format!("{}, by {} ({})", self.headline, self.author, self.location)
{
{

} ()fn main
} let article = NewsArticle
,headline: String::from("Penguins win the Stanley Cup Championship!")
,location: String::from("Pittsburgh, PA, USA")
,author: String::from("Iceburgh")
content: String::from("The Pittsburgh Penguins defeated the Detroit Red Wings 4-
,3 in Game 7 of the Stanley Cup Finals to win their fourth Stanley Cup championship!")
;{
```

```
;notify(&article)
{
```

در این مثال، تابع notify یک پارامتر به نام item می‌گیرد که هر نوع داده‌ای می‌تواند باشد به شرطی که Trait Summary را پیاده‌سازی کرده باشد.

سوالات:

۱. آیا می‌توان از Traitها به عنوان نوع پارامتر در توابع استفاده کرد؟
   - o بله.
۲. مزیت استفاده از Traitها به عنوان پارامتر چیست؟
   - o امکان نوشتن توابعی که می‌توانند بر روی انواع مختلف داده با رفتار مشابه عمل کنند.
۳. در مثال بالا، تابع notify چه نوع پارامتری می‌گیرد؟
   - o پارامتری که Trait Summary را پیاده‌سازی کرده باشد.

مثال کامل: Trait برای اشکال هندسی (مساحت و محیط)

هدف

- تعریف یک رفتار مشترک برای اشکال هندسی جهت محاسبه مساحت و محیط.
- پیاده‌سازی این رفتار برای اشکال Circle, Rectangle, Triangle, و Ellipse.
- ایجاد یک تابع عمومی که بتواند جزئیات (مساحت و محیط) هر شکلی که این رفتار را دارد، چاپ کند.
- درک عمیق‌تر مفهوم **Trait Bound**.

۱. بازتعریف **Trait Shape**

- Trait Shape دو متد دارد:
  - o area(): مساحت شکل را به صورت f64 برمی‌گرداند.
  - o perimeter(): محیط شکل را به صورت f64 برمی‌گرداند.

Rust

```
use std::f64::consts::PI; // برای استفاده از عدد π


// تعریف Trait برای اشکال هندسی با دو رفتار
```

```
} trait Shape
    ;fn area(&self) → f64
;fn perimeter(&self) → f64
{
```

۲. تعریف **Struct**ها برای اشکال

- ساختارهای لازم برای چهار شکل هندسی را تعریف می‌کنیم:

```
Rust
// دایره: فقط شعاع نیاز دارد
} struct Circle
    ,radius: f64
{

// مستطیل: به طول و عرض نیاز دارد
} struct Rectangle
    ,width: f64
    ,height: f64
{

// مثلث: طول سه ضلع را نگه میداریم (برای محاسبه محیط و مساحت با فرمول هرون)
} struct Triangle
    ,a: f64, // ضلع اول
    ,b: f64, // ضلع دوم
    ,c: f64, // ضلع سوم
{

// بیضی: به شعاع بزرگ (نیم قطر بزرگ) و شعاع کوچک (نیمقطر کوچک) نیاز دارد
} struct Ellipse
    semi_major_axis: f64, // a
    semi_minor_axis: f64, // b
{
```

۳. پیاده‌سازی **Trait Shape** برای هر **Struct**

- حالا برای هر یک از structها، متدهای area و perimeter را مطابق با فرمول‌های هندسی پیاده‌سازی می‌کنیم.

برای **Circle**:

```
Rust
} impl Shape for Circle
```

```
    } fn area(&self) → f64 {
        // مساحت دایره: π * r^2
        PI * self.radius * self.radius
    }

    } fn perimeter(&self) → f64 {
        // محیط دایره: π * r * 2
        PI * self.radius * 2.0
    }
}
```

برای **Rectangle**:

```
Rust
} impl Shape for Rectangle {
    } fn area(&self) → f64 {
        // مساحت مستطیل: width * height
        self.width * self.height
    }

    } fn perimeter(&self) → f64 {
        // محیط مستطیل: 2 * (width + height)
        (self.width + self.height) * 2.0
    }
}
```

برای **Triangle**:

- برای محاسبه مساحت از فرمول هرون استفاده می‌کنیم: $Area=s(s-a)(s-b)(s-c)$ که $s=(a+b+c)/2$ (نصف محیط) است.
- محیط برابر با جمع سه ضلع است.

```
Rust
} impl Shape for Triangle {
    } fn area(&self) → f64 {
        // محاسبه نصف محیط (s)
        let s = (self.a + self.b + self.c) / 2.0;
        // فرمول هرون برای مساحت
        // .sqrt() برای محاسبه جذر استفاده میشود
        (s * (s - self.a) * (s - self.b) * (s - self.c)).sqrt()
    }
```

```
    } fn perimeter(&self) → f64
    a + b + c :محیط مثلث  //
    self.a + self.b + self.c
                            {
{
```

برای **Ellipse**:

- مساحت بیضی: π×a×b

- محیط بیضی فرمول ساده و دقیقی ندارد. از یک تقریب رایج استفاده می‌کنیم: [P≈π[23(a+b)−ab.

```
Rust
    } impl Shape for Ellipse
        } fn area(&self) → f64
    π * a * b :مساحت بیضی  //
PI * self.semi_major_axis * self.semi_minor_axis
                            {

    } fn perimeter(&self) → f64
// استفاده از تقریب رامانوجان (ساده شده) برای محیط بیضی
                ;let a = self.semi_major_axis
                ;let b = self.semi_minor_axis
            P ≈ π [ 3/2(a+b) - sqrt(ab) ] //
    PI * (1.5 * (a + b) - (a * b).sqrt())
// توجه: این یک تقریب است، نه مقدار دقیق.
                            {
{
```

۴. توضیح در مورد Trait Bound

- وقتی تابعی مانند زیر را تعریف می‌کنیم:

```
Rust
} fn print_shape_details<T: Shape>(shape: &T)
                        ... //
{
```

- بخش <T: Shape> به عنوان **Trait Bound** شناخته می‌شود.

- **T** یک پارامتر نوع جنریک (**Generic**) است. یعنی T می‌تواند نماینده هر نوع داده‌ای باشد.

- علامت : بعد از T نشان می‌دهد که ما در حال اعمال یک محدودیت (**Constraint**) روی T هستیم.
- Shape نام Trait است که به عنوان محدودیت اعمال شده است.
- کل عبارت **T: Shape** به این معنی است: "**T** می‌تواند هر نوعی باشد، به شرطی که آن نوع، **Trait Shape** را پیاده‌سازی کرده باشد."

چرا **Trait Bound** مفید است؟

۱. اطمینان در زمان کامپایل: کامپایلر Rust با دیدن T: Shape مطمئن می‌شود که هر نوعی که به جای T قرار می‌گیرد، قطعاً متدهای تعریف شده در Trait Shape (یعنی area() و perimeter()) را دارد.

۲. فراخوانی متدهای **Trait**: به همین دلیل، درون تابع print_shape_details، ما می‌توانیم با اطمینان shape.area() و shape.perimeter() را فراخوانی کنیم. کامپایلر می‌داند این متدها وجود دارند، چون T مجبور بوده Shape را پیاده‌سازی کند.

۳. کد قابل استفاده مجدد (**Reusability**): به جای نوشتن توابع جداگانه print_circle_details, print_rectangle_details و غیره، یک تابع واحد (print_shape_details) می‌نویسیم که با همه انواع داده‌ای که Shape را پیاده‌سازی می‌کنند، کار می‌کند.

۴. پلی‌مورفیسم (**Polymorphism**) در زمان کامپایل: این تابع یک رفتار (چاپ جزئیات) را به صورت یکسان برای انواع مختلف (Circle, Rectangle, ...) اعمال می‌کند، در حالی که پیاده‌سازی واقعی متدهای area و perimeter برای هر نوع متفاوت است. این نوع پلی‌مورفیسم از طریق generics و trait bounds، معمولاً بسیار کارآمد است زیرا کامپایلر می‌تواند کد را برای هر نوع خاص بهینه‌سازی کند (static dispatch).

۵. استفاده از Trait و تابع عمومی

- حالا یک تابع عمومی می‌نویسیم که جزئیات هر Shape را چاپ کند و از آن در main استفاده می‌کنیم.

```Rust
// تابع عمومی که جزئیات هر شکلی که Shape را پیادهسازی کرده، چاپ میکند
fn print_shape_details<T: Shape>(shape_name: &str, shape: &T) {
    println!("--- Details for {} ---", shape_name);
    println!("Area: {:.2}", shape.area()); // چاپ مساحت با دو رقم اعشار
    println!("Perimeter: {:.2}", shape.perimeter()); // چاپ محیط با دو رقم اعشار
    println!("----------------------------");
}

// تابع اصلی برنامه
fn main() {
    let circle = Circle { radius: 5.0 };
```

```
    let rectangle = Rectangle { width: 10.0, height: 4.0 };
    // یک مثلث معتبر (مجموع هر دو ضلع بزرگتر از ضلع سوم)
    let triangle = Triangle { a: 3.0, b: 4.0, c: 5.0 }; // مثلث قائمالزاویه
    let ellipse = Ellipse { semi_major_axis: 7.0, semi_minor_axis: 3.0 };

    // استفاده از تابع عمومی برای چاپ جزئیات هر شکل
    print_shape_details("Circle", &circle);
    print_shape_details("Rectangle", &rectangle);
    print_shape_details("Triangle (3-4-5)", &triangle);
    print_shape_details("Ellipse", &ellipse);
}
```

کد کامل برنامه

```
Rust
use std::f64::consts::PI; // برای استفاده از عدد π

// ۱. تعریف Trait Shape
trait Shape {
    fn area(&self) -> f64;
    fn perimeter(&self) -> f64;
}

// ۲. تعریف Struct ها
struct Circle {
    radius: f64,
}

struct Rectangle {
    width: f64,
    height: f64,
}

struct Triangle {
    a: f64,
    b: f64,
    c: f64,
}

struct Ellipse {
```

```
                semi_major_axis: f64, // a
                semi_minor_axis: f64, // b
                                    {

// ۳. پیادهسازی Trait برای هر Struct

                    } impl Shape for Circle
                } fn area(&self) -> f64
        PI * self.radius * self.radius
                                    {
            } fn perimeter(&self) -> f64
              PI * self.radius * 2.0
                                    {
                                        {

                 } impl Shape for Rectangle
                } fn area(&self) -> f64
            self.width * self.height
                                    {
            } fn perimeter(&self) -> f64
        (self.width + self.height) * 2.0
                                    {
                                        {

                  } impl Shape for Triangle
                } fn area(&self) -> f64
                    ;let s = (self.a + self.b + self.c) / 2.0
Ensure the value inside sqrt is not negative due to floating point //
                                                    inaccuracies
                    .for degenerate triangles, though a 3-4-5 is fine //
   ;let area_squared = s * (s - self.a) * (s - self.b) * (s - self.c)
                                     } if area_squared < 0.0
                Return 0 for invalid/degenerate triangles // 0.0
                                                  } else {
                                ()area_squared.sqrt
                                                 {
                                                    {
                } fn perimeter(&self) -> f64
                self.a + self.b + self.c
                                    {
```

```
{

} impl Shape for Ellipse
} fn area(&self) -> f64
PI * self.semi_major_axis * self.semi_minor_axis
{
} fn perimeter(&self) -> f64
Approximation: P ≈ π [ 3/2(a+b) - sqrt(ab) ] //
;let a = self.semi_major_axis
;let b = self.semi_minor_axis
PI * (1.5 * (a + b) - (a * b).sqrt())
{
{

// ۴. تابع عمومی با استفاده از Trait Bound
} fn print_shape_details<T: Shape>(shape_name: &str, shape: &T)
;println!("--- Details for {} ---", shape_name)
formats the float to 2 decimal places {2.:} //
;println!("Area: {:.2}", shape.area())
;println!("Perimeter: {:.2}", shape.perimeter())
;println!("————————————————————")
{

// ۵. تابع اصلی برنامه
} ()fn main {
;let circle = Circle { radius: 5.0 }
;let rectangle = Rectangle { width: 10.0, height: 4.0 }
let triangle = Triangle { a: 3.0, b: 4.0, c: 5.0 }; // Right-angled triangle
;let ellipse = Ellipse { semi_major_axis: 7.0, semi_minor_axis: 3.0 }

;print_shape_details("Circle", &circle)
;print_shape_details("Rectangle", &rectangle)
;print_shape_details("Triangle (3-4-5)", &triangle)
;print_shape_details("Ellipse", &ellipse)
{
```

خروجی کد کامل:

```
--- Details for Circle ---
Area: 78.54
Perimeter: 31.42
```

```
--------------------------
--- Details for Rectangle ---
Area: 40.00
Perimeter: 28.00
--------------------------
--- Details for Triangle (3-4-5) ---
Area: 6.00
Perimeter: 12.00
--------------------------
--- Details for Ellipse ---
Area: 65.97
Perimeter: 32.28
--------------------------
```

سوالات برای مرور کلاس

۱. Trait Shape در این مثال چه متدهایی را تعریف می‌کند؟ (پاسخ: area و perimeter)

۲. چرا برای محاسبه مساحت Triangle از فرمول هرون استفاده کردیم؟ (پاسخ: چون فقط طول سه ضلع را داشتیم و این فرمول به ارتفاع نیاز ندارد).

۳. محیط Ellipse چگونه محاسبه شد؟ آیا دقیق بود؟ (پاسخ: با یک فرمول تقریبی محاسبه شد، چون فرمول دقیق و ساده‌ای ندارد).

۴. منظور دقیق از عبارت T: Shape در تابع print_shape_details چیست؟ (پاسخ: T هر نوعی است که Trait Shape را پیاده‌سازی کرده باشد).

۵. چگونه Trait Bound به ما اجازه می‌دهد تا متدهای area() و perimeter() را درون تابع print_shape_details فراخوانی کنیم؟ (پاسخ: چون کامپایلر تضمین می‌کند هر نوع T که به تابع داده شود، این متدها را خواهد داشت).

۶. اگر بخواهیم شکل Square (مربع) را اضافه کنیم، به طور خلاصه چه مراحلی را باید طی کنیم؟ (پاسخ: ۱. تعریف struct Square { side: f64 }. ۲. پیاده‌سازی impl Shape for Square با متدهای area (side*side) و perimeter (4*side)).

کار با فایل‌ها در Rust

این فصل به شما نشان می‌دهد چگونه با فایل‌ها در زبان برنامه‌نویسی Rust کار کنید. یاد می‌گیرید چگونه فایل‌ها را بخوانید، در آن‌ها بنویسید و سایر عملیات مرتبط با فایل سیستم را انجام دهید.

۱. مقدمه‌ای بر کار با فایل‌ها

- مفهوم ورودی/خروجی فایل (**File I/O**):

- برنامه‌ها اغلب نیاز به خواندن داده از فایل‌ها یا نوشتن داده در فایل‌ها دارند.
- این عملیات به عنوان ورودی/خروجی فایل (File I/O) شناخته می‌شود.
- فایل‌ها برای ذخیره‌سازی دائمی اطلاعات استفاده می‌شوند، حتی پس از پایان اجرای برنامه.

- ماژول **std::fs**:
  - کتابخانه استاندارد Rust یک ماژول به نام std::fs برای کار با فایل سیستم ارائه می‌دهد.
  - بیشتر توابع مورد نیاز برای عملیات پایه فایل در این ماژول قرار دارند.
  - برای استفاده از این توابع، معمولاً آن‌ها را به اسکوپ خود وارد می‌کنیم، مثلاً با use std::fs;.
- عملیات رایج فایل:
  - خواندن (**Reading**): دریافت محتوای یک فایل.
  - نوشتن (**Writing**): ذخیره داده‌ها در یک فایل. اگر فایل وجود نداشته باشد، معمولاً ایجاد می‌شود. اگر وجود داشته باشد، محتوای آن ممکن است بازنویسی شود.
  - اضافه کردن (**Appending**): افزودن داده‌ها به انتهای یک فایل موجود.
  - ایجاد (**Creating**): ساختن یک فایل یا دایرکتوری جدید.
  - حذف (**Deleting**): پاک کردن یک فایل یا دایرکتوری.
- اهمیت مدیریت خطا:
  - عملیات فایل می‌توانند با خطا مواجه شوند.
  - مثلاً، فایل مورد نظر برای خواندن ممکن است وجود نداشته باشد.
  - یا برنامه ممکن است مجوز لازم برای نوشتن در یک مکان خاص را نداشته باشد.
  - در Rust، توابعی که با فایل‌ها کار می‌کنند معمولاً یک نوع Result<T, E> برمی‌گردانند.
  - این به شما امکان می‌دهد خطاها را به شیوه‌ای ایمن و کنترل‌شده مدیریت کنید. نوع خطای رایج std::io::Error است.
- سوالات بخش ۱:
  ۰. عملیات ورودی/خروجی فایل (File I/O) به چه معناست؟
  ۱. کدام ماژول در کتابخانه استاندارد Rust برای کار با فایل سیستم استفاده می‌شود؟
  ۲. چهار عملیات رایج که می‌توان روی فایل‌ها انجام داد را نام ببرید.

۳. چرا مدیریت خطا هنگام کار با فایل‌ها مهم است و Rust چگونه به این موضوع کمک می‌کند؟

---

۲. خواندن از فایل‌ها

در این بخش، دو روش اصلی برای خواندن محتوای فایل‌ها در Rust را بررسی می‌کنیم.

- الف) خواندن کل محتوای فایل به یک رشته (**String**)
  - ساده‌ترین راه برای خواندن محتوای یک فایل متنی کوچک، استفاده از تابع std::fs::read_to_string() است.
  - این تابع مسیر فایل را به عنوان ورودی می‌گیرد.
  - در صورت موفقیت، یک Result<String, std::io::Error> برمی‌گرداند که حاوی محتویات فایل در یک String است.
  - در صورت بروز خطا (مثلاً فایل پیدا نشود)، یک Err حاوی اطلاعات خطا برمی‌گرداند.
  - مثال:

```Rust
use std::fs;
use std::path::Path;

fn main() {
    let file_path_str = "example_content.txt";
    let path = Path::new(file_path_str);

    // برای اینکه مثال قابل اجرا باشد، ابتدا یک فایل نمونه ایجاد میکنیم.
    // در برنامههای واقعی، این فایل احتمالاً از قبل وجود دارد.
    let initial_content = "Hello from Rust!\nThis is a test file.";
    if !path.exists() {
        match fs::write(path, initial_content) {
            Ok(_) => println!("Sample file '{}' created.", file_path_str),
            Err(e) => {
                eprintln!("Error creating sample file: {}", e);
                return; // Exit if we can't create the sample file
            }
        }
    }
```

```
                ;println!("Attempting to read file: {}", file_path_str)
                                } match fs::read_to_string(path)
                                        } ≤ Ok(content)
                            ;println!("--- File Content ---")
                                    ;println!("{}", content)
                            ;println!("--- End of Content ---")
                                                        {
                                                    } ≤ Err(e)
        ;eprintln!("Error reading file '{}': {}", file_path_str, e)
                                                        {
                                                            {


    Clean up the created file (optional, for keeping the directory clean) //
                                                } match fs::remove_file(path) //
        ,Ok(_) ⇒ println!("Sample file '{}' removed.", file_path_str)     //
            ,Err(e) ⇒ eprintln!("Error removing sample file: {}", e)     //
                                                                    { //
                                                                        {
```

use std::fs;

- std: این بخش به کتابخانه استاندارد (standard library) Rust اشاره دارد.
- fs: این بخش نام یک ماژول (module) در کتابخانه استاندارد است. ماژول fs (کوتاه شده‌ی "file system") شامل توابع و ساختارهایی برای کار با فایل سیستم کامپیوتر شما است. این عملیات می‌تواند شامل خواندن فایل‌ها، نوشتن در فایل‌ها، ایجاد دایرکتوری‌ها، حذف فایل‌ها و غیره باشد.
- کاربرد خط: با نوشتن use std::fs;، شما کل ماژول fs را به محدوده فعلی وارد می‌کنید. پس از این خط، به جای اینکه برای هر بار استفاده از توابع این ماژول مسیر کامل std::fs::نام_تابع را بنویسید (مثلاً std::fs::read_to_string("file.txt"))، می‌توانید به سادگی از fs::نام_تابع استفاده کنید (مثلاً fs::read_to_string("file.txt")). این کار کد شما را کوتاه‌تر و خواناتر می‌کند.
    - توضیح کد:
        - use std::fs; و use std::path::Path; ماژول‌های لازم را وارد می‌کنند.
        - Path::new("example_content.txt") یک مسیر فایل ایجاد می‌کند.

- path: این بخش نام یک ماژول دیگر در کتابخانه استاندارد است که برای کار با مسیرهای فایل سیستم (file system paths) به روشی مستقل از پلتفرم (یعنی چه در ویندوز، چه در لینوکس یا مک، به درستی کار کند) طراحی شده است
- متد exists() بررسی می‌کند که آیا فایل یا دایرکتوری مشخص شده توسط path واقعاً روی دیسک (در فایل سیستم) وجود دارد یا خیر.
- این متد یک مقدار بولین (boolean) برمی‌گرداند:
- true: اگر مسیر وجود داشته باشد.
- false: اگر مسیر وجود نداشته باشد.
- !: این عملگر "نقیض" (NOT) منطقی است.
- این عملگر مقدار بولین مقابل خود را معکوس می‌کند.
- بنابراین، !true برابر false می‌شود و !false برابر true می‌شود.
- ابتدا بررسی می‌کنیم فایل وجود دارد یا نه و اگر نبود، آن را با محتوای اولیه ایجاد می‌کنیم. این کار برای اجرای مستقل مثال مفید است.
- fs::read_to_string(path) تلاش می‌کند فایل را بخواند.
- از match برای بررسی نتیجه (Ok یا Err) استفاده می‌کنیم.
- اگر Ok(content) باشد، محتوای فایل در متغیر content قرار دارد و چاپ می‌شود.
- اگر Err(e) باشد، پیام خطا چاپ می‌شود.

- ب) خواندن فایل به صورت خط به خط
  - برای فایل‌های بسیار بزرگ، خواندن کل محتوا به یکباره در حافظه ممکن است کارآمد نباشد.
  - در این موارد، خواندن فایل به صورت خط به خط گزینه بهتری است.
  - برای این کار، ابتدا فایل را با std::fs::File::open() باز می‌کنیم.
  - سپس از std::io::BufReader برای ایجاد یک خواننده بافردار (buffered reader) استفاده می‌کنیم. بافر کردن عملیات خواندن را بهینه‌تر می‌کند.
  - BufReader یک متد به نام lines() دارد که یک iterator بر روی خطوط فایل برمی‌گرداند.

- مثال:

```rust
use std::fs::{self, File}; // fs::File for opening, fs for write (setup)
use std::io::{self, BufRead, BufReader}; // BufRead trait for .lines()
use std::path::Path;

fn main() -> io::Result<()> { // main can return Result for convenience
    let file_path_str = "lines_example.txt";
    let path = Path::new(file_path_str);

    // ایجاد فایل نمونه برای خواندن خط به خط
    let initial_lines = "First line of text.\nSecond line.\nAnd a third one.";
    if !path.exists() {
        // fs::write can also return a Result, we use expect for simplicity in setup
        fs::write(path, initial_lines).expect("Failed to create sample lines file.");
        println!("Sample file '{}' created for line-by-line reading.", file_path_str);
    }

    println!("\nReading file '{}' line by line:", file_path_str);

    // 1. Open the file
    let file = File::open(path)?; // The '?' operator propagates errors

    // 2. Create a buffered reader
    let reader = BufReader::new(file);

    // 3. Iterate over lines
    for (index, line_result) in reader.lines().enumerate() {
        match line_result {
            Ok(line_content) => {
                println!("Line {}: {}", index + 1, line_content);
            }
            Err(e) => {
                eprintln!("Error reading a line: {}", e);
            }
        }
    }

    // Optional: Clean up the created file
```

```
;fs::remove_file(path).expect("Failed to remove sample lines file.") //
;println!("Sample file '{}' removed.", file_path_str) //

Ok(()) // Indicate success
{
```

o توضیح کد:

- خط use std::fs::{self, File}; دو کار اصلی انجام می‌دهد:
- وارد کردن ماژول fs تحت نام fs: شما می‌توانید از سایر توابع و آیتم‌های درون std::fs با پیشوند fs:: استفاده کنید (مثلاً fs::remove_file، fs::write(...)(...) و غیره).
- **io**: این یک ماژول (module) در کتابخانه استاندارد است که عملیات ورودی/خروجی (Input/Output) را مدیریت می‌کند. این شامل خواندن از فایل‌ها، ورودی استاندارد، و نوشتن به فایل‌ها، خروجی استاندارد و غیره می‌شود.
- BufRead به طور خاص یک trait برای خواندن داده‌ها از یک منبع بافر شده است. این trait متدهای مفید زیادی را فراهم می‌کند که بر روی بافرها کار می‌کنند، از جمله متد بسیار رایج .**lines**() است.
- هدف اصلی BufReader بافر کردن عملیات خواندن است. به جای اینکه هر بار که شما یک بایت یا چند بایت کوچک را می‌خوانید، عملیات I/O واقعی انجام شود، BufReader حجم بیشتری از داده‌ها را به صورت یکجا از منبع اصلی می‌خواند و آن‌ها را در یک بافر داخلی ذخیره می‌کند. سپس، وقتی شما داده‌ها را درخواست می‌کنید، آن‌ها را از این بافر تحویل می‌دهد. این کار می‌تواند به طور قابل توجهی عملکرد خواندن (مخصوصاً از دیسک یا شبکه) را بهبود بخشد زیرا تعداد عملیات I/O کند را کاهش می‌دهد.
- تابع main طوری تعریف شده که io::Result<()> برگرداند. این به ما اجازه می‌دهد از عملگر ? استفاده کنیم.
- io::Result به طور خاص یک Result است که برای عملیات ورودی/خروجی (Input/Output) استفاده می‌شود. io اشاره به ماژول std::io در کتابخانه استاندارد Rust دارد.
- در واقع، io::Result<T> یک نام مستعار (alias) برای Result<T, io::Error> است. یعنی در حالت شکست، نوع خطای برگشتی از جنس io::Error خواهد بود.
- :()
- این علامت در Rust به "واحد" (unit type) یا "تاپل خالی" (empty tuple) معروف است.

- این به معنای "عدم وجود مقدار معنی‌دار" است. وقتی یک تابع Result<()> را برمی‌گرداند، به این معنی است که در صورت موفقیت‌آمیز بودن عملیات، هیچ مقدار خاصی برای برگرداندن وجود ندارد؛ تنها مهم است که عملیات بدون خطا انجام شده است. این معمولاً در توابعی استفاده می‌شود که هدف اصلی آن‌ها انجام یک اثر جانبی (side effect) است، مانند چاپ چیزی به کنسول یا نوشتن در یک فایل، و نه محاسبه و برگرداندن یک مقدار خاص.
- File::open(path)? فایل را باز می‌کند. اگر خطایی رخ دهد، با استفاده از ?، تابع main فوراً همان خطا را برمی‌گرداند.
- BufReader::new(file) یک BufReader جدید ایجاد می‌کند.
- reader.lines() یک iterator برمی‌گرداند. هر آیتم این iterator خود یک Result<String, io::Error> است، زیرا خواندن هر خط ممکن است با خطا مواجه شود.
- از enumerate() برای گرفتن شماره خط به همراه محتوای خط استفاده شده است.
- داخل حلقه for، دوباره از match برای هر خط استفاده می‌کنیم.

- سوالات بخش ۲:

۰. برای خواندن محتوای کامل یک فایل متنی کوچک به یک String، از کدام تابع در ماژول std::fs استفاده می‌شود؟

۱. خروجی تابع fs::read_to_string چیست و چرا به این شکل است؟

۲. چه زمانی خواندن فایل به صورت خط به خط به جای خواندن کل فایل ترجیح داده می‌شود؟

۳. برای خواندن خط به خط، ابتدا با چه تابعی فایل را باز می‌کنیم؟

۴. BufReader چیست و چه مزیتی دارد؟

۵. متد lines() روی چه نوعی فراخوانی می‌شود و هر آیتمی که برمی‌گرداند چه نوعی دارد؟

۶. عملگر ? در انتهای یک فراخوانی تابع که Result برمی‌گرداند، چه کاری انجام می‌دهد؟

---

۳. نوشتن در فایل‌ها (Writing to Files)

Rust روش‌های مختلفی برای نوشتن داده در فایل‌ها ارائه می‌دهد.

- الف) نوشتن یک رشته (**String** یا **str&**) در فایل
  - ساده‌ترین راه برای نوشتن داده‌های رشته‌ای در یک فایل، استفاده از تابع std::fs::write() است.

- این تابع دو آرگومان اصلی می‌گیرد: مسیر فایل، و داده‌هایی که باید نوشته شوند (می‌تواند &str، String یا &[u8] باشد).
- رفتار پیش‌فرض:
  - اگر فایل وجود نداشته باشد، ایجاد می‌شود.
  - اگر فایل وجود داشته باشد، محتوای قبلی آن بازنویسی (overwrite) می‌شود.
- این تابع Result<(), std::io::Error> برمی‌گرداند. () (unit type) نشان می‌دهد که در صورت موفقیت، مقدار معناداری برگردانده نمی‌شود.
- مثال:

```Rust
use std::fs;
use std::path::Path;

fn main() {
    let file_path_str = "output.txt";
    let path = Path::new(file_path_str);
    let content_to_write = "This is a new line written by Rust.\nAnd another line.";

    println!("Attempting to write to file: {}", file_path_str);
    match fs::write(path, content_to_write) {
        Ok(_) => {
            println!("Successfully wrote to '{}'.", file_path_str);
            // Optionally, read and print to verify
            match fs::read_to_string(path) {
                Ok(content) => println!("File content now:\n{}", content),
                Err(e) => eprintln!("Error reading back the file: {}", e),
            }
        }
        Err(e) => {
            eprintln!("Error writing to file '{}': {}", file_path_str, e);
        }
    }

    // Example of overwriting
    let new_content_to_write = "This content will overwrite the previous one.";
    println!("\nAttempting to overwrite file: {}", file_path_str);
```

```
                                                    } match fs::write(path, new_content_to_write)
                                                                                        } ⩵ (_)Ok
                                                ;println!("Successfully overwrote '{}'.", file_path_str)
                                                                    } match fs::read_to_string(path)
                                        ,Ok(content) ⇒ println!("File content now:\n{}", content)
                                        ,Err(e) ⇒ eprintln!("Error reading back the file: {}", e)
                                                                                                {
                                                                                                    {
                                                                                        } ⩵ Err(e)
                                                ;eprintln!("Error overwriting file '{}': {}", file_path_str, e)
                                                                                                {
                                                                                                {
                                                                                Optional: Clean up //
                                                                            ;()fs::remove_file(path).ok //
                                                                                                {
```

- توضیح کد:

- :: (عملگر **Path::new**):

- این یک عملگر برای فراخوانی یک "تابع مرتبط" (associated function) یا دسترسی به یک "آیتم مرتبط" (associated item) است.

- توابع مرتبط توابعی هستند که به جای اینکه روی یک نمونه خاص از struct فراخوانی شوند، مستقیماً روی خود struct (یا enum) فراخوانی می‌شوند. آن‌ها اغلب به عنوان سازنده (constructor) یا توابع کمکی عمل می‌کنند.

- **new**():

- این یک تابع مرتبط (constructor-like function) از Path است.

- وظیفه آن ایجاد یک نمونه جدید از Path از یک رشته (string slice) است.

- این تابع یک آرگومان می‌پذیرد که انتظار می‌رود یک &str (اسلایس رشته‌ای) باشد. در این حالت، آرگومان ما file_path_str است که قبلاً به عنوان "output.txt" تعریف شده است.

- fs::write(path, content_to_write) تلاش می‌کند محتوای content_to_write را در فایل بنویسد.

- اگر فایل output.txt وجود نداشته باشد، ایجاد می‌شود. اگر وجود داشته باشد، محتوای آن با content_to_write جایگزین می‌شود.

- بخش دوم مثال نشان می‌دهد که فراخوانی مجدد fs::write با محتوای جدید، فایل را بازنویسی می‌کند.

- ب) اضافه کردن به انتهای فایل (**Appending**)
  - گاهی اوقات نمی‌خواهیم محتوای فایل را بازنویسی کنیم، بلکه می‌خواهیم داده‌های جدید را به انتهای آن اضافه کنیم.
  - برای این کار، باید فایل را با گزینه‌های خاصی باز کنیم.
  - ماژول std::fs::OpenOptions برای این منظور استفاده می‌شود.
  - OpenOptions به شما اجازه می‌دهد تا نحوه باز شدن فایل را به دقت کنترل کنید (خواندن، نوشتن، اضافه کردن، ایجاد کردن و غیره).
  - مثال:

```Rust
use std::fs::{self, OpenOptions}; // Added OpenOptions
use std::io::Write; // For the .write_all() method
use std::path::Path;

fn main() -> std::io::Result<()> {
    let file_path_str = "append_example.txt";
    let path = Path::new(file_path_str);

    // 1. Create or clear the file initially for a clean example run
    fs::write(path, "Initial content.\n")?;
    println!("Initial content written to '{}'.", file_path_str);

    // 2. Open the file in append mode
    // .We also need .write(true) to be able to write, but .append(true) implies write
    // .create(true) ensures the file is made if it doesn't exist.
    let mut file = OpenOptions::new()
        .append(true)   // Enable append mode.
        .create(true)   // Create if it doesn't exist (though we created it above).
        .open(path)?;   // Open the file.

    // 3. Write new content
    let content_to_append = "This line is appended.\n";
    file.write_all(content_to_append.as_bytes())?; // write_all expects bytes
    println!("Appended first line to '{}'.", file_path_str);
```

```
                    ;"let another_content_to_append = "Another appended line.\n
                        ;?file.write_all(another_content_to_append.as_bytes())
                      ;println!("Appended second line to '{}'.", file_path_str)
    .file.sync_all()?; // getting sure that all contents on the disk is written
                                                Verify by reading (optional) //
            ;println!("\n--- Content of '{}' after appending ---", file_path_str)
                                    ;?let final_content = fs::read_to_string(path)
                                                ;print!("{}", final_content)
                                            ;println!("--- End of Content ---")

                                                        Optional: Clean up //
                                                ;()fs::remove_file(path).ok //

                                                                    (())Ok
                                                                            {
```

o توضیح کد:

- OpenOptions: این یک struct (ساختار داده) در ماژول std::fs است که به شما امکان می‌دهد گزینه‌های مختلفی را برای باز کردن یک فایل پیکربندی کنید.
- ::new(): این یک تابع مرتبط (associated function) است که بر روی خود struct OpenOptions فراخوانی می‌شود (نه روی یک نمونه از آن). این تابع یک نمونه جدید و پیش‌فرض از OpenOptions را برمی‌گرداند که هیچ گزینه‌ای در آن تنظیم نشده است.
- OpenOptions::new():
- OpenOptions: این یک struct (ساختار داده) در ماژول std::fs است که به شما امکان می‌دهد گزینه‌های مختلفی را برای باز کردن یک فایل پیکربندی کنید.
- ::new(): این یک تابع مرتبط (associated function) است که بر روی خود struct OpenOptions فراخوانی می‌شود (نه روی یک نمونه از آن). این تابع یک نمونه جدید و پیش‌فرض از OpenOptions را برمی‌گرداند که هیچ گزینه‌ای در آن تنظیم نشده است.
- use std::fs::OpenOptions; و use std::io::Write; اضافه شده‌اند. Write trait متد write_all را فراهم می‌کند.
- OpenOptions::new() یک سازنده برای گزینه‌های باز کردن فایل ایجاد می‌کند.

- append(true). مشخص می‌کند که می‌خواهیم در حالت اضافه کردن (append mode) فایل را باز کنیم. حالت append به طور ضمنی قابلیت نوشتن را هم فعال می‌کند.
- create(true). تضمین می‌کند که اگر فایل وجود نداشته باشد، ایجاد شود.
- open(path)?. فایل را با گزینه‌های مشخص شده باز می‌کند.
- file.write_all(content_to_append.as_bytes())?. داده‌ها را در فایل می‌نویسد. متد write_all یک برش از بایت‌ها (&[u8]) را به عنوان ورودی می‌گیرد، بنابراین رشته را با .as_bytes() تبدیل می‌کنیم.

۵. سایر عملیات فایل سیستم

ماژول std::fs علاوه بر خواندن و نوشتن، توابع دیگری برای کار با فایل سیستم ارائه می‌دهد.

- الف) ایجاد دایرکتوری:
  - std::fs::create_dir(path): یک دایرکتوری جدید در مسیر مشخص شده ایجاد می‌کند.
    - اگر دایرکتوری از قبل وجود داشته باشد، خطا برمی‌گرداند.
    - اگر هر یک از اجزای والد مسیر وجود نداشته باشند (مثلاً می‌خواهید a/b/c را بسازید و a/b وجود ندارد)، خطا برمی‌گرداند.
    - برمی‌گرداند. Result<(), io::Error>
  - std::fs::create_dir_all(path): یک دایرکتوری جدید و تمام دایرکتوری‌های والد لازم را ایجاد می‌کند (مانند دستور mkdir -p در لینوکس/مک).
    - اگر دایرکتوری از قبل وجود داشته باشد، خطایی نمی‌دهد و موفقیت برمی‌گرداند.
    - برمی‌گرداند. Result<(), io::Error>
  - مثال:

```Rust
use std::fs;
use std::path::Path;
use std::io;

fn main() -> io::Result<()> {
```

```
                                ;let path_single = Path::new("./my_new_directory")
                               ;let path_nested = Path::new("./parent_dir/child_dir")

                                                        Create a single directory .1 //
                                                             } ()if !path_single.exists
                                                  } match fs::create_dir(path_single)
                        Ok(_) ⇒ println!("Directory '{}' created successfully.",
                                                                    ,path_single.display())
,Err(e) ⇒ eprintln!("Failed to create '{}': {}", path_single.display(), e)
                                                                                    {
                                                                                } else {
                    ;println!("Directory '{}' already exists.", path_single.display())
                                                                                    {

                                                        Create nested directories .2 //
                 First, ensure parent_dir doesn't exist to properly test create_dir_all //
                                                  } ()if Path::new("./parent_dir").exists
                     fs::remove_dir_all("./parent_dir")?; // Clean up for the example
                                       ;println!("Cleaned up existing './parent_dir'")
                                                                                    {

                                                  } match fs::create_dir_all(path_nested)
               Ok(_) ⇒ println!("Directory '{}' (and parents) created successfully.",
                                                                    ,path_nested.display())
   ,Err(e) ⇒ eprintln!("Failed to create '{}': {}", path_nested.display(), e)
                                                                                    {

                    Attempt to create again - create_dir_all should succeed if it exists //
                                                  } match fs::create_dir_all(path_nested)
                 Ok(_) ⇒ println!("Calling create_dir_all again on '{}' succeeded (as
                                                         ,expected).", path_nested.display())
  Err(e) ⇒ eprintln!("create_dir_all failed unexpectedly on existing dir: {}",
                                                                                      ,e)
                                                                                    {

                                                                  Clean up (optional) //
                              if path_single.exists() { fs::remove_dir(path_single).ok(); }
 if Path::new("./parent_dir").exists() { fs::remove_dir_all("./parent_dir").ok(); }
                                                                                 (())Ok
                                                                                        {
```

- ج) کپی کردن فایل‌ها:
  - std::fs::copy(from_path, to_path): محتوای فایل مبدأ را در فایل مقصد کپی می‌کند.
    - اگر from_path یک دایرکتوری باشد، خطا برمی‌گرداند.
    - فایل مقصد بازنویسی می‌شود اگر از قبل وجود داشته باشد.
    - مجوزهای فایل (permissions) به طور پیش‌فرض کپی نمی‌شوند (رفتار دقیق ممکن است وابسته به پلتفرم باشد، اما معمولاً مجوزهای پیش‌فرض برای فایل جدید اعمال می‌شود).
    - Result<u64, io::Error> برمی‌گرداند که u64 تعداد بایت‌های کپی شده است.
  - مثال:

```Rust
use std::fs;
use std::path::Path;
use std::io;

fn main() -> io::Result<()> {
    let source_str = "source_file_for_copy.txt";
    let destination_str = "destination_file_copied.txt";
    let path_source = Path::new(source_str);
    let path_destination = Path::new(destination_str);

    // Setup: Create source file
    fs::write(path_source, "Content to be copied.")?;
    println!("Created source file: {}", path_source.display());

    // Copy file
    match fs::copy(path_source, path_destination) {
        Ok(bytes_copied) => {
            println!("Successfully copied {} bytes from '{}' to '{}'.", bytes_copied, path_source.display(), path_destination.display());

            // Verify content (optional)
            let content = fs::read_to_string(path_destination)?;
            println!("Content of destination: {}", content);
        }
```

```
                } ≤ Err(e)
        ;eprintln!("Error copying file: {}", e)
                                {
                                    {

                                    Clean up (optional) //
                                ;()fs::remove_file(path_source).ok
                            ;()fs::remove_file(path_destination).ok
                                    (())Ok
                                {
```

مثالی از یک پایگاه داده ساده با هش مپ

در کد زیر، HashMap شبیه به یک فرهنگ لغت (dictionary) است که در آن هر کلید یکتاست و به یک مقدار خاص نگاشت (map) می‌شود. این ساختار داده برای جستجو، اضافه کردن و حذف کردن عناصر بر اساس کلید، بسیار بهینه و سریع است.

در کد شما، HashMap<String, String> به این معنی است که:

کلیدها از نوع String (رشته متنی) هستند.

مقدارها نیز از نوع String (رشته متنی) هستند.

ساختار KVStore از این HashMap در فیلد store استفاده می‌کند تا یک پایگاه داده کلید–مقدار ساده را پیاده‌سازی کند. متدهای set, get, و remove مستقیماً از عملیات‌های متناظر HashMap برای دستکاری داده‌ها استفاده می‌کنند.

به طور خلاصه، HashMap در اینجا نقش ظرفی را دارد که داده‌های شما را به صورت جفت‌های کلید و مقدار نگه می‌دارد و امکان دسترسی سریع به مقدارها را با استفاده از کلید مربوطه فراهم می‌کند.

خط use serde::{Serialize, Deserialize}; در کد Rust مربوط به کتابخانه serde است.

توضیح کوتاه:

این خط دو ترِیت (Trait) به نام‌های Serialize و Deserialize را از کتابخانه serde وارد (import) می‌کند.

- **Serialize**: این تریت به کامپایلر می‌گوید که یک ساختار داده (مثل struct شما KVStore) قابل تبدیل شدن به فرمت‌های دیگر داده (مانند YAML، JSON و...) است. به عبارت دیگر، می‌توانید داده‌های این ساختار را "سریالایز" کنید تا در فایل ذخیره شوند یا از طریق شبکه ارسال شوند.
- **Deserialize**: این تریت به کامپایلر می‌گوید که یک ساختار داده قابل ساخته شدن از فرمت‌های دیگر داده است. یعنی می‌توانید داده‌هایی که قبلاً به فرمتی مثل JSON تبدیل شده‌اند را بخوانید و آن‌ها را به یک نمونه از این ساختار داده در برنامه Rust تبدیل کنید.

در کد شما، استفاده از # [derive(Serialize, Deserialize)] بالای struct KVStore به صورت خودکار این دو تریت را برای KVStore پیاده‌سازی می‌کند و به شما امکان می‌دهد تا از توابع serde_json (مثل to_string و from_str) برای تبدیل

KVStore به/از فرمت JSON استفاده کنید.

در Rust، #[derive] یک قابلیت بسیار مفید است که به شما اجازه می‌دهد پیاده‌سازی (implementation) برخی از تریِت (**Trait**) های استاندارد را برای struct یا enum خود به صورت خودکار تولید کنید.

توضیح خلاصه:

#[derive] در واقع یک نوع ماکروی رویه‌ای (**Procedural Macro**) است. وقتی شما آن را بالای یک struct یا enum قرار می‌دهید، به کامپایلر می گویید که کدهای لازم برای پیاده‌سازی تریت‌های مشخص شده را قبل از کامپایل نهایی به صورت خودکار تولید و اضافه کند.

چگونه کار می کند:

به جای اینکه شما مجبور باشید کدهای تکراری و boilerplate برای پیاده‌سازی تریت‌هایی مانند Debug (برای پرینت قابل خواندن)، Clone (برای کپی عمیق)، PartialEq (برای مقایسه برابری ==)، Serialize یا Deserialize (برای تبدیل به/از فرمت‌های داده) را خودتان بنویسید، #[derive(TraitName1, TraitName2, ...)] این کار را بر اساس ساختار struct یا enum شما انجام می‌دهد.

مثال در کد:

#[derive(Serialize, Deserialize, Debug)] بالای struct KVStore باعث شد که کامپایلر به صورت خودکار کدهای لازم برای:

- **Debug**: پرینت کردن آسان و قابل خواندن محتویات KVStore (مثلاً با {:?}).
- **Serialize** و **Deserialize**: تبدیل KVStore به/از فرمت JSON (با کمک کتابخانه serde_json).

را تولید و به برنامه اضافه کند، بدون اینکه شما نیاز به نوشتن دستی این کدها داشته باشید.

پس به زبان ساده، #[derive] یک میانبر هوشمندانه است که کامپایلر برای شما کدهای تکراری مربوط به تریت‌های رایج را می‌نویسد.

```
use std::collections::HashMap;
use std::fs::{self, File};
use std::io::{self, Write};
use serde::{Serialize, Deserialize};

#[derive(Serialize, Deserialize, Debug)]
struct KVStore {
    store: HashMap<String, String>,
}

impl KVStore {
    fn new() -> Self {
```

```
                                                    } KVStore
                                            ,()store: HashMap::new
                                                                {
                                                                    {

                                    } fn set(&mut self, key: String, value: String)
                                                ;self.store.insert(key, value)
                                                                {

                                      } <fn get(&self, key: &str) -> Option<&String
                                                        self.store.get(key)
                                                                {

                                                } fn remove(&mut self, key: &str)
                                                      ;self.store.remove(key)
                                                                {

                                   } <()>fn save_to_file(&self, filename: &str) -> io::Result
                                  ;()let serialized = serde_json::to_string(&self).unwrap
                                                ;?let mut file = File::create(filename)
                                                ;?file.write_all(serialized.as_bytes())
                                                                    (())Ok
                                                                {

                                       } <fn load_from_file(filename: &str) -> io::Result<Self
                                                ;?let data = fs::read_to_string(filename)
                                ;()let store: KVStore = serde_json::from_str(&data).unwrap
                                                                    Ok(store)
                                                                {
                                                                    {

                                                    } <()>fn main() -> io::Result
                                                ;()let mut kv_store = KVStore::new
                                        ;kv_store.set("name".to_string(), "Alice".to_string())
                                            ;kv_store.set("age".to_string(), "25".to_string())
                                                        ;?kv_store.save_to_file("store.json")
                                    ;?let loaded_store = KVStore::load_from_file("store.json")
                                                ;println!("{:?}", loaded_store)
                                                                    (())Ok
```

{

مثالی از یک پایگاه داده cli

سه جزء مهم از کتابخانه‌ی **clap** وارد برنامه می‌شوند. توضیح هرکدام:

---

۱. App

- **App** ساختاری است که برای تعریف برنامه‌ی خط فرمانی (Command Line Application) استفاده می‌شود.
- با استفاده از App می‌توانید نام برنامه، نسخه، توضیحات، نویسنده و همچنین زیردستورها و آرگومان‌ها را تعریف کنید.
- مثال:

```
let app = App::new("myapp")
        version("1.0").
    author("Author Name").
;about("This is a CLI app").
```

۲. Arg

- **Arg** برای تعریف آرگومان‌ها و گزینه‌های ورودی برنامه استفاده می‌شود.
- با این ساختار می‌توانید تعیین کنید برنامه شما چه ورودی‌هایی می‌پذیرد، اجباری یا اختیاری بودن آرگومان‌ها را مشخص کنید و ویژگی‌های دیگری مانند نوع داده را تنظیم نمایید.
- مثال:

```
Arg::with_name("filename")
        required(true).
help("Name of the file to process").
```

۳. SubCommand

- **SubCommand** برای تعریف زیردستورها (subcommands) به کار می‌رود.
- زیردستورها مانند بخش‌های مختلف یک برنامه CLI هستند. مثلا در برنامه مدیریت کلید–مقدار شما، زیردستورهای "set"، "get" و "remove" هرکدام یک SubCommand هستند.
- مثال:

```
                SubCommand::with_name("set")
                    about("Set a key-value pair").
                arg(Arg::with_name("key").required(true)).
                arg(Arg::with_name("value").required(true)).
```

در واقع این سه ساختار قلب برنامه‌های CLI با clap هستند و اکثر امکانات این کتابخانه با استفاده از همین‌ها پیاده‌سازی می‌شود.

```
                ;use std::collections::BTreeMap
                    ;use std::fs::{self, File}
                ;use std::io::{self, Write}
            ;use serde::{Serialize, Deserialize}
              ;use clap::{App, Arg, SubCommand}

        [derive(Serialize, Deserialize, Debug)]#
                        } struct KVStore
            ,<store: BTreeMap<String, String
                                        {

                                } impl KVStore
                            } fn new() -> Self
                                } KVStore
                    ,()store: BTreeMap::new
                                        {
                                            {

        } fn set(&mut self, key: String, value: String)
                        ;self.store.insert(key, value)
                                            {

          } <fn get(&self, key: &str) -> Option<&String
                                self.store.get(key)
                                            {

                        } fn remove(&mut self, key: &str)
                              ;self.store.remove(key)
                                            {

    } <()>fn save_to_file(&self, filename: &str) -> io::Result
```

```
;()let serialized = serde_json::to_string(&self).unwrap
                ;?let mut file = File::create(filename)
                ;?file.write_all(serialized.as_bytes())
                                                ((()Ok
                                                      {

    } <fn load_from_file(filename: &str) -> io::Result<Self
                ;?let data = fs::read_to_string(filename)
;()let store: KVStore = serde_json::from_str(&data).unwrap
                                                Ok(store)
                                                      {

                                              } fn list(&self)
                                    } ()if self.store.is_empty
                ;println!("No key-value pairs found.")
                                                } else {
                     } for (key, value) in &self.store
                   ;println!("{}: {}", key, value)
                                                   {
                                                       {
                                                          {
                                                              {

                                        } <()>fn main() -> io::Result
            let matches = App::new("Rust CLI Key-Value Store")
                                             version("1.0").
                                           author("Your Name").
                    about("A simple key-value store in Rust").
                                             )subcommand.
                          SubCommand::with_name("set")
                       about("Set a key-value pair").
      arg(Arg::with_name("key").required(true)).
    arg(Arg::with_name("value").required(true)).
                                                          (
                                             )subcommand.
                          SubCommand::with_name("get")
                      about("Get a value by key").
      arg(Arg::with_name("key").required(true)).
                                                          (
```

```
                                                )subcommand.
                                    SubCommand::with_name("remove")
                                about("Remove a key-value pair").
                            arg(Arg::with_name("key").required(true)).
                                                                (
                                                )subcommand.
                                        SubCommand::with_name("list")
                                about("List all key-value pairs").
                                                                (
                                            ;()get_matches.

                                        ;()let mut kv_store = KVStore::new
                                              ;"let filename = "store.json

                            } if let Ok(store) = KVStore::load_from_file(filename)
                                                    ;kv_store = store
                                                                {

                             } if let Some(matches) = matches.subcommand_matches("set")
                                ;()let key = matches.value_of("key").unwrap().to_string
                              ;()let value = matches.value_of("value").unwrap().to_string
                                                    ;kv_store.set(key, value)
                                                ;?kv_store.save_to_file(filename)
                                                 ;println!("Key-value pair set!")
                           } else if let Some(matches) = matches.subcommand_matches("get") {
                                            ;()let key = matches.value_of("key").unwrap
                                               } if let Some(value) = kv_store.get(key)
                                                    ;println!("Value: {}", value)
                                                                        } else {
                                                  ;println!("Key not found")
                                                                {
        } else if let Some(matches) = matches.subcommand_matches("remove") {
                                            ;()let key = matches.value_of("key").unwrap
                                                        ;kv_store.remove(key)
                                                ;?kv_store.save_to_file(filename)
                                                        ;println!("Key removed!")
                                } else if let Some(_) = matches.subcommand_matches("list") {
                                                    ;()kv_store.list
                                                                {
```

```
(())Ok
{
```

طول عمر (Lifetimes)

مقدمه و چرایی نیاز به طول عمر (Introduction and Why Lifetimes Are Needed)

- در Rust، کامپایلر باید مطمئن شود که ارجاع‌ها (references) همیشه به داده معتبر اشاره می‌کنند.
- این یعنی ارجاع نباید به حافظه‌ای اشاره کند که دیگر استفاده نمی‌شود (خطر dangling pointer).
- Rust بدون استفاده از Garbage Collector این امنیت را فراهم می‌کند.
- این کار با استفاده از سیستمی به نام Borrow Checker و مفهومی به نام طول عمر انجام می‌شود.
- طول عمر (lifetime) محدوده‌ای است که در آن یک ارجاع معتبر است.
- کامپایلر Rust از طول عمرها برای بررسی صحت استفاده از ارجاع‌ها استفاده می‌کند.

نحوه‌ی نوشتن و حدس کامپایلر (Syntax and Compiler Inference)

- طول عمرها با یک علامت ' و سپس یک نام (معمولاً یک حرف کوچک) مشخص می‌شوند.
- مثال: 'a، 'b
- در بیشتر مواقع، کامپایلر Rust می‌تواند طول عمرها را به طور خودکار حدس بزند.
- این قابلیت "lifetime elision" نام دارد.
- در این مواقع نیازی نیست طول عمر را صریحاً بنویسید.
- اما گاهی، برای کمک به کامپایلر در فهم رابطه بین ارجاع‌ها، باید طول عمر را مشخص کنید.
- این کار معمولاً در امضای توابع یا تعریف ساختارها انجام می‌شود.

طول عمر در امضای توابع (Lifetimes in Function Signatures)

- وقتی تابع شما ارجاعی به عنوان ورودی می‌گیرد و ارجاعی برمی‌گرداند:
- کامپایلر باید بداند که طول عمر ارجاع خروجی به کدام ارجاع ورودی وابسته است.

- اگر کامپایلر نتواند این را به طور قطعی حدس بزند، از شما می‌خواهد که طول عمرها را مشخص کنید.

مثال ۱: طول عمر حدس زده شده توسط کامپایلر (Compiler Inferred Lifetime)

- تابع first_word یک ارجاع به رشته (str&) می‌گیرد.
- یک ارجاع به قسمتی از آن رشته را برمی‌گرداند.
- کامپایلر می‌تواند حدس بزند که طول عمر ارجاع خروجی همان طول عمر ارجاع ورودی است.

```
} fn first_word(s: &str) → &str
    ;()let bytes = s.as_bytes

} ()for (i, &item) in bytes.iter().enumerate
        } ' 'if item == b
    ;return &s[0..i]
                {
                    {

                [..]s&
                    {

} ()fn main
    ;let my_string = String::from("hello world")

                .'word' is a reference to a part of 'my_string' //
.The compiler infers that 'word' is valid as long as 'my_string' is valid //
                ;let word = first_word(&my_string)

            .This is fine because 'my_string' is still valid here //
                ;println!("The first word is: {}", word)

            :Example of a scenario prevented by lifetimes //
                                Start a new scope // } //
                ;let my_string2 = String::from("short")     //
        .'word2' would reference data owned by 'my_string2' //     //
                ;let word2 = first_word(&my_string2)     //
        .my_string2' goes out of scope here and is dropped' //     //
                            'End of scope for 'my_string2 // { //
.If we could access 'word2' here, it would be a dangling reference // //
```

```
                                    .The compiler prevents this using lifetime analysis // //
    println!("The first word is: {}", word2); // Uncommenting this causes a // //
                                                                    compile-time error
                                                                                    {
```

- در این مثال، قواعد "lifetime elision" به کامپایلر اجازه می‌دهد طول عمر str& ورودی و خروجی را مرتبط کند.

مثال ۲: نیاز به مشخص کردن صریح طول عمر (Explicit Lifetime Annotation)

- تابع longest دو ارجاع به رشته (str&) می‌گیرد.
- ارجاع به طولانی‌ترین رشته را برمی‌گرداند.
- ارجاع خروجی می‌تواند به یکی از دو ورودی اشاره کند.
- کامپایلر به تنهایی نمی‌تواند بداند که ارجاع خروجی به کدام ورودی وابسته است.
- باید از Annotation طول عمر استفاده کنیم تا به کامپایلر کمک کنیم.

```
                                                    .Declare a lifetime parameter 'a //
                                                    Input x has lifetime 'a: &'a str //
                                                    Input y has lifetime 'a: &'a str //
                                                 Output has lifetime 'a: → &'a str //
                                 } fn longest<'a>(x: &'a str, y: &'a str) → &'a str
                                                        } ()if x.len() > y.len
                                                                              x
                                                                    } else {
                                                                              y
                                                                                  {
                                                                                      {


                                                                     } ()fn main
                                                  ;let string1 = String::from("long string is long")
                                               let result; // Declare 'result' outside the inner scope


                                                                    Start an inner scope // }
                                                ;let string2 = String::from("xyz")


The 'longest' function returns a reference that is valid as long as *both* //
                                                                        .inputs are valid
                                    The signature fn longest<'a>(x: &'a str, y: &'a str) → &'a str //
                                 .means the reference returned by longest is valid for the lifetime 'a //
```

```
            .Here, 'a is the shorter of the lifetimes of string1 and string2 //
                        .string2 goes out of scope at the end of this block //
                        .Thus, 'a is only valid until the end of this block //

This is okay because 'result' is used within the scope where both string1 and //
                                                                    .string2 are valid
                              ;result = longest(string1.as_str(), string2.as_str())
                            ;println!("The longest string in this scope is {}", result)

                            .string2 goes out of scope here. The lifetime 'a ends here // {

                                        .string1 is still valid here, but string2 is not //
 result' holds a reference whose validity was tied to *both* string1 and string2' //
                                                                    .by the lifetime annotation 'a
.Since 'a' ended when string2 went out of scope, 'result' is no longer valid here //
     .The compiler uses the 'a lifetime annotation to prevent using 'result' here //

      Uncommenting the line below causes a compile-time error because 'result' is //
                                                                            .invalid
                                  ;println!("The longest string is {}", result) //
                                                                                {
```

- در امضای تابع longest<'a>(x: &'a str, y: &'a str) -> &'a str، ما به کامپایلر اعلام می‌کنیم که:
  - طول عمر ارجاع خروجی 'a است.
  - این طول عمر 'a حداقل باید به اندازه طول عمر کوتاه‌ترین ارجاع ورودی (بین x و y) باشد.
- این تضمین می‌کند که ارجاع بازگشتی همیشه معتبر است و به داده‌ای اشاره نمی‌کند که زودتر از موعد از بین رفته است.

طول عمر 'static (The 'static Lifetime)

- طول عمر 'static یک طول عمر خاص و از پیش تعریف شده است.
- این طول عمر نشان دهنده داده‌ای است که در تمام طول اجرای برنامه وجود دارد و معتبر است.
- رشته‌های متنی که مستقیماً در کد می‌نویسید (String literals) دارای طول عمر 'static هستند.

```
                                                ;".let s: &'static str = "I have a static lifetime
.s' is a reference to data that is valid for the entire duration of the program' //
```

پرسش‌ها (Questions)

۱. منظور از "طول عمر یک ارجاع" در Rust چیست؟

۲. چرا مفهوم طول عمر برای امنیت حافظه در Rust ضروری است؟

۳. در چه مواقعی کامپایلر Rust به طور خودکار طول عمر را حدس می‌زند؟

۴. چه زمانی نیاز داریم طول عمر را در امضای توابع صریحاً مشخص کنیم؟

۵. طول عمر 'static چه کاربردی دارد؟

برنامه‌نویسی ناهمزمان (Asynchronous Programming) در Rust

برنامه‌نویسی ناهمزمان به شما اجازه می‌دهد تا برنامه‌ها کارهای مختلفی را همزمان انجام دهند. بدون اینکه منتظر بمانند یک کار خاص تمام شود. این موضوع به خصوص در کارهای ورودی/خروجی (**I/O**) مانند خواندن فایل‌ها یا درخواست‌های شبکه مفید است.

چرا برنامه‌نویسی ناهمزمان؟

در برنامه‌نویسی سنکرون (Synchronous)، وقتی یک تابع فراخوانی می‌شود، برنامه منتظر می‌ماند تا آن تابع کارش را تمام کند. سپس به کار بعدی می‌رود. این موضوع می‌تواند باعث مسدود شدن برنامه شود. به ویژه اگر عملیات طولانی باشد.

برنامه‌نویسی ناهمزمان این مشکل را با اجازه دادن به برنامه برای انجام کارهای دیگر در حین انتظار، حل می‌کند. این کار با استفاده از **Future**ها و **Executor**ها انجام می‌شود.

مفاهیم اصلی

۱. Future (آینده)

Future نوعی است که یک مقدار را در آینده تولید می‌کند. وقتی یک تابع ناهمزمان فراخوانی می‌شود، به جای برگرداندن مستقیم مقدار، یک **Future** برمی‌گرداند. این **Future** نشان‌دهنده عملیاتی است که هنوز کامل نشده است.

۲. async/await

Rust از کلمات کلیدی **async** و **await** برای برنامه‌نویسی ناهمزمان استفاده می‌کند.

- **async fn**: توابعی که با async fn تعریف می‌شوند، توابع ناهمزمان هستند. این توابع یک Future برمی‌گردانند.
- **await**.: زمانی که روی یک Future از .await استفاده می‌کنید، اجرای تابع فعلی متوقف می‌شود. این توقف تا زمانی است که Future کامل شود و مقدارش را برگرداند. در این مدت، Executor می‌تواند کارهای دیگری را انجام دهد.

مثال:

Rust

```
} ()async fn say_hello
;println!("Hello from async function!")
{
```

```
#[tokio::main] // برای اجرای توابع async نیاز به یک executor داریم
async fn main() {
    say_hello().await;
}
```

در این مثال:

- say_hello یک تابع async است.
- main هم یک تابع async است.
- say_hello().await; باعث می‌شود main منتظر بماند تا say_hello کامل شود. اما در این مدت، Executor می‌تواند کارهای دیگر را پردازش کند.

۳. Executor (اجراکننده)

یک **Executor** وظیفه اجرای **Future**ها را بر عهده دارد. Rust به صورت داخلی یک Executor ندارد. شما باید از کریت‌های شخص ثالث مانند **Tokio** یا **async-std** استفاده کنید. در این مثال، از **Tokio** استفاده می‌کنیم.

برای استفاده از Tokio، باید آن را به Cargo.toml اضافه کنید:

```
[dependencies]
tokio = { version = "1", features = ["full"] }
```

مثال با **Tokio**:

```
use tokio::time::{sleep, Duration};

async fn task_one() {
    println!("Task One: Start");
    sleep(Duration::from_secs(2)).await; // Simulate long operation
    println!("Task One: End");
}

async fn task_two() {
    println!("Task Two: Start");
    sleep(Duration::from_secs(1)).await; // Simulate long operation
    println!("Task Two: End");
}

#[tokio::main]
async fn main() {
    println!("Main: Start");
    // Both tasks will execute concurrently
```

```
    ;tokio::join!(task_one(), task_two())
                ;println!("Main: End")
}
```

در این مثال:

- task_one و task_two دو تابع async هستند.
- tokio::join! هر دو Future را به صورت همزمان اجرا می‌کند.
- شما خواهید دید که "Task One: شروع" و "Task Two: شروع" تقریباً همزمان چاپ می‌شوند. سپس "Task Two: پایان" قبل از "Task One: پایان" ظاهر می‌شود. این نشان‌دهنده اجرای ناهمزمان است.
- #[tokio::main] یک ماکرو (**Macro**) است که توسط کریت tokio فراهم شده است. نقش اصلی آن این است که تابع main شما را که یک تابع ناهمزمان (async function) است، قابل اجرا کند.
- به زبان ساده‌تر، Rust به صورت پیش‌فرض نمی‌داند چگونه یک تابع async fn main() را اجرا کند. Futureها (که توابع async تولید می‌کنند) برای اجرا شدن نیاز به یک **Executor** دارند. #[tokio::main] دقیقاً همین کار را می‌کند:
- بدون #[tokio::main] (یا ماکرو مشابهی از یک کریت async دیگر مانند async-std::main)، شما نمی‌توانید یک async fn main() داشته باشید، زیرا Rust یک Executor داخلی برای اجرای Futureها ندارد.

استفاده از spawn برای اجرای مستقل

اگر می‌خواهید یک Future را به صورت کاملاً مستقل از بقیه کد اجرا کنید، می‌توانید از tokio::spawn استفاده کنید.

Rust

```
;use tokio::time::{sleep, Duration}

} async fn long_running_task(id: u8)
    ;println!("Task {} started.", id)
  ;sleep(Duration::from_secs(3)).await
    ;println!("Task {} finished.", id)
                                    {

                                [tokio::main]#
                              } ()async fn main
  ;println!("Main program started.")
```

```
    let handle1 = tokio::spawn(long_running_task(1));
    let handle2 = tokio::spawn(long_running_task(2));

    // We can do other things here while tasks run in the background
    println!("Doing other work in the main program...");
    sleep(Duration::from_secs(1)).await;
    println!("Continuing other work in the main program...");

    // Wait for tasks to complete
    handle1.await.unwrap();
    handle2.await.unwrap();

    println!("Main program finished.");
}
```

در این مثال:

- token::spawn یک Future را به Executor می‌دهد تا اجرا شود.
- spawn یک JoinHandle برمی‌گرداند که می‌توانید از آن برای انتظار برای اتمام Future استفاده کنید.

سوالات کوتاه پاسخ:

۱. تفاوت اصلی برنامه‌نویسی سنکرون و ناهمزمان در چیست؟

۲. Future در Rust چه مفهومی دارد؟

۳. کاربرد کلمات کلیدی async و await چیست؟

۴. چرا Rust به یک Executor خارجی (مانند Tokio) برای برنامه‌نویسی ناهمزمان نیاز دارد؟

۵. وظیفه tokio::spawn چیست؟

برای اینکه یک Future در Rust اجرا شود، حتماً باید توسط یک Executor "پول" شود.

await و tokio::spawn (یا tokio::join!) راه‌هایی برای دادن Future شما به Executor هستند.

اگر Future را فقط تعریف کنید و هیچ‌کدام از این متدها را روی آن فراخوانی نکنید، هرگز اجرا نخواهد شد.

اگر با spawn اجرا کنید اما روی JoinHandle آن await نکنید، Task در پس‌زمینه اجرا می‌شود، اما اگر Executor قبل از اتمام Task خاموش شود، Task نیز خاتمه می‌یابد.

بازی مارها به عنوان درس

```
// ──────────────────────────────────────────────────────────
// Snake – نسخهٔ تک‌فایلی با کادر دور جدول
// وابستگی‌ها در Cargo.toml:
// tokio = { version = "1", features = ["macros", "rt-multi-thread", "time"] }
// crossterm = "0.27"
// rand = "0.8"
// ──────────────────────────────────────────────────────────
use crossterm::{
    cursor,
    event::{self, Event, KeyCode},
    execute, queue,
    style::Print as PrintChar,
    terminal,
};
use rand::Rng;
use std::{
    error::Error,
    io::{Write, stdout},
    time::Duration,
};
use tokio::time::sleep;

// اندازهٔ ناحیهٔ درونی (بدون کادر)
const W: u16 = 40;
const H: u16 = 20;

// ─── انواع داده
#[derive(Copy, Clone, PartialEq)]
enum Dir {
    Up,
    Down,
    Left,
    Right,
}

#[derive(Copy, Clone, PartialEq)]
struct Pt {
    x: u16,
```

```
                                                    ,y: u16
                                                          {

                                                      } impl Pt
                                                } fn step(self, d: Dir) → Pt
                                                            } match d
                                                      } Dir::Up ⇒ Pt
                                                       ,x: self.x
                                                ,y: self.y.saturating_sub(1)
                                                                   ,{
                                                    } Dir::Down ⇒ Pt
                                                       ,x: self.x
                                                    ,y: self.y + 1
                                                                   ,{
                                                    } Dir::Left ⇒ Pt
                                                ,x: self.x.saturating_sub(1)
                                                       ,y: self.y
                                                                   ,{
                                                   } Dir::Right ⇒ Pt
                                                    ,x: self.x + 1
                                                       ,y: self.y
                                                                   ,{
                                                                        {
                                                                            {
                                                                                {

                                                                   } struct Snake
                                                               ,<body: Vec<Pt
                                                                    ,dir: Dir
                                                                  ,grow: bool
                                                                          {

                                                                     } impl Snake
                                                                } fn new() → Self
                                                                       } Self
                                                ,body: vec![Pt { x: W / 2, y: H / 2 }]
                                                            ,dir: Dir::Right
                                                                ,grow: false
                                                                        {
```

```
                                                                {
                                                    } fn head(&self) → Pt
                                                *()self.body.first().unwrap
                                                                {
                                                 } fn change(&mut self, d: Dir)
                                                  // جلوگیری از برگشت ۱۸۰ درجه
                                                // جلوگیری از تغییر جهت ۱۸۰ درجه
                                        if (self.dir == Dir::Up && d == Dir::Down)
                                   (self.dir == Dir::Down && d == Dir::Up) ||
                                (self.dir == Dir::Left && d == Dir::Right) ||
                                (self.dir == Dir::Right && d == Dir::Left) ||
                                                                        }
                                                    return; // جهت جدید معتبر نیست
                                                                        {
                                                                self.dir = d;
                                                                        {
                                                             } fn tick(&mut self)
                                                let next = self.head().step(self.dir);
                                                            self.body.insert(0, next);
                                                                  } if !self.grow
                                                                 ()self.body.pop;
                                                                       } else {
                                                                self.grow = false
                                                                        {
                                                                            {
                                                             } fn crash(&self) → bool
                                                                 ()let h = self.head;
                                        h.x ≥ W || h.y ≥ H || self.body[1..].contains(&h)
                                                                        {
                                                                    {

                                                // ——— توابع کمکی
                                                  } fn rand_pt(except: &[Pt]) → Pt
                                                ()let mut rng = rand::thread_rng;
                                                                  } loop
                                                               } let p = Pt
                                                          ,x: rng.gen_range(0..W)
                                                          ,y: rng.gen_range(0..H)
                                                                        ;{
```

```
                } if !except.contains(&p)
                    ;return p
                {
                {
                {

} fn draw(s: &Snake, food: Pt, out: &mut std::io::Stdout)
    ;()queue!(out, cursor::MoveTo(0, 0)).unwrap

    //  ردیف بالای کادر
    } for _ in 0..=W + 1
        ;()queue!(out, PrintChar('#')).unwrap
    {
        ;()queue!(out, PrintChar('\n')).unwrap

    //  سطرهای درونی
    } for y in 0..H
        ;()queue!(out, PrintChar('#')).unwrap // لبهٔ چپ
    } for x in 0..W
} let ch = if s.head().x == x && s.head().y == y
        'O'
} else if s.body.iter().any(|p| p.x == x && p.y == y) {
        'o'
    } else if food.x == x && food.y == y {
        '*'
    } else {
        ' '
    ;{
    ;()queue!(out, PrintChar(ch)).unwrap
    {
    ;()queue!(out, PrintChar('#')).unwrap // لبهٔ راست
    ;()queue!(out, PrintChar('\n')).unwrap
    {

    //  ردیف پایین کادر
    } for _ in 0..=W + 1
        ;()queue!(out, PrintChar('#')).unwrap
    {
        ;()queue!(out, PrintChar('\n')).unwrap
```

```
                                                    ;()out.flush().unwrap
                                                                        {

                                                  // ——— تابع اصلی
                                                     #[tokio::main]
                                  } <<async fn main() -> Result<(), Box<dyn Error
                                                  ;()let mut out = stdout
                                                ;?()terminal::enable_raw_mode
;?execute!(out, terminal::EnterAlternateScreen, cursor::Hide)

                                                ;()let mut snake = Snake::new
                                             ;let mut food = rand_pt(&snake.body)

                                                                } game: loop'
                                                  // پردازش ورودی بدون بلوکه‌شدن
                                        } ?while event::poll(Duration::from_millis(0))
                                           } ?()if let Event::Key(k) = event::read
                                                        } match k.code
                                                ,KeyCode::Up => snake.change(Dir::Up)
                                              ,KeyCode::Down => snake.change(Dir::Down)
                                              ,KeyCode::Left => snake.change(Dir::Left)
                                            ,KeyCode::Right => snake.change(Dir::Right)
                                                ,KeyCode::Char('q') => break 'game
                                                                    {} => _
                                                                            {
                                                                                {
                                                                                    {

                                                                        ;()snake.tick
                                                                    } ()if snake.crash
                                                                      ;break 'game
                                                                                    {

                                                                } if snake.head() == food
                                                                    ;snake.grow = true
                                                             ;food = rand_pt(&snake.body)
                                                                                    {
```

```
            ;draw(&snake, food, &mut out)
        ;sleep(Duration::from_millis(150)).await
                                        {

    //  تمیزکاری و نمایش امتیاز
;?execute!(out, cursor::Show, terminal::LeaveAlternateScreen)
                                ;?()terminal::disable_raw_mode
    ;println!("Game over! Score: {}", snake.body.len() - 1)
                                    (())Ok
                                                            {
```

درس: ساخت بازی "مار" در ترمینال – گام به گام

هدف: جمع‌بندی تمام مباحث ترم دوم با یک نمونهٔ کامل ولی ساده.

مرحلهٔ ۱ – ایجاد پروژه و افزودن وابستگی‌ها

۱. یک پروژهٔ جدید بسازید:

cargo new snake

۲. در Cargo.toml خطوط زیر را اضافه کنید:

```
tokio = { version = "1", features = ["macros", "rt-multi-thread", "time"] }
                                                "crossterm = "0.27
                                                    "rand = "0.8
```

- مفهوم: tokio برای برنامه‌نویسی ناهمزمان، crossterm برای کنترل ترمینال، و rand برای تولید اعداد تصادفی.

پرسش کلاس

سؤال	پاسخ کوتاه
چرا از ویژگی macros در Tokio استفاده کردیم؟	برای استفاده از ماکروی #[tokio::main]

مرحلهٔ ۲ – ثابت‌ها و انواع دادهٔ پایه

```
;const W: u16 = 40
;const H: u16 = 20
```

- دو ثابت عددی برای عرض و ارتفاع جدول بدون کادر.

تعریف جهت‌ها و نقطه‌ها

```
#[derive(Copy, Clone, PartialEq)]
enum Dir { Up, Down, Left, Right }

#[derive(Copy, Clone, PartialEq)]
struct Pt { x: u16, y: u16 }
```

- enum برای چهار جهت.
- struct برای مختصات؛ با derive می‌توانیم آنها را کپی و مقایسه کنیم.

```
impl Pt {
    fn step(self, d: Dir) -> Pt { /* جابجایی یک قدم */ }
}
```

- متد عضو نشان‌دهندهٔ مفهوم «حرکت یک نقطه در جهت خاص».

پرسش کلاس

۱. تفاوت Copy و Clone چیست؟

- *Copy* کپیِ بیت‌به‌بیت خودکار؛ *Clone* کپی صریح با فراخوانی تابع.

مرحلهٔ ۳ — ساختار **Snake**

```
struct Snake { body: Vec<Pt>, dir: Dir, grow: bool }
```

- Vec<Pt> طول متغیر بدن.
- dir جهت فعلی.
- grow علامت رشد در فریم بعد.

متدهای کلیدی:

```
fn new() -> Self { /* شروع از مرکز */ }
fn head(&self) -> Pt { /* اولین عنصر بردار */ }
fn change(&mut self, d: Dir) { /* جلوگیری از چرخش ۱۸۰° */ }
fn tick(&mut self) { /* پیشروی و رشد */ }
fn crash(&self) -> bool { /* برخورد با دیوار یا خود */ }
```

پرسش کلاس

سؤال	پاسخ
چرا head مرجع قرضی	چون Pt کوچک و Copy است؛ برگشت مقدار هزینه

ندارد. نیست؟

مرحلهٔ ۴ — توابع کمکی

تولید غذای تصادفی

```
fn rand_pt(except: &[Pt]) → Pt { /* تا پیدا شدن نقطهٔ آزاد */ }
```

- استفاده از rand::thread_rng() و حلقهٔ loop.

رسم صحنه

```
fn draw(s: &Snake, food: Pt, out: &mut Stdout) { /* چاپ کادر و محتوا */ }
```

- ماکرو queue! دستورات را بافر می‌کند؛
- لبه‌ها با # ؛ سر با O؛ بدن با o؛ غذا با *.

پرسش کلاس

۱. چه تفاوتی بین queue! و execute! وجود دارد؟

o queue! فقط در بافر می‌نویسد؛ نیاز به flush() دارد.

مرحلهٔ ۵ — حلقهٔ اصلی ناهمزمان

```
#[tokio::main]
async fn main() → Result<(), Box<dyn Error>> { /* ... */ }
```

- فعال‌سازی حالت *raw* ترمینال و صفحهٔ جایگزین.
- حلقهٔ 'game:
  ۱. Poll ورودی بدون بلوکه‌شدن.
  ۲. به‌روزرسانی مار (tick).
  ۳. بررسی برخورد و خوردن غذا.
  ۴. فراخوانی draw.
  ۵. وقفهٔ ۱۵۰ms با sleep (غیر بلوکه).

پرسش کلاس

سؤال	پاسخ

تا CPU در حین انتظار آزاد باشد.

چرا زمان‌بندی را با sleep ناهمزمان انجام دادیم؟

مرحلهٔ ۶ – تمیزکاری و پایان

- خروج از حلقه با برخورد یا فشردن q.
- بازگرداندن ترمینال به حالت عادی و چاپ امتیاز.

تمرین‌ها

۱. سرعت بازی را پارامتری کنید.

۲. برای مار دو زندگی تعریف کنید؛ پس از اولین برخورد با دیوار فقط Life کم شود.

۳. قابلیت wrap-around اضافه کنید تا دیوار کشنده نباشد.

جمع‌بندی کوتاه

این پروژه تقریباً تمام مفاهیم زیر را پوشش داد:

- ثابت‌ها و انواع عددی
- enum و struct
- بردار پویا (Vec)
- مالکیت، Borrow و Traitهای مشتق‌شده
- حلقهٔ loop، شرط if، و الگوی match
- ماکروها (queue!, execute!)
- برنامه‌نویسی ناهمزمان با Tokio

سؤال پایانی برای امتحان کوتاه: چرا از Box<dyn Error> برای نوع بازگشتی main استفاده کردیم؟

- چون می‌خواهیم هر خطایی که Error پیاده می‌کند را به‌سادگی با ? برگردانیم.

ماژول‌ها و کریت‌ها در Rust

همانطور که برنامه‌ها بزرگتر می‌شوند، سازماندهی کد اهمیت پیدا می‌کند. Rust برای این منظور از سیستم ماژول و کریت استفاده می‌کند. این سیستم به شما کمک می‌کند کد خود را ساختاردهی کنید. همچنین به شما اجازه می‌دهد بخش‌هایی از کد را خصوصی یا عمومی کنید.

کریت چیست؟

یک کریت (**Crate**) کوچکترین واحد کامپایل در Rust است. کریت می‌تواند یک کتابخانه (library) یا یک فایل اجرایی (binary) باشد. وقتی دستور cargo build را اجرا می‌کنید، Cargo یک کریت را کامپایل می‌کند.

انواع کریت

کریت کتابخانه‌ای: کدی که قرار است در پروژه‌های دیگر استفاده شود. نقطه شروع آن معمولاً src/lib.rs است. کریت اجرایی: برنامه‌ای که قابل اجرا است. نقطه شروع آن معمولاً src/main.rs است.

ماژول چیست؟

یک ماژول (**Module**) بخشی از کد در یک کریت است. شما می‌توانید از ماژول‌ها برای تقسیم‌بندی کد خود استفاده کنید. ماژول‌ها می‌توانند شامل توابع، ساختارها، شمارشگرها، ثابت‌ها و حتی ماژول‌های دیگر باشند.

تعریف ماژول‌ها (با کلمه کلیدی mod)

برای تعریف یک ماژول جدید، از کلمه کلیدی mod استفاده می‌کنیم.

```
Rust
} mod my_module
  //  کد داخل ماژول در اینجا قرار میگیرد
    } ()pub fn hello
;println!("Hello from my_module!")
                                {


        } ()fn private_function
;println!("This is private.")
                                    {
                                      {


                                } ()fn main
                                ;()my_module::hello
// خطا: این تابع خصوصی است ;()my_module::private_function //
                                {
```

ماژول‌ها معمولاً در فایل‌های جداگانه قرار می‌گیرند. اگر ماژولی به نام my_module در src/main.rs تعریف کنید، Rust انتظار دارد کد آن در فایل src/my_module.rs یا در پوشه src/my_module/mod.rs باشد.

قابلیت مشاهده (Visibility با کلمه کلیدی pub)

به طور پیش‌فرض، تمام موارد در Rust خصوصی هستند. این یعنی از خارج از حوزه تعریف خود قابل دسترسی نیستند. برای اینکه یک آیتم (تابع، ساختار، شمارشگر، ماژول و غیره) عمومی باشد و از خارج ماژول قابل دسترسی باشد، از کلمه کلیدی pub استفاده می‌کنیم.

```
Rust
} mod outer_module
    } pub mod inner_module
        } ()pub fn public_function
            ;println!("This is public!")
                                    {

                } ()fn private_function
            ;println!("This is private.")
                                    {
                                        {

        } ()fn private_outer_function
                                ... //
                                    {
                                {

} ()fn main
    ;()outer_module::inner_module::public_function
// ;()outer_module::inner_module::private_function // خطا
    // ;()outer_module::private_outer_function // خطا
{
```

pub mod باعث می‌شود ماژول inner_module قابل دسترسی باشد. pub fn باعث می‌شود تابع public_function قابل دسترسی باشد. بدون pub، موارد خصوصی می‌مانند.

مسیرها (Paths)

برای دسترسی به آیتم‌ها در ماژول‌ها، از مسیرها استفاده می‌کنیم. مسیرها مانند مسیر فایل‌ها در سیستم عامل هستند. از :: برای جدا کردن نام‌ها در مسیر استفاده می‌شود.

مسیر مطلق (Absolute Path): مسیری که از ریشه کریت شروع می‌شود. ریشه کریت یا crate است (برای کریت فعلی) یا نام کریت خارجی است. مسیر نسبی (Relative Path): مسیری که از ماژول فعلی شروع می‌شود. می‌توانید از self برای اشاره به ماژول فعلی یا super برای اشاره به ماژول والد استفاده کنید.

```
Rust
```

```
    } mod a
} pub mod b
} ()pub fn hello_b
;println!("Hello from b!")
{
{

} ()pub fn hello_a
// استفاده از مسیر نسبی
;()self::b::hello_b
{
{

} ()fn main
// استفاده از مسیر مطلق
;()crate::a::b::hello_b
;()crate::a::hello_a
{
```

وارد کردن مسیرها (با کلمه کلیدی use)

نوشتن مسیرهای طولانی می‌تواند خسته‌کننده باشد. از کلمه کلیدی use برای آوردن یک مسیر به حوزه فعلی استفاده می‌کنیم. این کار نام آیتم را کوتاه‌تر می‌کند.

```
Rust
} mod network
} pub mod client
} ()pub fn connect
;println!("Connecting ... ")
{
{
{

use network::client; // network::client را به حوزه فعلی میآورد

} ()fn main
client::connect(); // حالا میتوانیم از نام کوتاهتر استفاده کنیم
{
```

می‌توانید چندین مسیر را در یک خط با استفاده از {} وارد کنید:

```
Rust
// به جای دو خط:
```

```
use std::collections::HashMap; //
use std::collections::HashSet; //

// از یک خط استفاده کنید:
use std::collections::{HashMap, HashSet};
```

با use می‌توانید نام مستعار (Alias) نیز تعیین کنید:

```
Rust
use std::fmt::Result;
use std::io::Result as IoResult; // نام مستعار برای Result در ماژول io

fn function1() -> Result {
    // ...
    Ok(())
}

fn function2() -> IoResult<()> {
    // ...
    Ok(())
}
```

استفاده از کریت‌های خارجی

اکثر پروژه‌های Rust از کریت‌های خارجی استفاده می‌کنند. این کریت‌ها معمولاً از رجیستری crates.io دانلود می‌شوند. برای استفاده از یک کریت خارجی، آن را به فایل Cargo.toml در بخش [dependencies] اضافه می‌کنید.

مثال Cargo.toml:

```
Ini, TOML
[package]
name = "my_project"
version = "0.1.0"
edition = "2021"

[dependencies]
rand = "0.8.5" # اضافه کردن کریت rand
```

بعد از اضافه کردن وابستگی در Cargo.toml و ذخیره فایل، Cargo به طور خودکار کریت را دانلود می‌کند. حالا می‌توانید در کد Rust خود از آیتم‌های عمومی آن کریت استفاده کنید. ریشه مسیر این کریت، نام کریت است.

```
Rust
use rand::Rng; // استفاده از trait Rng از کریت rand
```

```
} ()fn main
    rand از کریت thread_rng به تابع دسترسی // ;()let mut rng = rand::thread_rng
    ;let random_number = rng.gen_range(1..=100)
    ;println!("Random number: {}", random_number)
{
```

خلاصه: سازماندهی فایل‌ها

در یک پروژه با Cargo، سازماندهی فایل‌ها به این شکل است:

- src/main.rs: ریشه کریت اجرایی.
- src/lib.rs: ریشه کریت کتابخانه‌ای.
- src/mod_name.rs: حاوی کد ماژولی به نام mod_name که در main.rs یا lib.rs یا ماژول دیگری تعریف شده است.
- src/mod_name/mod.rs: روش جایگزین برای سازماندهی کد ماژول mod_name در یک پوشه.

سوالات کوتاه برای سنجش درک:

۱. تفاوت اصلی بین کریت (Crate) و ماژول (Module) چیست؟

۲. کلمه کلیدی pub چه کاری انجام می‌دهد؟

۳. برای دسترسی به یک تابع در یک ماژول دیگر، از چه چیزی استفاده می‌کنیم؟

۴. کلمه کلیدی use چه فایده‌ای دارد؟

۵. برای استفاده از یک کتابخانه خارجی مانند rand، اولین قدم چیست؟