https://simplifycpp.org

أساسيات لغة Rust

قاعدة عملية وواضحة لكل مبرمج

إعداد: أيمن الحراكي

أساسيات لغة Rust

قاعدة عملية وواضحة لكل مبرمج

تم دعم بعض أعمال الصياغة واستكشاف الأفكار باستخدام أدوات ذكاء اصطناعي حديثة،

وذلك تحت إشراف كامل، مع التحقق والمراجعة والتصحيح، وبمسؤولية وتأليف كاملين.

إعداد : أيمن الحراكي

simplifycpp.org

فبراير 2026

المحتويات

مقدمة المؤلف

تمثل Rust تطورًا مقصودًا في برمجة الأنظمة: أداءً عاليًا دون التضحية بالأمان. فهي تُزيل المقايضة التقليدية بين التحكم والموثوقية من خلال فرض الصحة البرمجية وقت الترجمة عبر مفاهيم ownership وborrowing وlifetimes — دون الاعتماد على garbage collector.

كُتب **Rust Essentials — A Practical and Clear Foundation for Every Programmer** لتقديم مسار مركز ومنظم لدخول اللغة. الهدف ليس تغطية كل موضوع متقدم، بل بناء نموذج ذهني قوي يمكّنك من التقدم بثقة إلى ما بعد الأساسيات.

رغم أنني حديث العهد نسبيًا بلغة Rust وقادم من خلفية طويلة في C++، فقد وجدت هذه اللغة سهلة التعلم بشكل مفاجئ، قوية، وغنية بمبادئ التصميم الحديثة. خلال هذه الفترة أصدرت عدة كتيبات حول Rust، ودائمًا أشعر بالحاجة إلى الكتابة عنها بكثافة لجعلها أسهل وأكثر إتاحة للجميع. وأنصح كل مبرمج C++ أن يضيف Rust إلى صندوق أدواته التقنية وأن يستخدمها حين تكون متطلبات الأمان وصحة الذاكرة والضمانات الصارمة عناصر حاسمة.

في هذا الكتيب نركّز على:

- فهم مفهومي ownership و borrowing بوضوح.
- كتابة شيفرة Rust بأسلوب قياسي باستخدام enums و pattern matching وiterators.
- التعامل مع الأخطاء بدقة ووضوح.
- تنظيم مشاريع Rust النظيفة والقابلة للتوسع.
- بناء مشروع مصغر متكامل يدمج المفاهيم الأساسية.

قد تبدو Rust صارمة في البداية، لكن صرامتها توجيه وإرشاد. كل رسالة من المترجم compiler تفرض قاعدة أمان تحمي برنامجك في النهاية. وبمجرد فهم المبادئ الجوهرية، تصبح Rust لغة متسقة، قابلة للتنبؤ، وعميقة المنطق.

يفترض هذا الكتاب وجود خبرة برمجية مسبقة، وهو موجّه للمطورين من أي خلفية يرغبون في تأسيس عملي وواضح في Rust.

ادرس الأمثلة، وجرّب الشيفرات، وابنِ أدوات صغيرة خاصة بك. الإتقان في Rust يأتي من الممارسة ومن التفكير الدقيق في مفاهيم ownership وصحة التصميم.

أيمن الحراكي

التمهيد

لماذا وُجد هذا الكتاب

برزت Rust كإحدى أكثر لغات برمجة الأنظمة تأثيرًا خلال العقد الماضي. صُممت لتوفير أمان الذاكرة ومنع سباقات البيانات data-race prevention وأداء عالٍ دون الاعتماد على garbage collection. يتيح نموذج ownership الخاص بها، وفحوصات وقت الترجمة الصارمة، ونظام الأنواع التعبيري type system كتابة برمجيات موثوقة وفعالة في مجالات مثل برمجة الأنظمة، وخدمات backend، والأنظمة المدمجة embedded development، والتطبيقات عالية الأداء.

ومع ذلك، يواجه كثير من المبرمجين Rust من خلال شروحات مجزأة أو مواد متقدمة تفترض خبرة سابقة عميقة على مستوى الأنظمة. هدف هذا الكتيب هو تقديم أساس واضح ومنظم وعملي يشرح المفاهيم الجوهرية في Rust بطريقة متاحة لأي مبرمج مهما كانت خلفيته.

يركّز هذا الكتاب على بناء نموذج ذهني قوي لمفاهيم ownership و borrowing و lifetimes ومعالجة الأخطاء وتنظيم المشاريع. وبنهاية هذا الكتيب، ينبغي للقارئ أن يكون قادرًا على كتابة برامج صحيحة بأسلوب قياسي في Rust وفهم المنطق وراء ضمانات المترجم الصارمة.

لمن هذا الكتاب

هذا الكتاب موجّه إلى:

- المبرمجين من أي خلفية (web، أو mobile، أو البيانات data، أو scripting، أو الأنظمة) ممن يرغبون في مقدمة راسخة إلى Rust.
- المطورين الملمين بلغات مثل C و C++ و Java و Python و Go و JavaScript ممن يريدون فهم نموذج الأمان والأداء في Rust.
- الطلاب الباحثين عن لغة أنظمة حديثة وآمنة بضمانات قوية من المترجم compiler.
- المهندسين الراغبين في الانتقال إلى مجالات برمجية تركز على الأداء العالي أو الموثوقية.

لا يُشترط وجود خبرة سابقة في إدارة الذاكرة منخفضة المستوى. يُفترض فقط الإلمام بالمفاهيم البرمجية الأساسية (المتغيرات، الدوال، تدفق التحكم).

كيفية استخدام هذا الكتاب بفعالية

تُتعلم Rust على أفضل وجه من خلال كتابة الشيفرة وملاحظة ملاحظات المترجم compiler. ولتحقيق أقصى فائدة:

- اكتب كل مثال يدوياً بدلاً من النسخ واللصق.
- قم بعملية الترجمة compile بشكل متكرر واقرأ رسائل المترجم بعناية.
- عدّل الأمثلة لاختبار الحالات الحدّية والتصاميم البديلة.
- أعد قراءة فصول Ownership و Borrowing عدة مرات؛ فهي تمثل القلب المفاهيمي للغة Rust.
- ابنِ المشروع المصغر في نهاية الكتاب من الصفر.

المترجم compiler ليس مجرد أداة لاكتشاف الأخطاء؛ بل هو دليل تفاعلي يفرض تصميماً صحيحاً للبرنامج. تعامل مع رسائل الخطأ باعتبارها شرحاً لضمانات Rust لا عوائق في طريقك.

ما الذي لا يغطيه هذا الكتاب

للحفاظ على الوضوح والتركيز ضمن عدد محدود من الصفحات، لا يغطي هذا الكتيب:

- البرمجة غير المتزامنة المتقدمة advanced asynchronous programming وبيئات التنفيذ executors.
- Unsafe Rust ومعالجة الذاكرة منخفضة المستوى.
- وحدات procedural macros وأنظمة macros المتقدمة.
- تصميم trait objects بعمق والاستخدامات المتقدمة لـ generics.
- بدائيات التزامن concurrency primitives بشكل موسع.
- أنماط البنية المعمارية واسعة النطاق.

هذه الموضوعات تتطلب معالجة منفصلة أكثر تقدمًا. هذا الكتاب يؤسس القاعدة اللازمة قبل التوسع في تلك المجالات.

مسار التعلم المقترح

لأفضل النتائج، اتبع هذا التسلسل المنظم:

١. إتقان المتغيرات وتدفق التحكم وأنواع البيانات الأساسية.
٢. تطوير فهم دقيق لمفهومي ownership و borrowing.
٣. دراسة lifetimes بعد الارتياح التام لقواعد الاستعارة.
٤. التدرب على استخدام الحاويات collections ومعالجة الأخطاء.
٥. تنظيم الشيفرة ضمن وحدات modules ومشاريع صغيرة.

٦. إنجاز المشروع المصغر النهائي بشكل مستقل.

بعد إنهاء هذا الكتيب، سيكون القارئ مستعداً لاستكشاف موضوعات متقدمة في Rust مثل البرمجة غير المتزامنة asynchronous programming، والتزامن concurrency، وضبط الأداء perfor-mance tuning، وتطوير الأنظمة منخفضة المستوى بثقة.
يهدف هذا الكتاب إلى تقديم أكثر من مجرد معرفة تركيبية syntax، بل إلى غرس عقلية هندسية منضبطة تنسجم مع فلسفة Rust الجوهرية: الأمان، والوضوح، والأداء المصمم بعناية.

الفصل ١: البدء باستخدام Rust

١.١ تثبيت Rust و Cargo

يتم توزيع Rust عبر مُثبّت رسمي يُسمى rustup. يتولى هذا المُثبّت إدارة المترجم compiler (rustc)، ومدير الحزم وأداة البناء (cargo)، وسلاسل الأدوات toolchains (stable، beta، nightly). الطريقة الموصى بها للتثبيت هي عبر rustup لأنه يضمن بقاء بيئة التطوير محدثة ومُهيأة بشكل صحيح.
على أنظمة شبيهة Unix (مثل Linux وmacOS):

```
curl --proto '=https' --tlsv1.2 -sSf https://sh.rustup.rs | sh
```

على نظام Windows يمكن تنزيل المُثبّت الرسمي من موقع Rust أو استخدام:

```
winget install Rustlang.Rustup
```

بعد التثبيت، تحقق من الإصدارات:

```
rustc --version
cargo --version
```

يتم تثبيت سلسلة الأدوات stable toolchain افتراضيًا. لتحديث Rust:

```
rustup update
```

للتحقق من سلاسل الأدوات المثبتة:

```
rustup show
```

تُركّز الأدوات الرسمية في Rust على قابلية إعادة الإنتاج reproducibility، وإدارة الإصدارات، والتعامل المتكامل مع الحزم عبر Cargo.

٢.١ فهم الفرق بين rustc وCargo

rustc هو مترجم Rust. يقوم بترجمة ملف مصدر واحد إلى ملف تنفيذي binary أو مكتبة library.
مثال:

```
rustc main.rs
```

ينتج عن ذلك ملف تنفيذي من main.rs. ومع ذلك، في التطوير الواقعي نادرًا ما يتم استخدام rustc مباشرة.
أما Cargo فهو نظام البناء build system ومدير الحزم package manager الخاص بـ Rust. وهو يتولى:

- هيكلة المشروع project structure
- إدارة الاعتماديات dependency management
- الترجمة compilation
- الاختبارات testing
- توليد التوثيق documentation generation
- بناء نسخ قابلة لإعادة الإنتاج reproducible builds

يقوم Cargo داخليًا باستدعاء rustc لكنه يضيف إعدادات التهيئة، وحل الاعتماديات، وتنظيم المشروع.
في تطوير Rust الحديث، يُعد Cargo أداة سير العمل القياسية.

٣.١ إنشاء أول مشروع لك

لإنشاء مشروع تنفيذي binary project جديد:

```
cargo new hello_rust
```

سيؤدي ذلك إلى إنشاء البنية التالية:

```
hello_rust/
    Cargo.toml
    src/
        main.rs
```

انتقل إلى مجلد المشروع:

```
cd hello_rust
```

يحتوي الملف الافتراضي main.rs على:

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

لتشغيل المشروع:

```
cargo run
```

يقوم Cargo بترجمة المشروع وتنفيذه بأمر واحد.
لإنشاء مشروع مكتبة library project بدلاً من ذلك:

```
cargo new my_library --lib
```

سيتم إنشاء ملف lib.rs بدلاً من main.rs.

٤.١ شرح بنية المشروع

يتضمن مشروع Cargo القياسي:

- Cargo.toml — بيانات تعريف المشروع والاعتماديات
- src/main.rs — نقطة الدخول للتطبيقات التنفيذية
- src/lib.rs — جذر المكتبة (عند الحاجة)
- target/ — مخرجات البناء build artifacts (يتم إنشاؤها تلقائيًا)

مثال على Cargo.toml:

```
[package]
name = "hello_rust"
version = "0.1.0"
edition = "2021"

[dependencies]
```

المكونات الأساسية:

- الحقل edition يحدد إصدار اللغة (إصدار 2021 هو المعيار المستقر الحالي).
- القسم [dependencies] يسرد الحزم الخارجية external crates.

لإضافة اعتماد (مثال: rand):

```
[dependencies]
rand = "0.8"
```

سيقوم Cargo تلقائيًا بتنزيل الاعتماديات وترجمتها أثناء عملية البناء.

٥.١ تشغيل وبناء التطبيقات

أهم أوامر Cargo:
التشغيل في وضع التطوير development mode:

```
cargo run
```

الترجمة دون تشغيل:

```
cargo build
```

بناء نسخة محسّنة للإصدار release:

```
cargo build --release
```

يتواجد الملف التنفيذي للإصدار في:

```
target/release/
```

تركّز نسخ التطوير على سرعة الترجمة، بينما تُفعّل نسخ release تحسينات مثل الإدراج inlining وتحسين توليد الشيفرة code generation.
لتشغيل الاختبارات:

```
cargo test
```

لتوليد التوثيق:

```
cargo doc --open
```

مثال على تعديل main.rs:

```
fn main() {
    let name = "Rust";
    println!("Welcome to {} programming!", name);
}
```

ثم أعد التشغيل:

```
cargo run
```

سيكتشف Cargo التغييرات، ويعيد ترجمة الأجزاء المعدلة فقط، ثم ينفذ البرنامج.
في هذه المرحلة تكون قد:

- ثبّتَّ وتهيّأت لاستخدام Rust.
- فهمت الفرق بين rustc وCargo.
- أنشأت ونفّذت مشروع Rust منظمًا.
- بنيت نسخ تطوير ونسخًا محسّنة optimized binaries.
- أدرت بيانات المشروع والاعتماديات.

يمثل هذا المسار أساس جميع عمليات التطوير الحديثة باستخدام Rust.

الفصل ٢: المتغيرات والأنواع وقابلية التغيير

٢.١ let وmut

في لغة Rust تكون المتغيرات غير قابلة للتغيير افتراضيًا. يشجع هذا التصميم على كتابة شيفرة أكثر أمانًا من خلال منع التعديل غير المقصود.

```
fn main() {
    let x = 10;
    println!("{}", x);
}
```

محاولة تعديل x ستؤدي إلى خطأ في وقت الترجمة compile-time error:

```
fn main() {
    let x = 10;
    x = 20; // error: cannot assign twice to immutable variable
}
```

للسماح بالتغيير استخدم mut:

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

```
    counter += 1;
    println!("{}", counter);
}
```

المبدأ الأساسي: عدم القابلية للتغيير افتراضيًا immutability by default يعزز الموثوقية ويقلل من تغييرات الحالة غير المقصودة accidental state changes.

٢.٢ أنواع البيانات الأولية Primitive Data Types

توفر Rust عدة أنواع أولية مدمجة.

أنواع الأعداد الصحيحة **Integer Types**

الموقّعة Signed: i8, i16, i32, i64, i128 isize

غير الموقّعة Unsigned: u8, u16, u32, u64, u128 usize

```
let a: i32 = -42;
let b: u64 = 100;
```

النوع الافتراضي للأعداد الصحيحة هو i32.

أنواع الفاصلة العائمة **Floating-Point Types**

f32 وf64 (الافتراضي هو f64)

```
let pi: f64 = 3.1415;
```

القيمة المنطقية **Boolean**

```
let is_active: bool = true;
```

المحرف **Character**

النوع char يمثل قيمة يونيكود Unicode scalar value.

```
let letter: char = 'A';
let emoji: char = ' ';
```

٣.٢ استدلال الأنواع Type Inference

تعتمد Rust على الكتابة الساكنة static typing مع استدلال قوي للأنواع strong type inference.
يحدد المترجم النوع تلقائيًا عندما يكون ذلك ممكنًا.

```
let x = 5;        // inferred as i32
let y = 3.14;     // inferred as f64
```

يلزم التصريح الصريح explicit annotation عندما يكون الاستدلال غير واضح:

```
let guess: u32 = "42".parse().expect("Not a number");
```

يفرض المترجم صحة الأنواع في وقت الترجمة، مما يمنع العديد من أخطاء وقت التشغيل runtime errors.

٤.٢ الصفوف المتغايرة Tuples والمصفوفات Arrays

الصفوف المتغايرة Tuples
تجمع Tuples عدة قيم بأنواع مختلفة.

```
let person: (i32, f64, char) = (30, 72.5, 'M');

let age = person.0;
let weight = person.1;
```

تفكيك البنية Destructuring:

```
let (age, weight, gender) = person;
```

المصفوفات Arrays
المصفوفات ذات حجم ثابت ونوع موحد.

```
let numbers: [i32; 3] = [10, 20, 30];
let first = numbers[0];
```

يتم تخصيص المصفوفات على المكدس stack-allocated ويكون حجمها معروفًا في وقت الترجمة.
مثال على التكرار:

```
for n in numbers {
    println!("{}", n);
}
```

٥.٢ تحويل الأنواع والتحليل Type Casting and Parsing

لا تقوم Rust بتحويلات ضمنية implicit conversions يجب أن يكون التحويل صريحًا باستخدام as

```
let x: i32 = 10;
let y: f64 = x as f64;
```

تحويل الأرقام من السلاسل النصية:

```
let input = "25";
let number: i32 = input.parse().expect("Invalid number");
```

المعالجة الآمنة للتحليل:

```
let result: Result<i32, _> = "42".parse();

match result {
    Ok(n) => println!("Parsed: {}", n),
    Err(e) => println!("Error: {}", e),
}
```

تتجنب Rust التحويلات الصامتة للحفاظ على الصحة والتنبؤ correctness and predictability.

٦.٢ الثوابت Constants والتظليل Shadowing

الثوابت Constants

يتم تعريف الثوابت باستخدام const ويجب أن تحتوي على تصريح نوع صريح.

```
const MAX_USERS: u32 = 100;
```

الثوابت غير قابلة للتغيير ويتم تقييمها في وقت الترجمة.

التظليل Shadowing

يسمح التظليل بإعادة تعريف متغير بنفس الاسم.

```
let x = 5;
let x = x + 1;
let x = x * 2;

println!("{}", x); // 12
```

يختلف التظليل عن التغيير mutation:

- التظليل ينشئ متغيرًا جديدًا.
- يمكنه تغيير النوع.
- لا يتطلب mut.

مثال على تغيير النوع عبر التظليل:

```
let spaces = "   ";
let spaces = spaces.len();
```

هذا غير مسموح به باستخدام mut لأن التغيير لا يسمح بتبديل النوع.

في هذه المرحلة أصبحت تفهم:

- نموذج عدم القابلية للتغيير الافتراضي في Rust.
- أنواع البيانات الأولية الأساسية.
- الكتابة الساكنة مع استدلال قوي للأنواع.
- المصفوفات ذات الحجم الثابت والصفوف المتغايرة متعددة الأنواع.
- التحويل الصريح والتحليل الآمن.
- الفرق بين التغيير mutation والتظليل shadowing.

تمثل هذه الأسس قاعدة ضرورية قبل الانتقال إلى مفاهيم الملكية ownership ودلالات الذاكرة memory semantics.

الفصل ٣: تدفق التحكم والدوال

١.٣ تعريف الدوال واستدعاؤها

تُعرَّف الدوال في Rust باستخدام الكلمة المفتاحية fn. يجب تحديد أنواع المعاملات بشكل صريح، كما يتم تحديد نوع القيمة المرجعة باستخدام ->.

```
fn greet(name: &str) {
    println!("Hello, {}!", name);
}

fn main() {
    greet("Rust");
}
```

يمكن للدوال إرجاع قيم:

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

fn main() {
```

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

يتم إرجاع آخر تعبير expression ضمن الدالة بشكل ضمني إذا لم ينتهِ بفاصلة منقوطة.
إرجاع صريح باستخدام return:

```
fn multiply(a: i32, b: i32) -> i32 {
    return a * b;
}
```

تفرض Rust تحديد أنواع المعاملات والقيم المرجعة بشكل صريح لضمان الوضوح وسلامة الترجمة compile-time safety.

٢.٣ التعابير مقابل التعليمات

تميّز Rust بين التعابير expressions والتعليمات statements.
التعليمات **Statements** تنفذ إجراءً لكنها لا تعيد قيمة.

```
let x = 5; // statement
```

التعابير **Expressions** تُقيَّم إلى قيمة.

```
let y = {
    let x = 3;
    x + 1
};
```

في هذا المثال، يتم تقييم الكتلة إلى القيمة 4 لأن x + 1 تعبير غير منتهٍ بفاصلة منقوطة.
إذا أُضيفت فاصلة منقوطة، يتحول التعبير إلى تعليمة ويُرجع النوع () (نوع الوحدة unit type):

```
let y = {
    let x = 3;
    x + 1;
};
```

فهم هذا الفرق أساسي لأن تراكيب التحكم في Rust هي تعابير.

٣.٣ if و match

تعبير if

في Rust تُعد if تعبيرًا ويمكن أن تعيد قيمة.

```
let number = 10;

let result = if number > 5 {
    "Greater"
} else {
    "Smaller or equal"
};

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

يجب أن تُرجع جميع الفروع أنواعًا متوافقة.

تعبير match

يوفر match مطابقة أنماط قوية pattern matching ويجب أن تكون شاملة exhaustive.

```
let value = 3;

match value {
    1 => println!("One"),
```

```
    2 => println!("Two"),
    3 => println!("Three"),
    _ => println!("Other"),
}
```

إرجاع قيم باستخدام match:

```
let description = match value {
    1 => "One",
    2 | 3 => "Two or Three",
    _ => "Other",
};
```

مطابقة مجالات ranges:

```
let grade = 85;

match grade {
    90..=100 => println!("A"),
    80..=89 => println!("B"),
    _ => println!("Below B"),
}
```

يُفضَّل استخدام match عند التعامل مع حالات متعددة أو بيانات مهيكلة.

٤.٣ for و while و loop

loop

حلقة لا نهائية يمكنها إرجاع قيمة باستخدام break.

```rust
let mut counter = 0;

let result = loop {
    counter += 1;
    if counter == 5 {
        break counter * 2;
    }
};

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

while

حلقة شرطية:

```rust
let mut n = 3;

while n > 0 {
    println!("{}", n);
    n -= 1;
}
```

for

تتكرر عبر المكررات iterators والمجالات ranges:

```rust
for i in 1..5 {
    println!("{}", i);
}
```

التكرار عبر المصفوفات:

```rust
let numbers = [10, 20, 30];
```

```
for n in numbers {
    println!("{}", n);
}
```

تُعد حلقة for البنية الأكثر شيوعًا والأسلوب القياسي idiomatic في Rust.

٥.٣ المجالات وأساسيات التكرار

يتم إنشاء المجالات باستخدام .. و..=.

- 1..5 تشمل القيم 1 و2 و3 و4
- 1..=5 تشمل القيم من 1 إلى 5

مثال:

```
for i in 1..=3 {
    println!("{}", i);
}
```

استخدام أساسي للمكررات iterators:

```
let values = [1, 2, 3, 4];

for v in values.iter() {
    println!("{}", v);
}
```

استخدام enumerate:

```rust
for (index, value) in values.iter().enumerate() {
    println!("{} -> {}", index, value);
}
```

تصفية بسيطة:

```rust
let even: Vec<i32> = (1..10)
    .filter(|x| x % 2 == 0)
    .collect();

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

يعتمد نموذج التكرار في Rust على السمة Iterator، مما يتيح تنقلاً آمنًا وتعبيريًا دون الحاجة إلى إدارة الفهارس يدويًا.

في هذه المرحلة أصبحت تفهم:

- كيفية تعريف الدوال وإرجاع القيم منها.
- الفرق بين التعابير expressions والتعليمات statements.
- منطق الشروط باستخدام if وmatch.
- بنيات الحلقات واستخدامها الأسلوبي القياسي.
- أنماط التكرار الأساسية باستخدام المجالات والمكررات.

تمثل هذه الأسس في تدفق التحكم قاعدة ضرورية قبل التعمق في نموذج الملكية ownership ونموذج الذاكرة في Rust.

الفصل ٤: الملكية Ownership: المفهوم الجوهري

٤.١ ما الذي تعنيه الملكية فعليًا

الملكية Ownership هي نظام انضباط في وقت الترجمة compile-time discipline لإدارة الذاكرة وأعمار الموارد resource lifetimes دون الحاجة إلى جامع نفايات garbage collection.
في Rust كل قيمة لها مالك owner واحد فقط في أي لحظة. وعندما يخرج المالك من النطاق scope يتم إسقاط القيمة dropped تلقائيًا (بأسلوب RAII-style) وتحرير مواردها بشكل حتمي deterministically.
القواعد الأساسية:

- لكل قيمة مالك واحد.
- لا يمكن أن يوجد أكثر من مالك في الوقت نفسه.
- عندما يخرج المالك من النطاق، تُسقط القيمة.

مثال بسيط على النطاق:

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

```
        let s = String::from("hello");
        println!("{}", s);
    } // s is dropped here (memory released)
}
```

الملكية لا تتعلق بالذاكرة فقط، بل تنطبق أيضًا على موارد مثل الملفات files والمقابس sockets والأقفال locks والمقابض handles. عند إسقاط المالك يتم تنفيذ التنظيف cleanup تلقائيًا.

٢.٤ النقل Move مقابل النسخ Copy

بعض الأنواع تطبق السمة Copy: عند إسنادها يتم نسخ القيمة (نسخ ثنائي bitwise copy) وتبقى المتغيرات صالحة.
معظم القيم العددية البسيطة scalars هي Copy (مثل الأعداد الصحيحة، bool، الأعداد العائمة، char، والصفوف المتجانسة من أنواع Copy).

```
fn main() {
    let a: i32 = 10;
    let b = a; // Copy
    println!("a = {}, b = {}", a, b);
}
```

أنواع أخرى لا تطبق Copy. عند إسنادها يتم نقل الملكية move. بعد النقل يصبح المتغير القديم غير صالح.

```
fn main() {
    let s1 = String::from("hi");
    let s2 = s1; // Move
    // println!("{}", s1); // error: value borrowed here after move
    println!("{}", s2);
}
```

لماذا النقل؟ النوع String يملك ذاكرة على الكومة heap. لو تم نسخه ضمنيًا لأصبح هناك مالكان يحاولان تحرير نفس الذاكرة، مما يؤدي إلى تحرير مزدوج double-free. تمنع Rust ذلك عبر نقل الملكية.

٣.٤ الكومة Heap مقابل المكدس Stack (رؤية عملية)

نموذج عملي:

- المكدس **Stack**: قيم سريعة ذات حجم ثابت تُدار حسب النطاق.
- الكومة **Heap**: تخصيصات ديناميكية الحجم تُدار عبر مؤشرات ويجب تحريرها بشكل صحيح.

النوع String عبارة عن قيمة صغيرة على المكدس تحتوي (مفهوميًا) على مؤشر وطول وسعة، بينما المخزن الفعلي للأحرف يوجد على الكومة.

```
fn main() {
    let s = String::from("Rust");
    // s is on the stack; its character buffer is on the heap.
    println!("len = {}", s.len());
}
```

أنواع Copy مثل i32 تعيش بالكامل على المكدس. تصبح قواعد الملكية حاسمة عندما تمتلك القيم ذاكرة على الكومة أو موارد خارجية.

٤.٤ الملكية والدوال

تمرير القيم إلى الدوال يتبع القواعد نفسها: أنواع Copy تُنسخ، والأنواع الأخرى تُنقل.
مثال على النسخ:

```rust
fn print_num(n: i32) {
    println!("n = {}", n);
}

fn main() {
    let x = 7;
    print_num(x); // x is copied
    println!("x still usable: {}", x);
}
```

مثال على النقل:

```rust
fn consume(s: String) {
    println!("consumed: {}", s);
} // s dropped here

fn main() {
    let s = String::from("owned");
    consume(s); // moved into function
    // println!("{}", s); // error: use of moved value
}
```

إرجاع الملكية:

```rust
fn make_message() -> String {
    String::from("hello")
}

fn main() {
    let msg = make_message(); // ownership returned to caller
    println!("{}", msg);
}
```

النقل إلى الدوال ومنها طبيعي وفعال: غالبًا ما يتم نسخ رأس صغير فقط (مثل رأس String) وليس كامل بيانات الكومة.

٥.٤ clone مقابل الاستعارة Borrowing

إذا احتجت إلى نسختين مستقلتين مملوكتين، يمكنك استخدام clone صراحة. هذا يؤدي إلى نسخ عميق deep copy عند الحاجة.

```
fn main() {
    let s1 = String::from("data");
    let s2 = s1.clone(); // deep copy
    println!("s1 = {}, s2 = {}", s1, s2);
}
```

لكن النسخ قد يكون مكلفاً. إذا كنت تحتاج وصولاً مؤقتًا فقط، فالأفضل استخدام الاستعارة borrowing عبر المراجع.
مثال على الاستعارة:

```
fn print_len(s: &String) {
    println!("len = {}", s.len());
}

fn main() {
    let s = String::from("borrow me");
    print_len(&s); // borrowed, not moved
    println!("still owned: {}", s);
}
```

الأسلوب الأكثر شيوعًا مع السلاسل النصية يستخدم &str:

```rust
fn print_len(s: &str) {
    println!("len = {}", s.len());
}

fn main() {
    let s = String::from("borrow me");
    print_len(&s);        // &String -> &str coercion
    print_len("literal"); // &str directly
}
```

إرشادات:

- استخدم clone عندما تحتاج فعلاً إلى نسخة مملوكة مستقلة.
- استخدم الاستعارة (&T، &mut T، &str) للوصول دون نقل الملكية.

٦.٤ أخطاء شائعة في الملكية

1) الاستخدام بعد النقل

```rust
fn main() {
    let s1 = String::from("hello");
    let s2 = s1;
    // println!("{}", s1); // error: use of moved value
    println!("{}", s2);
}
```

الحلول:

- الاستعارة بدل النقل: let s2 = &s1;

- النسخ الصريح: let s2 = s1.clone();

2) نقل قيمة من مرجع مستعار

```
fn take_owned(s: String) {}

fn main() {
    let s = String::from("x");
    let r = &s;
    // take_owned(*r); // error: cannot move out of borrowed content
}
```

الحل:

- تغيير الدالة لتقبل استعارة: take_borrowed(s: &str) {}

3) نقل جزئي من البنى **structs**

إذا احتوت البنية على حقول غير Copy فإن نقل أحد الحقول قد يجعل البنية غير قابلة للاستخدام جزئيًا.

```
struct Pair {
    a: String,
    b: String,
}

fn main() {
    let p = Pair { a: String::from("A"), b: String::from("B") };
    let a = p.a; // moves field a
    // println!("{}", p.b);
    println!("{}", a);
}
```

الحلول:

- استعارة الحقول: let a = &p.a;
- تفكيك البنية لنقل جميع الحقول:

```
fn main() {
    let p = Pair { a: String::from("A"), b: String::from("B") };
    let Pair { a, b } = p; // moves both fields
    println!("{} {}", a, b);
}
```

4) الخلط بين Copy وclone

```
fn main() {
    let x = 5;
    let y = x;          // Copy, cheap
    let z = x.clone(); // unnecessary for Copy types
    println!("{} {} {}", x, y, z);
}
```

الملكية هي الأساس الذي يمكّن ضمانات Rust. عندما تفهم النقل moves والنسخ copies ومقايضات الاستعارة borrowing trade-offs، تصبح الفصول التالية (قواعد الاستعارة وأعمار المراجع lifetimes) امتدادًا طبيعيًا لا عقبة.

الفصل ٥: الاستعارة والمراجع Borrowing and References

١.٥ المراجع غير القابلة للتغيير Immutable References

المرجع غير القابل للتغيير (&T) يسمح لك بقراءة قيمة دون أخذ ملكيتها ودون تعديلها. تُعد الاستعارة Borrowing الأداة الأساسية في Rust لمشاركة الوصول بأمان مع الحفاظ على مالك واحد واضح.

```
fn print_len(s: &String) {
    println!("len = {}", s.len());
}

fn main() {
    let s = String::from("borrowed");
    print_len(&s);              // s is borrowed, not moved
    println!("still owns: {}", s);
}
```

الأسلوب القياسي مع السلاسل النصية يستخدم &str:

```rust
fn print_len(s: &str) {
    println!("len = {}", s.len());
}

fn main() {
    let s = String::from("hello");
    print_len(&s);              // &String coerces to &str
    print_len("literal");       // &str directly
}
```

يمكن إنشاء عدة مراجع غير قابلة للتغيير في الوقت نفسه:

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

    let a = &v;
    let b = &v;

    println!("{:?} {:?}", a, b);
}
```

الاستعارة غير القابلة للتغيير آمنة لأنه لا يمكن تعديل القيمة أثناء وجود قرّاء.

٢.٥ المراجع القابلة للتغيير Mutable References

المرجع القابل للتغيير (T &mut) يمنح وصولاً حصريًا مؤقتًا للكتابة. لا ينقل الملكية، لكنه يتطلب التفرد uniqueness.

```rust
fn push_value(v: &mut Vec<i32>, x: i32) {
    v.push(x);
```

```
}

fn main() {
    let mut v = vec![1, 2, 3];
    push_value(&mut v, 4);
    println!("{:?}", v);
}
```

لا يمكن إنشاء مرجع قابل للتغيير إلا إذا كان الربط الأصلي قابلاً للتغيير (mut let).

```
fn main() {
    let mut s = String::from("hi");
    let r = &mut s;
    r.push_str(" rust");
    println!("{}", r);
}
```

المرجع القابل للتغيير حصري: أثناء وجوده لا يمكن وجود أي مراجع أخرى (قابلة أو غير قابلة للتغيير) لنفس القيمة.

٣.٥ شرح قواعد الاستعارة بوضوح

تفرض Rust مجموعة صغيرة من القواعد في وقت الترجمة لضمان أمان الذاكرة ومنع سباقات البيانات data races:

- يمكنك إنشاء أي عدد من المراجع غير القابلة للتغيير (&T) في الوقت نفسه.
- يمكنك إنشاء مرجع قابل للتغيير واحد فقط (T &mut) في الوقت نفسه.
- لا يمكن وجود مرجع قابل للتغيير أثناء وجود مراجع غير قابلة للتغيير.

- يجب أن تكون المراجع صالحة دائمًا (لا مراجع متدلية dangling references).

أمثلة شائعة:

عدة قرّاء مسموح بهم:

```
fn main() {
    let s = String::from("read");
    let r1 = &s;
    let r2 = &s;
    println!("{} {}", r1, r2);
}
```

كاتبان غير مسموح بهما:

```
fn main() {
    let mut s = String::from("write");
    let r1 = &mut s;
    // let r2 = &mut s; // error: cannot borrow `s` as mutable more than once
    r1.push_str("!");
    println!("{}", r1);
}
```

قارئ وكاتب في الوقت نفسه غير مسموح:

```
fn main() {
    let mut s = String::from("mix");
    let r = &s;
    // let w = &mut s; // error: cannot borrow as mutable because it is also
    ↪  borrowed as immutable
    println!("{}", r);
}
```

تستخدم Rust أيضًا مفهوم الأعمار غير المعجمية (non-lexical lifetimes (NLL: تنتهي الاستعارة عند آخر استخدام فعلي لها، وليس بالضرورة عند نهاية النطاق.

```
fn main() {
    let mut s = String::from("nll");

    let r = &s;
    println!("{}", r); // last use of r

    let w = &mut s;    // now allowed because r is no longer used
    w.push_str("_ok");

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

لا مراجع متدلية:
لن تسمح Rust بإرجاع مرجع إلى متغير محلي سيتم إسقاطه.

```
fn bad_ref() -> &String {
    let s = String::from("dangling");
    &s // error: returns a reference to data owned by the current function
}
```

الحل هو إرجاع قيمة مملوكة:

```
fn good_value() -> String {
    let s = String::from("owned");
    s
}
```

٤.٥ لماذا تمنع Rust سباقات البيانات

تحدث سباقات البيانات data races عندما تصل عدة خيوط threads إلى نفس موقع الذاكرة في الوقت نفسه، ويكون أحدها على الأقل للكتابة، دون مزامنة synchronization.
قواعد الاستعارة تمنع الشكل الأساسي لسباق البيانات حتى قبل إدخال التوازي concurrency:

- تعدد القراء آمن (&T).
- الكاتب يجب أن يكون وحيداً (T &mut).

في الشيفرة المتوازية تستخدم Rust قيود السمات trait bounds مثل Send وSyncg، وتفرض استخدام أنواع مزامنة مثل Mutex. الفكرة الجوهرية: لا يوجد وصول قابل للتغيير مشترك دون مزامنة صريحة.
مثال عملي أحادي الخيط:

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

    let a = &v[0];       // immutable borrow of element
    // v.push(40);       // could reallocate and invalidate `a`
    println!("{}", a);
}
```

لو سمح push بإعادة التخصيص reallocation فقد يصبح a مرجعًا متدليًا. تمنع Rust ذلك برفض التعديل أثناء وجود استعارة غير قابلة للتغيير.

٥.٥ تصميم الشيفرة حول الاستعارة

الاستعارة ليست شيئًا ``تقاومه''، بل تبني تصميمك وفقه. أكثر الشيفرات موثوقية هي التي تكون فيها الاستعارات قصيرة وواضحة ومحلية.

1) فضّل الاستعارة في تواقيع الدوال
بدل أخذ الملكية دون حاجة:

```
fn print_uppercase(s: String) {
    println!("{}", s.to_uppercase());
}
```

استخدم الاستعارة:

```
fn print_uppercase(s: &str) {
    println!("{}", s.to_uppercase());
}
```

الآن يمكن للدالة قبول String والنصوص الحرفية معًا.
2) أرجع قيمًا مملوكة عند إنشاء بيانات جديدة

```
fn make_tag(name: &str) -> String {
    format!("[{}]", name)
}
```

3) استخدم المقاطع **Slices** لعرض جزئي
المقاطع تستعير جزءًا من مجموعة دون نسخ.

```
fn sum(slice: &[i32]) -> i32 {
    slice.iter().sum()
}

fn main() {
    let v = vec![1, 2, 3, 4, 5];
    println!("{}", sum(&v[1..4])); // 2 + 3 + 4
}
```

4) قلّل عمر المراجع القابلة للتغيير

```
fn main() {
    let mut s = String::from("Hello");

    {
        let w = &mut s;
        w.push_str(", Rust");
    } // mutable borrow ends here

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

5) أعد هيكلة تدفق البيانات عند الحاجة
بدل إبقاء المراجع حية مدة طويلة، احسب أولاً ثم عدّل.

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

    let last = *v.last().unwrap(); // borrow ends here
    v.push(last + 1);

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

في هذه المرحلة ينبغي أن تكون مرتاحًا مع:

- استخدام &T للوصول المشترك للقراءة.
- استخدام &mut T للوصول الحصري للكتابة.

- فهم قواعد الاستعارة الجوهرية ولماذا وُجدت.
- إدراك كيف تمنع الاستعارة المراجع المتدلية وسباقات البيانات.
- تنظيم الشيفرة بحيث تكون الاستعارات قصيرة وسهلة الفهم.

الفصل التالي (الأعمار Lifetimes) يرسّخ رسميًا كيفية بقاء المراجع صالحة عبر حدود الدوال ويساعدك على التعبير عن أنماط استعارة أكثر تقدمًا بأمان.

الفصل ٦: الأعمار Lifetimes بطريقة مفهومة

١.٦ الغرض من الأعمار

الأعمار Lifetimes هي آلية Rust للإثبات في وقت الترجمة compile time أن المراجع (&T، &mut T) لا تعيش أطول من البيانات التي تشير إليها. هي لا تغيّر سلوك التنفيذ runtime behavior، بل تُستخدم كأداة تحليل ساكن static analysis tool لمنع المراجع المتدلية dangling references.
يكون المرجع صالحًا فقط طالما أن القيمة المملوكة التي يشير إليها ما زالت حية:

```
fn main() {
    let r: &i32;

    {
        let x = 10;
        r = &x;
        // r is valid here
    }
    // r would be dangling here, so Rust rejects this pattern
}
```

تستخدم Rust الأعمار للإجابة عن السؤال: ``إلى متى يُضمن أن هذا المرجع صالح؟''

٢.٦ تصريحات الأعمار Lifetime Annotations

في كثير من الحالات يستطيع المترجم استنتاج الأعمار تلقائيًا. وعندما لا يستطيع، نقوم بتحديد علاقات الأعمار بين المراجع.
معامل العمر (مثل 'a) يعني: ``هذه المراجع مرتبطة ببعضها؛ صلاحيتها الزمنية مترابطة.''
مثال كلاسيكي: إرجاع أحد مرجعين مدخلين:

```
fn longer<'a>(x: &'a str, y: &'a str) -> &'a str {
    if x.len() >= y.len() { x } else { y }
}

fn main() {
    let a = String::from("abcd");
    let b = "xyz";
    let r = longer(&a, b);
    println!("{}", r);
}
```

ما الذي تعبّر عنه 'a؟

- المرجع المُعاد سيكون صالحًا لمدة لا تتجاوز أقصر العمرين للمدخلات.
- الدالة لا تُنشئ بيانات مستعارة جديدة؛ بل تُرجع استعارة مشتقة من أحد المدخلات.

مهم: الأعمار لا تعني ``يعيش إلى الأبد''؛ بل تصف قيوداً constraints.

٣.٦ قواعد حذف الأعمار Lifetime Elision Rules

تطبّق Rust قواعد حذف الأعمار lifetime elision بحيث يمكن حذف كثير من التصريحات في الأنماط الشائعة:

- كل مرجع مدخل غير مصرّح بعمره يحصل على عمر مستقل.
- إذا وُجد مرجع مدخل واحد فقط، يُنسب عمره إلى جميع المراجع المُعادة.
- إذا وُجدت عدة مراجع مدخل وأحدها هو &self أو &mut self، فإن عمر self يُنسب إلى المخرجات.

مرجع مدخل واحد ومرجع مخرج (يتم الاستنتاج تلقائيًا):

```
fn first_char(s: &str) -> &str {
    &s[0..1]
}
```

يعامله المترجم كما لو كان:

```
fn first_char<'a>(s: &'a str) -> &'a str {
    &s[0..1]
}
```

الطرائق **Methods** مع self:

```
struct Text {
    s: String,
}

impl Text {
    fn as_str(&self) -> &str {
        &self.s
    }
}
```

هنا المرجع المُعاد مرتبط تلقائيًا بعمر &self.
مدخلان ومخرج (لا يكفي الحذف، يجب التصريح):

```rust
fn pick(x: &str, y: &str) -> &str {
    x
    // error: cannot infer which input lifetime the output comes from
}
```

الحل:

```rust
fn pick<'a>(x: &'a str, _y: &'a str) -> &'a str {
    x
}
```

٤.٦ إرجاع المراجع بأمان

يمكن للدالة إرجاع مرجع فقط إذا كان يشير إلى بيانات تعيش أطول من استدعاء الدالة. إرجاع مرجع إلى متغير محلي خطأ دائمًا لأن المتغيرات المحلية تُسقط عند نهاية الدالة.
غير صالح:

```rust
fn bad() -> &str {
    let s = String::from("temp");
    &s[..] // error: returns reference to local `s`
}
```

أنماط صحيحة:
1) إرجاع قيمة مملوكة

```rust
fn good() -> String {
    let s = String::from("owned");
    s
}
```

2) إرجاع مرجع مشتق من مدخل

```
fn head<'a>(s: &'a str) -> &'a str {
    if s.len() >= 3 { &s[..3] } else { s }
}
```

3) تخزين بيانات مملوكة داخل بنية ثم إرجاع مراجع إليها

```
struct Store {
    data: String,
}

impl Store {
    fn new(data: String) -> Self {
        Self { data }
    }

    fn view(&self) -> &str {
        &self.data
    }
}

fn main() {
    let st = Store::new(String::from("persistent"));
    let v = st.view();
    println!("{}", v);
}
```

4) استخدام 'static فقط عندما تكون البيانات ثابتة فعلاً

النصوص الحرفية من النوع &'static str:

```rust
fn banner() -> &'static str {
    "Rust is strict for a reason."
}
```

لا تستخدم 'static لإسكات الأخطاء؛ غالبًا ما يشير ذلك إلى مشكلة تصميم.

٥.٦ أخطاء أعمار شائعة وحلولها

الخطأ 1: إرجاع مرجع إلى بيانات محلية

```rust
fn make_view() -> &str {
    let s = String::from("view");
    &s
}
```

الحل: إرجاع String أو أخذ مدخل وإرجاع مقطع منه.

```rust
fn make_owned() -> String {
    String::from("view")
}

fn view_of<'a>(s: &'a str) -> &'a str {
    s
}
```

الخطأ 2: لا يمكن استنتاج عمر المخرج من عدة مدخلات

```rust
fn choose(x: &str, y: &str) -> &str {
    if x.len() > y.len() { x } else { y }
}
```

الحل:

```
fn choose<'a>(x: &'a str, y: &'a str) -> &'a str {
    if x.len() > y.len() { x } else { y }
}
```

الخطأ 3: القيمة المستعارة لا تعيش مدة كافية
يحدث هذا عندما يعيش المرجع أطول من مالكه.

```
fn main() {
    let r: &str;
    {
        let s = String::from("short");
        r = &s;
    }
    // println!("{}", r); // error: `s` dropped here
}
```

الحل: مدّ عمر المالك (نقله إلى نطاق أوسع) أو إرجاع بيانات مملوكة.

```
fn main() {
    let s = String::from("long enough");
    let r = &s;
    println!("{}", r);
}
```

الخطأ 4: خلط المراجع القابلة وغير القابلة للتغيير لفترة طويلة
أحيانًا يكون الحل تقصير مدة الاستعارة:

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

```
    let len = s.len(); // borrow ends immediately
    s.push('!');       // allowed

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

إذا احتفظت بمرجع مدة طويلة جداً فسيمنعك المترجم من التعديل لاحقًا. الحل العملي غالبًا هو حساب القيم المطلوبة أولاً، ثم التعديل، أو تقليل نطاق الاستعارة.

تصبح الأعمار بسيطة عندما تتذكر:

- يمكنك إرجاع مراجع فقط إلى بيانات تعيش أطول من استدعاء الدالة.
- تصريحات الأعمار تصف علاقات بين المراجع؛ لا تقوم بتخصيص ذاكرة.
- عندما يطلب المترجم أعمارًا، فهو يطلب منك توضيح نموذج تدفق البيانات والملكية بشكل صريح.

الفصل ٧: السلاسل النصية ومعالجة النصوص

١.٧ &str مقابل String

تحتوي Rust على شكلين أساسيين للسلاسل النصية:

- &str: مقطع نصي مستعار غير قابل للتغيير immutable borrowed string slice، وهو عرض view لبايتات مشفّرة بصيغة UTF-8. يُستخدم غالبًا كمعامل للدوال.
- String: سلسلة نصية مملوكة قابلة للنمو owned, growable UTF-8 buffer مخزّنة على الكومة heap.

النوع String يملك ذاكرته، بينما &str يستعيرها.

```
fn takes_str(s: &str) {
    println!("str: {}", s);
}

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

fn main() {
```

```
    let owned = String::from("Hello");
    let slice: &str = &owned;   // &String coerces to &str

    takes_str(slice);
    takes_str(&owned);          // also works

    takes_string(owned);        // moves ownership
    // println!("{}", owned);    // error: moved
}
```

إرشاد أسلوبي Idiomatic guideline:

- اقبل &str في واجهات APIs ما لم تكن بحاجة فعلية لأخذ الملكية.
- أعد String عندما تُنشئ أو تُحوّل نصًا إلى قيمة مملوكة جديدة.

٢.٧ إنشاء السلاسل النصية وتعديلها

إنشاء String

```
fn main() {
    let a = String::new();
    let b = String::from("Rust");
    let c = "literal".to_string();

    println!("{} {} {}", a, b, c);
}
```

الإضافة والتعديل

```rust
fn main() {
    let mut s = String::from("Hello");
    s.push(' ');
    s.push_str("Rust");

    println!("{}", s); // "Hello Rust"
}
```

الربط Concatenation

استخدام العامل + ينقل المعامل الأيسر (String) ويستعير المعامل الأيمن (&str):

```rust
fn main() {
    let s1 = String::from("Hello");
    let s2 = String::from("Rust");

    let s3 = s1 + " " + &s2; // s1 moved, s2 borrowed
    // println!("{}", s1);   // error: moved
    println!("{}", s3);
}
```

غالبًا ما يكون استخدام format! أوضح ويتجنب مفاجآت الملكية:

```rust
fn main() {
    let a = "Hello";
    let b = "Rust";
    let s = format!("{} {}", a, b);
    println!("{}", s);
}
```

استبدال النص (يعيد String جديدًا)

```rust
fn main() {
    let s = "I like Rust. Rust is fast.";
    let t = s.replace("Rust", "Rustacean");
    println!("{}", t);
}
```

٣.٧ تقطيع السلاسل String Slicing

مقاطع &str هي مجالات ضمن بايتات UTF-8. يجب أن يكون التقطيع عند حدود محارف صحيحة valid UTF-8 character boundaries.

مثال صحيح مع ASCII:

```rust
fn main() {
    let s = String::from("abcdef");
    let part = &s[1..4];
    println!("{}", part); // "bcd"
}
```

مقاربة آمنة:

- استخدم chars() للعمليات المعتمدة على المحارف.
- استخدم split_whitespace() وsplit() وlines() لتجزئة النص.

مثال: أول كلمة باستخدام split_whitespace:

```rust
fn first_word(s: &str) -> Option<&str> {
    s.split_whitespace().next()
}
```

```rust
fn main() {
    let s = "Rust makes safety practical";
    println!("{:?}", first_word(s));
}
```

٤.٧ التنسيق والإدراج Formatting and Interpolation

تستخدم Rust وحدات ماكرو macros للتنسيق. الأهم:

- println! للطباعة إلى stdout
- format! لبناء String

الاستخدام الأساسي:

```rust
fn main() {
    let lang = "Rust";
    let year = 2015;

    println!("{} appeared in {}", lang, year);

    let msg = format!("Welcome to {}", lang);
    println!("{}", msg);
}
```

تنسيق تصحيحي باستخدام :?:

```rust
fn main() {
    let v = vec![1, 2, 3];
    println!("{:?}", v);
}
```

أمثلة على المحاذاة والحشو:

```
fn main() {
    let n = 42;
    println!("{:>6}", n);  // right aligned width 6
    println!("{:<6}", n);  // left aligned width 6
    println!("{:06}", n);  // zero-padded width 6
}
```

٥.٧ UTF-8 وحدود المحارف

السلاسل في Rust مشفّرة بصيغة UTF-8. وهذا يعني:

- الدالة len() تعيد عدد البايتات وليس عدد المحارف.
- الفهرسة المباشرة s[i] غير مسموحة لأنها غير آمنة وملتبسة.
- بعض المحارف تشغل عدة بايتات.

مثال يوضح الفرق بين عدد البايتات وعدد المحارف:

```
fn main() {
    let s1 = "abc";
    let s2 = "é";      // 2 bytes in UTF-8
    let s3 = " ";    // 4 bytes in UTF-8

    println!("s1 bytes = {}", s1.len());
    println!("s2 bytes = {}", s2.len());
    println!("s3 bytes = {}", s3.len());
```

```
    println!("s2 chars = {}", s2.chars().count());
    println!("s3 chars = {}", s3.chars().count());
}
```

محاولة التقطيع داخل محرف متعدد البايتات ستؤدي إلى panic أثناء التنفيذ:

```
fn main() {
    let s = "é";          // UTF-8: two bytes
    // let bad = &s[0..1]; // panic: not on a char boundary
    let good = &s[0..2];  // valid boundary
    println!("{}", good);
}
```

التعامل الآمن مع النصوص Unicode:

التكرار عبر المحارف:

```
fn main() {
    let s = "Rust  ";
    for ch in s.chars() {
        println!("{}", ch);
    }
}
```

التكرار عبر البايتات (مستوى منخفض):

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

إرشادات عملية:

- استخدم &str للاستعارة النصوص في الواجهات.
- استخدم String عندما تحتاج إلى ملكية أو تعديل.
- تجنب التقطيع باستخدام فهارس رقمية ما لم تكن متأكداً من حدود UTF-8.
- فضّل المعالجة القائمة على المكررات iterators لضمان الصحة والوضوح.

الفصل ٨: المجموعات والتكرار Collections and Iteration

١.٨ المتجهات Vec<T>

النوع Vec<T> هو مصفوفة متجاورة قابلة للنمو growable, contiguous array مخزنة على الكومة heap. وهو أكثر أنواع المجموعات استخدامًا للأغراض العامة.

إنشاء المتجهات

```
fn main() {
    let v1: Vec<i32> = Vec::new();
    let v2 = vec![1, 2, 3];
    let v3 = vec![0; 5]; // five zeros

    println!("{:?} {:?} {:?}", v1, v2, v3);
}
```

الإضافة والحذف

```
fn main() {
    let mut v = Vec::new();
```

```
    v.push(10);
    v.push(20);

    let last = v.pop();
    println!("v = {:?}, popped = {:?}", v, last);
}
```

الفهرسة مقابل الوصول الآمن

الفهرسة قد تؤدي إلى panic إذا كان الفهرس خارج الحدود:

```
fn main() {
    let v = vec![10, 20, 30];
    println!("{}", v[1]); // ok
    // println!("{}", v[99]); // panic at runtime
}
```

يُفضَّل استخدام get للوصول الآمن:

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

    match v.get(99) {
        Some(x) => println!("value = {}", x),
        None => println!("out of bounds"),
    }
}
```

تحديث العناصر

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

```
    v[0] = 10;
    println!("{:?}", v);
}
```

تخزين أنواع متعددة

يحمل Vec<T> نوعًا واحدًا فقط T. لتخزين قيم ``مختلطة'' استخدم enum:

```
#[derive(Debug)]
enum Value {
    Int(i32),
    Text(String),
}

fn main() {
    let items = vec![
        Value::Int(7),
        Value::Text("Rust".to_string()),
    ];

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

٢.٨ شرح المكررات Iterators

يعتمد نموذج التكرار في Rust على السمة Iterator. توفر معظم المجموعات:

- iter() لمراجع غير قابلة للتغيير (&T)
- iter_mut() لمراجع قابلة للتغيير (&mut T)

- into_iter() لنقل القيم (نقل الملكية)

تكرار غير قابل للتغيير

```
fn main() {
    let v = vec![1, 2, 3];
    for x in v.iter() {
        println!("{}", x);
    }
    println!("{:?}", v); // still usable
}
```

تكرار قابل للتغيير

```
fn main() {
    let mut v = vec![1, 2, 3];
    for x in v.iter_mut() {
        *x *= 10;
    }
    println!("{:?}", v);
}
```

تكرار بالنقل

```
fn main() {
    let v = vec![String::from("a"), String::from("b")];

    for s in v.into_iter() {
        println!("{}", s);
    }

    // println!("{:?}", v); // error: moved
}
```

محوّلات المكررات Iterator Adapters

المكررات كسولة lazy. دوال مثل map وfilter وtake تبني سلسلة عمليات. لإنتاج مجموعة استخدم collect.

```
fn main() {
    let squares: Vec<i32> = (1..=10)
        .map(|x| x * x)
        .collect();

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

تصفية وتجميع:

```
fn main() {
    let evens: Vec<i32> = (1..=20)
        .filter(|x| x % 2 == 0)
        .collect();

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

التجميع Reducing

```
fn main() {
    let sum: i32 = (1..=5).sum();
    let product: i32 = (1..=5).product();

    println!("sum={}, product={}", sum, product);
}
```

٣.٨ HashMap

النوع HashMap<K, V> يخزن أزواج مفتاح-قيمة. يجب أن يكون المفتاح قابلاً للتجزئة hashable والمقارنة.

الإنشاء والإدراج

```
use std::collections::HashMap;

fn main() {
    let mut scores = HashMap::new();

    scores.insert(String::from("Alice"), 10);
    scores.insert(String::from("Bob"), 20);

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

البحث

```
use std::collections::HashMap;

fn main() {
    let mut scores = HashMap::new();
    scores.insert("Alice".to_string(), 10);

    match scores.get("Alice") {
        Some(v) => println!("Alice = {}", v),
        None => println!("Not found"),
    }
}
```

التكرار

```rust
use std::collections::HashMap;

fn main() {
    let mut scores = HashMap::new();
    scores.insert("Alice".to_string(), 10);
    scores.insert("Bob".to_string(), 20);

    for (name, score) in scores.iter() {
        println!("{} -> {}", name, score);
    }
}
```

التحديث باستخدام entry

```rust
use std::collections::HashMap;

fn main() {
    let mut counts = HashMap::new();

    let word = "rust";
    *counts.entry(word).or_insert(0) += 1;
    *counts.entry(word).or_insert(0) += 1;

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

عدّ الكلمات

```rust
use std::collections::HashMap;
```

```
fn main() {
    let text = "rust is fast and rust is safe";
    let mut freq: HashMap<&str, usize> = HashMap::new();

    for w in text.split_whitespace() {
        *freq.entry(w).or_insert(0) += 1;
    }

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

٤.٨ HashSet

النوع HashSet<T> يخزن قيماً فريدة (كمجموعة رياضية). مفيد للاختبار الانتماء وإزالة التكرار.

```
use std::collections::HashSet;

fn main() {
    let mut set = HashSet::new();

    set.insert("rust");
    set.insert("rust"); // duplicate ignored
    set.insert("safe");

    println!("contains rust? {}", set.contains("rust"));
    println!("{:?}", set);
}
```

عمليات المجموعات

```
use std::collections::HashSet;

fn main() {
    let a: HashSet<i32> = [1, 2, 3].into_iter().collect();
    let b: HashSet<i32> = [3, 4, 5].into_iter().collect();

    let union: HashSet<i32> = a.union(&b).copied().collect();
    let inter: HashSet<i32> = a.intersection(&b).copied().collect();
    let diff: HashSet<i32> = a.difference(&b).copied().collect();

    println!("union = {:?}", union);
    println!("inter = {:?}", inter);
    println!("diff  = {:?}", diff);
}
```

٥.٨ الملكية داخل المجموعات

تنطبق قواعد الملكية أيضًا داخل المجموعات:

- إدخال قيمة مملوكة إلى مجموعة يؤدي عادة إلى نقلها move.
- استعارة عنصر من مجموعة تعيد مرجعًا بعمر مرتبط باستعارة المجموعة.
- تعديل مجموعة قد يُبطل مراجع عناصرها (مثل إعادة تخصيص المتجه)، لذلك تقيّد Rust التعديل أثناء وجود استعارات.

نقل القيم إلى متجه

```rust
fn main() {
    let s = String::from("moved");
    let v = vec![s];
    // println!("{}", s); // error: moved into v
    println!("{:?}", v);
}
```

استعارة العناصر بأمان

```rust
fn main() {
    let v = vec![10, 20, 30];
    let r = &v[0];
    println!("{}", r);
    println!("{:?}", v);
}
```

لماذا يُقيَّد التعديل أثناء الاستعارة

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

    let first = &v[0];
    // v.push(4); // error: cannot borrow `v` as mutable because it is also
    ↪  borrowed as immutable
    println!("{}", first);
}
```

إذا أعاد push تخصيص الذاكرة reallocate فقد يصبح first مرجعًا متدليًا. تمنع Rust ذلك برفض الشيفرة.

نمط آمن: احسب ثم عدّل

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

    let first_value = v[0]; // Copy value out
    v.push(first_value + 10);

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

التكرار والتجميع دون مفاجآت ملكية

```rust
fn main() {
    let names = vec![String::from("A"), String::from("B")];

    let lens: Vec<usize> = names.iter().map(|s| s.len()).collect();
    println!("lens = {:?}", lens);

    println!("names still owned: {:?}", names);
}
```

في هذه المرحلة ينبغي أن تكون قادرًا على:

- استخدام Vec<T> كمصفوفة ديناميكية وفهم الوصول الآمن للعناصر.
- اختيار iter وiter_mut وinto_iter بشكل صحيح.
- استخدام سلاسل المكررات iterator pipelines مثل map وfilter وcollect وsum.
- إنشاء وتحديث HashMap باستخدام entry.
- استخدام HashSet لضمان التفرد وتنفيذ عمليات المجموعات.
- التنبؤ بمواضع نقل الملكية داخل المجموعات وفهم متى تقيّد قواعد الاستعارة التعديل.

الفصل ٩: الهياكل Structs والتعدادات Enums ومطابقة الأنماط Pattern Matching

١.٩ تعريف الهياكل Structs

تُستخدم الهياكل Structs لتعريف أنواع بيانات مخصصة عبر تجميع حقول مترابطة. تدعم Rust ثلاثة أشكال رئيسية:

- هياكل ذات حقول مسمّاة Named-field structs (الأكثر شيوعًا)
- هياكل على شكل صفّ Tuple structs (بدون أسماء للحقول)
- هياكل شبيهة بالوحدة Unit-like structs (بدون حقول)

هيكل ذو حقول مسمّاة

```
#[derive(Debug)]
struct User {
    id: u64,
    name: String,
    active: bool,
}
```

```
fn main() {
    let u = User {
        id: 1,
        name: String::from("Ayman"),
        active: true,
    };

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

هيكل صفّي

```
#[derive(Debug)]
struct Point(i32, i32);

fn main() {
    let p = Point(10, 20);
    println!("{:?} x={}, y={}", p, p.0, p.1);
}
```

هيكل شبيه بالوحدة

```
struct Marker;

fn main() {
    let _m = Marker;
}
```

صيغة تحديث الهيكل (تنقل الحقول غير Copy):

```rust
#[derive(Debug)]
struct User {
    id: u64,
    name: String,
    active: bool,
}

fn main() {
    let u1 = User { id: 1, name: "Alice".to_string(), active: true };

    let u2 = User {
        id: 2,
        name: "Bob".to_string(),
        ..u1
    };

    // println!("{:?}", u1); // error: u1.name moved into u2
    println!("{:?}", u2);
}
```

٢.٩ تنفيذ الطرائق Implementing Methods

تُعرّف الطرائق داخل كتلة impl وترتبط بنوع معيّن.
دالة مرتبطة (أسلوب المُنشئ)

```rust
#[derive(Debug)]
struct User {
    id: u64,
```

```
    name: String,
    active: bool,
}

impl User {
    fn new(id: u64, name: impl Into<String>) -> Self {
        Self {
            id,
            name: name.into(),
            active: true,
        }
    }
}

fn main() {
    let u = User::new(10, "Rustacean");
    println!("{:?}", u);
}
```

طرائق باستخدام &self و&mut self

```
impl User {
    fn deactivate(&mut self) {
        self.active = false;
    }

    fn is_active(&self) -> bool {
        self.active
    }
}
```

```
fn main() {
    let mut u = User::new(1, "Alice");
    println!("active? {}", u.is_active());

    u.deactivate();
    println!("active? {}", u.is_active());
}
```

الملكية داخل الطرائق

```
impl User {
    fn into_name(self) -> String {
        self.name // moves name out; consumes self
    }
}

fn main() {
    let u = User::new(1, "Alice");
    let name = u.into_name();
    println!("{}", name);
    // println!("{:?}", u); // error: u moved
}
```

اختيار &self أو &mut self أو self يعبّر بدقة عن عقد الملكية ownership contract للواجهة.

٣.٩ التعدادات Enums ومطابقة الأنماط القوية

يمثل enum قيمة يمكن أن تكون أحد عدة بدائل variants. وهو عنصر أساسي في تصميم Rust الآمن والتعبيري.

تعداد بسيط

```rust
#[derive(Debug)]
enum Direction {
    North,
    South,
    East,
    West,
}

fn main() {
    let d = Direction::East;

    match d {
        Direction::North => println!("N"),
        Direction::South => println!("S"),
        Direction::East  => println!("E"),
        Direction::West  => println!("W"),
    }
}
```

تعداد مع بيانات (أنواع مجموعية **sum types**)

```rust
#[derive(Debug)]
enum Message {
    Quit,
    Move { x: i32, y: i32 },
    Write(String),
    ChangeColor(u8, u8, u8),
}
```

```
fn main() {
    let msg = Message::Move { x: 10, y: 20 };

    match msg {
        Message::Quit => println!("quit"),
        Message::Move { x, y } => println!("move to ({}, {})", x, y),
        Message::Write(text) => println!("text: {}", text),
        Message::ChangeColor(r, g, b) => println!("rgb({}, {}, {})", r, g,
        ↪  b),
    }
}
```

ميزات مطابقة الأنماط

- أنماط بديلة باستخدام |
- حراس guards باستخدام if
- فكّ البنية destructuring للهياكل والصفوف والتعدادات

```
fn main() {
    let x = 7;

    match x {
        1 | 2 => println!("one or two"),
        3..=9 if x % 2 == 1 => println!("odd between 3 and 9"),
        _ => println!("other"),
    }
}
```

let whileg let if

تُستخدم لمطابقة نمط واحد بشكل مختصر.

```
fn main() {
    let msg = Message::Write("hello".to_string());

    if let Message::Write(text) = msg {
        println!("written: {}", text);
    }
}
```

٤.٩ Option وResult بعمق

تتجنب Rust القيمة الفارغة null وأكواد الأخطاء غير المفحوصة. بدلاً من ذلك تستخدم:

- Option<T> لتمثيل ``قد توجد قيمة''
- Result<T, E> لتمثيل ``نجاح أو خطأ''

Option<T>

```
fn find_first_even(v: &[i32]) -> Option<i32> {
    for &x in v {
        if x % 2 == 0 {
            return Some(x);
        }
    }
    None
}
```

```
fn main() {
    let v = [1, 3, 5, 8, 9];

    match find_first_even(&v) {
        Some(x) => println!("found: {}", x),
        None => println!("no even number"),
    }
}
```

دوال مساعدة شائعة:

```
fn main() {
    let maybe = Some(10);

    let v1 = maybe.unwrap_or(0);
    let v2 = maybe.map(|x| x * 2);

    println!("v1={}, v2={:?}", v1, v2);
}
```

E> Result<T,

```
use std::fs;

fn read_file(path: &str) -> Result<String, std::io::Error> {
    fs::read_to_string(path)
}
```

```rust
fn main() {
    match read_file("data.txt") {
        Ok(text) => println!("len={}", text.len()),
        Err(e) => println!("error: {}", e),
    }
}
```

تمرير الخطأ باستخدام ?:

```rust
use std::fs;

fn read_and_trim(path: &str) -> Result<String, std::io::Error> {
    let text = fs::read_to_string(path)?;
    Ok(text.trim().to_string())
}
```

تحويل النتائج:

```rust
fn main() {
    let r: Result<i32, &str> = Ok(21);

    let doubled = r.map(|x| x * 2);
    println!("{:?}", doubled);
}
```

إرشاد تصميمي:

- استخدم Option عندما يكون غياب القيمة أمرًا طبيعيًا.
- استخدم Result عندما يكون الفشل ذا معنى ويجب حمل معلومات الخطأ.

٥.٩ تصميم نماذج بيانات نظيفة

يُبنى نموذج البيانات النظيف في Rust بدمج الهياكل والتعدادات لتمثيل القواعد داخل نظام الأنواع.
1) استخدم التعداد لتمثيل الحالات بدلاً من bool
بدلاً من:

```
struct Order {
    paid: bool,
}
```

يفضّل:

```
#[derive(Debug)]
enum PaymentStatus {
    Unpaid,
    Paid { transaction_id: String },
    Refunded,
}

#[derive(Debug)]
struct Order {
    id: u64,
    status: PaymentStatus,
}
```

الآن الحالات غير الصحيحة غير قابلة للتمثيل.
2) نمذجة الحقول الاختيارية باستخدام Option

```
#[derive(Debug)]
struct Profile {
    name: String,
```

```
    email: Option<String>,
}

fn main() {
    let p = Profile {
        name: "Alice".to_string(),
        email: None,
    };

    let email = p.email.as_deref().unwrap_or("<no email>");
    println!("{} {}", p.name, email);
}
```

3) استخدام Result عند الحدود

حوّل العمليات القابلة للفشل إلى Result مبكرًا، وتعامل معها عند حدود البرنامج مثل main.

```
use std::fs;

fn load_config(path: &str) -> Result<String, std::io::Error> {
    fs::read_to_string(path)
}

fn main() {
    let config = match load_config("config.txt") {
        Ok(c) => c,
        Err(e) => {
            println!("config error: {}", e);
            return;
        }
    };
```

```
    println!("config loaded: {} bytes", config.len());
}
```

4) الاستعارة في الواجهات، والملكية في التخزين

```
#[derive(Debug)]
struct User {
    name: String,
}

impl User {
    fn new(name: impl Into<String>) -> Self {
        Self { name: name.into() }
    }

    fn name(&self) -> &str {
        &self.name
    }
}

fn main() {
    let u = User::new("Rust");
    println!("{}", u.name());
}
```

في هذه المرحلة ينبغي أن تكون مرتاحًا مع:

- استخدام الهياكل لنمذجة سجلات بملكية واضحة.
- استخدام كتل impl لتعريف مُنشئات وطرائق بعقود استعارة دقيقة.

- استخدام التعدادات لتمثيل البدائل والحالات الصحيحة.
- استخدام match وlet if وحراس الأنماط للتفرع الآمن.
- تصميم نماذج بيانات تجعل الحالات غير الصحيحة غير قابلة للتمثيل باستخدام Option والتعدادات.
- تمثيل الغياب والفشل بشكل صريح باستخدام Option وResult.

الفصل ١٠: معالجة الأخطاء بأسلوب Rust

١.١٠ panic! مقابل Result

تميّز Rust بين نوعين من الإخفاقات:

- إخفاقات غير قابلة للاسترجاع تُعالج باستخدام panic! (يتوقف البرنامج أو يتم فك المكدس unwind).
- إخفاقات قابلة للاسترجاع تُمثَّل باستخدام Result<T, E>.

متى نستخدم panic!

استخدم panic! في الحالات التي تمثل أخطاء برمجية bugs، أو خرقًا لثوابت invariants، أو حالات ``لا ينبغي أن تحدث أبدًا''.

```
fn divide(a: i32, b: i32) -> i32 {
    if b == 0 {
        panic!("division by zero");
    }
    a / b
}
```

```rust
fn main() {
    println!("{}", divide(10, 2));
    // println!("{}", divide(10, 0)); // panic
}
```

متى نستخدم Result

استخدم Result في حالات الفشل المتوقعة مثل غياب ملف، إدخال غير صالح، مهلة شبكة timeout، وغيرها.

```rust
use std::fs;

fn read_text(path: &str) -> Result<String, std::io::Error> {
    fs::read_to_string(path)
}

fn main() {
    match read_text("data.txt") {
        Ok(text) => println!("len = {}", text.len()),
        Err(e) => println!("error: {}", e),
    }
}
```

إرشاد عام

- استخدم panic! للأخطاء البرمجية والحالات المستحيلة.
- استخدم Result للأخطاء القابلة للاسترجاع أو المعتمدة على البيئة أو إدخال المستخدم.

٢.١٠ المعامل ?

المعامل ? هو الطريقة الأسلوبية idiomatic للتعامل مع Result (وكذلك Option) عبر الإرجاع المبكر عند الفشل.
إذا كانت القيمة Ok(v) فإن ? تعيد v. وإذا كانت Err(e) فإن ? تُرجع Err(e) من الدالة الحالية.

```
use std::fs;

fn load(path: &str) -> Result<String, std::io::Error> {
    let text = fs::read_to_string(path)?; // early return on Err
    Ok(text)
}
```

مع عدة خطوات:

```
use std::fs;

fn load_trimmed(path: &str) -> Result<String, std::io::Error> {
    let text = fs::read_to_string(path)?;
    Ok(text.trim().to_string())
}
```

يسمح ? بكتابة شيفرة خطية واضحة مع الحفاظ على تمرير الأخطاء بدقة.

٣.١٠ تمرير الأخطاء Propagating Errors

تمرير الأخطاء يعني إرجاعها إلى المستدعي ومعالجتها عند حدود البرنامج (غالبًا في main).
مثال: تمرير من دوال مساعدة

```
use std::fs;
use std::io;

fn read_config(path: &str) -> Result<String, io::Error> {
    fs::read_to_string(path)
}

fn parse_port(text: &str) -> Result<u16, std::num::ParseIntError> {
    text.trim().parse::<u16>()
}

fn main() {
    let config = match read_config("config.txt") {
        Ok(c) => c,
        Err(e) => {
            eprintln!("config read error: {}", e);
            return;
        }
    };

    let port = match parse_port(&config) {
        Ok(p) => p,
        Err(e) => {
            eprintln!("port parse error: {}", e);
            return;
        }
    };

    println!("port = {}", port);
```

```
}
```

إرجاع Result من main

تسمح Rust بأن تُرجع main قيمة من نوع Result، مما يبسّط نقطة دخول البرنامج.

```
use std::fs;

fn main() -> Result<(), std::io::Error> {
    let text = fs::read_to_string("data.txt")?;
    println!("bytes = {}", text.len());
    Ok(())
}
```

٤.١٠ إنشاء أخطاء مخصصة

مع نمو البرنامج، تحتاج غالبًا إلى أخطاء خاصة بمجال التطبيق بدلاً من أخطاء I/O أو التحليل parse الخام. النهج الشائع هو تعريف enum يمثل فئات الأخطاء في التطبيق.

تعداد أخطاء مخصص مع تحويلات

```
use std::fmt;

#[derive(Debug)]
enum AppError {
    Io(std::io::Error),
    Parse(std::num::ParseIntError),
    EmptyInput,
}

impl fmt::Display for AppError {
```

```
    fn fmt(&self, f: &mut fmt::Formatter<'_>) -> fmt::Result {
        match self {
            AppError::Io(e) => write!(f, "io error: {}", e),
            AppError::Parse(e) => write!(f, "parse error: {}", e),
            AppError::EmptyInput => write!(f, "empty input"),
        }
    }
}

impl From<std::io::Error> for AppError {
    fn from(e: std::io::Error) -> Self {
        AppError::Io(e)
    }
}

impl From<std::num::ParseIntError> for AppError {
    fn from(e: std::num::ParseIntError) -> Self {
        AppError::Parse(e)
    }
}
```

الآن يمكن تحويل أي std::io::Error أو ParseIntError تلقائيًا باستخدام ?.

```
fn parse_port(text: &str) -> Result<u16, AppError> {
    let t = text.trim();
    if t.is_empty() {
        return Err(AppError::EmptyInput);
    }
    let p = t.parse::<u16>()?;
    Ok(p)
```

```
}
```

٥.١٠ مثال عملي لمعالجة الملفات

مثال واقعي يجمع بين الإدخال/الإخراج I/O، والتحليل، ومعالجة الأخطاء المخصصة.
الهدف:

- قراءة ملف يحتوي على رقم منفذ port.
- التحقق من صحته.
- إرجاع خطأ تطبيقي مخصص.

```
use std::fs;

fn read_port_from_file(path: &str) -> Result<u16, AppError> {
    let text = fs::read_to_string(path)?;   // Io -> AppError via From
    let port = parse_port(&text)?;          // Parse/EmptyInput -> AppError
    if port == 0 {
        return Err(AppError::EmptyInput);
    }
    Ok(port)
}

fn main() {
    match read_port_from_file("port.txt") {
        Ok(port) => println!("port = {}", port),
        Err(e) => eprintln!("error: {}", e),
    }
}
```

ما الذي يوضحه هذا المثال

- الإخفاقات قيم صريحة (Result) وليست تدفقًا مخفيًا للتحكم.
- المعامل ? يمرّر الأخطاء بشكل نظيف ومتوقع.
- التعدادات المخصصة تمثل أخطاء المجال بوضوح.
- تُدفع معالجة الأخطاء إلى حدود البرنامج (main) حيث يجب عرض الرسائل للمستخدم.

هذا الأسلوب قابل للتوسّع: مع نمو البرنامج، تضيف بدائل جديدة للأخطاء وتحافظ على تمرير متّسق وواضح.

الفصل ١١: الوحدات Modules والحِزم Crates وتنظيم المشروع

١.١١ الكلمة المفتاحية mod

الحِزمة crate في Rust هي وحدة ترجمة compilation unit (إما برنامج تنفيذي binary أو مكتبة library). داخل الحِزمة يُنظَّم الكود باستخدام الوحدات modules.
الوحدة هي حدّ لنطاق الأسماء namespace boundary تُجمّع العناصر (الدوال، الهياكل، التعدادات، السمات) وتتحكم في الرؤية visibility.
تُستخدم mod لتعريف وحدة. يمكن أن يكون جسم الوحدة داخل الملف نفسه أو محمّلاً من ملف منفصل.
وحدة داخلية **Inline Module**

```
mod math {
    pub fn add(a: i32, b: i32) -> i32 {
        a + b
    }
}

fn main() {
```

```
    println!("{}", math::add(2, 3));
}
```

وحدة من ملف

بنية شائعة (حِزمة تنفيذية):

```
src/
  main.rs
  math.rs
```

```
// src/main.rs
mod math;

fn main() {
    println!("{}", math::add(2, 3));
}
```

```
// src/math.rs
pub fn add(a: i32, b: i32) -> i32 {
    a + b
}
```

تدعم Rust أيضًا وحدات مبنية على مجلدات:

```
src/
  main.rs
  net/
    mod.rs
    client.rs
```

```rust
// src/main.rs
mod net;

fn main() {
    net::client::ping();
}
```

```rust
// src/net/mod.rs
pub mod client;
```

```rust
// src/net/client.rs
pub fn ping() {
    println!("ping");
}
```

٣.١١ الرؤية باستخدام pub

العناصر خاصة private افتراضيًا، مما يشجع على الإحكام encapsulation وتصميم واجهات متعمد.
خاص افتراضيًا

```rust
mod secret {
    fn internal() {}
}
```

لجعل عنصر مرئيًا خارج وحدته، نضع pub:

```rust
mod api {
    pub fn visible() {}
    fn hidden() {}
}
```

رؤية حقول الهيكل

تعريف الهيكل كـ pub لا يجعل حقوله عامة تلقائيًا؛ يجب تحديد كل حقل بـ pub إذا لزم.

```
pub struct User {
    pub name: String,
    age: u32, // private
}

impl User {
    pub fn new(name: impl Into<String>, age: u32) -> Self {
        Self { name: name.into(), age }
    }

    pub fn age(&self) -> u32 {
        self.age
    }
}
```

إعادة التصدير **Re-exporting**

تُستخدم pub use لإعادة تصدير عنصر وصياغة واجهة عامة نظيفة.

```
mod inner {
    pub struct Engine;
}

pub use inner::Engine;
```

٣.١١ use ونطاقات الأسماء

تُستخدم المسارات paths للإشارة إلى العناصر:

- مسارات مطلقة تبدأ من جذر الحِزمة: `crate::...`
- مسارات نسبية من الوحدة الحالية: `self::...` و`super::...`

الاستيراد باستخدام `use`

```
mod math {
    pub fn add(a: i32, b: i32) -> i32 { a + b }
}

use math::add;

fn main() {
    println!("{}", add(1, 2));
}
```

استيراد عناصر من المكتبة القياسية

```
use std::collections::HashMap;

fn main() {
    let mut m: HashMap<&str, i32> = HashMap::new();
    m.insert("a", 1);
    println!("{:?}", m);
}
```

تسمية بديلة لتجنب التعارض

```
use std::io::Result as IoResult;

fn read() -> IoResult<()> {
    Ok(())
}
```

استيراد عدة أسماء

```
use std::cmp::{min, max};

fn main() {
    println!("{} {}", min(3, 7), max(3, 7));
}
```

استخدام crate:: و self:: و super::

```
mod outer {
    pub fn a() {}

    pub mod inner {
        pub fn b() {
            super::a(); // access parent module item
        }
    }
}
```

٤.١١ تنظيم المشاريع متعددة الملفات

نهج عملي هو فصل:

- أنواع المجال domain types (هياكل/تعدادات)
- المنطق الأساسي core logic
- حدود الإدخال/الإخراج I/O boundaries

بنية نظيفة لحِزمة تنفيذية:

```
src/
  main.rs
  lib.rs
  domain.rs
  logic.rs
  io.rs
```

استخدم lib.rs لاستضافة أغلب الكود واجعل main.rs خفيفًا

```rust
// src/main.rs
fn main() {
    if let Err(e) = myapp::run() {
        eprintln!("error: {}", e);
    }
}
```

```rust
// src/lib.rs
mod domain;
mod io;
mod logic;

pub fn run() -> Result<(), String> {
    let input = io::read_input().map_err(|e| e.to_string())?;
    let out = logic::process(&input);
    io::write_output(&out).map_err(|e| e.to_string())?;
    Ok(())
}
```

```rust
// src/io.rs
pub fn read_input() -> std::io::Result<String> {
    Ok(String::from("data"))
}

pub fn write_output(_s: &str) -> std::io::Result<()> {
    Ok(())
}
```

```rust
// src/logic.rs
pub fn process(s: &str) -> String {
    s.trim().to_uppercase()
}
```

هذا الأسلوب قابل للتوسّع ويدعم الاختبار وإعادة الاستخدام.
وحدات مجلدية لمكونات أكبر

```
src/
  lib.rs
  net/
    mod.rs
    client.rs
    protocol.rs
```

```rust
// src/net/mod.rs
pub mod client;
mod protocol; // internal details
```

٥.١١ ممارسات بنية مشروع نظيفة

1) اجعل الواجهة العامة صغيرة ومقصودة
اكشف فقط ما يحتاجه المستخدم. ابدأ بالخصوصية ثم أعد تصدير السطح النظيف.

```
mod internal {
    pub struct Engine;
    impl Engine {
        pub fn new() -> Self { Self }
    }
}

pub use internal::Engine; // public API surface
```

2) فضّل معاملات &str وإرجاع قيم مملوكة في واجهات النصوص

```
pub fn normalize(s: &str) -> String {
    s.trim().to_lowercase()
}
```

3) ضع العمليات القابلة للفشل عند الحواف وابقِ المنطق الأساسي نقيًا

```
pub fn compute(xs: &[i32]) -> i32 {
    xs.iter().sum()
}
```

4) افصل بين التنفيذ والمكتبة
حتى لو كان المشروع تنفيذيًا فقط، فإن بنية ``مكتبة + main خفيف'' تحسّن الاختبار والتنظيم.
5) تجنّب الاعتماد الدائري بين الوحدات
إذا وُجد اعتماد متبادل، فكّر في:

- استخراج الأنواع المشتركة إلى وحدة مثل domain

- إدخال وحدة أعلى تنسّق الاستدعاءات
- تقليل الرؤية وتمرير البيانات عبر حدود الدوال

6) سمِّ الوحدات حسب المسؤولية
يفضَّل:

- domain، model، types
- parse، format، io
- logic، service، engine

بدل أسماء عامة مثل utils ما لم تكن بسيطة جدًا.

في هذه المرحلة ينبغي أن تكون قادرًا على:

- تعريف الوحدات باستخدام mod وتحميلها من ملفات.
- التحكم في السطح العام باستخدام pub وpub use.
- التنقل بين النطاقات باستخدام use و::crate و::self و::super.
- تنظيم المشروع إلى طبقات واضحة: المجال، المنطق، وحدود الإدخال/الإخراج.
- الحفاظ على بنية نظيفة قابلة للتوسّع في برامج Rust الواقعية.

الفصل ١٢: بناء مشروع مصغّر متكامل

١.١٢ متطلبات المشروع

في هذا الفصل نبني أداة سطر أوامر صغيرة ولكن واقعية تُظهر سير العمل الكامل في Rust وتجمع أهم المفاهيم التي تم تعلمها في الفصول السابقة.
المشروع: rwc (Rust Word Count)
الأداة تقرأ ملف نصي UTF-8 وتطبع:

- عدد الأسطر
- عدد الكلمات (مقسومة حسب الفراغات)
- عدد البايتات

كما تدعم أعلاماً اختيارية لتحديد العدّادات المطلوبة، بأسلوب مشابه للأداة الكلاسيكية wc.
السلوك

- الوضع الافتراضي: طباعة الأسطر والكلمات والبايتات ثم اسم الملف.
- عند تمرير أعلام: طباعة العدّادات المحددة فقط.
- إذا لم يوجد الملف أو تعذّر قراءته: عرض رسالة خطأ واضحة والخروج.

- إذا كان الإدخال غير صحيح: عرض رسالة استخدام بسيطة.

أمثلة تشغيل

```
$ rwc data.txt
12 98 623 data.txt

$ rwc -w data.txt
98 data.txt

$ rwc -l -b data.txt
12 623 data.txt
```

٢.١٢ تصميم نموذج البيانات

النموذج النظيف يجعل الشيفرة قابلة للاختبار وسهلة التوسعة.
نقوم بتمثيل:

- خيارات المستخدم (الأعلام + المسار)
- الإحصائيات المحسوبة (أسطر، كلمات، بايتات)
- أخطاء التطبيق

الخيارات والإحصائيات

```
#[derive(Debug, Clone, Copy)]
struct Options {
    show_lines: bool,
    show_words: bool,
```

```
    show_bytes: bool,
}

#[derive(Debug, Default)]
struct Stats {
    lines: usize,
    words: usize,
    bytes: usize,
}
```

خطأ التطبيق

```
use std::fmt;

#[derive(Debug)]
enum AppError {
    Usage(String),
    Io(std::io::Error),
}

impl fmt::Display for AppError {
    fn fmt(&self, f: &mut fmt::Formatter<'_>) -> fmt::Result {
        match self {
            AppError::Usage(msg) => write!(f, "{}", msg),
            AppError::Io(e) => write!(f, "io error: {}", e),
        }
    }
}

impl From<std::io::Error> for AppError {
```

```
    fn from(e: std::io::Error) -> Self {
        AppError::Io(e)
    }
}
```

٣.١٢ تنفيذ المنطق الأساسي

نفصل بين:

- تحليل الوسائط إلى Options ومسار ملف
- قراءة الملف
- حساب Stats
- تنسيق الإخراج

تحليل الوسائط

```
fn parse_args(args: &[String]) -> Result<(Options, String), AppError> {
    if args.len() < 2 {
        return Err(AppError::Usage(usage()));
    }

    let mut opt = Options {
        show_lines: false,
        show_words: false,
        show_bytes: false,
    };
```

```
    let mut path: Option<String> = None;

    for a in &args[1..] {
        match a.as_str() {
            "-l" => opt.show_lines = true,
            "-w" => opt.show_words = true,
            "-b" => opt.show_bytes = true,
            _ if a.starts_with('-') => {
                return Err(AppError::Usage(
                    format!("Unknown flag: {}\n\n{}", a, usage())
                ));
            }
            _ => {
                if path.is_some() {
                    return Err(AppError::Usage(
                        format!("Too many inputs.\n\n{}", usage())
                    ));
                }
                path = Some(a.clone());
            }
        }
    }

    let path = path.ok_or_else(|| AppError::Usage(usage()))?;

    if !(opt.show_lines || opt.show_words || opt.show_bytes) {
        opt.show_lines = true;
        opt.show_words = true;
        opt.show_bytes = true;
```

```
    }

    Ok((opt, path))
}

fn usage() -> String {
    let msg =
"Usage:
  rwc [FLAGS] <file>

Flags:
  -l    print lines
  -w    print words
  -b    print bytes

Examples:
  rwc data.txt
  rwc -w data.txt
  rwc -l -b data.txt
";
    msg.to_string()
}
```

حساب الإحصائيات

```
fn compute_stats(text: &str) -> Stats {
    Stats {
        lines: text.lines().count(),
        words: text.split_whitespace().count(),
        bytes: text.as_bytes().len(),
```

```
    }
}
```

تنسيق الإخراج

```
fn format_output(opt: Options, st: &Stats, path: &str) -> String {
    let mut parts: Vec<String> = Vec::new();

    if opt.show_lines {
        parts.push(st.lines.to_string());
    }
    if opt.show_words {
        parts.push(st.words.to_string());
    }
    if opt.show_bytes {
        parts.push(st.bytes.to_string());
    }

    parts.push(path.to_string());
    parts.join(" ")
}
```

٤.١٢ معالجة الأخطاء بالشكل الصحيح

البنية العملية:

- run() تُعيد Result<(), AppError>
- main() تطبع الخطأ وتخرج بشكل نظيف

```rust
use std::fs;

fn run() -> Result<(), AppError> {
    let args: Vec<String> = std::env::args().collect();
    let (opt, path) = parse_args(&args)?;

    let text = fs::read_to_string(&path)?;
    let st = compute_stats(&text);
    let out = format_output(opt, &st, &path);

    println!("{}", out);
    Ok(())
}

fn main() {
    if let Err(e) = run() {
        eprintln!("{}", e);
        std::process::exit(1);
    }
}
```

هذا يحقق:

- فصل واضح بين المنطق وحدود البرنامج
- استخدام اصطلاحي للمُعامل ?
- رسائل خطأ واضحة ومتوقعة

٥.١٢ إعادة الهيكلة والتحسين

بعد أن يعمل المشروع:

- اجعل compute_stats دالة نقية قابلة للاختبار.
- افصل تحليل الوسائط عن عمليات الملفات.
- اجعل main خفيفًا ويفوض إلى run.

إضافة اختبارات وحدات

```
#[cfg(test)]
mod tests {
    use super::*;

    #[test]
    fn stats_basic() {
        let text = "a b\nc\n";
        let st = compute_stats(text);
        assert_eq!(st.lines, 2);
        assert_eq!(st.words, 3);
        assert_eq!(st.bytes, text.as_bytes().len());
    }

    #[test]
    fn output_default_order() {
        let opt = Options {
            show_lines: true,
            show_words: true,
            show_bytes: true
```

```
        };

        let st = Stats { lines: 2, words: 3, bytes: 6 };
        let s = format_output(opt, &st, "x.txt");
        assert_eq!(s, "2 3 6 x.txt");
    }
}
```

تشغيل الاختبارات

```
cargo test
```

تحسينات اختيارية

- محاذاة الأعمدة بعرض ثابت.
- إضافة -h أو --help.
- دعم القراءة من stdin.
- دعم عدة ملفات وحساب المجموع الكلي.

٦.١٢ مراجعة نهائية للمشروع

هذا المشروع المصغّر يدمج أهم مهارات Rust:

- الملكية والاستعارة: النص مملوك كـ String والمنطق يعمل على &str.
- معالجة الأخطاء: Result + ? + نوع خطأ مخصص.
- التكرار والمجموعات: العد باستخدام المكررات iterators.

- بنية نظيفة: فصل الإدخال، المنطق، العرض، وحدود التنفيذ.

النسخة الكاملة المصغّرة (ملف واحد)

```
use std::fmt;
use std::fs;

#[derive(Debug, Clone, Copy)]
struct Options {
    show_lines: bool,
    show_words: bool,
    show_bytes: bool,
}

#[derive(Debug, Default)]
struct Stats {
    lines: usize,
    words: usize,
    bytes: usize,
}

#[derive(Debug)]
enum AppError {
    Usage(String),
    Io(std::io::Error),
}

impl fmt::Display for AppError {
    fn fmt(&self, f: &mut fmt::Formatter<'_>) -> fmt::Result {
        match self {
```

```
            AppError::Usage(msg) => write!(f, "{}", msg),
            AppError::Io(e) => write!(f, "io error: {}", e),
        }
    }
}

impl From<std::io::Error> for AppError {
    fn from(e: std::io::Error) -> Self {
        AppError::Io(e)
    }
}

fn usage() -> String {
    let msg =
"Usage:
  rwc [FLAGS] <file>

Flags:
  -l    print lines
  -w    print words
  -b    print bytes

Examples:
  rwc data.txt
  rwc -w data.txt
  rwc -l -b data.txt
";
    msg.to_string()
}
```

```
fn parse_args(args: &[String]) -> Result<(Options, String), AppError> {
    if args.len() < 2 {
        return Err(AppError::Usage(usage()));
    }

    let mut opt = Options {
        show_lines: false,
        show_words: false,
        show_bytes: false
    };

    let mut path: Option<String> = None;

    for a in &args[1..] {
        match a.as_str() {
            "-l" => opt.show_lines = true,
            "-w" => opt.show_words = true,
            "-b" => opt.show_bytes = true,
            _ if a.starts_with('-') => {
                return Err(AppError::Usage(
                    format!("Unknown flag: {}\n\n{}", a, usage())
                ));
            }
            _ => {
                if path.is_some() {
                    return Err(AppError::Usage(
                        format!("Too many inputs.\n\n{}", usage())
                    ));
```

```
                }
                path = Some(a.clone());
            }
        }
    }

    let path = path.ok_or_else(|| AppError::Usage(usage()))?;

    if !(opt.show_lines || opt.show_words || opt.show_bytes) {
        opt.show_lines = true;
        opt.show_words = true;
        opt.show_bytes = true;
    }

    Ok((opt, path))
}

fn compute_stats(text: &str) -> Stats {
    Stats {
        lines: text.lines().count(),
        words: text.split_whitespace().count(),
        bytes: text.as_bytes().len(),
    }
}

fn format_output(opt: Options, st: &Stats, path: &str) -> String {
    let mut parts: Vec<String> = Vec::new();

    if opt.show_lines {
```

```
        parts.push(st.lines.to_string());
    }
    if opt.show_words {
        parts.push(st.words.to_string());
    }
    if opt.show_bytes {
        parts.push(st.bytes.to_string());
    }

    parts.push(path.to_string());
    parts.join(" ")
}

fn run() -> Result<(), AppError> {
    let args: Vec<String> = std::env::args().collect();
    let (opt, path) = parse_args(&args)?;

    let text = fs::read_to_string(&path)?;
    let st = compute_stats(&text);
    let out = format_output(opt, &st, &path);

    println!("{}", out);
    Ok(())
}

fn main() {
    if let Err(e) = run() {
        eprintln!("{}", e);
        std::process::exit(1);
```

```
    }
}
```

بهذا تكون قد بنيت أداة كاملة باستخدام أهم أسس Rust: ملكية دقيقة، منطق معتمد على المكررات، ومعالجة أخطاء اصطلاحية مع بنية مشروع نظيفة وقابلة للتوسّع.

الفصل ١٣: الخلاصة النهائية وأهم الدروس الجوهرية

١.١٣ النموذج الذهني للغة Rust

أفضل طريقة لفهم Rust هي اعتبارها لغة تجعل إدارة الموارد والصحة البرمجية صريحة في وقت الترجمة.
بدلاً من الاعتماد على آليات وقت التشغيل (مثل جامع القمامة) أو على انضباط المبرمج فقط، تقوم Rust بترميز قواعد الأمان داخل نظام الأنواع ونموذج الاستعارة.
نموذج ذهني عملي:

- الملكية (Ownership) تجيب: من المسؤول عن تحرير / إغلاق / إسقاط هذا المورد؟
- الاستعارة (Borrowing) تجيب: من يحق له القراءة أو الكتابة على هذا المورد الآن؟
- الأعمار (Lifetimes) تجيب: إلى متى يُضمن أن يكون هذا المرجع صالحًا؟
- Enums + المطابقة (Pattern Matching) تجيب: ما الحالات الممكنة؟ وهل تعاملنا مع جميعها صراحة؟
- Option / Result تجيب: كيف نمثل الفشل أو غياب القيمة دون غموض؟

إذا كتبت الشيفرة وأنت تطرح هذه الأسئلة باستمرار، ستصبح اللغة متوقعة ومنطقية؛ أغلب رسائل المترجم ما هي إلا تطبيق صارم لأحد هذه العقود.

٢.١٣ القواعد الذهبية الثلاث

في البرمجة اليومية باستخدام Rust، ثلاث قواعد تغطي معظم ما تحتاجه:

١. مالك واحد: كل قيمة لها مالك واحد مسؤول عن تنظيفها (drop).

٢. قراء متعددون أو كاتب واحد: يمكن وجود عدة &T في نفس الوقت، أو &mut T واحدة فقط (وليس كلاهما).

٣. لا مراجع معلّقة: لا يجوز أن يعيش المرجع أطول من القيمة التي يشير إليها.

مثال موجز:

```
fn main() {
    let mut s = String::from("Rust");

    let r1 = &s;
    let r2 = &s;
    println!("{} {}", r1, r2); //

    let w = &mut s;            //              (NLL)
    w.push_str("!");
    println!("{}", s);
}
```

عندما يرفض المترجم شيفرة ما، فهو في الغالب يطبق واحدة من هذه القواعد الثلاث.

٣.١٣ التفكير بعقلية الملكية

لكتابة Rust موثوقة، صمّم واجهاتك البرمجية باختيار عقد الملكية المناسب:

- استخدم &T أو &str عندما تحتاج إلى القراءة فقط.
- استخدم &mut T عندما تحتاج إلى التعديل دون نقل الملكية.
- استخدم T (قيمة مملوكة) عندما يجب على الدالة استهلاك القيمة أو تخزينها.
- أعد T عندما تنشئ بيانات جديدة.
- استخدم clone فقط عندما تحتاج فعلاً إلى نسخة مستقلة مملوكة.

أمثلة لنفس الفكرة بعقود مختلفة:
قراءة فقط (استعارة)

```
fn count_words(s: &str) -> usize {
    s.split_whitespace().count()
}
```

تعديل في المكان (استعارة قابلة للتغيير)

```
fn normalize_in_place(s: &mut String) {
    *s = s.trim().to_lowercase();
}
```

استهلاك الملكية (نقل وتخزين)

```
struct Store {
    items: Vec<String>,
}
```

```
impl Store {
    fn add(&mut self, item: String) {
        self.items.push(item); //            Store
    }
}
```

إنشاء مخرجات جديدة (إرجاع قيمة مملوكة)

```
fn normalized(s: &str) -> String {
    s.trim().to_lowercase()
}
```

قاعدة عملية مهمة: استعر في الإدخال، واملك في التخزين والإخراج. هذا يجعل واجهاتك مرنة وفعّالة وسهلة الاستخدام.

٤.١٣ كتابة Rust بأسلوب اصطلاحي

الأسلوب الاصطلاحي في Rust لا يتعلق بالشكل فقط، بل بالتعبير الواضح عن النية وترك المترجم يفرض الصحة.

1) فضّل المكررات على الفهرسة اليدوية

```
fn sum(xs: &[i32]) -> i32 {
    xs.iter().sum()
}
```

2) استخدم Result و ? لتدفق أخطاء خطي وواضح

```
use std::fs;

fn load_trimmed(path: &str) -> Result<String, std::io::Error> {
```

```
    let s = fs::read_to_string(path)?;
    Ok(s.trim().to_string())
}
```

3) استخدم match لجعل الحالات صريحة

```
fn describe(x: Option<i32>) -> &'static str {
    match x {
        Some(v) if v > 0 => "positive",
        Some(0) => "zero",
        Some(_) => "negative",
        None => "missing",
    }
}
```

4) اجعل main خفيفة وادفع المنطق إلى run()

```
fn run() -> Result<(), String> {
    Ok(())
}

fn main() {
    if let Err(e) = run() {
        eprintln!("{}", e);
        std::process::exit(1);
    }
}
```

5) استخدم وحدات صغيرة بحدود واضحة

قسّم حسب المسؤولية: نماذج البيانات، المنطق الأساسي، وحدود الإدخال/الإخراج. هذا يقلل تعارضات الاستعارة ويحسن قابلية الاختبار.

6) اجعل الحالات غير الصحيحة غير قابلة للتمثيل
استخدم enum و Option بدل الأعلام الغامضة.

```
#[derive(Debug)]
enum Status {
    Draft,
    Published { url: String },
}

#[derive(Debug)]
struct Post {
    title: String,
    status: Status,
}
```

٥.١٣ إلى أين بعد ذلك؟

بعد إنهاء هذا الكتيب، يمكنك التوسع بشكل منظم:

- الاختبارات والأدوات: cargo test، rustfmt، clippy، التوثيق عبر cargo doc.
- السمات (Traits) والقوالب (Generics) بعمق أكبر.
- تصميم الأخطاء المتقدم: طبقات أخطاء واضحة في التطبيقات الكبيرة.
- البرمجة غير المتزامنة: async/await، التنفيذيات، وأنماط التزامن.
- التوازي والمزامنة: Arc، Mutex، تمرير الرسائل.
- الأداء والتحليل: فهم التخصيصات وبناء الإصدارات النهائية.

- برمجة الأنظمة: FFI، واجهات أنظمة التشغيل، الشبكات.

توصية عملية أخيرة: ابنِ 2–3 أدوات صغيرة إضافية، كل واحدة تركز على مهارة جديدة (تحليل نصوص، شبكة، توازي، أو async). تتحول Rust إلى لغة طبيعية عندما تمارس نموذج الملكية والأخطاء في برامج حقيقية مرارًا.

إذا استطعت أن تشرح بوضوح:

- من يملك القيمة،
- من يستعيرها،
- وكيف تنتقل الأخطاء عبر البرنامج،

فأنت بالفعل تفكر بعقلية Rust.

الملاحق

الملحق :A ملخص سريع للملكية والاستعارة

القواعد الأساسية

- لكل قيمة مالك واحد فقط.
- عند خروج المالك من النطاق، تُنفّذ عملية drop ويُحرَّر المورد بشكل حتمي.
- يمكن وجود عدة مراجع غير قابلة للتغيير &T أو مرجع واحد قابل للتغيير &mut T (وليس الاثنين معًا).
- يجب ألا يعيش المرجع أطول من البيانات التي يشير إليها (لا مراجع معلّقة).

Move مقابل Copy

```
fn main() {
    let a = 10;       // i32 is Copy
    let b = a;        // Copy: both usable
    println!("{} {}", a, b);

    let s1 = String::from("hi");
```

```
    let s2 = s1;      // Move: s1 invalid
    // println!("{}", s1); // error
    println!("{}", s2);
}
```

استعر لتجنب النقل

```
fn len(s: &str) -> usize { s.len() }

fn main() {
    let s = String::from("borrow");
    let n = len(&s);          // borrow, no move
    println!("n={}, s={}", n, s);
}
```

استعارة قابلة للتغيير (كتابة حصرية)

```
fn main() {
    let mut s = String::from("Hello");
    {
        let w = &mut s;
        w.push_str(" Rust");
    } //
    println!("{}", s);
}
```

تقليص نطاق الاستعارة لفك الحظر عن التعديل

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

```
    let first = v[0];  //        (Copy)
    v.push(first + 10);

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

دليل سريع لتصميم الواجهات

- الإدخال: فضّل الاستعارة (&T, &str, الشرائح).
- الإخراج: أعد قيمة مملوكة (T, String, Vec<T>) عند إنشاء بيانات جديدة.
- التخزين: خذ الملكية (T) عندما تحتاج الاحتفاظ بالقيمة.
- clone: استخدمه صراحة وبنية واضحة (نسخ عميق في كثير من الأنواع).

الملحق B: شرح أشهر أخطاء المترجم

هذا الملحق يقدّم: ماذا يعني الخطأ؟ وكيف تصلحه؟
1) استخدام قيمة بعد نقلها (use of moved value)

```
fn main() {
    let s1 = String::from("x");
    let s2 = s1;
    // println!("{}", s1); // error: moved
    println!("{}", s2);
}
```

الحلول:

- الاستعارة: let s2 = &s1; (وتعديل التواقيع).

- النسخ العميق: let s2 = s1.clone();
- إعادة تصميم الدالة لتأخذ &str بدل String.

2) لا يمكن الاستعارة كتغيير لأنها مستعارة قراءة

```
fn main() {
    let mut s = String::from("hi");
    let r = &s;
    // let w = &mut s; // error
    println!("{}", r);
}
```

الحلول:

- أنهِ الاستعارة غير القابلة للتغيير أولاً.
- ضع التعديل في نطاق أصغر.

```
fn main() {
    let mut s = String::from("hi");
    let len = s.len(); //
    s.push('!');
    println!("len={}, s={}", len, s);
}
```

3) لا يمكن وجود استعارتين قابلتين للتغيير في نفس الوقت

```
fn main() {
    let mut x = 0;
    let a = &mut x;
    // let b = &mut x; // error
    *a += 1;
}
```

الحل: استعارة واحدة في كل نطاق.

```
fn main() {
    let mut x = 0;

    {
        let a = &mut x;
        *a += 1;
    }

    {
        let b = &mut x;
        *b += 2;
    }

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

4) القيمة المستعارة لا تعيش مدة كافية

```
fn main() {
    let r: &str;
    {
        let s = String::from("short");
        r = &s;
    }
    // println!("{}", r); // error
}
```

الحل: وسّع نطاق المالك أو أعد قيمة مملوكة.

5) لا يمكن إعادة مرجع إلى متغير محلي

```
fn bad() -> &str {
    let s = String::from("tmp");
    &s // error
}
```

الحل: أعد String، أو أعد مرجعًا مشتقًا من إدخال.

الملحق C: مرجع سريع لأوامر Cargo

إنشاء مشروع

```
cargo new my_app
cargo new my_lib --lib
```

بناء وتشغيل

```
cargo run
cargo build
cargo build --release
```

فحص سريع دون إنتاج ملف تنفيذي

```
cargo check
```

الاختبارات

```
cargo test
cargo test test_name
```

التنسيق والتحليل

```
cargo fmt
cargo clippy
```

التوثيق

```
cargo doc
cargo doc --open
```

تحديث الاعتماديات

```
cargo update
```

تنظيف الملفات الناتجة

```
cargo clean
```

ملفات الضبط

- dev: بناء سريع، تحسينات أقل.
- release: تحسينات عالية للأداء.

الملحق D: نموذج الذاكرة المبسّط

المكدس مقابل الكومة

- Stack: تخصيص سريع، تنظيف تلقائي عند نهاية النطاق.
- Heap: تخصيص ديناميكي، تُدار عبر مالك يضمن تحريرها عند drop.

String كمثال

- قيمة String نفسها (مؤشر + طول + سعة) على المكدس.

- المحتوى النصي على الكومة.
- نقل String ينقل المقبض الصغير فقط، لا كل البافر.

```
fn main() {
    let s1 = String::from("hello");
    let s2 = s1; //
    // println!("{}", s1); //    double free
    println!("{}", s2);
}
```

لماذا تُقيَّد المراجع؟
تعديل Vec قد يعيد تخصيص الذاكرة، ما قد يُبطل المراجع. لذلك تمنع Rust التعديل أثناء وجود استعارات قراءة.

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

    let first = &v[0];
    // v.push(4); // :
    println!("{}", first);
}
```

التنظيف الحتمي (Drop)

```
struct Guard;

impl Drop for Guard {
    fn drop(&mut self) {
        println!("cleaning up");
```

```
    }
}

fn main() {
    let _g = Guard;
    println!("work");
} //    cleaning up
```

الملحق E: إرشادات أسلوب كتابة Rust

التسمية

- الأنواع: CamelCase
- الدوال والمتغيرات: snake_case
- الثوابت: SCREAMING_SNAKE_CASE

فضّل &str لمدخلات النص

```
fn normalize(s: &str) -> String {
    s.trim().to_lowercase()
}
```

اجعل الملكية واضحة

- استعر عند الإمكان.
- أعد قيماً مملوكة عند إنشاء بيانات جديدة.
- تجنب clone() إلا عند الحاجة الفعلية.

استخدم Result و ? عند الحدود

```
use std::fs;

fn load(path: &str) -> Result<String, std::io::Error> {
    Ok(fs::read_to_string(path)?)
}
```

فضّل المكررات والتراكيب التعبيرية

```
fn even_squares(n: i32) -> Vec<i32> {
    (1..=n)
        .map(|x| x * x)
        .filter(|x| x % 2 == 0)
        .collect()
}
```

طابق جميع الحالات، واستخدم enums للنمذجة

```
enum Mode {
    Fast,
    Safe,
}

fn describe(m: Mode) -> &'static str {
    match m {
        Mode::Fast => "fast",
        Mode::Safe => "safe",
    }
}
```

هيكلة المشروع

- اجعل main.rs بسيطًا.
- ضع المنطق في lib.rs ووحدات منفصلة.
- افصل بين النماذج والمنطق وحدود الإدخال/الإخراج.

الأدوات

- fmt cargo لتنسيق موحّد.
- clippy cargo لاكتشاف الأخطاء الشائعة وتحسين الأسلوب.

صُمّمت هذه الملاحق كمرجع سريع أثناء بناء مشاريعك الحقيقية: ارجع إلى الملحق A وB عند مواجهة أخطاء المترجم، الملحق C أثناء العمل مع Cargo، الملحق D لفهم الذاكرة والاستعارة، والملحق E للحفاظ على شيفرة نظيفة واحترافية.

المراجع

التوثيق الرسمي للغة Rust

يُعد التوثيق الرسمي للغة Rust المصدر الأساسي والأكثر اعتمادًا لفهم اللغة ونظامها البيئي. تتم صيانته من قِبل مشروع Rust ويعكس سلوك الإصدار المستقر الحالي بدقة.
يتضمن هذا التوثيق:

- توثيق المكتبة القياسية، والمتاح محليًا عبر الأمر cargo doc --open.
- توثيق الواجهات البرمجية (API) للأنواع الأساسية مثل String و Vec<T> و Option<T> و Result<T, E> والمكررات (iterators) والمجموعات.
- توثيق أدوات السلسلة البرمجية (toolchain) مثل Cargo و rustc و rustfmt و clippy.

مثال على استكشاف توثيق نوع معيّن:

```
fn main() {
    let v = vec![1, 2, 3];
    println!("{}", v.len());
}
```

يشرح التوثيق الضمانات، وتنفيذ السمات (traits)، وخصائص الأداء، وعقود السلامة. ويجب أن يكون المرجع الأول عند تعلم واجهة جديدة.

The Rust Programming Language (The Book)

يُعرف عادة باسم ``The Book''، وهو المقدمة الرسمية والشاملة للغة Rust. يغطي:

- الملكية (Ownership) والاستعارة (Borrowing)
- مدد الحياة (Lifetimes)
- البُنى (Structs) والتعدادات (Enums) والمطابقة (Pattern Matching)
- معالجة الأخطاء باستخدام Result و ?
- الوحدات وتنظيم المشاريع
- أساسيات التزامن (Concurrency)

تكمن قوته التعليمية في شرح لماذا تفرض Rust قواعد معينة، وليس فقط كيفية استخدامها. مثال من موضوع الملكية عند استدعاء الدوال:

```
fn take(s: String) {}

fn main() {
    let s = String::from("data");
    take(s); // ownership moved
}
```

يوفر الكتاب وضوحًا مفاهيميًا وتسلسلاً منهجيًا مناسبًا للمبتدئين والمتوسطين.

المرجع الرسمي Rust Reference

يُعد Rust Reference المواصفة الرسمية الدقيقة للغة. يصف:

- قواعد البنية النحوية (Syntax Grammar)
- قواعد نظام الأنواع (Type System)
- دلالات المطابقة (Pattern Matching)
- قواعد مدد الحياة والاستعارة
- آلية حل السمات (Trait Resolution)
- ضمانات الذاكرة وتخطيط البيانات

على عكس ``The Book''، فإن المرجع ليس تعليميًا، بل تقني ودقيق، موجّه لمن يحتاج إلى فهم القواعد الرسمية بشكل صريح.
من الموضوعات التي يوضحها بدقة:

- كيفية التحقق من شمولية match
- قواعد حذف مدد الحياة (Lifetime Elision)
- سلوك ربط الأنماط (Pattern Bindings)

عند ظهور سلوك يبدو مفاجئًا، يكون المرجع هو المصدر الحاسم لتحديد القاعدة الصحيحة.

Rustonomicon

يُعد Rustonomicon الدليل المتقدم للبرمجة غير الآمنة باستخدام unsafe في Rust. وهو يشرح:

- كتل unsafe والثوابت (Invariants)
- المؤشرات الخام (Raw Pointers)
- السمات التلقائية Send و Sync

- أنماط القابلية الداخلية للتغيير (Interior Mutability)
- ترتيب Drop ودلالات المُدمِّرات (Destructors)
- حدود السلامة عند الربط مع لغات أخرى (FFI)

مثال على استخدام مؤشر خام:

```
fn main() {
    let x = 5;
    let ptr = &x as *const i32;

    unsafe {
        println!("{}", *ptr);
    }
}
```

يوضح Rustonomicon الضمانات التي تقدمها Rust، وكذلك الالتزامات التي يجب على المطوّر احترامها عند الخروج عن نطاق Rust safe.
يُنصح به بعد إتقان الملكية والاستعارة ومدد الحياة.

المجتمع ومصادر التعلم

يدعم نظام Rust البيئي مجتمع تقني قوي ومنضبط.
من مسارات التعلم المهمة:

- أمثلة وأنماط برمجية معيارية من المجتمع.
- دراسة مشاريع مفتوحة المصدر مكتوبة بلغة Rust.
- مناقشات القضايا (Issues) في الحزم الكبرى، والتي تتضمن كثيرًا من مبررات التصميم.

- أدوات مثل clippy (للتحليل والتنبيه) و rustfmt (للتنسيق)، والتي تجسّد معايير الأسلوب المجتمعية.

أدوات عملية شائعة:

```
cargo fmt
cargo clippy
cargo test
```

دراسة الحزم المفتوحة المصدر الجيدة تعزز فهم:

- الهيكلة المعيارية (Modular Architecture)
- أنماط تمرير الأخطاء
- تصميم واجهات تراعي الملكية
- التجريد القائم على السمات (Trait-Based Abstraction)
- الكتابة المراعية للأداء

للاستمرار في التطور:

- اقرأ وادرس الشيفرة المصدرية للمكتبة القياسية.
- ابنِ أدوات ومكتبات صغيرة.
- ساهم في تحسين التوثيق أو إصلاح الأخطاء في مشاريع مفتوحة المصدر.
- حلّل أنماط معالجة الأخطاء والتزامن في مشاريع حقيقية.

توفر هذه المراجع مجتمعة:

- الأسس المفاهيمية (The Book)

- الدقة الرسمية Rust) (Reference
- الفهم المتقدم للذاكرة والسلامة (Rustonomicon)
- معرفة الواجهات والنظام البيئي (التوثيق الرسمي)
- أنماط الهندسة العملية (موارد المجتمع)

ومعًا تشكّل أساسًا متكاملاً وموثوقًا لتطوير احترافي باستخدام Rust.