https://simplifycpp.org

البرمجة باستخدام لغة Rust

البداية العملية

إعداد: أيمن الحراكي

البرمجة باستخدام لغة Rust

بداية عملية

تم دعم بعض مراحل الصياغة واستكشاف الأفكار باستخدام أدوات ذكاء اصطناعي حديثة، وذلك تحت إشراف كامل، مع التحقق والتدقيق والتصحيح، وبمسؤولية وتأليف المؤلف بالكامل.

إعداد : أيمن الحراكي

simplifycpp.org

فبراير 2026

المحتويات

مقدمة المؤلف

كانت لغة البرمجة الأساسية لدي لأكثر من ثلاثة عقود هي C++. لقد عايشت تطورها، ونقاط قوتها، وتحدياتها عبر أجيال متعددة من العتاد، وأنظمة التشغيل، ونماذج البرمجيات المختلفة. ولا تزال C++ واحدة من أقوى لغات برمجة الأنظمة التي تم ابتكارها على الإطلاق، ويتطلب إتقانها فهمًا عميقًا، وانضباطًا عاليًا، واحترامًا صارمًا للواقع منخفض المستوى.

ومع ذلك، فإن المقارنة العادلة بين لغات البرمجة تتطلب أكثر من مجرد الولاء أو طول الخبرة. إنها تتطلب كفاءة مهنية حقيقية في البدائل الأخرى. وقد كُتب هذا الكتيّب الصغير من هذا المنطلق تحديدًا: الحاجة إلى إتقان لغة Rust ليس بوصفها بديلاً عن C++، بل كنظير حديث يستحق مقارنة جادة وعميقة.

تمثل Rust استجابة مختلفة للمشكلات التي اعتاد مطورو C++ على حلها عبر الانضباط، والاتفاقيات، والأدوات، والمراجعة المكثفة. ففي حين تمنح C++ أقصى درجات الحرية وتضع المسؤولية تقريبًا بالكامل على عاتق المبرمج، تقوم Rust عمدًا بترميز العديد من قواعد السلامة مباشرة داخل اللغة والمترجم. إن مفاهيم مثل الملكية، والاستعارة، والتعامل الصريح مع الأخطاء ليست تفضيلات أسلوبية، بل آليات صُممت لمنع فئات كاملة من الأخطاء قبل أن يعمل البرنامج أصلاً.

بالنسبة للأنظمة الحرجة — وخاصة تلك التي تكون فيها الأمان، والموثوقية، والتزامن عناصر محورية — تقدم اللغات الحديثة مثل Rust ضمانات قوية ومقنعة. هذه الضمانات لا تأتي من فحوصات وقت التشغيل أو من جامع قمامة، بل من التحليل الساكن والتطبيق الصارم في وقت الترجمة. إن فهم هذا النموذج بشكل صحيح أمر أساسي قبل إطلاق أي أحكام تتعلق بالتفوق، أو الملاءمة، أو المفاضلات بين اللغات.

وُجد هذا الكتيّب لدعم هذا الفهم. فهو يركز على أساسيات Rust التي تهم مبرمجي الأنظمة، وخصوصًا أولئك القادمين من C أو C++. الهدف ليس الترويج، بل الوضوح: تعلم Rust بالقدر

الكافي لتقييمها بإنصاف، واستخدامها بكفاءة حيثما كانت مناسبة، ومعرفة المواضع التي تقدم فيها اختياراتها التصميمية مزايا ملموسة في السلامة والصحة البرمجية للبرمجيات الحساسة أمنيًا.
المادة المقدمة عملية ومرتكزة على الواقع عن قصد. فهي تتجنب المقارنات السطحية، وبدلاً من ذلك تبني الأساس المفاهيمي اللازم لكتابة برامج Rust حقيقية، والتفكير السليم في الذاكرة وأعمار الكائنات، وفهم كيفية فرض Rust للثوابت البرمجية التي تتركها C++ لانضباط البشر. ومن هذا المستوى من الفهم فقط يمكن أن تكون أي مقارنة بين C++ وRust عادلة، وتقنية، ومهنية. وفي النهاية، يعكس هذا العمل مبدأً أوسع: اللغات أدوات، والاحتراف يعني معرفة متى وكيف يُستخدم كل أداة بمسؤولية. تُعد Rust إحدى هذه الأدوات — قوية، ذات فلسفة واضحة، ومتزايدة الأهمية في الأنظمة الحديثة الحساسة أمنيًا. ويُعد هذا الكتيّب خطوة نحو إتقانها بالجدية نفسها التي تتطلبها عقود من الخبرة.

أيمن الحراكي

المقدمة

تُعد Rust لغة برمجة أنظمة حديثة صُممت لمعالجة تحديات قديمة في موثوقية البرمجيات، والأداء، والسلامة. وقد أُنشئت لتمكين المطورين من كتابة شيفرة منخفضة المستوى وعالية الأداء مع القضاء على العديد من فئات الأخطاء التي لا تظهر تقليديًا إلا أثناء وقت التشغيل. وبدلاً من الاعتماد على جامع قمامة أو فحوصات مكثفة في وقت التنفيذ، تفرض Rust الصحة البرمجية من خلال نظام الأنواع والتحليل في وقت الترجمة.

يوفر هذا الكتيّب الصغير مقدمة مركزة وعملية إلى Rust للمبرمجين الذين يرغبون في فهم اللغة كما تُستخدم اليوم. لا ينصب التركيز على حفظ الصياغة، بل على تعلم المفاهيم الجوهرية التي تحدد نهج Rust: الملكية والاستعارة، والتعامل الصريح مع الأخطاء، والإدارة الحتمية للموارد، وبنية البرامج المعيارية. تشكل هذه المفاهيم الأساس لكتابة برامج Rust صحيحة وقابلة للصيانة على أي نطاق.

المحتوى موجز وتدريجي عن قصد. يبني كل فصل على ما سبقه، بدءًا من البنى اللغوية الأساسية وصولاً إلى التطبيق العملي. وقد اختيرت الأمثلة لتعكس أسلوب Rust الحديث والمعياري، مع الاعتماد حصريًا على المكتبة القياسية للحفاظ على تجربة تعلم واضحة وقابلة لإعادة الإنتاج.

هذا الكتيّب مناسب للقراء الذين يمتلكون خبرة برمجية سابقة والجدد على Rust، أو لأولئك الذين يرغبون في ترسيخ فهمهم لأساسياتها. ولا يسعى إلى تغطية كل ميزة في اللغة، بل يهدف إلى بناء نموذج ذهني صحيح يُعد القارئ لموضوعات متقدمة مثل التزامن، والبرمجة غير المتزامنة، والعمل منخفض المستوى على الأنظمة.

بحلول نهاية هذا الكتيّب، ينبغي أن تكون قادرًا على قراءة وكتابة برامج Rust صغيرة، وتنظيم الشيفرة في وحدات وحِزم، والتعامل الصريح مع الأخطاء، والتفكير في الذاكرة وأعمار الكائنات بثقة. والأهم من ذلك، ستكون مزودًا بالأدوات المفاهيمية اللازمة لمواصلة تعلم Rust بفعالية ومسؤولية.

الفصل ١: ما هي **Rust** ولماذا وُجدت

١.١ أهداف تصميم **Rust**

تُعد Rust لغة برمجة أنظمة صُممت لتوفير سلامة الذاكرة، وسلامة التزامن، وأداء يمكن التنبؤ به دون استخدام جامع قمامة. ويتمثل هدفها الأساسي في تمكين المطورين من كتابة برامج تكون فعّالة وموثوقة افتراضيًا. ويرتكز تصميم Rust على ثلاثة مبادئ جوهرية:

- السلامة: تفرض Rust قواعد تقضي على فئات كاملة من أخطاء البرمجة الشائعة مثل فك الإشارة إلى مؤشرات فارغة، والاستخدام بعد تحرير الذاكرة، وسباقات البيانات في الشيفرة المتزامنة.

- الأداء: تقدم Rust أداءً مماثلاً للغات مثل C و++C من خلال تجنب الكلفة وقت التشغيل، وإتاحة تحكم دقيق في الذاكرة واستخدام الموارد.

- التجريد دون كلفة: تدعم Rust مفهوم التجريد الصفري الكلفة، أي أن البنى عالية المستوى لا تضيف أي كلفة وقت تشغيل مقارنة بالشيفرة منخفضة المستوى المكافئة.

وبشكلٍ جماعي، تضع هذه الأهداف Rust كلغة تجعل برمجة الأنظمة الآمنة والفعّالة أكثر سهولة وموثوقية.

٢.١ المفاضلة بين السلامة والأداء

يُعد تحقيق التوازن بين السلامة والأداء أحد التحديات المركزية في لغات برمجة الأنظمة. فاللغات التقليدية مثل C و++C توفر تحكمًا أقصى في الذاكرة والموارد، لكنها تترك فرض السلامة بالكامل على عاتق المبرمج. وغالبًا ما يؤدي ذلك إلى سلوك غير معرّف، وتلف في الذاكرة، وثغرات أمنية. تقدم Rust نموذج ملكية صارمًا يُفرض في وقت الترجمة ويضمن سلامة الذاكرة دون الحاجة إلى جامع قمامة. ويضمن نموذج الملكية ومدقق الاستعارة ما يلي:

- لكل قيمة مالك واحد فقط.
- لا يمكن أن تتجاوز المراجع عمر القيمة التي تشير إليها.
- يتم التحكم في القابلية للتغيير والتشارك الصوري بشكل صارم.

وبما أن هذه الفحوصات تُجرى في وقت الترجمة، فإن برامج Rust لا تتحمل أي كلفة وقت تشغيل مقابل فرض السلامة. ويضمن هذا التصميم ألا تضحي Rust بالأداء من أجل السلامة، بل يجعل السلامة جزءًا جوهريًا من دلالات اللغة يُفرض قبل تشغيل البرنامج.
يتجنب نهج Rust الآليات الشائعة في وقت التشغيل المستخدمة في اللغات المُدارة، مثل جمع القمامة، والتي تؤدي إلى توقفات غير متوقعة وكلفة إضافية في الذاكرة. ومن خلال ضمان السلامة في وقت الترجمة، تحقق Rust أداءً عاليًا وضمانات قوية للسلامة في آنٍ واحد.

٣.١ موضع **Rust** بين **C** و**C++** واللغات المُدارة

تحتل Rust موقعًا مميزًا في مشهد لغات البرمجة:

١.٣.١ مقارنة **Rust** مع **C** و**C++**

تُعد Rust الأقرب مقارنةً بلغتي C و++C بوصفها لغة برمجة أنظمة. إذ تتيح اللغات الثلاث الوصول منخفض المستوى إلى العتاد، وإدارة الذاكرة يدويًا، والتحكم الدقيق في الأداء. إلا أن Rust تتميز بما يلي:

- سلامة الذاكرة دون جامع قمامة: تقضي قواعد الملكية والاستعارة في Rust على فئات كاملة من الأخطاء الشائعة في C و++C.

- ميزات لغوية حديثة: تجعل مطابقة الأنماط، وأنواع البيانات الجبرية، والاستدلال التعبيري للأنواع شيفرة Rust أكثر تعبيرًا وأعلى أمانًا.

- أدوات البناء وإدارة الحزم: توفر أدوات Rust، والمتمركزة حول cargo، إدارة مدمجة للاعتمادات وأتمتة لعمليات البناء.

في المقابل، تضع C و++C مسؤولية السلامة بالكامل على المبرمج، مع الاعتماد على الانضباط اليدوي والأدوات الخارجية لمنع أخطاء الذاكرة والتزامن.

٢.٣.١ مقارنة **Rust** مع اللغات المُدارة

توفر اللغات المُدارة مثل Java و#C وPython ضمانات للسلامة من خلال إدارة تلقائية للذاكرة وفحوصات في وقت التشغيل. وتُبسّط هذه اللغات إنتاجية المبرمج، لكنها تُدخل كلفة أداء وسلوكًا غير متوقع أثناء التنفيذ نتيجة جمع القمامة.
توفر Rust العديد من مزايا السلامة الموجودة في اللغات المُدارة، مع الحفاظ على أداء يعادل اللغات غير المُدارة. وتحقق ذلك من خلال إجراء جميع فحوصات السلامة في وقت الترجمة، وتوفير تحكم مباشر في تخصيص الذاكرة وأعمار الموارد.

٣.٣.١ خلاصة موقع **Rust**

صُممت Rust للحالات التي تكون فيها السلامة، والأداء، والتحكم عناصر حاسمة في آنٍ واحد. وهي مناسبة لبرمجة الأنظمة، والتطوير المضمّن، وWebAssembly، والشبكات، والأنظمة المتزامنة، وغيرها من المجالات. ومن خلال الجمع بين خصائص الأداء في C و++C ومبادئ السلامة المرتبطة عادةً باللغات المُدارة، تقدم Rust توازنًا فريدًا يلبي متطلبات تطوير البرمجيات الحديثة.

٤.١ مثال بلغة Rust

يوضح المثال التالي مبدأ ربط المتغيرات والسلامة في Rust:

```
fn main() {
    let x = String::from("hello");
    let y = x; // ownership of the value is moved to y
    println!("{}", y);
}
```

في هذا المثال، تنتقل ملكية القيمة من x إلى y، مما يمنع استخدام مرجع قد يكون غير صالح، ويقضي على فئة من أخطاء وقت التشغيل الشائعة في لغات أخرى.

الفصل ٢: سلسلة أدوات Rust وسير العمل

١.٢ تثبيت Rust واختيار سلسلة الأدوات

صُممت Rust لتُثبَّت وتُدار على شكل سلسلة أدوات (toolchain) تضم المترجم، والمكتبة القياسية، وأدوات المطور المرتبطة بها والتي تتطور معًا. ويُعد النهج الموصى به هو استخدام rustup، والذي يتولى إدارة التثبيت، والتحديثات، والتبديل بين سلاسل الأدوات المختلفة.

١.١.٢ التثبيت والتحقق

على الأنظمة الشبيهة بيونكس (Linux، macOS، WSL)، يتم التثبيت عادةً عبر أمر تمهيدي واحد يقوم بتثبيت rustup وسلسلة أدوات Rust الافتراضية:

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

بعد اكتمال التثبيت، افتح نافذة طرفية جديدة (أو أعد تحميل جلسة الطرفية الحالية) وتحقق من التثبيت:

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

على نظام Windows، يتم تثبيت Rust باستخدام مُثبّت rustup-init، ويُعد اختيار سلسلة أدوات MSVC الخيار الأكثر شيوعًا للحصول على أفضل تكامل مع أدوات Visual Studio.

٢.١.٢ قنوات سلسلة الأدوات: stable، beta، nightly

تُنشر سلاسل أدوات Rust عبر قنوات متعددة:

- **stable**: الخيار الافتراضي للإنتاج والتعلم؛ يتميز بأفضل توافق واستقرار.
- **beta**: الإصدار المستقر القادم؛ مفيد للاختبار المبكر.
- **nightly**: يتضمن ميزات تجريبية؛ ولا يُستخدم إلا عند الحاجة إلى ميزات حصرية أو أدوات ترحيل مبكرة للإصدارات.

يتمثل سير العمل العملي في إبقاء **stable** كخيار افتراضي، واستخدام **nightly** فقط عندما تتطلب مهمة معينة ذلك:

```
rustup default stable
rustup update
rustup toolchain install nightly
```

يمكنك تشغيل أمر واحد باستخدام سلسلة أدوات بديلة دون تغيير الإعداد الافتراضي:

```
cargo +nightly --version
cargo +nightly check
```

٣.١.٢ تثبيت سلسلة أدوات خاصة بالمشروع

غالبًا ما تقوم المشاريع الاحترافية بتثبيت سلسلة الأدوات الخاصة بها لضمان قابلية إعادة البناء عبر الأجهزة المختلفة وأنظمة التكامل المستمر. وتُعد آلية شائعة لذلك استخدام ملف rust-toolchain.toml في جذر المشروع:

```
[toolchain]
channel = "stable"
components = ["rustfmt", "clippy"]
```

يضمن هذا أن يستخدم جميع المساهمين قناة مترجم موحدة، مع تثبيت المكونات الأساسية تلقائيًا.

٤.١.٢ الأهداف والمكونات

تدعم Rust الترجمة العابرة للمنصات عبر تثبيت الأهداف (targets):

```
rustup target list --installed
rustup target add x86_64-unknown-linux-gnu
```

من مكونات المطور الشائعة:

- **rustfmt**: أداة التنسيق الرسمية (cargo fmt)
- **clippy**: أداة التنبيهات والتحقق الرسمية (cargo clippy)

لتثبيت المكونات (إن لم تكن مثبتة بالفعل):

```
rustup component add rustfmt clippy
```

٢.٢ أساسيات Cargo

يُعد cargo أداة سير العمل القياسية في Rust، ويشمل إنشاء المشاريع، والبناء، وحل الاعتمادات، والاختبارات، وقياس الأداء، وتوليد التوثيق، والنشر.

١.٢.٢ إنشاء مشروع وتشغيله

لإنشاء مشروع جديد:

```
cargo new hello_rust
cd hello_rust
cargo run
```

ويكون ملف src/main.rs البسيط على النحو التالي:

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

٢.٢.٢ الأوامر الأساسية التي يجب إتقانها

تشكل الأوامر التالية جوهر سير العمل اليومي:

```
cargo build      # compile (default dev profile)
cargo run        # build and run
cargo check      # type-check quickly without producing a full binary
cargo test       # run tests (unit + integration)
cargo doc        # build local documentation for dependencies and your crate
cargo fmt        # format code (rustfmt component)
cargo clippy     # lint code (clippy component)
```

إرشادات عملية:

- استخدم `cargo check` بشكل متكرر للحصول على تغذية راجعة سريعة أثناء التطوير.
- استخدم `cargo fmt` للحفاظ على تنسيق موحد.
- استخدم `cargo clippy` لاكتشاف الأخطاء الشائعة مبكرًا.

٣.٢.٢ الاعتمادات، الإصدارات، وملف القفل

تُصرّح الاعتمادات في ملف Cargo.toml. ويقوم cargo بحل الإصدارات المتوافقة وتسجيل الإصدارات الدقيقة في ملف Cargo.lock لضمان قابلية إعادة البناء.
يتضمن ملف التوصيف الأدنى في Rust الحديثة عادةً تحديدًا صريحًا للإصدار (edition). تستخدم المشاريع الجديدة أحدث إصدار مستقر افتراضيًا، مثل:

```
[package]
name = "hello_rust"
version = "0.1.0"
edition = "2024"

[dependencies]
```

بالنسبة للتطبيقات، يُنصح بإضافة ملف Cargo.lock إلى نظام التحكم بالإصدارات لضمان بناء متطابق. أما المكتبات، فغالبًا لا يتم تضمين الملف، إلا إذا فرضت سياسات المؤسسة خلاف ذلك.

٤.٢.٢ الميزات كواجهة برمجية

تتيح ميزات Cargo توفير وظائف اختيارية وترجمة شرطية، وتُستخدم على نطاق واسع في:

- الاعتمادات الاختيارية،
- مفاضلات الأداء أو المنصات،
- اختيار الواجهات الخلفية.

نمط أساسي للتعريف:

```
[features]
default = []
fast-path = []
```

ثم استخدام cfg داخل الشيفرة:

```
pub fn compute(x: u64) -> u64 {
    #[cfg(feature = "fast-path")]
    {
        return x.wrapping_mul(3);
```

```
    }
    #[cfg(not(feature = "fast-path"))]
    {
        return x + x + x;
    }
}
```

لتفعيل ميزة أثناء البناء:

```
cargo build --features fast-path
```

٣.٢ بنية المشروع وأنماط البناء

يقوم cargo بتوحيد بنية المشاريع لجعل مشاريع Rust سهلة التصفح ومتسقة بين الفرق.

١.٣.٢ البنية القياسية

بنية حزمة نموذجية:

```
my_project/
  Cargo.toml
  Cargo.lock
  src/
    main.rs
    lib.rs
  tests/
  examples/
  benches/
```

أهم الاصطلاحات:

- src/main.rs: نقطة دخول التطبيق التنفيذي.
- src/lib.rs: جذر المكتبة (اختياري).
- tests/: اختبارات تكامل (تُبنى كحِزم مستقلة).
- examples/: أمثلة قابلة للتشغيل.
- benches/: اختبارات قياس الأداء.

يمكن وضع عدة ملفات تنفيذية ضمن src/bin/:

```
src/bin/admin.rs
src/bin/worker.rs
```

٢.٣.٢ الحِزم والوحدات

تُعد الحِزمة (crate) وحدة الترجمة الأساسية: إما مكتبة أو برنامجًا تنفيذيًا. وداخل الحِزمة، تُنظَّم الشيفرة في وحدات (modules) باستخدام mod وملفات أو مجلدات. ومن الاستراتيجيات النظيفة للبدء:

- إبقاء main.rs صغيرًا،
- نقل المنطق إلى lib.rs أو وحدات فرعية،
- كشف واجهة الاستخدام العامة المقصودة فقط باستخدام pub.

٣.٣.٢ أنماط البناء وملفات التعريف: dev مقابل release

يبني cargo المشاريع باستخدام ملف تعريف (profile) يتحكم في مستوى التحسين، ومعلومات التصحيح، وإعدادات المترجم الأخرى.

- **dev** (الافتراضي): أزمنة ترجمة سريعة وإعدادات مناسبة للتصحيح.
- **release**: تفعيل التحسينات لأجل الأداء.

استخدم البناء بنمط release عند قياس الأداء:

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

٤.٣.٢ إعداد ملفات التعريف

يمكن تخصيص ملفات التعريف في Cargo.toml. ونمط شائع هو إبقاء التطوير مريحًا مع جعل بناء الإصدار النهائي أكثر تحسينًا:

```
[profile.dev]
opt-level = 0
debug = true

[profile.release]
opt-level = 3
lto = true
codegen-units = 1
panic = "abort"
```

التفسير:

- opt-level: القيم الأعلى تُحسّن سرعة التنفيذ عادةً (مع احتمال زيادة زمن الترجمة).
- lto: تحسين وقت الربط قد يُحسّن الأداء ويُقلل الحجم (مع إبطاء الربط).
- codegen-units: تقليل الوحدات قد يؤدي إلى تحسين أفضل (مع إبطاء الترجمة).
- panic = "abort": ملفات تنفيذية أصغر ونمط فشل أبسط، مقابل فقدان بعض سلوكيات التفريغ.

٥.٣.٢ قائمة تدقيق عملية لسير العمل

للحصول على سير عمل نظيف واحترافي:

- التثبيت باستخدام rustup مع اعتماد **stable** افتراضيًا.
- تثبيت سلسلة الأدوات في rust-toolchain.toml لضمان القابلية للإعادة البناء.
- استخدام `cargo check` للتكرار السريع.
- تنسيق الشيفرة باستخدام `cargo fmt` والتحقق باستخدام `cargo clippy`.
- قياس الأداء باستخدام --release وليس dev.
- الالتزام بالبنية القياسية للمشاريع لتقليل كلفة الانضمام للمشروع.

الفصل ٣: أساسيات اللغة الجوهرية

١.٣ المتغيرات وعدم القابلية للتغيير

تستخدم Rust ما يُعرف بـ الروابط (bindings) التي تُنشأ باستخدام let. وبشكل افتراضي، تكون هذه الروابط غير قابلة للتغيير. ويُعد هذا اختيارًا تصميميًا مقصودًا يحدّ من التغييرات العرضية في الحالة، ويُسهّل التفكير المنطقي في الشيفرة، ويُحسّن السلامة في البرامج المتزامنة.

١.١.٣ عدم القابلية للتغيير افتراضيًا

```
fn main() {
    let x = 10;
    // x = 11; // error: cannot assign to immutable variable `x`
    println!("{x}");
}
```

٢.١.٣ القابلية للتغيير تكون صريحة

عند الحاجة إلى التغيير، يجب استخدام mut بشكل صريح:

```
fn main() {
```

```
    let mut counter = 0;
    counter += 1;
    println!("{counter}");
}
```

٣.١.٣ التظليل (Shadowing)

تسمح Rust بما يُعرف بـ التظليل، أي إعادة تعريف الاسم نفسه لإنشاء رابط جديد. يختلف التظليل عن التغيير؛ إذ يتم إنشاء قيمة جديدة تمامًا، وغالبًا ما يُحسّن الوضوح عند تحويل البيانات.

```
fn main() {
    let x = "42";
    let x = x.parse::<i32>().unwrap(); // shadow with a new type
    let x = x + 1;                     // shadow with a new value
    println!("{x}");
}
```

الاستخدامات الشائعة:

- التحويل بين الأنواع دون الحاجة إلى ابتكار أسماء متغيرات جديدة،
- تضييق نطاق قيمة سابقة،
- الإبقاء على الروابط غير قابلة للتغيير أثناء إجراء التحويلات.

٤.١.٣ الثوابت والمتغيرات الساكنة

تدعم Rust الثوابت التي يتم تقييمها في وقت الترجمة:

```
const MAX_RETRIES: u32 = 5;

fn main() {
    println!("{MAX_RETRIES}");
}
```

تُضمَّن الثوابت مباشرة في أماكن استخدامها. أما في حال الحاجة إلى حالة عامة ذات عنوان ثابت في الذاكرة، فتوفّر Rust الكلمة المفتاحية static، والتي تُستخدم عادةً للبيانات العامة المقروءة فقط.

٢.٣ الدوال والتعابير

تُعد Rust لغة مبنية على التعابير. إذ تُنتج العديد من البنى قيمًا، وغالبًا ما تُعيد الدوال آخر تعبير دون الحاجة إلى استخدام return بشكل صريح.

١.٢.٣ تواقيع الدوال

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

fn main() {
    let s = add(2, 3);
    println!("{s}");
}
```

نقاط أساسية:

- أنواع المعاملات وقيمة الإرجاع تكون صريحة.

- صيغة السهم -> T تُحدد نوع الإرجاع.
- حذف الفاصلة المنقوطة يجعل التعبير هو قيمة الإرجاع.

٣.٢.٣ التعليمات مقابل التعابير

تُحوّل الفاصلة المنقوطة التعبير إلى تعليمة، مما يؤدي إلى تجاهل قيمته:

```
fn main() {
    let a = 3 + 4;    // expression, value used
    let b = {         // block expression
        let t = a * 2;
        t + 1          // returned from the block
    };
    println!("{a} {b}");
}
```

٣.٢.٣ الإرجاع المبكر والدوال غير العائدة

لا يزال بالإمكان استخدام return للخروج المبكر من الدالة:

```
fn safe_div(a: i32, b: i32) -> i32 {
    if b == 0 {
        return 0;
    }
    a / b
}
```

الدالة التي لا تُعيد قيمة أبدًا (على سبيل المثال، لأنها تُطلق panic) يكون لها نوع خاص يُسمى نوع العدم !. ويُعد هذا مهمًا لأن ! يمكن أن يتحول ضمن تدفق التحكم إلى أنواع أخرى:

```rust
fn fail(msg: &str) -> ! {
    panic!("{msg}");
}
```

٤.٢.٣ الإغلاقات (Closures) – أساسيات

تلتقط الإغلاقات المتغيرات من بيئتها المحيطة:

```rust
fn main() {
    let factor = 3;
    let mul = |x: i32| x * factor;
    println!("{}", mul(7));
}
```

تختار Rust نمط الالتقاط (بالمرجع، أو بمرجع قابل للتغيير، أو بالنقل) بناءً على الاستخدام. وللإجبار على الالتقاط بالنقل:

```rust
fn main() {
    let s = String::from("data");
    let f = move || s.len(); // moves ownership of `s` into the closure
    println!("{}", f());
}
```

٣.٣ تدفق التحكم ومطابقة الأنماط

توفر Rust بنيات تدفق تحكم قياسية (if، الحلقات)، بالإضافة إلى نظام قوي لمطابقة الأنماط يتمحور حول match. وتُستخدم الأنماط على نطاق واسع مع التعدادات ومعالجة الأخطاء.

١.٣.٣ if كتعبير

في Rust، يمكن لـ if أن تُنتج قيمة:

```
fn main() {
    let n = 10;
    let kind = if n % 2 == 0 { "even" } else { "odd" };
    println!("{kind}");
}
```

يجب أن تُقيّم جميع الفروع إلى النوع نفسه.

٢.٣.٣ الحلقات: for، while، loop

تُكرر loop التنفيذ إلى أن يتم استدعاء break، ويمكنها أيضًا إعادة قيمة:

```
fn main() {
    let mut i = 0;
    let result = loop {
        i += 1;
        if i == 5 {
            break i * 2; // value returned from the loop
        }
    };
    println!("{result}");
}
```

تُعد while حلقة تعتمد على شرط:

```
fn main() {
    let mut n = 3;
```

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

أما for فتُستخدم للتكرار على المُكرِّرات وهي الخيار الأسلوبي الموصى به:

```
fn main() {
    for x in 0..3 {
        println!("{x}");
    }

    let v = vec![10, 20, 30];
    for item in &v {
        println!("{item}");
    }
}
```

٣.٣.٣ match: مطابقة شاملة

يجب أن تكون match شاملة لكل الحالات الممكنة، مما يمنع إغفال أي حالة:

```
fn describe(n: i32) -> &'static str {
    match n {
        0 => "zero",
        1..=9 => "small",
        _ => "large",
    }
```

```
}

fn main() {
    println!("{}", describe(7));
}
```

تدعم الأنماط:

- النطاقات (1..=9),
- المحارف العامة (_),
- تفكيك التراكيب (destructuring) للتراكيب والتعدادات.

٤.٣.٣ تفكيك الأزواج والتراكيب

```
struct Point { x: i32, y: i32 }

fn main() {
    let t = (1, 2);
    let (a, b) = t;
    println!("{a} {b}");

    let p = Point { x: 10, y: 20 };
    let Point { x, y } = p;
    println!("{x} {y}");
}
```

٥.٣.٣ التعدادات و Option و Result

تستخدم Rust التعدادات لنمذجة الغياب والعمليات القابلة للفشل.

Option<T> للقيم الاختيارية:

```
fn first(v: &[i32]) -> Option<i32> {
    if v.is_empty() { None } else { Some(v[0]) }
}

fn main() {
    let v = vec![5, 6, 7];
    match first(&v) {
        Some(x) => println!("first = {x}"),
        None => println!("empty"),
    }
}
```

Result<T, E> لمعالجة الأخطاء:

```
use std::num::ParseIntError;

fn parse_u32(s: &str) -> Result<u32, ParseIntError> {
    s.parse::<u32>()
}

fn main() {
    match parse_u32("123") {
        Ok(n) => println!("{n}"),
        Err(e) => println!("error: {e}"),
    }
}
```

٦.٣.٣ `if let` و `while let`

عند الاهتمام بنمط واحد فقط والرغبة في شيفرة مختصرة:

```
fn main() {
    let maybe = Some(42);

    if let Some(x) = maybe {
        println!("{x}");
    }

    let mut stack = vec![1, 2, 3];
    while let Some(v) = stack.pop() {
        println!("{v}");
    }
}
```

٧.٣.٣ الحراس والربط داخل الأنماط

تضيف حراس match شروطًا إضافية، بينما تلتقط الروابط القيم:

```
fn classify(n: i32) -> &'static str {
    match n {
        x if x < 0 => "negative",
        x if x == 0 => "zero",
        x if x > 0 => "positive",
        _ => "unreachable",
    }
}
```

٤.٣ ملخص مُكثف

- الروابط في Rust غير قابلة للتغيير افتراضيًا؛ ويُستخدم mut فقط عند الحاجة.
- Rust لغة مبنية على التعابير؛ إذ يمكن للكتل و if وحتى loop إعادة قيم.
- تُعد match شاملة ومحورية لتدفق تحكم آمن، خاصةً مع التعدادات.
- يُفضَّل التكرار الأسلوبي باستخدام for وأنماط المُكرِّرات.

الفصل ٤: الملكية والاستعارة

١.٤ قواعد الملكية

يُعد نظام الملكية في Rust الأساس الذي تقوم عليه ضمانات السلامة دون الحاجة إلى جامع قمامة. يفرض المترجم مجموعة صغيرة من القواعد التي تمنع أخطاء مثل الاستخدام بعد التحرير، والتحرير المزدوج، وسباقات البيانات في الشيفرة الآمنة.

١.١.٤ القواعد الثلاث الأساسية

- لكل قيمة مالك واحد فقط (يمتلك رابط المتغير القيمة).
- لا يمكن أن يوجد سوى مالك واحد في أي وقت. ويمكن نقل الملكية (حركة move).
- عند خروج المالك من النطاق، يتم إسقاط القيمة. تُحرَّر الموارد بشكل حتمي.

٢.١.٤ النطاق والتنظيف الحتمي

```
fn main() {
    let s = String::from("hello"); // s owns the heap allocation
    println!("{s}");
} // s goes out of scope here, String::drop runs, memory is released
```

تستخدم Rust نموذج تدمير حتميًا يشبه من حيث الفكرة مبدأ RAII في C++: فعند خروج القيمة من نطاقها، يُستدعى المُدمِّر (Drop) تلقائيًا.

٣.١.٤ نقل الملكية (الحركة)

تُنقل ملكية العديد من الأنواع (بما في ذلك String و Vec<T> ومعظم التراكيب) افتراضيًا عند الإسناد أو عند تمريرها إلى دالة.

```
fn main() {
    let a = String::from("data");
    let b = a;              // move: ownership transfers to b
    // println!("{a}");      // error: a was moved
    println!("{b}");
}
```

٢.٤ الحركة مقابل النسخ

تُميز Rust بين الحركة (move) والنسخ (copy). ويكون هذا التمييز ساكنًا وصريحًا ضمن نظام الأنواع.

١.٢.٤ أنواع Copy

الأنواع التي يكون نسخها رخيصًا وآمنًا تطبّق السمة Copy. وبالنسبة لأنواع Copy، فإن الإسناد ينسخ البتّات، وتظل الروابط جميعها صالحة للاستخدام.
تشمل أنواع Copy الشائعة: الأعداد الصحيحة، والأعداد العشرية، والقيم المنطقية، والمحارف، والأزواج المكوّنة من أنواع Copy.

```
fn main() {
    let x: i32 = 7;
```

```
    let y = x;            // copy
    println!("{x} {y}"); // both valid
}
```

٢.٣.٤ الأنواع غير القابلة للنسخ (أنواع الحركة)

الأنواع التي تمتلك تخصيصات على الكومة أو تدير موارد تكون عادةً غير Copy. وتُنقل افتراضيًا لمنع أخطاء التحرير المزدوج والتشارك غير الآمن.

```
fn main() {
    let v = vec![1, 2, 3];
    let w = v;            // move
    // println!("{:?}", v); // error: moved
    println!("{:?}", w);
}
```

٣.٣.٤ الاستنساخ يكون صريحًا

عند الحاجة إلى نسخة عميقة، يُستخدم clone() (عندما يكون مدعومًا). ويكون الاستنساخ صريحًا حتى لا تُخفى كلفته.

```
fn main() {
    let a = String::from("abc");
    let b = a.clone();     // heap allocation copy
    println!("{a} {b}");
}
```

٤.٢.٤ الحركة عبر استدعاءات الدوال

تمرير قيمة غير Copy بالقيمة يؤدي إلى نقلها إلى الدالة:

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

fn main() {
    let s = String::from("hello");
    consume(s);            // move into consume
    // println!("{s}");    // error: moved
}
```

للاستمرار في استخدام القيمة بعد الاستدعاء، يجب تمرير مرجع بدلاً من ذلك.

٣.٤ الاستعارة والقابلية للتغيير

تسمح الاستعارة بالوصول إلى قيمة دون امتلاكها. وتكون المراجع إما غير قابلة للتغيير (&T) أو قابلة للتغيير (&mut T). وتفرض Rust قواعد التشارك والتغيير في وقت الترجمة.

١.٣.٤ قواعد الاستعارة

- يمكن وجود أي عدد من المراجع غير القابلة للتغيير للقيمة (&T).
- يمكن وجود مرجع واحد فقط قابل للتغيير للقيمة (&mut T).
- لا يمكن وجود مرجع قابل للتغيير أثناء وجود مراجع غير قابلة للتغيير.
- يجب أن تكون المراجع صالحة دائمًا (لا توجد مراجع متدلية).

تضمن هذه القواعد ألا يحدث التغيير بالتزامن مع عمليات قراءة عبر مراجع مشتركة.

٢.٣.٤ الاستعارة غير القابلة للتغيير

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

fn main() {
    let s = String::from("hello");
    let n = len_of(&s); // borrow: s is still owned by main
    println!("{s} {n}");
}
```

ومن التحسينات الشائعة الاستعارة بصيغة أعم &str بدلاً من &String:

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

fn main() {
    let s = String::from("hello");
    println!("{}", len_of(&s));
    println!("{}", len_of("literal"));
}
```

٣.٣.٤ الاستعارة القابلة للتغيير

تتطلب المراجع القابلة للتغيير أن يكون رابط المالك مُعلناً باستخدام mut:

```
fn push_exclamation(s: &mut String) {
    s.push('!');
```

```
}

fn main() {
    let mut s = String::from("hi");
    push_exclamation(&mut s);
    println!("{s}");
}
```

٤.٣.٤ عدم الجمع بين الاستعارة القابلة وغير القابلة للتغيير

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

    let r1 = &s;          // immutable borrow
    let r2 = &s;          // another immutable borrow
    println!("{r1} {r2}");

    let r3 = &mut s;      // mutable borrow allowed only after r1/r2 are no
    ↪  longer used
    r3.push('!');
    println!("{r3}");
}
```

الفكرة الأساسية هي أعمار مبنية على الاستخدام: تنتهي الاستعارات غير القابلة للتغيير بعد آخر استخدام لها، وليس بالضرورة عند نهاية النطاق.

٥.٣.٤ الشرائح: استعارة أجزاء من المجموعات

تتيح الشرائح استعارة مقاطع من التسلسلات دون نسخ:

```rust
fn first_two(v: &[i32]) -> &[i32] {
    &v[0..2]
}

fn main() {
    let v = vec![10, 20, 30, 40];
    let a = first_two(&v);
    println!("{:?}", a);
}
```

تدعم السلاسل النصية شرائح &str، لكن يجب أن تحترم الفهرسة حدود UTF-8:

```rust
fn main() {
    let s = String::from("hello");
    let sub: &str = &s[0..2]; // ok: ASCII, valid UTF-8 boundaries
    println!("{sub}");
}
```

٦.٣.٤ إرجاع المراجع يتطلب أعمارًا صالحة

تمنع Rust إرجاع مرجع إلى قيمة محلية:

```rust
fn bad_ref() -> &String {
    let s = String::from("no");
    &s
}
```

يفشل هذا لأن s تُسقط عند خروج الدالة. وتجبرك Rust على إرجاع قيمة مملوكة أو الاستعارة من مُدخل.
الأساليب الصحيحة:

```rust
fn make_string() -> String {
    String::from("ok")
}

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

٤.٤ أخطاء شائعة لدى المبتدئين

أكثر الأخطاء شيوعاً لدى المبتدئين ليست عيوباً؛ بل هي حالات يمنع فيها المترجم أخطاء حقيقية. ومع الممارسة، تصبح هذه الأنماط طبيعية.

١.٤.٤ خطأ: النقل عندما كان المقصود الاستعارة

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

fn main() {
    let s = String::from("hello");
    print_len(s);
    // println!("{s}"); // error: moved
}
```

التصحيح: استخدام الاستعارة:

```rust
fn print_len(s: &str) {
```

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

fn main() {
    let s = String::from("hello");
    print_len(&s);
    println!("{s}");
}
```

٢.٤.٤ خطأ: محاولة التغيير عبر مرجع غير قابل للتغيير

```
fn add_x(s: &String) {
    // s.push('x'); // error: cannot borrow as mutable
}
```

التصحيح: استخدام &mut:

```
fn add_x(s: &mut String) {
    s.push('x');
}
```

٣.٤.٤ خطأ: الاحتفاظ باستعارة أثناء تغيير المالك

مثال شائع يتمثل في استعارة عنصر ثم الدفع إلى المتجه نفسه:

```
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}");
}
```

السبب: قد تؤدي عملية الدفع إلى إعادة تخصيص المتجه، مما يُبطل first. ويتم التصحيح عبر حصر نطاق الاستعارة:

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

    {
        let first = v[0];
        println!("{first}");
    } // borrow ends here

    v.push(4);
    println!("{:?}", v);
}
```

٤.٤.٤ خطأ: افتراض أن المراجع يمكن أن تعيش أطول من المالِك

```
fn main() {
    let r: &str;
    {
        let s = String::from("hello");
        // r = &s; // error: borrowed value does not live long enough
    }
    // println!("{r}");
}
```

التصحيح: نقل المالك إلى نطاق خارجي، أو إرجاع بيانات مملوكة.

٥.٤.٤ خطأ: الاستنساخ غير الضروري

يلجأ المبتدئون كثيرًا إلى clone() لإرضاء مدقق الاستعارة. يعمل ذلك لكنه قد يسبب تخصيصات يمكن تجنبها. يُفضَّل الاستعارة ونقل الملكية بوضوح، مع الاستنساخ فقط عند الحاجة الفعلية للتكرار.

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

    // Prefer borrowing:
    let n = s.len();
    println!("{s} {n}");

    // Clone only when required:
    let a = s.clone();
    println!("{a}");
}
```

٥.٤ ملخص مُكثف

- تضمن الملكية وجود مالك واحد مسؤول عن كل قيمة مع تنظيف حتمي للموارد.
- تُنسخ أنواع Copy عند الإسناد؛ بينما تُنقل معظم الأنواع المديرة للموارد افتراضيًا.
- توفر الاستعارة وصولاً دون نقل الملكية، مع قواعد صارمة للتشارك والقابلية للتغيير.
- معظم أخطاء المبتدئين هي أخطاء مُنعت قبل وقوعها: مراجع غير صالحة، وتشارك مع تغيير، وسوء استخدام للموارد.

الفصل ٥: المراجع وأعمارها

١.٥ المراجع في التطبيق العملي

تتيح المراجع الوصول إلى البيانات دون امتلاكها. في Rust، تكون المراجع دائمًا صالحة، وغير فارغة، ويقوم المترجم بالتحقق من استخدامها الآمن. ويوجد شكلان رئيسيان للمراجع:

- `&T`: مرجع مشترك (غير قابل للتغيير)
- `&mut T`: مرجع حصري (قابل للتغيير)

١.١.٥ المراجع المشتركة (&T)

تسمح المراجع المشتركة بالوصول للقراءة فقط. ويمكن وجود عدة مراجع مشتركة للقيمة نفسها في الوقت ذاته.

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

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

```
    let a = &s;          // &String coerces to &str where needed
    let b = &s;
    print_len(a);
    print_len(b);
    println!("{s}");
}
```

٢.١.٥ المراجع القابلة للتغيير (&mut T)

تسمح المراجع القابلة للتغيير بالتعديل ويجب أن تكون حصرية: مرجع واحد فقط من نوع &mut في أي وقت، ولا يجوز وجود مراجع مشتركة بالتزامن معه.

```
fn push_mark(s: &mut String) {
    s.push('!');
}

fn main() {
    let mut s = String::from("hi");
    push_mark(&mut s);
    println!("{s}");
}
```

٣.١.٥ نطاق الاستعارة ينتهي عند آخر استخدام

تستخدم Rust مفهوم الأعمار المعتمدة على الاستخدام: تنتهي الاستعارة بعد آخر استخدام لها، وليس بالضرورة عند نهاية الكتلة البرمجية.

```
fn main() {
```

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

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

    let m = &mut s;       // allowed after r is no longer used
    m.push('!');
    println!("{m}");
}
```

٤.١.٥ مراجع لأجزاء من البيانات: الشرائح

الشرائح هي مراجع لنطاق متصل داخل مجموعة بيانات.

```
fn head(v: &[i32]) -> &[i32] {
    &v[..2]
}

fn main() {
    let v = vec![10, 20, 30, 40];
    println!("{:?}", head(&v));
}
```

بالنسبة للسلاسل النصية، يجب أن تحترم عملية التقسيم حدود UTF-8، ولا يُسمح بالفهرسة المباشرة.

```
fn main() {
    let s = String::from("hello");
    let sub = &s[0..2];
    println!("{sub}");
}
```

٢.٥ لماذا توجد الأعمار

العمر (lifetime) هو النطاق الذي يكون فيه المرجع صالحًا. وتوجد الأعمار كي يتمكن المترجم من ضمان ما يلي:

- ألا يتجاوز أي مرجع عمر البيانات التي يشير إليها،
- عدم وجود مراجع متدلية،
- فرض قواعد التشارك الآمن عند التغيير.

لا تتعلق الأعمار بقياس الزمن أثناء التشغيل، بل هي علاقات تُحل في وقت الترجمة بين المراجع والقيم المملوكة.

١.٢.٥ المشكلة الجوهرية: المراجع المتدلية

تمنع Rust إرجاع مرجع إلى قيمة محلية، لأن القيمة المحلية تُسقط عند خروج الدالة.

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

الأساليب الصحيحة هي إرجاع قيمة مملوكة، أو الاستعارة من مُدخل.

```
fn good_owned() -> String {
    String::from("owned")
}

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

٢.٢.٥ الأعمار تُعبّر عن العلاقات

عندما تُعيد دالة مرجعًا، يحتاج Rust إلى معرفة أي مرجع مُدخل يرتبط به المرجع المُعاد. وتُستخدم الأعمار لترميز هذه العلاقة صراحةً عندما لا يكون الاستدلال كافيًا.

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

في هذا المثال، يُضمن أن يكون المرجع المُعاد صالحًا طالما أن كلا المُدخلين صالحان، لأن عمر الإخراج 'a مقيَّد بأعمار المُدخلات.

٣.٥ استدلال المترجم مقابل الأعمار الصريحة

تستخدم معظم شيفرات Rust المراجع دون كتابة الأعمار صراحةً، وذلك لأن المترجم يطبق قواعد حذف الأعمار والاستدلال التلقائي.

١.٣.٥ حذف الأعمار (الحالات الشائعة)

يمكن لـ Rust استنتاج الأعمار في الأنماط الشائعة:

- إذا وُجد مرجع مُدخل واحد فقط، يُسند عمره إلى المرجع المُعاد.
- إذا وُجدت عدة مراجع مُدخلة دون مرجع مُعاد، تُستنتج الأعمار بشكل مستقل.
- في الطرائق، غالبًا ما يزوّد &self (أو &mut self) عمر الإخراج عند إرجاع مرجع مرتبط بالكائن نفسه.

مثال يعمل فيه الحذف تلقائيًا:

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

ويتعامل المترجم معه كما لو كان:

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

٢.٣.٥ متى تكون الأعمار الصريحة مطلوبة

إذا أخذت دالة عدة مراجع وأعادت أحدها، فلا يستطيع Rust معرفة المرجع المرتبط بالإخراج دون تحديد صريح.

```
fn pick(a: &str, b: &str) -> &str {
    if a.len() >= b.len() { a } else { b }
}
```

يفشل هذا لأن الإخراج قد يرتبط بـ a أو b. والتصحيح هو:

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

٣.٣.٥ التراكيب التي تخزن مراجع

إذا احتوى تركيب على مراجع، فيجب عليه التصريح بمعاملات أعمار لضمان عدم تجاوز عمر التركيب لعمر البيانات المشار إليها.

```
struct View<'a> {
    data: &'a [u8],
}

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

٤.٣.٥ الطرائق وإرجاع مراجع مرتبطة بـ self

يُعد إرجاع مرجع إلى بيانات يملكها التركيب أمرًا شائعًا، وغالبًا ما يُستدل عمره تلقائيًا:

```
struct Buffer {
    data: Vec<u8>,
}

impl Buffer {
    fn as_slice(&self) -> &[u8] {
        &self.data
    }
}
```

في هذه الحالة، يكون المرجع المُعاد مرتبطًا بعمر &self.

٥.٣.٥ أخطاء الأعمار الشائعة ومعناها

رسائل المترجم الشائعة:

- **borrowed value does not live long enough**: المرجع يعيش أطول من مالكه.
- **cannot return reference to local variable**: سيؤدي ذلك إلى مرجع متدلٍ.
- **lifetime may not live long enough**: العلاقة بين المُدخلات والإخراج غير واضحة أو غير ممكنة.

تُعد هذه الرسائل إشارة لإعادة تصميم الملكية:

- إرجاع بيانات مملوكة (مثل `Vec<T>` و `String`)،
- الاستعارة من المُدخلات،
- تخزين بيانات مملوكة داخل التراكيب بدلاً من المراجع عند الاقتضاء.

٤.٥ ملخص مُكثف

- تُمكّن المراجع من الوصول دون نقل الملكية: `&T` للقراءة المشتركة، و`&mut T` للتعديل الحصري.
- وُجدت الأعمار لضمان ألا تتجاوز المراجع عمر البيانات التي تشير إليها، ولتطبيق قواعد التشارك الآمن.
- يستدل المترجم على معظم الأعمار تلقائيًا؛ وتكون الأعمار الصريحة مطلوبة عند إرجاع مراجع مرتبطة بأحد عدة مُدخلات أو عند تخزين مراجع داخل التراكيب.

الفصل ٦: أنواع البيانات ونمذجتها

١.٦ التراكيب والتعدادات

تشجّع Rust على نمذجة مفاهيم المجال باستخدام أنواع الضرب (product types – التراكيب) وأنواع الجمع (sum types – التعدادات). ويجعل هذا النهج تمثيل الحالات غير الصحيحة أصعب، ويُمكّن المترجم من فرض الصحة عبر نظام الأنواع.

١.١.٦ التراكيب: تجميع البيانات المرتبطة

تجمع struct الحقول المسماة في نوع واحد. وتُعد التراكيب الأداة الأساسية لنمذجة السجلات وكائنات البيانات.

```
#[derive(Debug, Clone, Copy)]
struct Point {
    x: i32,
    y: i32,
}

fn main() {
    let p = Point { x: 10, y: 20 };
```

```
    println!("{p:?}");
}
```

أفكار أساسية:

- فضّل الأنواع الدلالية الواضحة على "أكياس البدائيات".
- استخدم #[derive(...)] لتوليد السمات القياسية عند الاقتضاء.
- اجعل الحقول خاصة افتراضيًا، واكشف فقط الواجهات المقصودة.

٢.١.٦ تراكيب الأزواج والتراكيب أحادية الشكل

توفر تراكيب الأزواج أنواعًا مسماة بحقول موضعية:

```
struct UserId(u64);
struct Celsius(f64);

fn main() {
    let id = UserId(7);
    let t = Celsius(36.5);
    let _ = (id.0, t.0);
}
```

أما التراكيب أحادية الشكل فهي مفيدة كعلامات وأنواع ذات حجم صفري:

```
struct Admin;

fn is_admin(_: Admin) -> bool { true }
```

٣.١.٦ التعدادات: تمثيل البدائل

تمثل التعدادات قيَمًا يمكن أن تكون إحدى عدة حالات. ويمكن للحالات حمل بيانات، مما يتيح نمذجة تعبيرية دون الحاجة إلى الوراثة.

```
#[derive(Debug)]
enum Payment {
    Cash,
    Card { last4: u16 },
    Transfer { iban: String },
}

fn main() {
    let p = Payment::Card { last4: 1234 };
    println!("{p:?}");
}
```

تُعد التعدادات محورية في التصميم الآمن لـ Rust، بما في ذلك الأنواع القياسية مثل Option<T> و Result<T,E>.

٢.٦ مطابقة الأنماط مع التعدادات

تُعد مطابقة الأنماط الوسيلة الأساسية للتفرّع الآمن بناءً على حالات التعداد. وتكون match في Rust شاملة، مما يمنع إغفال أي حالة.

١.٢.٦ match مع التفكيك

```
#[derive(Debug)]
enum Message {
```

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

fn handle(m: Message) -> i32 {
    match m {
        Message::Quit => 0,
        Message::Move { x, y } => x + y,
        Message::Text(s) => s.len() as i32,
    }
}

fn main() {
    let a = Message::Move { x: 3, y: 4 };
    println!("{}", handle(a));
}
```

٢.٣.٦ أنماط مفيدة: `while let` و `if let`

عند الاهتمام بحالة واحدة والرغبة في شيفرة مختصرة:

```
fn main() {
    let maybe: Option<i32> = Some(10);

    if let Some(x) = maybe {
        println!("{x}");
    }
```

```
    let mut stack = vec![1, 2, 3];
    while let Some(v) = stack.pop() {
        println!("{v}");
    }
}
```

٣.٢.٦ الحراس والروابط

تضيف حراس match شروطًا إضافية، بينما تلتقط الروابط البيانات:

```
fn classify(n: i32) -> &'static str {
    match n {
        x if x < 0 => "negative",
        0 => "zero",
        x if x > 0 => "positive",
        _ => "unreachable",
    }
}
```

٣.٦ تصميم البيانات بأمان

تعني نمذجة البيانات الآمنة في Rust استخدام الأنواع لترميز الثوابت ومنع تمثيل الحالات غير الصحيحة.

١.٣.٦ اجعل الحالات غير الصحيحة غير قابلة للتمثيل

بدلاً من استخدام القيم المنطقية أو الأرقام السحرية أو المؤشرات القابلة للإفراغ، استخدم التعدادات والأنواع الجديدة.

```
enum Status {
    Draft,
    Published { at_unix: u64 },
}

struct Post {
    title: String,
    status: Status,
}
```

يمنع هذا تراكيب غير صحيحة مثل “منشور دون طابع زمني”.

٢.٣.٦ فضّل Option على القيم الدالّة

لا تستخدم السلاسل الفارغة أو -1 أو 0 كعلامات على “عدم وجود قيمة” عندما يكون الغياب ذا دلالة.

```
struct User {
    name: String,
    phone: Option<String>,
}

fn main() {
    let u = User { name: "A".to_string(), phone: None };
    match &u.phone {
        Some(p) => println!("{p}"),
        None => println!("no phone"),
    }
}
```

٣.٣.٦ استخدم Result للحالات القابلة للفشل

نمذج العمليات التي قد تفشل باستخدام Result<T,E> بدلاً من رموز الأخطاء.

```
use std::num::ParseIntError;

fn parse_port(s: &str) -> Result<u16, ParseIntError> {
    s.parse::<u16>()
}
```

٤.٣.٦ التغليف: اجعل الحقول خاصة وتحقق في البُنّاء

لفرض الثوابت، اجعل حقول التركيب خاصة، واكشف بُنّاءات وطرائق تتحقق من الصحة.

```
#[derive(Debug, Clone, Copy, PartialEq, Eq)]
struct NonZeroU32(u32);

impl NonZeroU32 {
    fn new(x: u32) -> Option<Self> {
        if x == 0 { None } else { Some(Self(x)) }
    }

    fn get(self) -> u32 {
        self.0
    }
}

fn main() {
    let v = NonZeroU32::new(10).unwrap();
    println!("{}", v.get());
```

```
}
```

يضمن هذا النمط ألا يحمل NonZeroU32 قيمة غير صالحة بعد إنشائه.

٥.٣.٦ اختر الملكية عن قصد

عند تصميم أنواع البيانات، قرر ما إذا كانت الحقول ستمتلك البيانات أم ستستعيرها:

- الحقول المالِكة (مثل String و Vec<T>) تُبسّط الأعمار وتجعل النوع أسهل للتخزين والنقل.
- الحقول المُستعيرة (مثل &str و &[T]) قد تتجنب التخصيص لكنها تتطلب معاملات أعمار وإدارة دقيقة للصلاحية.

خيار افتراضي شائع ومناسب للمبتدئين هو: امتلك البيانات داخل التراكيب ما لم يُثبت القياس العكسي الحاجة إلى الاستعارة.

٦.٣.٦ فضّل الأنواع الصغيرة القابلة للتركيب

بدلاً من تركيب كبير واحد يحوي العديد من الحقول الاختيارية والأعلام، قسّم المفاهيم إلى أنواع أصغر وركّبها معًا.

```
struct Email(String);
struct UserName(String);

struct Account {
    name: UserName,
    email: Email,
}
```

٤.٦ ملخص مُكثف

- استخدم التراكيب لنمذجة السجلات والحالة المملوكة؛ واستخدم التعدادات لنمذجة البدائل والحالات.
- استخدم match لمعالجة حالات التعداد بشكل شامل؛ واستخدم `if let` للحالات المركّزة.
- صمّم للأمان بترميز الثوابت داخل الأنواع، واستخدام Option/Result، وفرض التحقق عبر البُنّاءات والتغليف.

الفصل ٧: معالجة الأخطاء بأسلوب Rust

١.٧ Option و Result

تنمذج Rust غياب القيم والعمليات القابلة للفشل بشكل صريح داخل نظام الأنواع. ويحل هذا محل أنماط شائعة مثل المؤشرات الفارغة، والقيم الدالّة، ورموز الأخطاء غير المُتحقق منها.

١.١.٧ Option<T>: التعبير عن الغياب

يمثل Option<T> إما Some(T) أو None. ويُستخدم عندما تكون القيمة قد تكون مفقودة دون أن يُعد ذلك حالة استثنائية.

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

```rust
fn main() {
    let v = [1, 3, 5, 8, 9];
    match find_first_even(&v) {
        Some(x) => println!("found {x}"),
        None => println!("no even value"),
    }
}
```

أنماط شائعة الاستخدام:

- map: تحويل القيمة المحتواة
- and_then: تسلسل عمليات قد تُعيد Option
- unwrap_or / unwrap_or_else: توفير قيم افتراضية

```rust
fn main() {
    let maybe_port: Option<&str> = Some("8080");

    let port: u16 = maybe_port
        .and_then(|s| s.parse::<u16>().ok())
        .unwrap_or(80);

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

٢.١.٧ التعبير عن الفشل: Result<T, E>

يمثل Result<T, E> إما Ok(T) أو Err(E). ويُستخدم للعمليات التي قد تفشل، خصوصًا الإدخال/الإخراج، والتحليل، والشبكات، والتحقق من الصحة.

```rust
use std::fs;

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

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

خصائص أساسية:

- يجب على المستدعي معالجة مسار الخطأ صراحةً،
- يمكن للأخطاء حمل معلومات مُهيكلة،
- تصبح الواجهات ذاتية التوثيق: يصرّح التوقيع بإمكانية الفشل.

٢.٧ عامل ?

يُعد عامل ? الآلية القياسية في Rust لتمرير الأخطاء. ويُطبّق على Result (وكذلك على Option) ويُعيد مبكرًا من الدالة الحالية عندما تكون القيمة خطأً أو غيابًا.

١.٢.٧ تمرير أخطاء Result

```rust
use std::fs;
```

```rust
fn config_len(path: &str) -> Result<usize, std::io::Error> {
    let s = fs::read_to_string(path)?; // if Err, returns immediately
    Ok(s.len())
}
```

وهذا مكافئ دلاليًا للمطابقة الصريحة:

```rust
use std::fs;

fn config_len(path: &str) -> Result<usize, std::io::Error> {
    let s = match fs::read_to_string(path) {
        Ok(v) => v,
        Err(e) => return Err(e),
    };
    Ok(s.len())
}
```

٢.٢.٧ تمرير Option باستخدام ?

```rust
fn second(v: &[i32]) -> Option<i32> {
    let _first = v.get(0)?;  // returns None if missing
    let s = v.get(1)?;       // returns None if missing
    Some(*s)
}
```

٣.٢.٧ دمج عدة خطوات بشكل نظيف

```rust
use std::fs;
```

```rust
fn read_and_parse_u32(path: &str) -> Result<u32, Box<dyn std::error::Error>>
↪  {
    let s = fs::read_to_string(path)?;
    let n: u32 = s.trim().parse()?;
    Ok(n)
}
```

تستخدم الدالة أعلاه نوع خطأ واسعًا للتبسيط. في المكتبات الحقيقية، يُفضَّل عادةً استخدام تعداد أخطاء أكثر تحديدًا (سيُعرض لاحقًا في هذا الفصل).

٣.٧ تصميم الأخطاء القابلة للاسترداد

تميّز Rust بين الأخطاء القابلة للاسترداد (المُمثّلة بـ Result) وبين حالات الفشل غير القابلة للاسترداد (وغالبًا ما تكون panic!). ويصمّم الكود الاحترافي الأخطاء بحيث يستطيع المستدعي اتخاذ القرار المناسب: إعادة المحاولة، أو التراجع، أو الإبلاغ، أو الإنهاء.

١.٣.٧ استخدم panic! نادرًا

استخدم panic! فقط عندما:

- يُنتهك ثابت جوهري ويكون الاستمرار غير آمن أو غير منطقي،
- لا يمكن للبرنامج المتابعة (مثل تلف المنطق الداخلي)،
- تكون بصدد كتابة أمثلة أو اختبارات يُفضَّل فيها الاختصار.

بالنسبة لإدخال المستخدم، والإدخال/الإخراج، والتحليل، وفشل الشبكات، فَضِّل Result.

٢.٣.٧ فضّل أنواع أخطاء خاصة بالمجال في المكتبات

للكود القابل لإعادة الاستخدام، عرّف تعداد أخطاء يمثّل فئات فشل ذات معنى. يحافظ هذا على المعلومات ويجعل المعالجة مريحة.

```
use std::fmt;

#[derive(Debug)]
pub enum ConfigError {
    Io(std::io::Error),
    MissingField(&'static str),
    InvalidNumber(std::num::ParseIntError),
}

impl fmt::Display for ConfigError {
    fn fmt(&self, f: &mut fmt::Formatter<'_>) -> fmt::Result {
        match self {
            ConfigError::Io(e) => write!(f, "I/O error: {e}"),
            ConfigError::MissingField(k) => write!(f, "missing field: {k}"),
            ConfigError::InvalidNumber(e) => write!(f, "invalid number:
            ↪ {e}"),
        }
    }
}

impl std::error::Error for ConfigError {}

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

```
impl From<std::num::ParseIntError> for ConfigError {
    fn from(e: std::num::ParseIntError) -> Self {
    ↪  ConfigError::InvalidNumber(e) }
}
```

ومع تحويلات From، يصبح استخدام ? سلسًا:

```
use std::fs;

fn parse_timeout(s: &str) -> Result<u64, ConfigError> {
    let n: u64 = s.trim().parse()?; // ParseIntError -> ConfigError via From
    Ok(n)
}

fn load_timeout(path: &str) -> Result<u64, ConfigError> {
    let txt = fs::read_to_string(path)?; // io::Error -> ConfigError via From
    parse_timeout(&txt)
}
```

٣.٣.٧ أنواع الإرجاع: إرشادات عملية

- يمكن لتطبيقات المستخدم استخدام نوع خطأ أوسع للتبسيط عندما يكون الهدف الأساسي هو الإبلاغ.
- ينبغي للمكتبات كشف نوع خطأ محدد لتمكين المعالجة الدقيقة.
- فَضِّل إرجاع `Result<T, E>` من الدوال القابلة للفشل، وتجنّب إخفاء الإخفاقات.

٤.٣.٧ لا تفقد سياق الخطأ

أضِف سياقًا للأخطاء عندما يُحسّن ذلك التصحيح والتشخيص. وحتى دون استخدام مكتبات خارجية، يمكن إضافة السياق بتحويل الأخطاء إلى متغيرات أغنى:

```
#[derive(Debug)]
pub enum LoadError {
    Io { path: String, source: std::io::Error },
}

impl std::fmt::Display for LoadError {
    fn fmt(&self, f: &mut std::fmt::Formatter<'_>) -> std::fmt::Result {
        match self {
            LoadError::Io { path, source } => write!(f, "failed to read
            ↪ {path}: {source}"),
        }
    }
}

impl std::error::Error for LoadError {}

fn read_with_path(path: &str) -> Result<String, LoadError> {
    std::fs::read_to_string(path).map_err(|e| LoadError::Io {
        path: path.to_string(),
        source: e,
    })
}
```

٤.٧ ملخص مُكثف

- استخدم `Option<T>` للتعبير عن الغياب و `Result<T,E>` للتعبير عن الفشل القابل للاسترداد.
- استخدم `?` لتمرير الأخطاء بشكل نظيف ومتّسق.
- صمّم الأخطاء القابلة للاسترداد بحيث يستطيع المستدعي اتخاذ القرار؛ وفضّل تعدادات أخطاء خاصة بالمجال في المكتبات.
- تجنّب `panic!` في حالات الفشل المتوقعة مثل الإدخال/الإخراج أو التحليل أو أخطاء إدخال المستخدم.

الفصل ٨: أساسيات إدارة الذاكرة

١.٨ المكدس مقابل الكومة

تستخدم Rust نموذج ذاكرة مألوفًا لمبرمجي الأنظمة: يمكن للقيم أن تعيش على المكدس (stack) أو على الكومة (heap). ولا يقتصر الفرق الجوهري على الأداء فقط، بل يشمل الملكية، والعمر، وسلوك تغيير الحجم.

١.١.٨ المكدس

يخزن المكدس:

- القيم ذات الحجم الثابت المعروف في وقت الترجمة،
- إطارات استدعاء الدوال (المتغيرات المحلية، عناوين العودة، السجلات المحفوظة)،
- القيم ذات العمر التلقائي المرتبط بنطاق التنفيذ.

يتميز التخصيص على المكدس بالسرعة والحتمية. وعند انتهاء النطاق، تُستعاد مساحة المكدس تلقائيًا.

```
fn main() {
    let a: i32 = 7;             // stack
```

```
    let b: (i32, i32) = (1, 2); // stack (fixed-size tuple)
    println!("{a} {:?}", b);
}
```

٢.١.٨ الكومة

تخزن الكومة:

- القيم ذات الحجم الديناميكي (مثل السلاسل النصية والمتجهات)،
- القيم الكبيرة التي ترغب في تجنب نسخها على المكدس،
- البيانات التي يجب أن تعيش أطول من إطار مكدس واحد.

تُدار تخصيصات الكومة في Rust صراحةً عبر أنواع مالِكة (مثل Box وVec وString). ولا تستخدم Rust جامع قمامة؛ بل تعتمد على الملكية والتنظيف الحتمي.

٢.٨ Box و Vec و String

تُعد هذه الأنواع المالِكة القياسية أهم أنواع الكومة للمبتدئين. ويُعد فهم ملكيتها، وتخطيط الذاكرة، وعملياتها الشائعة أمرًا أساسيًا.

١.٢.٨ Box<T>: امتلاك تخصيص واحد على الكومة

يخصص `Box<T>` قيمة واحدة من النوع `T` على الكومة ويمتلكها. أما الصندوق نفسه فهو قيمة صغيرة شبيهة بالمؤشر تُخزَّن عادةً على المكدس.
استخدامات شائعة:

- نقل القيم الكبيرة دون نسخها،

- بناء أنواع تكرارية (حيث يجب أن يكون الحجم معروفًا)،
- نقل الملكية عبر حدود الواجهات البرمجية.

```
fn main() {
    let b = Box::new(123u64); // heap allocation for the u64
    println!("{b}");
}
```

مثال على نوع تكراري:

```
enum List {
    Nil,
    Node(i32, Box<List>),
}

fn main() {
    let xs = List::Node(1, Box::new(List::Node(2, Box::new(List::Nil))));
    let _ = xs;
}
```

٢.٢.٨ `Vec<T>`: مصفوفة متجاورة قابلة للنمو

`Vec<T>` هو المتجه القياسي القابل للنمو في Rust:

- تُخزَّن العناصر بشكل متجاور على الكومة،
- يمتلك المتجه تخصيصه،
- ينمو عبر إعادة التخصيص عند تجاوز السعة.

تحتوي قيمة `Vec<T>` منطقيًا على:

- مؤشر إلى ذاكرة الكومة،
- طول (عدد العناصر المهيأة)،
- سعة (المساحة المخصصة).

```
fn main() {
    let mut v: Vec<i32> = Vec::new();
    v.push(10);
    v.push(20);
    println!("{:?}", v);
}
```

يقلل التخصيص المسبق للسعة من عمليات إعادة التخصيص:

```
fn main() {
    let mut v = Vec::with_capacity(100);
    for i in 0..100 {
        v.push(i);
    }
    println!("len={}, cap={}", v.len(), v.capacity());
}
```

ملاحظة أمان مهمة: قد تؤدي عملية `push` إلى إعادة التخصيص، مما يُبطل المراجع إلى العناصر. تمنع Rust الأنماط غير الآمنة عبر فرض قواعد الاستعارة.

```
fn main() {
    let mut v = vec![1, 2, 3];
    let first = v[0];
    v.push(4);
    println!("{first} {:?}", v);
}
```

٣.٢.٨ String: مخزن نصي مملوك بترميز UTF-8

String هو مخزن نصي مملوك وقابل للنمو بترميز UTF-8، مبني فوق Vec<u8>. ويمتلك تخصيصه على الكومة ويمكنه النمو.

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

تستخدم الشرائح النصية المُستعارة النوع &str:

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

fn main() {
    let s = String::from("world");
    greet(&s);
    greet("everyone");
}
```

وبما أن السلاسل النصية مُرمَّزة بـ UTF-8، فلا يُسمح بالفهرسة الموضعية المباشرة؛ استخدم المُكرِّرات، أو التقسيم على حدود صالحة، أو التحويل الصريح إلى بايتات عند الاقتضاء.

٣.٨ الإسقاط والتنظيف الحتمي

تستخدم Rust تنظيفًا حتميًا: عندما يخرج المالِك من النطاق، يُنفَّذ المُدمِّر تلقائيًا وتُحرَّر الموارد. وهذا مضمون من اللغة ولا يعتمد على جامع قمامة.

١.٣.٨ الإسقاط عند نهاية النطاق

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

٢.٣.٨ الإسقاط المبكر

يمكنك إنهاء الملكية صراحةً مبكرًا باستخدام drop:

```
fn main() {
    let s = String::from("temp");
    drop(s); // resources released here
    // println!("{s}"); // error: use after move (s was dropped)
}
```

٣.٣.٨ تنظيف مخصص باستخدام Drop

يمكن للأنواع تنفيذ Drop لتشغيل منطق تنظيف مخصص. ويُستخدم ذلك لإدارة موارد مثل الملفات، والأقفال، والمقابس، ومخازن الذاكرة.

```
struct Guard;

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

```rust
fn main() {
    let _g = Guard;
    println!("work");
} // drop runs automatically here
```

٤.٣.٨ التنظيف الحتمي والسلامة

يُمكّن التنظيف الحتمي من:

- تحرير متوقع للذاكرة وموارد نظام التشغيل،
- أنماط آمنة لإدارة الموارد،
- تركيب موثوق للتجريدات دون توقفات خفية وقت التشغيل.

يُعد هذا النموذج بالغ الأهمية في برمجة الأنظمة، حيث يجب التحكم بتسرب الموارد، والتوقفات غير المتوقعة، ومخاطر التزامن.

٤.٨ ملخص مُكثف

- يخزن المكدس القيم ثابتة الحجم ذات عمر مرتبط بالنطاق؛ وتخزن الكومة البيانات الديناميكية أو الكبيرة المُدارة بواسطة أنواع مالِكة.
- يمتلك `Box<T>` تخصيصًا واحدًا على الكومة؛ ويمتلك `Vec<T>` مخزنًا متجاورًا قابلاً للنمو؛ ويمتلك `String` نصًا قابلاً للنمو بترميز UTF-8.
- تُجري Rust تنظيفًا حتميًا: تُسقَط القيم عند نهاية النطاق، ويُمكّن `Drop` من إدارة الموارد بأمان دون جامع قمامة.

الفصل ٩: الوحدات، والحِزم، وتنظيم الشيفرة

١.٩ الحِزم كوحدات ترجمة

تُعد الحِزمة (crate) وحدة الترجمة في Rust. وتكون الحِزمة إما:

- حِزمة تنفيذية (binary crate) تُنتج ملفًا تنفيذيًا،
- حِزمة مكتبة (library crate) تُنتج مكتبة قابلة للإعادة الاستخدام.

يمكن لحزمة Cargo واحدة (package) أن تحتوي على:

- صفر أو حِزمة مكتبة واحدة (src/lib.rs),
- حزمة تنفيذية واحدة أو أكثر (src/main.rs و/أو src/bin/*.rs).

```
my_pkg/
  Cargo.toml
  src/
    lib.rs        (optional)
    main.rs       (optional)
    bin/
      tool.rs     (optional)
```

نصائح عملية:

- اجعل main.rs صغيرًا، وفوّض المنطق إلى حِزمة مكتبة (lib.rs) عندما يكبر المشروع.
- عامِل lib.rs على أنه "المحرّك"، والحِزم التنفيذية على أنها "واجهات أمامية".

٢.٩ الوحدات ونطاق الرؤية

توفر الوحدات (modules) مساحات أسماء وتتحكم في نطاق الرؤية. وتُستخدم لتنظيم الشيفرة في وحدات مترابطة ولفرض مبدأ التغليف.

١.٢.٩ التصريح عن الوحدات

يمكن التصريح عن وحدة ضمنية داخل الملف:

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

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

أو التصريح عنها في ملفات منفصلة (وهو الأسلوب الشائع في المشاريع الحقيقية):

```
src/
  lib.rs
  math.rs
```

```rust
// src/lib.rs
mod math;

pub fn demo() -> i32 {
    math::add(1, 2)
}
```

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

٢.٢.٩ قواعد نطاق الرؤية

نطاق الرؤية في Rust صريح وواضح:

- العناصر خاصة افتراضيًا.
- pub تجعل العنصر عامًا للكود الخارجي (مع مراعاة مسار الوحدة).
- pub(crate) تقيد الرؤية داخل الحِزمة الحالية فقط.
- pub(super) تقيد الرؤية للوحدة الأب.
- pub(in path) تقيد الرؤية لمسار وحدات محدد.

```rust
pub fn public_api() {}

fn internal_helper() {}

pub(crate) fn crate_only() {}
```

```
mod inner {
    pub(super) fn parent_only() {}
}
```

عادة تصميمية مفيدة:

- اكشف فقط ما تنوي اعتباره واجهة مستقرة،
- أبقِ تفاصيل التنفيذ خاصة،
- استخدم pub(crate) للمشاركة داخل الحِزمة مع حماية المستخدمين الخارجيين من التفاصيل الداخلية.

٣.٢.٩ المسارات والاستيراد

تدعم Rust المسارات المطلقة من جذر الحِزمة والمسارات النسبية داخل الوحدات. ويُعد استخدام use الأسلوب الاصطلاحي للاستيراد:

```
mod a {
    pub fn f() -> i32 { 1 }
}

mod b {
    use crate::a::f;

    pub fn g() -> i32 { f() + 1 }
}
```

ويُعد إعادة التصدير باستخدام pub use شائعًا لبناء سطح عام نظيف:

```
mod net {
```

```
    pub struct Client;
}

pub use net::Client; // users can write `mycrate::Client`
```

٣.٩ تصميم الواجهات العامة

ينبغي أن تكون الواجهة العامة:

- محدودة ومقصودة،
- مستقرة في التسمية والدلالات،
- آمنة للاستخدام الصحيح بطبيعتها.

١.٣.٩ اكشف الأنواع وأخفِ الحقول

فَضِّل الحقول الخاصة مع بُنّاءات وطرائق عامة للحفاظ على الثوابت.

```
pub struct Port(u16);

impl Port {
    pub fn new(p: u16) -> Option<Self> {
        if p == 0 { None } else { Some(Self(p)) }
    }

    pub fn get(self) -> u16 {
        self.0
    }
}
```

يمنع هذا إنشاء حالات غير صحيحة من قِبل كود خارجي.

٢.٣.٩ أنواع الأخطاء جزء من الواجهة

إذا كانت الدالة قد تفشل، فأعِد `Result<T, E>`. وبالنسبة للمكتبات، فَضِّل نوع خطأ خاصًا بالمجال لتمكين المعالجة الصحيحة:

```
#[derive(Debug)]
pub enum ParseError {
    Empty,
    Invalid,
}

pub fn parse_id(s: &str) -> Result<u64, ParseError> {
    let t = s.trim();
    if t.is_empty() { return Err(ParseError::Empty); }
    t.parse::<u64>().map_err(|_| ParseError::Invalid)
}
```

٣.٣.٩ وثّق سطح الواجهة عبر البنية

نمط شائع هو إعادة تصدير مجموعة صغيرة من الأنواع/الدوال عند جذر الحِزمة، وترك الوحدات كتنظيم داخلي:

```
mod internal;

pub use internal::{Client, ClientConfig};
```

يحافظ هذا على قصة عامة متماسكة للحِزمة مع السماح بإعادة هيكلة داخلية.

٤.٩ حدود الحِزم وتنظيم المشروع

مع نمو المشاريع، تصبح حدود الحِزم حدوداً معمارية. ويساهم الحد الجيد في تقليل أزمنة الترجمة، وتحسين الوضوح، وفصل المسؤوليات.

١.٤.٩ متى تُقسّم إلى عدة حِزم

فكّر في تعدد الحِزم عندما:

- يمكن إعادة استخدام "النواة" المكتبية عبر عدة حِزم تنفيذية،
- تحتاج إلى حد واجهة مستقر بين الأنظمة الفرعية،
- يجب عزل الميزات أو الشيفرة الخاصة بالمنصات.

٢.٤.٩ مساحات العمل

تدير Cargo مساحات العمل (workspaces) عدة حِزم معًا بملف قفل مشترك وبناء منسّق.

```
workspace/
  Cargo.toml
  crates/
    core/
      Cargo.toml
      src/lib.rs
    cli/
      Cargo.toml
      src/main.rs
```

ملف توصيف مساحة عمل بسيط:

```
[workspace]
members = ["crates/core", "crates/cli"]
```

يمكّن هذا الهيكل فصلاً نظيفًا:

- core: منطق المجال والمكوّنات القابلة لإعادة الاستخدام،
- cli: واجهة سطر الأوامر وتفاعل المستخدم.

٥.٩ ملخص مُكثف

- الحِزمة هي وحدة الترجمة؛ ويمكن للحزمة الواحدة أن تحتوي على حِزمة مكتبة وعدة حِزم تنفيذية.
- تنظّم الوحدات الشيفرة وتتحكم في نطاق الرؤية؛ والعناصر خاصة افتراضيًا ويُكشف عنها بقصد.
- صمّم الواجهات العامة بكشف سطح صغير، وإخفاء التفاصيل، والحفاظ على الثوابت، وإرجاع أخطاء مُهيكلة.
- استخدم حدود الحِزم ومساحات العمل لفصل المسؤوليات ودعم الأنظمة الكبيرة بشكل نظيف.

الفصل ١٠: أول مشروع عملي

١.١٠ هدف المشروع

يبني هذا الفصل أداة بسيطة وعملية لسطر الأوامر (CLI) تقوم بقراءة ملف نصي وطباعة إحصاءات أساسية:

- عدد الأسطر،
- عدد الكلمات،
- عدد البايتات.

يركّز المشروع على ثلاثة أساسيات:

- استخدام Cargo لبناء وتشغيل برنامج CLI,
- التعامل الصحيح مع إدخال/إخراج الملفات باستخدام المكتبة القياسية،
- تطبيق مفاهيم الملكية والاستعارة ومعالجة الأخطاء باستخدام Result والمعامل ?.

٢.١٠ إنشاء المشروع

أنشئ حِزمة تنفيذية جديدة:

```
cargo new rstat
cd rstat
```

نقطة الدخول للبرنامج هي src/main.rs.

٣.١٠ أداة CLI صغيرة: استقبال مسار الإدخال

كمشروع أول، سننفّذ تحليل الوسائط باستخدام std::env::args للحفاظ على أقل قدر من الاعتمادات. يتوقع البرنامج وسيطًا واحدًا فقط: مسار ملف.

```
use std::env;

fn usage(program: &str) {
    eprintln!("Usage: {program} <file>");
}

fn main() {
    let mut it = env::args();
    let program = it.next().unwrap_or_else(|| "rstat".to_string());
    let path = match it.next() {
        Some(p) => p,
        None => {
            usage(&program);
            std::process::exit(2);
        }
    };

    // the real work will be added next
    println!("Input: {path}");
}
```

ملاحظات:

- env::args() تُرجع قيم String مملوكة، لذلك لا توجد مشاكل أعمار (lifetimes).
- نعالج غياب الوسيط صراحةً ونخرج برمز غير صفري.

٤.١٠ إدخال/إخراج الملفات باستخدام المكتبة القياسية

هناك طريقتان شائعتان ومناسبتان للمبتدئين:

- قراءة الملف كاملاً إلى الذاكرة باستخدام read_to_string للبساطة،
- المعالجة سطرًا بسطر للمدخلات الكبيرة (قارئ مُخزَّن).

سنبدأ بالطريقة البسيطة لقراءة الملف كاملاً.

١.٤.١٠ قراءة ملف إلى String

```
use std::fs;

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

٢.٤.١٠ حساب الإحصاءات دون تخصيصات إضافية

نحسب:

- الأسطر: بعدّ مُكرِّر lines(),
- الكلمات: بعدّ تقسيم المسافات البيضاء،

- البايتات: باستخدام as_bytes().len() لطول UTF-8 بالبايت.

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

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

نقطة مهمة متعلقة بالملكية: الدالة compute_stats تستعير &str، لذا لا تأخذ ملكية String ولا تنسخ البيانات.

٥.١٠ تطبيق الملكية ومعالجة الأخطاء

الآن نربط كل شيء باستخدام Result والمعامل ? لنشر الأخطاء بشكل نظيف.

١.٥.١٠ دالة run نظيفة

أسلوب شائع في Rust هو وضع العمل القابل للفشل داخل دالة تُرجع Result، والإبقاء على main بسيطة.

```
use std::env;
use std::fs;

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

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

fn run(path: &str) -> Result<Stats, std::io::Error> {
    let text = fs::read_to_string(path)?;   // may fail: returns early on Err
    Ok(compute_stats(&text))                // borrow: no copy of file
    ↪ content
}

fn usage(program: &str) {
    eprintln!("Usage: {program} <file>");
}

fn main() {
    let mut it = env::args();
```

```
    let program = it.next().unwrap_or_else(|| "rstat".to_string());

    let path = match it.next() {
        Some(p) => p,
        None => {
            usage(&program);
            std::process::exit(2);
        }
    };

    match run(&path) {
        Ok(st) => {
            println!("lines: {}", st.lines);
            println!("words: {}", st.words);
            println!("bytes: {}", st.bytes);
        }
        Err(e) => {
            eprintln!("error: {e}");
            std::process::exit(1);
        }
    }
}
```

ما الذي يوضحه هذا المثال:

- الملكية: read_to_string تُرجع String مملوكة ومخصصة على الهيب.
- الاستعارة: compute_stats تستقبل &str، فتُحسب الإحصاءات دون نقل أو نسخ السلسلة.
- معالجة الأخطاء: المعامل ? ينشر أخطاء الإدخال/الإخراج بشكل نظيف؛ و main تُبلّغ وتضبط رموز الخروج.

٦.١٠ توسيع الأداة للتعامل مع ملفات كبيرة

قراءة الملف كاملاً بسيطة، لكن للملفات الكبيرة قد تفضّل المعالجة المتدفقة. يبقى التصميم كما هو: دالة run قابلة للفشل، أخطاء مُهيكلة، واستعارة مدروسة.

```
use std::fs::File;
use std::io::{self, BufRead, BufReader};

fn run_streaming(path: &str) -> Result<Stats, io::Error> {
    let f = File::open(path)?;
    let mut reader = BufReader::new(f);

    let mut buf = String::new();
    let mut lines = 0usize;
    let mut words = 0usize;
    let mut bytes = 0usize;

    loop {
        buf.clear();
        let n = reader.read_line(&mut buf)?;
        if n == 0 { break; }          // EOF
        lines += 1;
        bytes += n;                    // bytes read from the file
        words += buf.split_whitespace().count();
    }

    Ok(Stats { lines, words, bytes })
}
```

ملاحظات:

- BufReader يُقلّل عدد نداءات النظام، مما يحسّن الأداء.
- read_line تُلحِق البيانات بمخزن مؤقت قابل للإعادة الاستخدام؛ ومسحه يمنع التخصيصات المتكررة.
- قيمة bytes هنا تعدّ البايتات المقروءة من الملف، بما فيها محارف نهاية السطر كما تُسلَّم بواسطة read_line.

٧.١٠ تمارين

- أضِف خيار --json لطباعة الإحصاءات بصيغة JSON (بأقل قدر ممكن، دون مكتبات خارجية).
- أضِف نمطًا لعدّ المحارف (قيم يونيكود العددية) وقارنه بعدّ البايتات.
- وسّع الأداة لقبول عدة مسارات وطباعة سطر إخراج واحد لكل ملف.

٨.١٠ ملخص مُكثف

- يمكن بناء أداة CLI بسيطة باستخدام std::env::args دون أي اعتمادات.
- استخدم fs::read_to_string للبساطة؛ واستخدم المعالجة المتدفقة المخبّأة للمدخلات الكبيرة.
- ضع العمل القابل للفشل داخل دالة run تُرجع Result، واستخدم ? لنشر الأخطاء بشكل نظيف.
- استعِر البيانات (&str) عند حساب النتائج المُشتقّة لتجنّب النسخ غير الضروري والحفاظ على وضوح الملكية.

الفصل ١١: ماذا تتعلم بعد ذلك

١.١١ نظرة عامة على التزامن

تقوم فلسفة التزامن في Rust على فكرة أن العديد من خصائص الصحة يمكن فرضها أثناء الترجمة. يمنع نظام الأنواع وقواعد الملكية حدوث سباقات البيانات في الشيفرة الآمنة عبر التحكم في التشاركية (aliasing) والتعديل. بعد إتقان الملكية والاستعارة ومعالجة الأخطاء، يصبح التزامن القدرة الكبرى التالية التي ينبغي تعلمها.

١.١.١١ المفاهيم الأساسية التي يجب تعلمها

- الخيوط ونقل الملكية: نقل البيانات إلى الخيوط بأمان.
- الحالة المشتركة مع التزامن: استخدام الأقفال عندما تحتاج عدة خيوط إلى تعديل بيانات مشتركة.
- تمرير الرسائل: استخدام القنوات للتواصل بين الخيوط دون مشاركة حالة قابلة للتعديل.
- معنى Send وSync: سمات وسمية تعبّر عن خصائص الأمان بين الخيوط للأنواع.

٢.١.١١ الأنماط الأولى

يُعد تمرير الرسائل باستخدام القنوات غالبًا أبسط نقطة بداية آمنة:

```
use std::sync::mpsc;
use std::thread;

fn main() {
    let (tx, rx) = mpsc::channel();

    thread::spawn(move || {
        tx.send(String::from("work done")).unwrap();
    });

    let msg = rx.recv().unwrap();
    println!("{msg}");
}
```

أما الحالة المشتركة فتستخدم عادةً `Arc<T>` للملكية المشتركة و `Mutex<T>` (أو `RwLock<T>` للوصول المتزامن:

```
use std::sync::{Arc, Mutex};
use std::thread;

fn main() {
    let counter = Arc::new(Mutex::new(0u64));
    let mut handles = Vec::new();

    for _ in 0..4 {
        let c = Arc::clone(&counter);
        handles.push(thread::spawn(move || {
```

```
            let mut g = c.lock().unwrap();
            *g += 1;
        }));
    }

    for h in handles {
        h.join().unwrap();
    }

    println!("{}", *counter.lock().unwrap());
}
```

إرشادات عملية:

- ابدأ بالقنوات والأقفال ذات النطاق الواسع.
- تعلّم التفكير في حدود الملكية: من يملك البيانات، من يستعيرها، وأين يحدث التعديل.
- قِس الأداء فقط بعد ضمان الصحة؛ وتجنّب التصاميم الخالية من الأقفال قبل الأوان.

٢.١١ الوعي بـ Rust Unsafe

توجد Unsafe Rust لتمكين قدرات منخفضة المستوى لا يستطيع المصرّف إثبات أمانها، مثل التعامل مع المؤشرات الخام، وبعض الأنماط الحساسة للأداء، وواجهات الاستدعاء الخارجي. لا تعني `unsafe` ``إيقاف القواعد''، بل تعني أنك أصبحت مسؤولاً يدوياً عن الحفاظ على ثوابت أمان محددة.

١.٢.١١ ما الذي تسمح به `unsafe`

داخل كتلة `unsafe`، يسمح Rust بعمليات مثل:

- فك إشارة المؤشرات الخام (`*const T`، `*mut T`)،
- استدعاء دوال موسومة بـ `unsafe`،
- الوصول إلى أو تعديل `static mut`،
- تنفيذ سمات `unsafe`.

مثال بسيط:

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

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

٢.٢.١١ كيف تتعلم unsafe بشكل صحيح

منهج تعلّم آمن:

- لا تكتب `unsafe` مبكرًا إلا إذا كان هناك هدف واضح (FFI، الأنظمة المضمنة، المخصّصات، حدود الأداء).
- ادرس الثوابت الكامنة وراء التجريدات الآمنة (الشرائح، Vec، المُكرِّرات، قواعد الاستعارة).
- عامِل كتل `unsafe` كنوى ``موثوقة'' صغيرة محاطة بواجهات آمنة.

العقلية الصحيحة هي: `unsafe` أسلوب تنفيذ، بينما `safe` هو عقد الواجهة. ينبغي أن تبقى معظم شيفرة التطبيقات في Rust آمنة بالكامل.

٣.١١ متى تنتقل إلى Async والموضوعات المتقدمة

تُعد Async Rust قوية، لكنها تُدخل تعقيداً إضافياً: المنفّذات، المهام، التثبيت (pinning)، ودلالات الإدخال/الإخراج غير المتزامن. ينبغي الانتقال إلى async بعد أن تصبح مرتاحًا في كتابة Rust المتزامنة بوضوح.

١.٣.١١ المتطلبات قبل async

ينبغي أن تكون قادرًا على:

- تصميم دوال تُرجع Result ونشر الأخطاء باستخدام ?,
- تنظيم المشروع إلى وحدات وحِزم بشكل نظيف،
- التفكير في حدود الملكية والاستعارة داخل واجهات البرمجة،
- كتابة الاختبارات واستخدام الأدوات (`cargo test`، `cargo fmt`، clippy cargo).

٢.٣.١١ متى يكون async هو الأداة المناسبة

يكون async مبرَّرًا عادةً عندما:

- تتعامل مع عدد كبير من عمليات الإدخال/الإخراج المتزامنة (خوادم الشبكة، الوسطاء، الزواحف)،
- تحتاج إلى قابلية توسّع مع عدد كبير من الاتصالات وخيوط محدودة،
- تريد جدولة تعاونية بدل خيط نظام لكل مهمة.

بالنسبة للتوازي المعتمد على المعالج (حسابات كثيفة)، ابدأ بالخيوط وأنماط التوازي المُقيّدة؛ فـ async لا يجعل العمل الحسابي أسرع تلقائيًا.

١١.٣.٣ خارطة طريق للموضوعات المتقدمة

بعد هذا الكتيّب المصغّر، يكون مسار التعلّم العملي كالتالي:

- الاختبار والأدوات بعمق: اختبارات الوحدات، اختبارات التكامل، اختبارات التوثيق، إعداد clippy، ملفات البناء.
- السمات والتعميمات عمليًا: قيود السمات، الأعمار في السمات، كائنات سمات الأخطاء مقابل التعدادات.
- التزامن بعمق: القنوات مقابل الحالة المشتركة، تصميم الأقفال، حالات الجمود، أساسيات الذرّيات.
- أساسيات async: async/await، المستقبلات، المنفّذات، أنماط الإلغاء.
- FFI وحدود unsafe: استدعاء C، بناء مغلّفات آمنة، الثوابت والتدقيق.

١١.٤ ملخص مُكثف

- يبدأ التزامن في Rust بأنماط آمنة: تمرير الرسائل، Arc، والأقفال؛ مع فهم Send وSync.
- تمكّن Unsafe Rust من قدرات منخفضة المستوى لكنها تتطلب ثوابت يدوية؛ اجعل unsafe صغيرًا ومغلفًا بواجهات آمنة.
- انتقل إلى async بعد إتقان Rust المتزامنة؛ واختر async أساسًا للإدخال/إخراج عالي التزامن، لا لتسريع الحسابات.

الخلاصة

قدّم هذا الكتيّب المصغّر Rust كما ينبغي أن تُتعلَّم: لغة برمجة نظم حديثة تُعطي الأولوية للصحة والأمان والأداء عبر تصميم صريح، لا عبر آليات تشغيل خفية. ومن خلال التركيز على مبادئ اللغة الجوهرية بدل الاكتفاء بالصياغة السطحية، كان الهدف بناء نموذج ذهني متين يتدرّج من الأدوات الصغيرة إلى الأنظمة الكبيرة.

لقد رأيت كيف تستبدل قواعد الملكية والاستعارة في Rust فئات كاملة من أخطاء وقت التشغيل بضمانات وقت الترجمة. إدارة الذاكرة صريحة لكنها متوقعة، مع تنظيف حتمي ودون جامع قمامة. تُنمذج معالجة الأخطاء مباشرةً داخل نظام الأنواع، ما يفرض وضوحًا حول ما يمكن أن يفشل وكيف ينبغي التعامل مع الإخفاقات. وتوفّر الوحدات والحِزم حدودًا قوية تدعم قواعد الشيفرة القابلة للصيانة، بينما تُظهر المشاريع العملية كيف تتآلف هذه الأفكار في برامج حقيقية.

تكافئ Rust التصميم المنضبط. فعندما تُشفَّر الثوابت داخل الأنواع، وتكون الملكية واضحة، وتُنمذج الأخطاء صراحةً، تصبح البرامج أسهل في الفهم وأكثر أمانًا عند التطوير. ويعمل المصرّف ليس كعقبة، بل كمراجع صارم يفرض العقود بثبات عبر قاعدة الشيفرة.

هذا الكتيّب ليس نهاية الطريق، بل أساسًا. ومع ترسيخ الأساسيات، تصبح مستعدًا للانتقال بثقة إلى التزامن، والبرمجة غير المتزامنة، والأنظمة الحساسة للأداء، وشيفرة `unsafe` المدقّقة بعناية عند الحاجة. وتبقى المبادئ نفسها التي قُدِّمت هنا—الصراحة، والصحة بالبناء، والتصميم المتعمّد—مركزية في كل مستوى متقدم من تطوير Rust.

أهم عادة ينبغي حملها إلى الأمام هي الإصغاء إلى المصرّف، ونمذجة المشكلات بأنواع دقيقة، وتفضيل الوضوح على الحِيَل الذكية. بهذا النهج، تصبح Rust ليس مجرد أداة قوية، بل شريكًا موثوقًا في بناء برمجيات صحيحة وطويلة العمر.