Comprehensive Rust 🦀

Martin Geisler

Зміст

70 Кредити **395**

Ласкаво просимо в Comprehensive Rust 🦀

Це безкоштовний курс Rust, розроблений командою Android у Google. Курс охоплює весь спектр Rust, від базового синтаксиса до складних тем, таких як узагальнення (generics) и обробка помилок.

> Останню версію курсу можна знайти за адресою https://google.github.io/comprehensive-rust/. Якщо ви читаєте десь в іншому місці, перевіряйте там на оновлення.
>
> Курс доступний на інших мовах. Виберіть потрібну мову у верхньому правому куті сторінки або перегляньте сторінку Переклади, щоб ознайомитися зі списком усіх доступних перекладів.
>
> Курс також доступний у форматі PDF.

Ціль курсу навчити вас мові Rust. Ми припускаємо, що ви нічого не знаєте про Rust та сподіваємося:

- Дати вам повне уявлення про синтаксис та семантику мови Rust.
- Навчити працювати з існуючим кодом та писати нові програми на Rust.
- Показати розповсюджені ідіоми мови Rust.

Перші чотири дні курсу ми називаємо Rust Fundamentals.

Спираючись на це, вам пропонується зануритися в одну або кілька спеціалізованих тем:

- Android: розрахований на половину дня курс з використання Rust для розробки на платформі Android (AOSP). Сюди входить взаємодія з C, C++ та Java.
- Chromium: розрахований на половину дня курс курс із використання Rust у браузерах на основі Chromium. Сюди входить взаємодія з C++ та як включити крейти сторонніх розробників у Chromium.
- Голе залізо: одноденне заняття з використання Rust для низькорівневої (embedded) розробки, що охоплює як мікроконтролери, так і звичайні процесори.
- Concurrency: повний день занять з вивчення конкурентності у Rust. Ми розглянемо як класичну конкурентність (витіснююча багатозадачність з використанням потоків і м'ютексів), так і async/await конкурентність (кооперативна багатозадачність з використанням futures).

За рамками курсу

Rust це об'ємна мова, і ми не зможемо охопити її за кілька днів. Теми, що виходять за межі курсу:

- Написання макросів, будь ласка подивіться Розділ 19.5 у The Rust Book та Rust by Example.

Припущення

Передбачається, що ви вже можете програмувати. Rust це статично типізована мова, і іноді ми порівнюватимемо і зіставлятимемо її з C та C++, щоб краще пояснити чи підкреслити різницю у підходах до написання коду на Rust.

Якщо ви знаєте, як програмувати мовою з динамічною типізацією, наприклад Python або JavaScript, ви зможете також успішно пройти цей курс.

Це приклад *нотаток для викладача*. Ми будемо використовувати їх для додавання додаткової інформації до слайдів. Це можуть бути ключові моменти, які викладач повинен висвітлити, а також відповіді на типові питання, що виникають під час проходження курсу.

Розділ 1

Проведення курсу

Ця сторінка призначена для викладача курсу.

Ось коротка довідкова інформація про те, як ми проводили цей курс всередині Google.

Зазвичай ми проводимо заняття з 9:00 до 16:00, з 1-годинною перервою на обід посередині. Це залишає 3 години для ранкового заняття та 3 години для післяобіднього заняття. Обидві сесії містять кілька перерв і час для роботи студентів над вправами.

Перед проведенням курсу бажано:

1. Ознайомитись з матеріалами курсу. Ми додали нотатки для викладача на деяких сторінках, щоб виділити ключові моменти (будь ласка, допомагайте нам, додаючи свої нотатки для викладачів!). Під час презентації переконайтеся, що відкрили нотатки для викладача у спливаючому вікні (натисніть на посилання з маленькою стрілкою поруч з "Нотатки для викладача"). Таким чином ви матимете чистий екран, який можна представити класу.
2. Визначитись з датами. Оскільки курс вимагає щонайменше чотири дні, ми рекомендуємо вам запланувати ці дні протягом двох тижнів. Учасники курсу сказали, що вони вважають корисним наявність прогалин у курсі, оскільки це допомагає їм обробити всю інформацію, яку ми їм надаємо.
3. Знайти приміщення досить просторе для очної участі. Ми рекомендуємо, щоб у класі було 15-20 чоловік. Це досить небагато для того, щоб людям було комфортно ставити запитання --- також достатньо мало, щоб один інструктор мав час відповісти на запитання. Переконайтеся, що в кімнаті є *парти* для вас і для студентів: ви всі повинні мати можливість сидіти і працювати за своїми ноутбуками. Зокрема, ви будете виконувати багато програмування в реальному часі як інструктор, тому кафедра вам не дуже допоможе.
4. У день заняття приходьте в кімнату трохи раніше, щоби все підготувати. Ми рекомендуємо презентувати безпосередньо за допомогою `mdbook serve`, запущеного на вашому ноутбуці (дивиться installation instructions). Це забезпечує оптимальну продуктивність без затримок під час зміни сторінок. Використання ноутбука також дозволить вам виправляти друкарські помилки в міру їх виявлення вами або учасниками курсу.
5. Дозвольте учасникам вирішувати вправи самостійно або у невеликих групах.

Зазвичай ми приділяємо вправам по 30-45 хвилин вранці та у другій половині дня (включаючи час на розбір рішень). Обов'язково запитуйте людей, чи не мають вони труднощів і чи є щось, з чим ви можете допомогти. Коли ви бачите, що у кількох людей одна і та ж проблема, повідомте про цей клас і запропонуйте рішення, наприклад, показавши, де знайти відповідну інформацію у стандартній бібліотеці.

На цьому все, удачі у проходженні курсу! Ми сподіваємося, що вам буде так само весело, як і нам!

Будь ласка, залишіть відгук, щоб ми могли продовжувати удосконалювати курс. Ми хотіли б почути, що було добре і що можна зробити краще. Ваші студенти також можуть надіслати нам свої відгуки!

1.1 Структура курсу

Ця сторінка призначена для викладача курсу.

Основи Rust

Перші чотири дні складають Основи Rust. Дні протікають швидко, і ми багато робимо!

Структура курсу

- День 1 Ранок (2 години 5 хвилин, включаючи перерви)

Сегмент	Тривалість
Ласкаво просимо	5 хвилин
Hello World!	15 хвилин
Типи та значення	40 хвилин
Основи потоку керування	40 хвилин

- День 1 Полудень (2 години 35 хвилин, включаючи перерви)

Сегмент	Тривалість
Кортежі та масиви	35 хвилин
Посилання	55 хвилин
Типи, які визначаються користувачем	50 хвилин

- День 2 Ранок (2 години та 10 хвилин, включаючи перерви)

Сегмент	Тривалість
Ласкаво просимо	3 хвилини
Зіставлення зразків	1 година
Методи та Трейти	50 хвилин

- День 2 Полудень (3 години та 15 хвилин, включаючи перерви)

Сегмент	Тривалість
Узагальнені типи	45 хвилин
Типи стандартної бібліотеки	1 година
Трейти стандартної бібліотеки	1 година та 10 хвилин

- День 3 Ранок (2 години та 20 хвилин, включаючи перерви)

Сегмент	Тривалість
Ласкаво просимо	3 хвилини
Управління пам'яттю	1 година
Розумні вказівники	55 хвилин

- День 3 Полудень (1 година 55 хвилин, включаючи перерви)

Сегмент	Тривалість
Запозичення	55 хвилин
Тривалість життя	50 хвилин

- День 4 Ранок (2 години та 40 хвилин, включаючи перерви)

Сегмент	Тривалість
Ласкаво просимо	3 хвилини
Ітератори	45 хвилин
Модулі	40 хвилин
Тестування	45 хвилин

- День 4 Полудень (2 години та 20 хвилин, включаючи перерви)

Сегмент	Тривалість
Обробка помилок	1 година та 5 хвилин
Небезпечний Rust	1 година та 5 хвилин

Глибоке занурення

На додаток до 4-денного курсу з основ Rust, ми розглянемо ще кілька спеціалізованих тем:

Rust в Android

Rust в Android --- це напівденний курс з використання Rust для розробки на Android платформі. Сюди входить взаємодія з C, C++ та Java.

Вам знадобиться AOSP. Завантажте репозиторій курсу на той же комп'ютер, що і курс та перемістіть каталог `src/android/` в кореневий каталог вашого AOSP. Це гарантує, що система збирання Android побачить файли `Android.bp` в `src/android/`.

Переконайтеся, що `adb sync` працює з вашим емулятором або реальним пристроєм, та попередньо зберіть усі приклади Android, використовуючи `src/android/build_all.sh`. Прочитайте скрипт, щоб побачити команди, які він запускає, і переконайтеся, що вони працюють, коли ви запускаєте їх вручну.

Rust в Chromium

Глибоке занурення Rust in Chromium — це південний курс із використання Rust як частини браузера Chromium. Він включає використання Rust у системі збирання gn Chromium, залучення сторонніх бібліотек (”крейтів”) і взаємодію з C++.

Вам потрібно буде мати можливість зібрати Chromium --- налагодженна, компонентна побудова рекомендується для швидкості, але будь-яка збірка буде працювати. Переконайтеся, що ви можете запустити веб-переглядач Chromium, який ви побудували.

Rust на голому залізі

Rust на голому залізі: заняття на повний день з використання Rust для низькорівневої (embedded) розробки. Розглядаються як мікроконтролери, так і прикладні процесори.

Щодо частини мікроконтролерів, то вам потрібно буде заздалегідь придбати плату розробки BBC micro:bit v2. Усім потрібно встановити кілька пакетів, як описано на сторінці привітання.

Конкурентність в Rust

Конкурентність в Rust це цілий день занять з класичної, а також `async`/`await` конкурентності.

Вам знадобиться налаштований новий крейт, а також завантажені залежності готові до роботи. Потім ви сможете скопіювати приклади в `src/main.rs`, щоб поекспериментувати з ними:

```
cargo init concurrency
cd concurrency
cargo add tokio --features full
cargo run
```

Структура курсу

- Ранок (3 години та 20 хвилин, включаючи перерви)

Сегмент	Тривалість
Потоки	30 хвилин
Канали	20 хвилин
Send та Sync	15 хвилин
Спільний стан	30 хвилин
Вправи	1 година та 10 хвилин

- Полудень (3 години та 20 хвилин, включаючи перерви)

Сегмент	Тривалість
Основи асинхронізації	30 хвилин
Канали та потік управління	20 хвилин
Підводні камені	55 хвилин
Вправи	1 година та 10 хвилин

Формат

Курс задуманий дуже інтерактивним, і ми рекомендуємо, щоб питання сприяли вивченню Rust!

1.2 Гарячі клавіші

У mdBook є кілька корисних поєднань клавіш:

- Стрілка вліво

  : Перехід на попередню сторінку.

- Стрілка вправо

  : Перехід до наступної сторінки.

- Ctrl + Enter

  : Виконати приклад коду, який знаходиться у фокусі.

- s

  : Активувати панель пошуку.

1.3 Переклади

Курс був перекладений іншими мовами групою чудових волонтерів:

- Бразильська Португальська від @rastringer, @hugojacob, @joaovicmendes, та @henrif75.
- Китайська (спрощена) від @suetfei, @wnghl, @anlunx, @kongy, @noahdragon, @superwhd, @SketchK, та @nodmp.
- Китайська (традиційна) від @hueich, @victorhsieh, @mingyc, @kuanhungchen, and @johnathan79717.
- Фарсі від @Alix1383, @DannyRavi, @hamidrezakp, @javad-jafari і @moaminsharifi, @hamidrezakp та @mehrad77.
- Японьска від @CoinEZ-JPN, @momotaro1105, @HidenoriKobayashi і @kantasv.
- Корейска від @keispace, @jiyongp, @jooyunghan та @namhyung.
- Іспанська від @deavid.
- Українська від @git-user-cpp, @yaremam і @reta.
- Фарсі від @Alix1383, @DannyRavi, @hamidrezakp, @javad-jafari і @moaminsharifi, @hamidrezakp та @mehrad77.

Використовуйте кнопку вибору мови у верхньому правому куті для перемикання між мовами.

Незавершені переклади

Існує велика кількість незавершених перекладів. Ми посилаємося на останні оновлені переклади:

- Арабська від @younies.
- Бенгальська від @raselmandol.
- Французька від @KookaS, @vcaen і @AdrienBaudemont.
- Німецька від @Throvn і @ronaldfw.
- Італійська від @henrythebuilder і @detro.

Повний список перекладів з їхнім поточним статусом також доступний або на момент останнього оновлення, або синхронізований з останньою версією курсу.

Якщо ви хочете допомогти в цьому, будь ласка, ознайомтеся з нашими інструкціями про те, як розпочати роботу. Переклади координуються за допомогою трекера проблем.

Розділ 2

Використання Cargo

Коли ви почнете читати про Rust, то незабаром познайомитеся з Cargo, стандартним інструментом, що використовується в екосистемі Rust для створення та запуску програм. Тут ми хочемо дати короткий огляд того, що таке Cargo і як він вписується в ширшу екосистему і в цей курс.

Встановлення

> **Дотримуйтесь інструкцій на https://rustup.rs/.**

Як результат, ви отримаэте інструмент побудови Cargo (`cargo`) та компілятор Rust (`rustc`). Ви також отримаэте `rustup`, утиліту командної стрічки, яку виможете використовувати для встановлення різних версій компілятора.

Після встановлення Rust вам слід налаштувати редактор або IDE для роботи з Rust. Більшість редакторів роблять це, звертаючись до rust-analyzer, який забезпечує автозаповнення та функцію переходу до визначення для VS Code, Emacs, Vim/Neovim, та багато інших. Існує також інша доступна IDE під назвою RustRover.

- У Debian/Ubuntu ви можете встановити Cargo, вихідний код Rust та Rust formatter за допомогою `apt`. Однак це може призвести до встановлення застарілої версії Rust і неочікуваної поведінки. Використовуйте таку команду:

  ```
  sudo apt install cargo rust-src rustfmt
  ```

- На macOS ви можете використовувати Homebrew для установки Rust, але він може надати застарілу версію. Тому рекомендується встановлювати Rust з офіційного сайту.

2.1 Екосистема Rust

Екосистема Rust складається з ряду інструментів, основними з яких є:

- `rustc`: компілятор Rust, який перетворює файли `.rs` на бінарні файли та інші проміжні формати.

- `cargo`: менеджер залежностей Rust та інструмент збірки. Cargo знає, як завантажити залежності, розміщені на https://crates.io, і передати їх `rustc` при збірці вашого проекту. Cargo також поставляється з вбудованим інструментом запуску тестів, який використовується для виконання модульних тестів.
- `rustup`: програма встановлення та оновлення набору інструментів Rust. Цей інструмент використовується для встановлення та оновлення `rustc` і `cargo` при виході нових версій Rust. Окрім того, `rustup` також може завантажувати документацію стандартної бібліотеки. Ви можете встановити кілька версій Rust одночасно і `rustup` дозволить вам перемикатися між ними за необхідності.

Ключові моменти:

- У Rust стрімкий графік релізів: нова версія виходить кожні шість тижнів. Нові версії підтримують зворотну сумісність із старими версіями --- на додаток вони надають нові функціональні можливості.
- Існує три канали релізів: ”stable”, ”beta” та ”nightly”.
- Нові функції тестуються на ”nightly”, ”beta” це те, що стає ”stable” кожні шість тижнів.
- Залежності також можна вирішити за допомогою альтернативних реєстрів, git, папок тощо.
- Rust також має [редакції]: поточна редакція це Rust 2021. Попередніми редакціями були Rust 2015 та Rust 2018.
  - Редакціям дозволено вносити зворотно-несумісні зміни до мови.
  - Щоб уникнути збоїв коду, редакцію для свого пакета можна явно вказати у файлі `Cargo.toml`.
  - Щоб уникнути поділу екосистеми, компілятор Rust може змішувати код, написаний для різних редакцій.
  - Варто нагадати, що використання компілятора безпосередньо, а не через `cargo`, є рідкісним явищем (більшість користувачів ніколи цього не роблять).
  - Варто зазначити, що Cargo сам по собі є надзвичайно потужним і всеосяжним інструментом. Він має багато додаткових функцій, включаючи, але не обмежуючись:
    * Структуру проекту/пакета
    * робочі області
    * Управління/кешування залежностями для розробки (dev) та часу виконання (runtime)
    * сценарії побудови
    * глобальна установка
    * Він також розширюється за допомогою плагінів підкоманд (таких як cargo clippy).
  - Докладніше читайте в офіційній Cargo Book

2.2 Приклади коду в цьому курсі

У цьому курсі ми в основному вивчатимемо мову Rust на прикладах, які можуть бути виконані у вашому браузері. Це значно спрощує налаштування та забезпечує однаковий досвід для всіх.

Встановлення Cargo, як і раніше, рекомендується: це полегшить виконання вправ. В останній день ми виконаємо більш масштабну вправу, яка покаже вам як працювати із залежностями, і для цього вам знадобиться Cargo.

Блоки коду в цьому курсі є повністю інтерактивними:

```
fn main() {
    println!("Відредагуте мене!");
}
```

Ви можете використовувати

Ctrl + Enter

для виконання коду, коли фокус введення знаходиться в текстовому полі.

Більшість прикладів коду доступні для редагування, як показано вище. Кілька прикладів коду недоступні для редагування з різних причин:

- Вбудований у сторінку редактор коду не може запускати модульні тести. Скопіюйте код і відкрийте його в справжньому Playground, щоб продемонструвати модульні тести.
- Вбудовані в сторінку редактори коду втрачають свій стан у той момент, коли ви йдете зі сторінки! Саме з цієї причини учні повинні виконувати вправи, використовуючи локальну установку Rust або Rust Playground.

2.3 Запуск коду локально за допомогою Cargo

Якщо ви хочете поекспериментувати з кодом на своїй системі, то вам потрібно буде спочатку встановити Rust. Зробіть це, дотримуючись інструкцій у The Rust Book. У вашій системі з'являться інструменти `rustc` та `cargo`. На момент написання статті останній стабільний випуск Rust має такі версії:

```
% rustc --version
rustc 1.69.0 (84c898d65 2023-04-16)
% cargo --version
cargo 1.69.0 (6e9a83356 2023-04-12)
```

Ви також можете використовувати будь-яку пізнішу версію, оскільки Rust підтримує зворотну сумісність.

Після цього виконайте такі кроки, щоб зібрати виконуваний файл на основі одного з прикладів у цьому курсі:

1. Натисніть кнопку ”Copy to clipboard” на прикладі коду, який потрібно скопіювати.
2. Використовуйте `cargo new exercise`, щоб створити нову директорію `exercise/` для вашого коду:

```
$ cargo new exercise
     Created binary (application) `exercise` package
```

3. Перейдіть в директорію `exercise/` і виконайте `cargo run` для побудови та запуску виконуваного файлу:

```
$ cd exercise
$ cargo run
   Compiling exercise v0.1.0 (/home/mgeisler/tmp/exercise)
    Finished dev [unoptimized + debuginfo] target(s) in 0.75s
     Running `target/debug/exercise`
Hello, world!
```

4. Замініть шаблонний код у `src/main.rs` на свій код. Наприклад, використовуючи приклад коду з попередньої сторінки, зробіть `src/main.rs` схожим на

```
fn main() {
    println!("Відредагуте мене!");
}
```

5. Використовуйте `cargo run` для побудови та запуску оновленого виконуваного файлу:

```
$ cargo run
   Compiling exercise v0.1.0 (/home/mgeisler/tmp/exercise)
    Finished dev [unoptimized + debuginfo] target(s) in 0.24s
     Running `target/debug/exercise`
Edit me!
```

6. Використовуйте `cargo check` для швидкої перевірки проекту на наявність помилок і `cargo build` для компіляції проекту без його запуску. Ви знайдете результат у директорії `target/debug/` для налагоджувальної збірки. Використовуйте `cargo build --release` для створення оптимізованої фінальної збірки в `target/release/`.

7. Ви можете додати залежності для вашого проекту, відредагувавши файл `Cargo.toml`. Коли ви запустите команду `cargo`, вона автоматично завантажить і скомпілює відсутні залежності для вас.

Запропонуйте учасникам заняття встановити Cargo та використовувати локальний редактор. Це полегшить їм життя, тому що у них буде відповідне середовище розробки.

Частина I

День 1: Ранок

Розділ 3

Ласкаво просимо до Дня 1

Це перший день Rust Fundamentals. Сьогодні ми розглянемо багато питань:

- Базовий синтаксис Rust: змінні, скалярні та складені типи, переліки, структури, посилання, функції та методи.
- Типи та виведення типів.
- Конструкції потоку управління: цикли, умовні переходи і так далі.
- Типи, визначені користувачем: структури та переліки.
- Зіставлення шаблонів: деструктуризація переліків, структур і масивів.

Розклад

Враховуючи 10-хвилинні перерви, ця сесія триватиме приблизно 2 години та 5 хвилин. Вона містить:

Сегмент	Тривалість
Ласкаво просимо	5 хвилин
Hello World!	15 хвилин
Типи та значення	40 хвилин
Основи потоку керування	40 хвилин

Будь ласка, нагадайте учням, що:

- Вони повинні задавати питання, коли вони їх мають, а не зберігати їх до кінця.
- Клас має на меті бути інтерактивним, тому дискусії дуже заохочуються!
  - Як інструктор, ви повинні намагатися підтримувати обговорення актуальними, тобто підтримувати обговорення, пов'язані з тим, як Rust щось робить на відміну від іншої мови. Буває важко знайти правильний баланс, але краще дозволити дискусії, оскільки вони залучають людей набагато більше, ніж одностороннє спілкування.
- Запитання, швидше за все, означатимуть, що ми обговорюємо речі, які випереджають слайди.

– Це цілком нормально! Повторення є важливою частиною навчання. Пам’ятайте, що слайди є лише підтримкою, і ви можете пропускати їх, якщо забажаєте.

Ідея першого дня полягає в тому, щоб показати ”базові” речі в Rust, які повинні мати безпосередні паралелі в інших мовах. Більш просунуті частини Rust будуть розглянуті в наступні дні.

Якщо ви викладаєте цей курс у класі, це гарний момент, щоб ознайомитися з розкладом. Зверніть увагу, що в кінці кожного сегмента є вправа, після якої слідує перерва. Плануйте розглянути рішення вправи після перерви. Час, вказаний тут, є рекомендацією для того, щоб тримати курс за розкладом. Не соромтеся бути гнучкими і вносити корективи за потреби!

Розділ 4

Hello World!

Цей сегмент повинен зайняти близько 15 хвилин. Він містить:

Слайд	Тривалість
Що таке Rust?	10 хвилин
Переваги Rust	3 хвилини
Ігровий майданчик	2 хвилини

4.1 Що таке Rust?

Rust — це нова мова програмування, яка мала 1.0 випуск у 2015 році:

- Rust — це статично скомпільована мова, яка виконує таку саму роль, як C++
  - `rustc` використовує LLVM як бекенд.
- Rust підтримує багато платформ і архітектур:
  - x86, ARM, WebAssembly, ...
  - Linux, Mac, Windows, ...
- Rust використовується для широкого спектру пристроїв:
  - прошивки та завантажувачі,
  - розумні дисплеї,
  - мобільні телефони,
  - робочі станції,
  - сервери.

Rust вписується в ту ж саму область, що й C++:

- Висока гнучкість.
- Високий рівень контролю.
- Може бути зменьшений до дуже обмежених пристроїв, таких як мікроконтролери.
- Не має часу виконання або збирання сміття.
- Зосереджений на надійності та безпеці без шкоди для продуктивності.

4.2 Переваги Rust

Деякі унікальні переваги Rust:

- *Безпека пам'яті під час компіляції* - цілі класи помилок пам'яті запобігаються на етапі компіляції
  - Немає неініціалізованих змінних.
  - Ніяких подвійних звільнень.
  - Немає використання після звільнення.
  - Немає вказівників `NULL`.
  - Немає забутих заблокованих м'ютексів.
  - Немає перегонів даних між потоками.
  - Немає недійсності ітератора.
- *Ніякої невизначеної поведінки під час виконання* - те, що робить оператор Rust, ніколи не залишається невизначеним
  - Доступ до масиву перевірено на межі.
  - Поведінка цілочисельного переповнення визначена (паніка або обертання).
- *Можливості сучасної мови* - така ж виразна та ергономічна, як і мови вищих рівнів
  - Переліки та зіставлення шаблонів.
  - Узагальнені типи.
  - FFI без накладних витрат.
  - Абстракції без витрат.
  - Чудово деталізовані помилки компілятора.
  - Вбудований менеджер залежностей.
  - Вбудована підтримка тестування.
  - Чудова підтримка протоколу мовного сервера (LSP).

Не витрачайте тут багато часу. Всі ці пункти будуть розглянуті більш детально пізніше.

Обов’язково запитайте клас, з якими мовами вони мають досвід. Залежно від відповіді ви можете виділити різні особливості Rust:

- Досвід роботи з C або C++: Rust усуває цілий клас *помилок виконання* за допомогою засобу перевірки запозичень. Ви отримуєте продуктивність, як у C і C++, але у вас немає проблем із небезпекою пам’яті. Крім того, ви отримуєте сучасну мову з такими конструкціями, як зіставлення шаблонів і вбудоване керування залежностями.
- Досвід роботи з Java, Go, Python, JavaScript...: Ви отримуєте таку саму безпеку пам’яті, що й у цих мовах, а також подібне відчуття мови високого рівня. Крім того, ви отримуєте швидку та передбачувану продуктивність як C і C++ (без збиральника сміття), а також доступ низького рівня до апаратного забезпечення (якщо воно вам знадобиться)

4.3 Ігровий майданчик

Rust Playground надає простий спосіб виконання коротких Rust-програм і є основою для прикладів і вправ у цьому курсі. Спробуйте запустити програму ”hello-world”, з якої він починається. Вона має декілька зручних можливостей:

- У розділі ”Інструменти” скористайтеся опцією `rustfmt` для форматування вашого коду у ”стандартний” спосіб.
- Rust має два основних ”профілі” для генерації коду: Налагодження (додаткові перевірки під час виконання, менше оптимізації) та Випуску (менше перевірок під час виконання, багато оптимізацій). Вони доступні у розділі ”Налагодження” у верхній частині вікна.
- Якщо вам цікаво, скористайтеся командою ”ASM” в ”...”, щоб переглянути згенерований асемблерний код.

Коли студенти підуть на перерву, заохотьте їх відкрити майданчик і трохи поекспериментувати. Заохочуйте їх залишати вкладку відкритою і пробувати щось протягом решти курсу. Це особливо корисно для досвідчених студентів, які хочуть дізнатися більше про оптимізацію Rust або згенеровану збірку.

Розділ 5

Типи та значення

Цей сегмент повинен зайняти близько 40 хвилин. Він містить:

Слайд	Тривалість
Hello World!	5 хвилин
Змінні	5 хвилин
Значення	5 хвилин
Арифметика	3 хвилини
Виведення типів	3 хвилини
Вправа: Фібоначчі	15 хвилин

5.1 Hello World!

Перейдемо до найпростішої програми Rust, класичної програми Hello World:

```
fn main() {
    println!("Привіт 🌍!");
}
```

Що ви бачите:

- Функції вводяться за допомогою `fn`.
- Блоки розділені фігурними дужками, як у C і C++.
- Функція `main` є точкою входу в програму.
- Rust має гігієнічні макроси, `println!` є прикладом цього.
- Рядки в Rust мають кодування UTF-8 і можуть містити будь-які символи Unicode.

Цей слайд спрямований на те, щоб студенти звикли працювати з кодом Rust. Вони побачать масу цього протягом наступних чотирьох днів, тож ми починаємо з чогось малого та знайомого.

Ключові моменти:

- Rust дуже схожий на інші традиційні мови як C/C++/Java. Це навмисно, і він не намагається винайти щось заново, якщо це не є абсолютно необхідним.
- Rust сучасний із повною підтримкою таких речей, як Unicode.

- Rust використовує макроси для ситуацій, коли потрібно мати змінну кількість аргументів (немає перевантаження функцій).
- Макроси є «гігієнічними» що означає, що вони випадково не захоплюють ідентифікатори з області, у якій вони використовуються. Макроси Rust насправді лише [частково гігієнічні](https://veykril.github.io/tlborm/decl-macros/minutiae/hygiene .html).
- Rust є мультипарадигмою. Наприклад, він має потужні функції об'єктно-орієнтованого програмування, і, хоча це не функціональна мова, він включає діапазон функціональних понять.

5.2 Змінні

Rust забезпечує безпеку типів за допомогою статичної типізації. Прив'язки змінних створюються за допомогою `let`:

```
fn main() {
    let x: i32 = 10;
    println!("x: {x}");
    // x = 20;
    // println!("x: {x}");
}
```

- Відкоментуйте `x = 20`, щоб продемонструвати, що змінні за замовчуванням є незмінними. Додайте ключове слово `mut`, щоб дозволити зміну.
- Тут `i32` - це тип змінної. Він має бути відомий під час компіляції, але виведення типів (розглядається пізніше) дозволяє програмісту у багатьох випадках не вказувати його.

5.3 Значення

Нижче наведено деякі основні вбудовані типи та синтаксис для літеральних значень кожного типу.

	Типи	Літерали
Цілі числа зі знаком	`i8`, `i16`, `i32`, `i64`, `i128`, `isize`	`-10`, `0`, `1_000`, `123_i64`
Беззнакові цілі числа	`u8`, `u16`, `u32`, `u64`, `u128`, `usize`	`0`, `123`, `10_u16`
Числа з плаваючою комою	`f32`, `f64`	`3.14`, `-10.0e20`, `2_f32`
Скалярні значення Unicode	`char`	`'a'`, `'α'`, `'∞'`

Типи	Літерали
Логічні значення `bool`	`true`, `false`

Типи мають наступну ширину:

- `iN`, `uN` і `fN` мають ширину *N* біт,
- `isize` і `usize` – це ширина вказівника,
- `char` має ширину 32 біти,
- `bool` має ширину 8 біт.

Є кілька синтаксисів, які не показано вище:

- Усі підкреслення у числах можна опускати, вони призначені лише для розбірливості. Отже, `1_000` можна записати як `1000` (або `10_00`), а `123_i64` можна записати як `123i64`.

5.4 Арифметика

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

fn main() {
    println!("результат: {}", interproduct(120, 100, 248));
}
```

Це перший раз, коли ми бачимо функцію, відмінну від `main`, але її значення повинно бути зрозумілим: вона отримує три цілих числа і повертає ціле число. Функції буде розглянуто більш детально пізніше.

Арифметика дуже схожа на інші мови, зі схожими пріоритетами.

Як бути з переповненням цілих чисел? У мовах C та C++ переповнення цілих чисел *зі знаком* фактично не визначено, і може робити невідомі речі під час виконання. У Rust воно визначене.

Замініть `i32` на `i16`, щоб побачити цілочисельне переповнення, яке панікує (перевіряється) у налагоджувальній збірці і загортається у релізній збірці. Існують і інші варіанти, такі як переповнення, перенасичення і перенесення. Доступ до них здійснюється за допомогою синтаксису методу, наприклад, `(a * b).saturating_add(b * c).saturating_add(c * a)`.

Насправді, компілятор виявить переповнення константних виразів, тому приклад вимагає окремої функції.

5.5 Виведення типів

Rust перевірить, як *використовується* змінна для визначення типу:

```
fn takes_u32(x: u32) {
    println!("u32: {x}");
```

```
}

fn takes_i8(y: i8) {
    println!("i8: {y}");
}

fn main() {
    let x = 10;
    let y = 20;

    takes_u32(x);
    takes_i8(y);
    // takes_u32(y);
}
```

На цьому слайді показано, як компілятор Rust виводить типи на основі обмежень, заданих оголошеннями змінних та їх використанням.

Дуже важливо підкреслити, що змінні, оголошені таким чином, не належать до якогось динамічного «будь-якого типу», який може містити будь-які дані. Машинний код, згенерований такою декларацією, ідентичний явному оголошенню типу. Компілятор виконує роботу за нас і допомагає нам писати більш стислий код.

Якщо тип цілочисельного літерала не обмежено, Rust за замовчуванням використовує тип `i32`. Іноді у повідомленнях про помилки це позначається як `{integer}`. Подібно до цього, літерали з плаваючою комою за замовчуванням мають тип `f64`.

```
fn main() {
    let x = 3.14;
    let y = 20;
    assert_eq!(x, y);
    // ПОМИЛКА: немає реалізації для `{float} == {integer}`
}
```

5.6 Вправа: Фібоначчі

Послідовність Фібоначчі починається з `[0,1]`. Для n>1 n-те число Фібоначчі обчислюється рекурсивно як сума n-1-го та n-2-го чисел Фібоначчі.

Напишіть функцію `fib(n)`, яка обчислює n-те число Фібоначчі. Коли ця функція запанікує?

```
fn fib(n: u32) -> u32 {
    if n < 2 {
        // Базовий випадок.
        todo!("Реалізуйте це")
    } else {
        // Рекурсивний випадок.
        todo!("Реалізуйте це")
    }
}

fn main() {
```

```
    let n = 20;
    println!("fib({n}) = {}", fib(n));
}
```

5.6.1 Рішення

```
fn fib(n: u32) -> u32 {
    if n < 2 {
        return n;
    } else {
        return fib(n - 1) + fib(n - 2);
    }
}

fn main() {
    let n = 20;
    println!("fib({n}) = {}", fib(n));
}
```

Розділ 6

Основи потоку керування

Цей сегмент повинен зайняти близько 40 хвилин. Він містить:

Слайд	Тривалість
Вирази if	4 хвилини
Цикли	5 хвилин
break та continue	4 хвилини
Блоки та області застосування	5 хвилин
Функції	3 хвилини
Макроси	2 хвилини
Вправа: Послідовність Коллатца	15 хвилин

- We will now cover the many kinds of flow control found in Rust.
- Most of this will be very familiar to what you have seen in other programming languages.

6.1 Вирази `if`

Ви використовуєте вирази `if` так само, як і вирази `if` в інших мовах:

```
fn main() {
    let x = 10;
    if x == 0 {
        println!("нуль!");
    } else if x < 100 {
        println!("великий");
    } else {
        println!("величезний");
    }
}
```

Крім того, ви можете використовувати `if` як вираз. Останній вираз кожного блоку стає значенням виразу `if`:

```rust
fn main() {
    let x = 10;
    let size = if x < 20 { "маленький" } else { "великий" };
    println!("розмір числа: {}", size);
}
```

Оскільки `if` є виразом і повинен мати певний тип, обидва його блоки розгалужень повинні мати той самий тип. Покажіть, що станеться, якщо додати `;` після `"маленький"` у другому прикладі.

Вираз `if` слід використовувати так само, як і інші вирази. Наприклад, якщо він використовується в операторі `let`, цей оператор також має завершуватися символом `;`. Видаліть `;` перед `println!`, щоб побачити помилку компілятора.

6.2 Цикли

У Rust є три ключові слова циклу: `while`, `loop` і `for`:

`while`

Ключове слово `while` працює так само, як і в інших мовах, виконуючи тіло циклу доти, доки умова виконується.

```rust
fn main() {
    let mut x = 200;
    while x >= 10 {
        x = x / 2;
    }
    println!("Final x: {x}");
}
```

6.2.1 `for`

Цикл `for` виконує ітерації над діапазонами значень або елементами колекції:

```rust
fn main() {
    for x in 1..5 {
        println!("x: {x}");
    }

    for elem in [1, 2, 3, 4, 5] {
        println!("elem: {elem}");
    }
}
```

- Під капотом циклів `for` використовується концепція, яка називається ”ітератори”, для обробки ітерацій над різними типами діапазонів/колекцій. Ітератори буде розглянуто більш детально пізніше.
- Зверніть увагу, що перший цикл `for` виконує ітерацію тільки до `4`. Покажіть синтаксис `1..=5` для включеного діапазону.

6.2.2 `loop`

Оператор `loop` просто повторюється до нескінченності, поки не трапиться `break`.

```
fn main() {
    let mut i = 0;
    loop {
        i += 1;
        println!("{i}");
        if i > 100 {
            break;
        }
    }
}
```

- The `loop` statement works like a `while true` loop. Use it for things like servers which will serve connections forever.

6.3 `break` та `continue`

Якщо ви хочете негайно почати наступну ітерацію, використовуйте `continue`.

Якщо ви хочете достроково вийти з будь-якого типу циклу, використовуйте `break`. З `loop` це може бути необов'язковий вираз, який стане значенням виразу `loop`.

```
fn main() {
    let mut i = 0;
    loop {
        i += 1;
        if i > 5 {
            break;
        }
        if i % 2 == 0 {
            continue;
        }
        println!("{}", i);
    }
}
```

Зверніть увагу, що `loop` - це єдина циклічна конструкція, яка може повертати нетривіальне значення. Це пов'язано з тим, що вона гарантовано повертає значення лише при виконанні оператора `break` (на відміну від циклів `while` і `for`, які також можуть повертати значення при невиконанні умови).

6.3.1 Мітки

І `continue`, і `break` можуть додатково приймати аргумент мітки, який використовується для виходу з вкладених циклів:

```
fn main() {
    let s = [[5, 6, 7], [8, 9, 10], [21, 15, 32]];
    let mut elements_searched = 0;
    let target_value = 10;
```

```
    'outer: for i in 0..=2 {
        for j in 0..=2 {
            elements_searched += 1;
            if s[i][j] == target_value {
                break 'outer;
            }
        }
    }
    print!("елементів переглянуто: {elements_searched}");
}
```

- Позначене переривання також працює на довільних блоках, наприклад
```
'label: {
    break 'label;
    println!("Цей рядок пропускається");
}
```

6.4 Блоки та області застосування

Блоки

Блок у Rust містить послідовність виразів, взятих у фігурні дужки `{}`. Кожен блок має значення і тип, які відповідають значенню і типу останнього виразу в блоці:

```
fn main() {
    let z = 13;
    let x = {
        let y = 10;
        println!("y: {y}");
        z - y
    };
    println!("x: {x}");
}
```

Якщо останній вираз закінчується символом `;`, то результуюче значення і тип буде `()`.

- Ви можете показати, як змінюється значення блоку, змінивши останній рядок у блоці. Наприклад, додаванням/видаленням крапки з комою або використанням `return`.

6.4.1 Області видимості та затінення

Область видимості змінної обмежується блоком, що її охоплює.

Ви можете затіняти змінні, як із зовнішніх областей, так і змінні з тієї ж області:

```
fn main() {
    let a = 10;
    println!("до: {a}");
    {
        let a = "привіт";
        println!("внутрішня область видимості: {a}");
```

```
        let a = true;
        println!("затінений у внутрішній області видимості: {a}");
    }

    println!("після: {a}");
}
```

- Покажіть, що область видимості змінної обмежена, додавши b у внутрішньому блоці в останньому прикладі, а потім спробувавши отримати доступ до неї за межами цього блоку.
- Затінення відрізняється від мутації тим, що після затінення обидві ділянки пам'яті змінних існують одночасно. Обидві змінні доступні під одним і тим же ім'ям, залежно від того, де ви їх використовуєте у коді.
- Змінна затінення може мати інший тип.
- Затінення спочатку виглядає незрозумілим, але є зручним для збереження значень після `.unwrap()`.

6.5 Функції

```
fn gcd(a: u32, b: u32) -> u32 {
    if b > 0 {
        gcd(b, a % b)
    } else {
        a
    }
}

fn main() {
    println!("gcd: {}", gcd(143, 52));
}
```

- Параметри оголошення супроводжуються типом (у зворотному порядку порівняно з деякими мовами програмування), а потім типом повернення.
- Останній вираз у тілі функції (або будь-якого блоку) стає значенням, що повертається. Просто опустіть `;` в кінці виразу. Ключове слово `return` можна використовувати для дострокового повернення, але форма "голого значення" є ідіоматичною у кінці функції (рефактор `gcd` щоб використовувати `return`).
- Деякі функції не мають значення, що повертається, і повертають 'тип агрегату', `()`. Компілятор визначить це, якщо тип повернення пропущено.
- Перевантаження не підтримується - кожна функція має єдину реалізацію.
  - Завжди приймає фіксовану кількість параметрів. Аргументи за замовчуванням не підтримуються. Для підтримки варіаційних функцій можна використовувати макроси.
  - Завжди приймає єдиний набір типів параметрів. Ці типи можуть бути загальними, що буде розглянуто пізніше.

6.6 Макроси

Макроси розгортаються у код Rust під час компіляції і можуть приймати змінну кількість аргументів. Вони відрізняються символом `!` у кінці. До стандартної бібліотеки

Rust входить набір корисних макросів.

- `println!(format, ..)` виводить рядок у стандартний вивід, застосовуючи форматування, описане у `std::fmt`.
- `format!(format, ..)` працює так само, як `println!`, але повертає результат у вигляді рядка.
- `dbg!(вираз)` записує значення виразу і повертає його.
- `todo!()` позначає частину коду як таку, що ще не виконана. Якщо цей код буде виконано, він викличе паніку.
- `unreachable!()` позначає ділянку коду як недосяжну. Якщо цей код буде виконано, він викличе паніку.

```
fn factorial(n: u32) -> u32 {
    let mut product = 1;
    for i in 1..=n {
        product *= dbg!(i);
    }
    product
}

fn fizzbuzz(n: u32) -> u32 {
    todo!()
}

fn main() {
    let n = 4;
    println!("{n}! = {}", factorial(n));
}
```

Висновок з цього розділу полягає в тому, що ці загальні зручності існують, і те, як ними користуватися. Чому вони визначені як макроси і на що вони поширюються, не є особливо важливим.

У цьому курсі не розглядається визначення макросів, але в наступному розділі буде описано використання похідних макросів.

6.7 Вправа: Послідовність Коллатца

Послідовність Коллатца визначається наступним чином, для довільного n

1

більшого за нуль:

- Якщо *n

  i

  * є 1, то послідовність завершується при *n

  i

  *.

- Якщо *n

i

* є парним, то *n

i+1

= n

i

/ 2*.

- Якщо *n

  i

  * є непарним, то *n

  i+1

  = 3 * n

  i

  + 1*.

Наприклад, починаючи з *n

1

* = 3:

- 3 є непарним, таким чином *n

  2

  * = 3 * 3 + 1 = 10;
- 10 є парним, таким чином *n

  3

  * = 10 / 2 = 5;
- 5 є непарним, таким чином *n

  4

  * = 3 * 5 + 1 = 16;
- 16 є парним, таким чином *n

  5

  * = 16 / 2 = 8;
- 8 є парним, таким чином *n

  6

  * = 8 / 2 = 4;
- 4 є парним, таким чином *n

  7

  * = 4 / 2 = 2;

- 2 є парним, таким чином *n

  8

  * = 1; та

- послідовність завершується.

Напишіть функцію, яка обчислює довжину коллатц-послідовності для заданого початкового n.

```
/// Визначте довжину послідовності колатів, яка починається з `n`.
fn collatz_length(mut n: i32) -> u32 {
  todo!("Реалізуйте це")
}

fn test_collatz_length() {
    assert_eq!(collatz_length(11), 15);
}

fn main() {
    println!("Довжина: {}", collatz_length(11));
}
```

6.7.1 Рішення

```
/// Визначте довжину послідовності колатів, яка починається з `n`.
fn collatz_length(mut n: i32) -> u32 {
    let mut len = 1;
    while n > 1 {
        n = if n % 2 == 0 { n / 2 } else { 3 * n + 1 };
        len += 1;
    }
    len
}

fn test_collatz_length() {
    assert_eq!(collatz_length(11), 15);
}

fn main() {
    println!("Довжина: {}", collatz_length(11));
}
```

Частина II

День 1: Полудень

Розділ 7

Ласкаво просимо назад

Враховуючи 10-хвилинні перерви, ця сесія триватиме приблизно 2 години 35 хвилин. Віна містить:

Сегмент	Тривалість
Кортежі та масиви	35 хвилин
Посилання	55 хвилин
Типи, які визначаються користувачем	50 хвилин

Розділ 8

Кортежі та масиви

Цей сегмент повинен зайняти близько 35 хвилин. Він містить:

Слайд	Тривалість
Масиви	5 хвилин
Кортежі	5 хвилин
Ітерація масиву	3 хвилини
Патерни та деструктуризація	5 хвилин
Вправа: Вкладені масиви	15 хвилин

- We have seen how primitive types work in Rust. Now it's time for you to start building new composite types.

8.1 Масиви

```
fn main() {
    let mut a: [i8; 10] = [42; 10];
    a[5] = 0;
    println!("a: {a:?}");
}
```

- Значення типу масиву `[T; N]` містить `N` (константа часу компіляції) елементів того самого типу `T`. Зверніть увагу, що довжина масиву є *частиною його типу*, що означає, що `[u8; 3]` і `[u8; 4]` вважаються двома різними типами. Зрізи, розмір яких визначається під час виконання, покриваються пізніше.
- Спробуйте доступ до елементу масиву, що знаходиться за межами масиву. Доступ до масиву перевіряється під час виконання. Зазвичай Rust може оптимізувати ці перевірки, і їх можна уникнути, використовуючи небезпечний Rust.
- Ми можемо використовувати літерали для присвоєння значень масивам.
- Макрос `println!` запитує реалізацію налагодження за допомогою параметра формату `?`: `{}` - виведення за замовчуванням, `{:?}` - виведення налагодження. Такі типи, як цілі числа і рядки, реалізують виведення за замовчуванням, але

масиви реалізують лише виведення для налагодження. Це означає, що тут ми повинні використовувати налагоджувальний вивід.

- Додавання #, наприклад `{a:#?}`, викликає формат "гарного друку", який може бути легшим для читання.

8.2 Кортежі

```
fn main() {
    let t: (i8, bool) = (7, true);
    println!("t.0: {}", t.0);
    println!("t.1: {}", t.1);
}
```

- Як і масиви, кортежі мають фіксовану довжину.
- Кортежі групують значення різних типів у складений тип.
- Доступ до полів кортежу можна отримати після крапки з індексом значення, наприклад. `t.0`, `t.1`.
- Порожній кортеж `()` називається "типом одиниці" і означає відсутність значення, що повертається, подібно до `void` в інших мовах.

8.3 Ітерація масиву

Оператор `for` підтримує ітерацію над масивами (але не кортежами).

```
fn main() {
    let primes = [2, 3, 5, 7, 11, 13, 17, 19];
    for prime in primes {
        for i in 2..prime {
            assert_ne!(prime % i, 0);
        }
    }
}
```

Ця функціональність використовує трейт `IntoIterator`, але ми ще не розглядали його.

Макрос `assert_ne!` тут новий. Існують також макроси `assert_eq!` та `assert!`. Вони завжди перевіряються, тоді як варіанти лише для налагодження, такі як `debug_assert!`, не компілюються у релізних збірках.

8.4 Патерни та деструктуризація

При роботі з кортежами та іншими структурованими значеннями часто виникає потреба витягти внутрішні значення у локальні змінні. Це можна зробити вручну шляхом прямого доступу до внутрішніх значень:

```
fn print_tuple(tuple: (i32, i32)) {
    let left = tuple.0;
```

```
    let right = tuple.1;
    println!("left: {left}, right: {right}");
}
```

Однак, Rust також підтримує використання зіставлення шаблонів для розбиття більшого значення на складові частини:

```
fn print_tuple(tuple: (i32, i32)) {
    let (left, right) = tuple;
    println!("left: {left}, right: {right}");
}
```

- Шаблони, що використовуються тут, є ”неспростовними”, тобто компілятор може статично перевірити, що значення праворуч від = має таку саму структуру, як і шаблон.
- Ім'я змінної - це неспростовний шаблон, який завжди відповідає будь-якому значенню, тому ми також можемо використовувати `let` для оголошення однієї змінної.
- Rust також підтримує використання шаблонів в умовних операторах, що дозволяє виконувати порівняння на рівність і деструкцію одночасно. Ця форма порівняння шаблонів буде розглянута більш детально пізніше.
- Відредагуйте приклади вище, щоб показати помилку компілятора, коли шаблон не збігається зі значенням, що порівнюється.

8.5 Вправа: Вкладені масиви

Масиви можуть містити інші масиви:

```
let array = [[1, 2, 3], [4, 5, 6], [7, 8, 9]];
```

Який тип цієї змінної?

Використовуйте масив, подібний до наведеного вище, для написання функції `transpose`, яка транспонує матрицю (перетворює рядки у стовпці):

```
           ([1 2 3])      [1 4 7]
"transpose"|4 5 6|  "=="|2 5 8|
           ([7 8 9])      [3 6 9]
```

Скопіюйте наведений нижче код на https://play.rust-lang.org/ і реалізуйте функцію. Ця функція працює лише з матрицями 3x3.

```
// TODO: видаліть це, коли закінчите реалізацію.

fn transpose(matrix: [[i32; 3]; 3]) -> [[i32; 3]; 3] {
    unimplemented!()
}

fn test_transpose() {
    let matrix = [
        [101, 102, 103], //
        [201, 202, 203],
        [301, 302, 303],
    ];
```

```
    let transposed = transpose(matrix);
    assert_eq!(
        transposed,
        [
            [101, 201, 301], //
            [102, 202, 302],
            [103, 203, 303],
        ]
    );
}

fn main() {
    let matrix = [
        [101, 102, 103], // <-- коментар змушує rustfmt додати новий рядок
        [201, 202, 203],
        [301, 302, 303],
    ];

    println!("матриця: {:#?}", matrix);
    let transposed = transpose(matrix);
    println!("транспонована: {:#?}", transposed);
}
```

8.5.1 Рішення

```
fn transpose(matrix: [[i32; 3]; 3]) -> [[i32; 3]; 3] {
    let mut result = [[0; 3]; 3];
    for i in 0..3 {
        for j in 0..3 {
            result[j][i] = matrix[i][j];
        }
    }
    result
}

fn test_transpose() {
    let matrix = [
        [101, 102, 103], //
        [201, 202, 203],
        [301, 302, 303],
    ];
    let transposed = transpose(matrix);
    assert_eq!(
        transposed,
        [
            [101, 201, 301], //
            [102, 202, 302],
            [103, 203, 303],
        ]
    );
}
```

```rust
fn main() {
    let matrix = [
        [101, 102, 103], // <-- коментар змушує rustfmt додати новий рядок
        [201, 202, 203],
        [301, 302, 303],
    ];

    println!("матриця: {:#?}", matrix);
    let transposed = transpose(matrix);
    println!("транспонована: {:#?}", transposed);
}
```

Розділ 9

Посилання

Цей сегмент повинен зайняти близько 55 хвилин. Він містить:

Слайд	Тривалість
Спільні посилання	10 хвилин
Ексклюзивні посилання	10 хвилин
Зрізи	10 хвилин
Рядки	10 хвилин
Вправа: Геометрія	15 хвилин

9.1 Спільні посилання

Посилання забезпечує доступ до іншого значення без отримання права власності на нього, і також називається ”запозиченням”. Спільні посилання доступні лише для читання, і дані, на які вони посилаються, не можуть бути змінені.

```
fn main() {
    let a = 'A';
    let b = 'B';
    let mut r: &char = &a;
    println!("r: {}", *r);
    r = &b;
    println!("r: {}", *r);
}
```

Спільне посилання на тип T має тип &T. Значення посилання робиться за допомогою оператора &. Оператор * ”розіменовує” посилання, повертаючи його значення.

Rust статично забороняє висячі посилання:

```
fn x_axis(x: &i32) -> &(i32, i32) {
    let point = (*x, 0);
    return &point;
}
```

- Посилання ніколи не можуть бути null у Rust, тому перевірка на null не є обов'язковою.
- Кажуть, що посилання ”позичає” значення, на яке воно посилається, і це гарна модель для слухачів, які не знайомі з вказівниками: код може використовувати посилання для доступу до значення, але все одно залишається ”власністю” вихідної змінної. Більш детально про володіння буде розглянуто на третьому дні курсу.
- Посилання реалізовано як вказівники, і ключовою перевагою є те, що вони можуть бути набагато меншими за об'єкт, на який вони вказують. Слухачі, знайомі з C або C++, розпізнають посилання як вказівники. У наступних частинах курсу буде розглянуто, як Rust запобігає помилкам, пов'язаним з безпекою пам'яті, які виникають при використанні сирих вказівників.
- Rust не створює посилання автоматично - завжди потрібно використовувати `&`.
- У деяких випадках Rust виконує автоматичне розіменування, зокрема під час виклику методів (спробуйте `r.is_ascii()`). Тут не потрібен оператор `->`, як у C++.
- У цьому прикладі `r` є мутабельним, тому його можна перепризначити (`r = &b`). Зверніть увагу, що це повторно зв'язує `r`, так що він посилається на щось інше. Це відрізняється від C++, де присвоювання посилання змінює значення, на яке воно посилається.
- Спільне посилання не дозволяє змінювати значення, на яке воно посилається, навіть якщо це значення було змінним. Спробуйте `*r = 'X'`.
- Rust відстежує час життя всіх посилань, щоб переконатися, що вони живуть достатньо довго. У безпечному Rust'і не може бути ”висячих” посилань. Функція `x_axis` поверне посилання на `point`, але `point` буде звільнено, коли функція повернеться, тому це не буде скомпільовано.
- Про запозичення ми поговоримо більше, коли дійдемо до володіння.

9.2 Ексклюзивні посилання

Ексклюзивні посилання, також відомі як мутабельні посилання, дозволяють змінювати значення, на яке вони посилаються. Вони мають тип `&mut T`.

```
fn main() {
    let mut point = (1, 2);
    let x_coord = &mut point.0;
    *x_coord = 20;
    println!("point: {point:?}");
}
```

Ключові моменти:

- ”Ексклюзивне” означає, що тільки це посилання може бути використане для доступу до значення. Жодні інші посилання (спільні або ексклюзивні) не можуть існувати одночасно, і до значення, на яке посилаються, не можна отримати доступ, поки існує ексклюзивне посилання. Спробуйте створити `&point.0` або змінити `point.0`, поки існує `x_coord`.

- Обов’язково зверніть увагу на різницю між `let mut x_coord: &i32` і `let x_coord: &mut i32`. Перший представляє спільне посилання, яке можна прив'язати до різних значень, тоді як другий представляє ексклюзивне посилання на значення, що змінюється.

9.3 Зрізи

Зріз дає змогу поглянути на більшу колекцію:

```
fn main() {
    let mut a: [i32; 6] = [10, 20, 30, 40, 50, 60];
    println!("a: {a:?}");

    let s: &[i32] = &a[2..4];

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

- Зрізи запозичують дані зі зрізаного типу.
- Ми створюємо зріз, запозичуючи `a` та вказуючи початковий і кінцевий індекси в дужках.
- Якщо зріз починається з індексу 0, синтаксис діапазону Rust дозволяє нам відкинути початковий індекс, тобто `&a[0..a.len()]` і `&a[..a.len()]` ідентичні.
- Теж саме стосується останнього індексу, тому `&a[2..a.len()]` і `&a[2..]` ідентичні.
- Щоб легко створити повний зріз масиву, ми можемо використовувати `&a[..]`.
- `s` є посиланням на зріз `i32`. Зверніть увагу, що тип `s` (`&[i32]`) більше не згадує довжину масиву. Це дозволяє нам виконувати обчислення на зрізах різного розміру.
- Зрізи завжди запозичуються з іншого об'єкта. У цьому прикладі `a` має залишатися 'живим' (в області застосування) принаймні стільки ж, скільки і наш зріз.

9.4 Рядки

Тепер ми можемо зрозуміти два типи рядків у Rust:

- `&str` is a slice of UTF-8 encoded bytes, similar to `&[u8]`.
- `String` is an owned buffer of UTF-8 encoded bytes, similar to `Vec<T>`.

```
fn main() {
    let s1: &str = "Світ";
    println!("s1: {s1}");

    let mut s2: String = String::from("Привіт ");
    println!("s2: {s2}");
    s2.push_str(s1);
    println!("s2: {s2}");
```

```
    let s3: &str = &s2[s2.len() - s1.len()..];
    println!("s3: {s3}");
}
```

- `&str` представляє зріз рядка, який є незмінним посиланням на дані рядка в кодуванні UTF-8, що зберігаються в блоці пам'яті. Рядкові літерали (`"Hello"`) зберігаються у бінарному файлі програми.
- Тип `String` в Rust — це оболонка навколо вектора байтів. Як і у випадку з `Vec<T>`, він знаходиться у володінні.
- Як і у багатьох інших типів, `String::from()` створює рядок із рядкового літералу; `String::new()` створює новий порожній рядок, до якого дані рядка можна додати за допомогою методів `push()` і `push_str()`.
- Макрос `format!()` є зручним способом створення рядка, яким володіють, з динамічних значень. Він приймає таку саму специфікацію формату, як і `println!()`.
- Ви можете запозичувати зрізки `&str` з `String` за допомогою `&` і, за бажанням, вибору діапазону. Якщо ви виберете діапазон байт, який не вирівняно за межами символів, вираз запанікує. Ітератор `chars` перебирає символи, і йому надається перевага перед спробами вирівняти межі символів.
- Для програмістів на C++: думайте про `&str` як про `std::string_view` з C++, але такий, що завжди вказує на дійсний рядок у пам'яті. Rust `String` є приблизним еквівалентом `std::string` з C++ (головна відмінність: він може містити лише байти у кодуванні UTF-8 і ніколи не використовує оптимізацію малих рядків)..
- Літерали байтових рядків дозволяють створювати значення `&[u8]` безпосередньо:

  ```
  fn main() {
      println!("{:?}", b"abc");
      println!("{:?}", &[97, 98, 99]);
  }
  ```

- Необроблені рядки дозволяють створювати значення `&str` з відключеним екрануванням: `r"\n" == "\\n"`. Ви можете вставити подвійні лапки, використовуючи однакову кількість `#` з обох боків лапок:

  ```
  fn main() {
      println!(r#"<a href="link.html">link</a>"#);
      println!("<a href=\"link.html\">link</a>");
  }
  ```

9.5 Вправа: Геометрія

Ми створимо декілька утиліт для тривимірної геометрії, що представляють точку у вигляді `[f64;3]`. Ви самі визначаєте сигнатури функцій.

```
// Обчисліть величину вектора шляхом додавання квадратів його координат
// і вилучення квадратного кореня. Використовуйте метод `qrt()` для для обчислення квадр
// кореня, наприклад `v.sqrt()`.
```

```
fn magnitude(...) -> f64 {
    todo!()
}

// Нормалізуйте вектор, обчисливши його величину і поділивши всі його
// координати на цю величину.


fn normalize(...) {
    todo!()
}
```

Використовуйте наступний `main` для тестування вашої роботи.

```
fn main() {
    println!("Величина одиничного вектора: {}", magnitude(&[0.0, 1.0, 0.0]));

    let mut v = [1.0, 2.0, 9.0];
    println!("Величина {v:?}: {}", magnitude(&v));
    normalize(&mut v);
    println!("Величина {v:?} після нормалізації: {}", magnitude(&v));
}
```

9.5.1 Рішення

```
/// Обчисліть величину заданого вектора.
fn magnitude(vector: &[f64; 3]) -> f64 {
    let mut mag_squared = 0.0;
    for coord in vector {
        mag_squared += coord * coord;
    }
    mag_squared.sqrt()
}

/// Змініть величину вектора на 1.0, не змінюючи його напрямок.
fn normalize(vector: &mut [f64; 3]) {
    let mag = magnitude(vector);
    for item in vector {
        *item /= mag;
    }
}

fn main() {
    println!("Величина одиничного вектора: {}", magnitude(&[0.0, 1.0, 0.0]));

    let mut v = [1.0, 2.0, 9.0];
    println!("Величина {v:?}: {}", magnitude(&v));
    normalize(&mut v);
    println!("Величина {v:?} після нормалізації: {}", magnitude(&v));
}
```

Розділ 10

Типи, які визначаються користувачем

Цей сегмент повинен зайняти близько 50 хвилин. Він містить:

Слайд	Тривалість
Іменовані структури	10 хвилин
Кортежні структури	10 хвилин
Перелічувані типи	5 хвилин
Статика	5 хвилин
Псевдоніми типу	2 хвилини
Вправа: події в ліфті	15 хвилин

10.1 Іменовані структури

Подібно до C і C++, Rust підтримує користувальницькі структури:

```
struct Person {
    name: String,
    age: u8,
}

fn describe(person: &Person) {
    println!("{} віком {} років", person.name, person.age);
}

fn main() {
    let mut peter = Person { name: String::from("Peter"), age: 27 };
    describe(&peter);

    peter.age = 28;
    describe(&peter);
```

```
    let name = String::from("Avery");
    let age = 39;
    let avery = Person { name, age };
    describe(&avery);

    let jackie = Person { name: String::from("Jackie"), ..avery };
    describe(&jackie);
}
```

Ключові моменти:

- Структури працюють як у C або C++.
  - Як і в C++, і на відміну від C, для визначення типу не потрібен typedef.
  - На відміну від C++, між структурами немає успадкування.
- Це може бути вдалий час, щоб повідомити людям, що існують різні типи структур.
  - Структури нульового розміру (наприклад, `struct Foo;`) можуть бути використані при реалізації трейту на якомусь типі, але не мають даних, які ви хочете зберігати у самому значенні.
  - На наступному слайді буде представлено структури кортежу, які використовуються, коли імена полів не важливі.
- Якщо у вас уже є змінні з правильними іменами, ви можете створити структуру за допомогою скорочення:
- Синтаксис `..avery` дозволяє нам скопіювати більшість полів зі старої структури без необхідності явного введення всіх полів. Це завжди має бути останнім елементом.

10.2 Кортежні структури

Якщо імена полів неважливі, ви можете використати структуру кортежу:

```
struct Point(i32, i32);

fn main() {
    let p = Point(17, 23);
    println!("({}, {})", p.0, p.1);
}
```

Це часто використовується для обгорток з одним полем (так званих newtypes):

```
struct PoundsOfForce(f64);
struct Newtons(f64);

fn compute_thruster_force() -> PoundsOfForce {
    todo!("Запитайте вченого-ракетника з NASA")
}

fn set_thruster_force(force: Newtons) {
    // ...
}

fn main() {
    let force = compute_thruster_force();
```

```
    set_thruster_force(force);
}
```

- Newtypes — чудовий спосіб закодувати додаткову інформацію про значення в примітивному типі, наприклад:
  - Число вимірюється в деяких одиницях: у наведеному вище прикладі `Newtons`.
  - Значення пройшло певну перевірку під час створення, тому вам більше не потрібно перевіряти його знову при кожному використанні: `PhoneNumber(String)` або `OddNumber(u32)`.
- Продемонструйте, як додати значення `f64` до типу `Newtons`, отримавши доступ до єдиного поля в newtype.
  - Rust зазвичай не любить неявних речей, таких як автоматичне розгортання або, наприклад, використання логічних значень як цілих чисел.
  - Перевантаження операторів обговорюється в день 3 (дженерики).
- Цей приклад є тонким посиланням на невдачу Mars Climate Orbiter.

10.3 Перелічувані типи

Ключове слово `enum` дозволяє створити тип, який має кілька різних варіантів:

```
enum Direction {
    Left,
    Right,
}

enum PlayerMove {
    Pass,                       // Простий варіант
    Run(Direction),             // Варіант кортежу
    Teleport { x: u32, y: u32 }, // Варіант структури
}

fn main() {
    let player_move: PlayerMove = PlayerMove::Run(Direction::Left);
    println!("На цьому повороті: {player_move:?}");
}
```

Ключові моменти:

- Переліки дозволяють збирати набір значень під одним типом.
- Напрямок - це тип з варіантами. Існує два значення `Direction`: `Direction::Left` та `Direction::Right`.
- `PlayerMove` - це тип з трьома варіантами. На додаток до корисного навантаження, Rust зберігатиме дискримінант, щоб під час виконання знати, який варіант є у значенні `PlayerMove`.
- Це може бути гарний час для порівняння структури та переліки:
  - В обох ви можете мати просту версію без полів (структура одиниць) або з різними типами полів (різні варіанти корисного навантаження).
  - Ви навіть можете реалізувати різні варіанти переліку окремими структурами, але тоді вони не будуть одного типу, як якщо б всі вони були визначені в переліку.
- Rust використовує мінімальний обсяг пам'яті для зберігання дискримінанта.
  - Якщо потрібно, він зберігає ціле число найменшого необхідного розміру

– Якщо допустимі значення варіантів не покривають усіх бітових шаблонів, для кодування дискримінанта буде використано неприпустимі бітові шаблони (”нішева оптимізація”). Наприклад, `Option<&u8>` зберігає або вказівник на ціле число, або `NULL` для варіанта `None`.
– За потреби можна керувати дискримінантом (наприклад, для сумісності з C):

```
enum Bar {
    A, // 0
    B = 10000,
    C, // 10001
}

fn main() {
    println!("A: {}", Bar::A as u32);
    println!("B: {}", Bar::B as u32);
    println!("C: {}", Bar::C as u32);
}
```

Без `repr` тип дискримінанта займає 2 байти, оскільки 10001 вміщує 2 байти.

Більше інформації для вивчення

Rust має декілька оптимізацій, які можна застосувати, щоб зменшити розмір переліків.

- Оптимізація нульового вказівника: для деяких типів Rust гарантує, що `size_of::<T>()` дорівнює `size_of::<Option <T>>()`.

  Приклад коду, якщо ви хочете показати, як *може* виглядати побітове представлення на практиці. Важливо зазначити, що компілятор не надає жодних гарантій щодо цього представлення, тому це абсолютно небезпечно.

```
use std::mem::transmute;

macro_rules! dbg_bits {
    ($e:expr, $bit_type:ty) => {
        println!("- {}: {:#x}", stringify!($e), transmute::<_, $bit_type>($e));
    };
}

fn main() {
    unsafe {
        println!("bool:");
        dbg_bits!(false, u8);
        dbg_bits!(true, u8);

        println!("Option<bool>:");
        dbg_bits!(None::<bool>, u8);
        dbg_bits!(Some(false), u8);
        dbg_bits!(Some(true), u8);

        println!("Option<Option<bool>>:");
        dbg_bits!(Some(Some(false)), u8);
        dbg_bits!(Some(Some(true)), u8);
        dbg_bits!(Some(None::<bool>), u8);
```

```
            dbg_bits!(None::<Option<bool>>, u8);

            println!("Option<&i32>:");
            dbg_bits!(None::<&i32>, usize);
            dbg_bits!(Some(&0i32), usize);
        }
    }
```

10.4 `const`

Константи обчислюються під час компіляції, а їхні значення вставляються всюди, де вони використовуються:

```
const DIGEST_SIZE: usize = 3;
const ZERO: Option<u8> = Some(42);

fn compute_digest(text: &str) -> [u8; DIGEST_SIZE] {
    let mut digest = [ZERO.unwrap_or(0); DIGEST_SIZE];
    for (idx, &b) in text.as_bytes().iter().enumerate() {
        digest[idx % DIGEST_SIZE] = digest[idx % DIGEST_SIZE].wrapping_add(b);
    }
    digest
}

fn main() {
    let digest = compute_digest("Hello");
    println!("digest: {digest:?}");
}
```

Відповідно до Книги Rust RFC вони підставляються під час використання.

Лише функції з позначкою `const` можна викликати під час компіляції для створення значень `const`. Однак функції `const` можна викликати під час виконання.

- Зауважте, що `const` поводиться семантично подібно до `constexpr` C++.
- Не так часто виникає потреба у константі, що обчислюється під час виконання, але це корисно та безпечніше, ніж використовувати static.

10.5 `static`

Статичні змінні будуть жити протягом усього часу виконання програми, тому не будуть переміщатися:

```
static BANNER: &str = "Ласкаво просимо до RustOS 3.14";

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

Як зазначено в Книзі Rust RFC, вони не підставляються під час використання та мають реальну асоційовану ділянку пам'яті. Це корисно для небезпечного та вбудованого

коду, і змінна живе протягом усього виконання програми. Якщо значення глобальної області видимості не потребує ідентичності об'єкта, перевага надається `const`.

- `static` схожий на мутабельні глобальні змінні в C++.
- `static` забезпечує ідентичність об'єкта: адресу в пам'яті та стан відповідно до типів із внутрішньою змінністю, таких як `Mutex<T>`.

Більше інформації для вивчення

Оскільки `static` змінні доступні з будь-якого потоку, вони повинні бути `Sync`. Внутрішня змінність можлива через `Mutex`, atomic або подібні до них.

Локальні дані потоку можна створити за допомогою макросу `std::thread_local`.

10.6 Псевдоніми типу

Псевдонім типу створює ім'я для іншого типу. Ці два типи можна використовувати взаємозамінно.

```
enum CarryableConcreteItem {
    Left,
    Right,
}

type Item = CarryableConcreteItem;

// Псевдоніми більш корисні для довгих, складних типів:
use std::cell::RefCell;
use std::sync::{Arc, RwLock};
type PlayerInventory = RwLock<Vec<Arc<RefCell<Item>>>>;
```

- A newtype is often a better alternative since it creates a distinct type. Prefer `struct InventoryCount(usize)` to `type InventoryCount = usize`.
- Програмісти на C впізнають це як схоже на `typedef`.

10.7 Вправа: події в ліфті

Ми створимо структуру даних для представлення події в системі керування ліфтом. Ви самі визначаєте типи та функції для створення різних подій. Використовуйте `#[derive(Debug)]`, щоб дозволити форматування типів за допомогою `{:?}`.

У цій вправі потрібно лише створити і заповнити структури даних так, щоб `main` працював без помилок. Наступна частина курсу буде присвячена отриманню даних з цих структур.

```
/// Подія в ліфтовій системі, на яку повинен реагувати контролер.
enum Event {
    // TODO: додайте необхідні варіанти
}
```

```rust
/// Напрямок руху.
enum Direction {
    Up,
    Down,
}

/// Кабіна ліфта прибув на вказаний поверх.
fn car_arrived(floor: i32) -> Event {
    todo!()
}

/// Двері у кабіні ліфта відчинилися.
fn car_door_opened() -> Event {
    todo!()
}

/// Двері у кабіні ліфта зачинилися.
fn car_door_closed() -> Event {
    todo!()
}

/// У ліфтовому холі на даному поверсі була натиснута кнопка виклику в заданому напрямку
fn lobby_call_button_pressed(floor: i32, dir: Direction) -> Event {
    todo!()
}

/// У кабіні ліфта була натиснута кнопка поверху.
fn car_floor_button_pressed(floor: i32) -> Event {
    todo!()
}

fn main() {
    println!(
        "Пасажир першого поверху натиснув кнопку вгору: {:?}",
        lobby_call_button_pressed(0, Direction::Up)
    );
    println!("Кабіна ліфта заїхала на перший поверх: {:?}", car_arrived(0));
    println!("Двері у кабіні ліфта відчинилися: {:?}", car_door_opened());
    println!(
        "Пасажир натиснув кнопку 3-го поверху: {:?}",
        car_floor_button_pressed(3)
    );
    println!("Двері у кабіні ліфта зачинилися: {:?}", car_door_closed());
    println!("Кабіна ліфта заїхала на 3-й поверх: {:?}", car_arrived(3));
}
```

10.7.1 Рішення

```rust
/// Подія в ліфтовій системі, на яку повинен реагувати контролер.
enum Event {
    /// Була натиснута кнопка.
```

```
    ButtonPressed(Button),

    /// Кабіна ліфта прибула на вказаний поверх.
    CarArrived(Floor),

    /// Двері кабіни ліфта відчинилися.
    CarDoorOpened,

    /// Двері кабіни ліфта зачинилися.
    CarDoorClosed,
}

/// Поверх задається цілим числом.
type Floor = i32;

/// Напрямок руху.
enum Direction {
    Up,
    Down,
}

/// Кнопка, доступна для користувача.
enum Button {
    /// Кнопка в холі ліфта на даному поверсі.
    LobbyCall(Direction, Floor),

    /// Кнопка поверху в кабіни ліфта.
    CarFloor(Floor),
}

/// Кабіна ліфта прибув на вказаний поверх.
fn car_arrived(floor: i32) -> Event {
    Event::CarArrived(floor)
}

/// Двері у кабіні ліфта відчинилися.
fn car_door_opened() -> Event {
    Event::CarDoorOpened
}

/// Двері у кабіні ліфта зачинилися.
fn car_door_closed() -> Event {
    Event::CarDoorClosed
}

/// У ліфтовому холі на даному поверсі була натиснута кнопка виклику в заданому напрямку
fn lobby_call_button_pressed(floor: i32, dir: Direction) -> Event {
    Event::ButtonPressed(Button::LobbyCall(dir, floor))
}

/// У кабіні ліфта була натиснута кнопка поверху.
```

```rust
fn car_floor_button_pressed(floor: i32) -> Event {
    Event::ButtonPressed(Button::CarFloor(floor))
}

fn main() {
    println!(
        "Пасажир першого поверху натиснув кнопку вгору: {:?}",
        lobby_call_button_pressed(0, Direction::Up)
    );
    println!("Кабіна ліфта заїхала на перший поверх: {:?}", car_arrived(0));
    println!("Двері у кабіні ліфта відчинилися: {:?}", car_door_opened());
    println!(
        "Пасажир натиснув кнопку 3-го поверху: {:?}",
        car_floor_button_pressed(3)
    );
    println!("Двері у кабіні ліфта зачинилися: {:?}", car_door_closed());
    println!("Кабіна ліфта заїхала на 3-й поверх: {:?}", car_arrived(3));
}
```

Частина III

День 2: Ранок

Розділ 11

Ласкаво просимо до Дня 2

Тепер, коли ми побачили достатню кількість Rust, ми зосередимося на системі типів Rust:

- Зіставлення шаблонів: вилучення даних зі структур.
- Методи: зв'язування функцій з типами.
- Трэйти: поведінка, спільна для кількох типів
- Узагальнення: параметризація типів в інших типах.
- Типи та властивості стандартної бібліотеки: екскурсія по багатій стандартній бібліотеці Rust.

Розклад

Враховуючи 10-хвилинні перерви, ця сесія триватиме приблизно 2 години 10 хвилин. Вона містить:

Сегмент	Тривалість
Ласкаво просимо	3 хвилини
Зіставлення зразків	1 година
Методи та Трейти	50 хвилин

Розділ 12

Зіставлення зразків

Цей сегмент повинен зайняти близько 1 години. Він містить:

Слайд	Тривалість
Співставлення значень	10 хвилин
Деструктурування структур	4 хвилини
Деструктурування переліків	4 хвилини
Потік контролю Let	10 хвилин
Вправа: обчислення виразу	30 хвилин

12.1 Співставлення значень

Ключове слово `match` дозволяє зіставити значення з одним або декількома *шаблонами*. Порівняння відбуваються зверху вниз, і виграє перший збіг.

Шаблони можуть бути простими значеннями, подібно до `switch` у C та C++:

```
fn main() {
    let input = 'x';
    match input {
        'q'                       => println!("Виходжу"),
        'a' | 's' | 'w' | 'd'     => println!("Пересування"),
        '0'..='9'                 => println!("Введення числа"),
        key if key.is_lowercase() => println!("Нижній регістр: {key}"),
        _                         => println!("Щось інше"),
    }
}
```

Шаблон _ - це шаблон підстановки, який відповідає будь-якому значенню. Вирази *повинні* бути вичерпними, тобто охоплювати всі можливі варіанти, тому _ часто використовується як остаточний всеохоплюючий випадок.

Match можна використовувати як вираз. Як і у випадку з `if`, кожна гілка зіставлення повинно мати однаковий тип. Тип - це останній вираз у блоці, якщо такий є. У наведеному вище прикладі тип `()`.

Змінна у шаблоні (у цьому прикладі - `key`) створить прив'язку, яку можна використовувати у гілці зіставлення.

Запобіжник зіставлення призводить до гілці зіставлення, тільки якщо умова істинна.

Ключові моменти:

- Ви можете вказати, як деякі конкретні символи використовуються в шаблоні
  - `|` як `or`
  - `..` може розширюватися настільки, наскільки це потрібно
  - `1..=5` представляє включний діапазон
  - `_` - символ підстановки
- Запобіжники зіставлення як окрема функція синтаксису є важливою та необхідною, коли ми хочемо стисло висловити більш складні ідеї, ніж це дозволили б самі шаблони.
- Це не те саме, що окремий вираз `if` всередині гілкі зіставлення. Вираз `if` всередині блоку розгалуження (після `=>`) виникає після вибору гілкі зіставлення. Невиконання умови `if` всередині цього блоку не призведе до розгляду інших частин вихідного виразу `match`.
- Умова, визначена в запобіжнику, застосовується до кожного виразу в шаблоні з `|`.

Більше інформації для вивчення

- Ще одним елементом синтаксису шаблону, який ви можете показати учням, є синтаксис `@`, який прив'язує частину шаблону до змінної. Наприклад:

  ```
  let opt = Some(123);
  match opt {
      outer @ Some(inner) => {
          println!("outer: {outer:?}, inner: {inner}");
      }
      None => {}
  }
  ```

  У цьому прикладі `inner` має значення 123, яке він витягнув з `Option` за допомогою деструктуризації, `outer` перехоплює весь вираз `Some(inner)`, тому він містить повний вираз `Option::Some(123)`. Це рідко використовується, але може бути корисним у більш складних шаблонах.

12.2 Структури

Як і кортежі, структури також можуть бути деструктуровані шляхом зіставлення:

```
struct Foo {
    x: (u32, u32),
    y: u32,
}

fn main() {
    let foo = Foo { x: (1, 2), y: 3 };
```

```
    match foo {
        Foo { x: (1, b), y } => println!("x.0 = 1, b = {b}, y = {y}"),
        Foo { y: 2, x: i }   => println!("y = 2, x = {i:?}"),
        Foo { y, .. }        => println!("y = {y}, інші поля були проігноровані"),
    }
}
```

- Змініть значення літералів у `foo`, відповідно до інших шаблонів.
- Додайте нове поле до `Foo` і внесіть потрібні зміни до шаблону.
- Різницю між захопленням і постійним виразом може бути важко помітити. Спробуйте змінити `2` у другій гілці на змінну, і побачте, що це непомітно не працює. Змініть її на `const` і подивіться, що це знову запрацює.

12.3 Перелічувані типи

Як і кортежі, переліки також можуть бути деструктуровані шляхом зіставлення:

Шаблони також можна використовувати для прив’язки змінних до частин ваших значень. Таким чином ви перевіряєте структуру ваших типів. Давайте розпочнемо з простого типу `enum`:

```
enum Result {
    Ok(i32),
    Err(String),
}

fn divide_in_two(n: i32) -> Result {
    if n % 2 == 0 {
        Result::Ok(n / 2)
    } else {
        Result::Err(format!("не можна поділити {n} на дві рівні частини"))
    }
}

fn main() {
    let n = 100;
    match divide_in_two(n) {
        Result::Ok(half) => println!("{n} поділена навпіл, це {half}"),
        Result::Err(msg) => println!("вибачте, сталася помилка: {msg}"),
    }
}
```

Тут ми використали гілки для *деструктурування* значення `Result`. У першій гілці `half` прив'язано до значення всередині варіанту `Ok`. У другій гілці `msg` прив'язано до повідомлення про помилку.

- Вираз `if`/`else` повертає перелік, який пізніше розпаковується за допомогою `match`.
- Ви можете спробувати додати третій варіант до визначення переліку і відобразити помилки під час виконання коду. Вкажіть місця, де ваш код зараз є невичерпним, і як компілятор намагається дати вам підказки.

- Доступ до значень у варіантах переліку можливий лише після зіставлення з шаблоном.
- Продемонструйте, що відбувається, коли пошук є невичерпним. Зверніть увагу на перевагу, яку надає компілятор Rust, підтверджуючи що всі випадки оброблено.

12.4 Потік контролю Let

Rust має кілька конструкцій потоку керування, які відрізняються від інших мов. Вони використовуються для зіставлення шаблонів:

- вирази `if let`
- вирази `let else`
- вирази `while let`

вирази `if let`

Вираз `if let` дозволяє виконувати інший код залежно від того, чи відповідає значення шаблону :

```
use std::time::Duration;

fn sleep_for(secs: f32) {
    if let Ok(duration) = Duration::try_from_secs_f32(secs) {
        std::thread::sleep(duration);
        println!("проспав {duration:?}");
    }
}

fn main() {
    sleep_for(-10.0);
    sleep_for(0.8);
}
```

вирази `let else`

Для загального випадку зіставлення шаблону і повернення з функції використовуйте `let else`. Випадок "else" повинен відрізнятися (`return`, `break` або паніка - що завгодно, але не випадання з кінця блоку).

```
fn hex_or_die_trying(maybe_string: Option<String>) -> Result<u32, String> {
    if let Some(s) = maybe_string {
        if let Some(first_byte_char) = s.chars().next() {
            if let Some(digit) = first_byte_char.to_digit(16) {
                Ok(digit)
            } else {
                return Err(String::from("не шістнадцяткова цифра"));
            }
        } else {
            return Err(String::from("отримав порожній рядок"));
```

```
        }
    } else {
        return Err(String::from("отримав None"));
    }
}

fn main() {
    println!("результат: {:?}", hex_or_die_trying(Some(String::from("foo"))));
}
```

Подібно до `if let`, існує варіант `while let`, який багаторазово перевіряє значення на відповідність шаблону:

```
fn main() {
    let mut name = String::from("Comprehensive Rust 🦀");
    while let Some(c) = name.pop() {
        println!("character: {c}");
    }
    // (There are more efficient ways to reverse a string!)
}
```

Тут `String::pop` повертає `Some(c)` поки рядок не стане порожнім, після чого поверне None. Використання `while let` дозволяє нам продовжувати ітерацію по всіх елементах.

if-let

- На відміну від `match`, `if let` не має охоплювати всі гілки. Це може зробити його більш лаконічним, ніж `match`.
- Загальним використанням є обробка значень `Some` під час роботи з `Option`.
- На відміну від `match`, `if let` не підтримує захисні вирази для збігу шаблонів.

let-else

`if-let` може накопичуватись, як показано. Конструкція `let-else` підтримує згладжування цього вкладеного коду. Перепишіть незручну версію для студентів, щоб вони могли побачити перетворення.

Переписана версія така:

```
fn hex_or_die_trying(maybe_string: Option<String>) -> Result<u32, String> {
    let Some(s) = maybe_string else {
        return Err(String::from("отримав None"));
    };

    let Some(first_byte_char) = s.chars().next() else {
        return Err(String::from("отримав порожній рядок"));
    };

    let Some(digit) = first_byte_char.to_digit(16) else {
        return Err(String::from("не шістнадцяткова цифра"));
    };
```

```
    return Ok(digit);
}
```

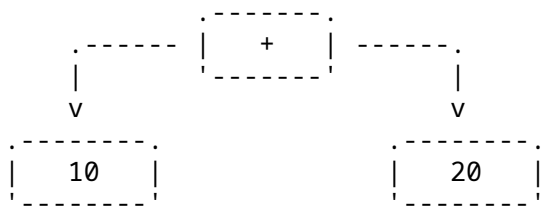
while-let

- Зверніть увагу, що цикл `while let` триватиме, доки значення відповідає шаблону.
- Ви можете переписати цикл `while let` як нескінченний цикл з оператором if, який переривається, коли для `name.pop()` немає значення для розгортання. Цикл `while let` забезпечує синтаксичний цукор для наведеного вище сценарію.

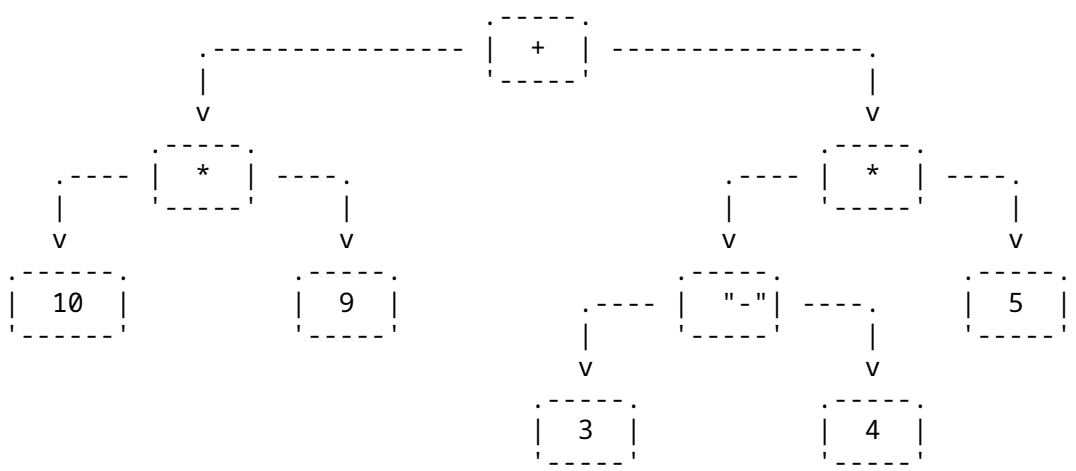
12.5 Вправа: обчислення виразу

Давайте напишемо простий рекурсивний обчислювач арифметичних виразів.

Прикладом невеликого арифметичного виразу може бути `10 + 20`, який обчислюється як `30`. Ми можемо представити цей вираз у вигляді дерева:



Більшим і складнішим виразом буде `(10 * 9) + ((3 - 4) * 5)`, що обчислюється як `85`. Ми зобразимо його у вигляді набагато більшого дерева:



У коді ми будемо представляти дерево двома типами:

```
/// Операція для виконання над двома під-виразами.
enum Operation {
    Add,
    Sub,
    Mul,
    Div,
```

```
}

/// Вираз у вигляді дерева.
enum Expression {
    /// Операція над двома підвиразами.
    Op { op: Operation, left: Box<Expression>, right: Box<Expression> },

    /// Літеральне значення
    Value(i64),
}
```

Тип Box тут є розумним вказівником і буде детально розглянутий пізніше у курсі. Вираз може бути ”упаковано” за допомогою `Box::new`, як показано у тестах. Щоб обчислити вираз, використайте оператор розіменування (`*`), щоб ”розпакувати” його: `eval(*boxed_expr)`.

Деякі вирази не можуть бути обчислені і повертають помилку. Стандартний тип `Result<Value, String>` - це перелік, який представляє або успішне значення (`Ok(Value)`), або помилку (`Err(String)`). Ми розглянемо цей тип більш детально пізніше.

Скопіюйте та вставте код у середовище Rust і почніть реалізацію `eval`. Кінцевий продукт повинен пройти тести. Може бути корисно використати `todo!()` і змусити тести проходити один за одним. Ви також можете тимчасово оминути тест за допомогою `#[ignore]`:

```
#[test]
#[ignore]
fn test_value() { .. }
```

```
/// Операція для виконання над двома під-виразами.
enum Operation {
    Add,
    Sub,
    Mul,
    Div,
}

/// Вираз у вигляді дерева.
enum Expression {
    /// Операція над двома підвиразами.
    Op { op: Operation, left: Box<Expression>, right: Box<Expression> },

    /// Літеральне значення
    Value(i64),
}

fn eval(e: Expression) -> Result<i64, String> {
    todo!()
}

fn test_value() {
    assert_eq!(eval(Expression::Value(19)), Ok(19));
```

```
}

fn test_sum() {
    assert_eq!(
        eval(Expression::Op {
            op: Operation::Add,
            left: Box::new(Expression::Value(10)),
            right: Box::new(Expression::Value(20)),
        }),
        Ok(30)
    );
}

fn test_recursion() {
    let term1 = Expression::Op {
        op: Operation::Mul,
        left: Box::new(Expression::Value(10)),
        right: Box::new(Expression::Value(9)),
    };
    let term2 = Expression::Op {
        op: Operation::Mul,
        left: Box::new(Expression::Op {
            op: Operation::Sub,
            left: Box::new(Expression::Value(3)),
            right: Box::new(Expression::Value(4)),
        }),
        right: Box::new(Expression::Value(5)),
    };
    assert_eq!(
        eval(Expression::Op {
            op: Operation::Add,
            left: Box::new(term1),
            right: Box::new(term2),
        }),
        Ok(85)
    );
}

fn test_zeros() {
    assert_eq!(
        eval(Expression::Op {
            op: Operation::Add,
            left: Box::new(Expression::Value(0)),
            right: Box::new(Expression::Value(0))
        }),
        Ok(0)
    );
    assert_eq!(
        eval(Expression::Op {
            op: Operation::Mul,
            left: Box::new(Expression::Value(0)),
```

```
            right: Box::new(Expression::Value(0))
        }),
        Ok(0)
    );
    assert_eq!(
        eval(Expression::Op {
            op: Operation::Sub,
            left: Box::new(Expression::Value(0)),
            right: Box::new(Expression::Value(0))
        }),
        Ok(0)
    );
}

fn test_error() {
    assert_eq!(
        eval(Expression::Op {
            op: Operation::Div,
            left: Box::new(Expression::Value(99)),
            right: Box::new(Expression::Value(0)),
        }),
        Err(String::from("ділення на нуль"))
    );
}
```

12.5.1 Рішення

```
/// Операція для виконання над двома під-виразами.
enum Operation {
    Add,
    Sub,
    Mul,
    Div,
}

/// Вираз у вигляді дерева.
enum Expression {
    /// Операція над двома підвиразами.
    Op { op: Operation, left: Box<Expression>, right: Box<Expression> },

    /// Літеральне значення
    Value(i64),
}

fn eval(e: Expression) -> Result<i64, String> {
    match e {
        Expression::Op { op, left, right } => {
            let left = match eval(*left) {
                Ok(v) => v,
                Err(e) => return Err(e),
            };
```

```
            let right = match eval(*right) {
                Ok(v) => v,
                Err(e) => return Err(e),
            };
            Ok(match op {
                Operation::Add => left + right,
                Operation::Sub => left - right,
                Operation::Mul => left * right,
                Operation::Div => {
                    if right == 0 {
                        return Err(String::from("ділення на нуль"));
                    } else {
                        left / right
                    }
                }
            })
        }
        Expression::Value(v) => Ok(v),
    }
}

fn test_value() {
    assert_eq!(eval(Expression::Value(19)), Ok(19));
}

fn test_sum() {
    assert_eq!(
        eval(Expression::Op {
            op: Operation::Add,
            left: Box::new(Expression::Value(10)),
            right: Box::new(Expression::Value(20)),
        }),
        Ok(30)
    );
}

fn test_recursion() {
    let term1 = Expression::Op {
        op: Operation::Mul,
        left: Box::new(Expression::Value(10)),
        right: Box::new(Expression::Value(9)),
    };
    let term2 = Expression::Op {
        op: Operation::Mul,
        left: Box::new(Expression::Op {
            op: Operation::Sub,
            left: Box::new(Expression::Value(3)),
            right: Box::new(Expression::Value(4)),
        }),
        right: Box::new(Expression::Value(5)),
    };
```

```
    assert_eq!(
        eval(Expression::Op {
            op: Operation::Add,
            left: Box::new(term1),
            right: Box::new(term2),
        }),
        Ok(85)
    );
}

fn test_zeros() {
    assert_eq!(
        eval(Expression::Op {
            op: Operation::Add,
            left: Box::new(Expression::Value(0)),
            right: Box::new(Expression::Value(0))
        }),
        Ok(0)
    );
    assert_eq!(
        eval(Expression::Op {
            op: Operation::Mul,
            left: Box::new(Expression::Value(0)),
            right: Box::new(Expression::Value(0))
        }),
        Ok(0)
    );
    assert_eq!(
        eval(Expression::Op {
            op: Operation::Sub,
            left: Box::new(Expression::Value(0)),
            right: Box::new(Expression::Value(0))
        }),
        Ok(0)
    );
}

fn test_error() {
    assert_eq!(
        eval(Expression::Op {
            op: Operation::Div,
            left: Box::new(Expression::Value(99)),
            right: Box::new(Expression::Value(0)),
        }),
        Err(String::from("ділення на нуль"))
    );
}

fn main() {
    let expr = Expression::Op {
        op: Operation::Sub,
```

```
        left: Box::new(Expression::Value(20)),
        right: Box::new(Expression::Value(10)),
    };
    println!("expr: {expr:?}");
    println!("результат: {:?}", eval(expr));
}
```

Розділ 13

Методи та Трейти

Цей сегмент повинен зайняти близько 50 хвилин. Він містить:

Слайд	Тривалість
Методи	10 хвилин
Трейти	15 хвилин
Виведення	3 хвилини
Вправа: Загальний логгер	20 хвилин

13.1 Методи

Rust дозволяє пов'язувати функції з новими типами. Ви робите це за допомогою блоку `impl`:

```
struct Race {
    name: String,
    laps: Vec<i32>,
}

impl Race {
    // Немає отримувача, статичний метод
    fn new(name: &str) -> Self {
        Self { name: String::from(name), laps: Vec::new() }
    }

    // Ексклюзивний запозичений доступ на читання та запис до себе
    fn add_lap(&mut self, lap: i32) {
        self.laps.push(lap);
    }

    // Спільний та запозичений доступ тільки на читання до себе
    fn print_laps(&self) {
        println!("Записано {} кіл для {}:", self.laps.len(), self.name);
        for (idx, lap) in self.laps.iter().enumerate() {
```

```
            println!("Коло {idx}: {lap} sec");
        }
    }

    // Виключне володіння собою (про це пізніше)
    fn finish(self) {
        let total: i32 = self.laps.iter().sum();
        println!("Гонка {} завершена, загальний час проходження кола: {}", self.name, to
    }
}

fn main() {
    let mut race = Race::new("Гран-прі Монако");
    race.add_lap(70);
    race.add_lap(68);
    race.print_laps();
    race.add_lap(71);
    race.print_laps();
    race.finish();
    // race.add_lap(42);
}
```

Аргументи `self` визначають ”отримувача” - об'єкт, на який діє метод. Існує декілька типових отримувачів для методу:

- `&self`: запозичує об’єкт у викликувача за допомогою спільного та незмінного посилання. Після цього об’єкт можна використовувати знову.
- `&mut self`: запозичує об’єкт у викликувача, використовуючи унікальне та мутабельне посилання. Після цього об’єкт можна використовувати знову.
- `self`: приймає право власності на об'єкт і переміщує його від викликувача. Метод стає власником об'єкта. Об’єкт буде видалено (звільнено), коли метод завершиться, якщо володіння їм не передано явно. Повне володіння не означає автоматичної мутабельності.
- `mut self`: те саме, що й вище, але метод може змінювати об’єкт.
- Немає отримувача: це стає статичним методом у структурі. Зазвичай використовується для створення конструкторів, які за домовленістю називаються `new`.

Ключові моменти:

- Може бути корисно представити методи, порівнюючи їх із функціями.
  - Методи викликаються для екземпляра типу (такі як структура або перелік), перший параметр представляє екземпляр як `self`.
  - Розробники можуть використовувати методи, щоб скористатися перевагами синтаксису отримувача методів і допомогти їм бути більш організованими. Використовуючи методи, ми можемо зберігати весь код реалізації в одному передбачуваному місці.
- Зверніть увагу на використання ключового слова `self`, отримувача методу.
  - Покажіть, що це скорочений термін для `self: Self` і, можливо, покажіть, як можна також використовувати назву структури.
  - Поясніть, що `Self` — це псевдонім типу для типу, до якого входить блок `impl`, і його можна використовувати деінде в блоці.
  - Зауважте, що `self` використовується, як і інші структури, і крапкова нотація

може використовуватися для посилання на окремі поля.
  - Це може бути гарний час, щоб продемонструвати, чим `&self` відрізняється від `self`, спробувавши запустити `finish` двічі.
  - Окрім варіантів `self`, існують також спеціальні типи обгорток, які можуть бути типами отримувачів, наприклад `Box<Self>`.

13.2 Трейти

Rust дозволяє абстрагування над типами за допомогою трейтів. Вони схожі на інтерфейси:

```
trait Pet {
    /// Повертає речення від цього вихованця.
    fn talk(&self) -> String;

    /// Виводить на термінал рядок привітання цього вихованця.
    fn greet(&self);
}
```

- Трейт визначає ряд методів, які повинні мати типи, щоб реалізувати цій трейт.
- Далі у розділі "Узагальнення" ми побачимо, як побудувати функціональність, яка є загальною для всіх типів, що реалізують трейт.

13.2.1 Реалізація трейтів

```
trait Pet {
    fn talk(&self) -> String;

    fn greet(&self) {
        println!("Який же ти милий! Як тебе звати? {}", self.talk());
    }
}

struct Dog {
    name: String,
    age: i8,
}

impl Pet for Dog {
    fn talk(&self) -> String {
        format!("Гав, мене звуть {}!", self.name)
    }
}

fn main() {
    let fido = Dog { name: String::from("Фідо"), age: 5 };
    fido.greet();
}
```

- Щоб реалізувати `Trait` для `Type`, ви використовуєте `impl Trait for Type { .. }` блок.

- На відміну від інтерфейсів Go, просто мати відповідні методи недостатньо: тип `Cat` з методом `talk()` не буде автоматично задовольняти `Pet`, якщо він не знаходиться у блоці `impl Pet`.
- Трейти можуть надавати реалізації за замовчуванням для деяких методів. Реалізації за замовчуванням можуть покладатися на всі методи трейту. У цьому випадку надається `greet`, який покладається на `talk`.

13.2.2 Супертрейти

Трейт може вимагати, щоб типи, які його реалізують, також реалізовували інші трейти, так звані *супертрейти*. У цьому випадку, будь-який тип, що реалізує `Pet`, повинен реалізувати `Animal`.

```
trait Animal {
    fn leg_count(&self) -> u32;
}

trait Pet: Animal {
    fn name(&self) -> String;
}

struct Dog(String);

impl Animal for Dog {
    fn leg_count(&self) -> u32 {
        4
    }
}

impl Pet for Dog {
    fn name(&self) -> String {
        self.0.clone()
    }
}

fn main() {
    let puppy = Dog(String::from("Рекс"));
    println!("{} має {} ніг", puppy.name(), puppy.leg_count());
}
```

Іноді це називають ”успадкуванням трейтів”, але студенти не повинні очікувати, що це буде схоже на успадкування об'єктів ОО. Це просто вказує додаткову вимогу до реалізації трейту.

13.2.3 Асоційовані типи

Асоціативні типи - це типи-заповнювачі, які надаються реалізацією трейту.

```
struct Meters(i32);
struct MetersSquared(i32);
```

```
trait Multiply {
    type Output;
    fn multiply(&self, other: &Self) -> Self::Output;
}

impl Multiply for Meters {
    type Output = MetersSquared;
    fn multiply(&self, other: &Self) -> Self::Output {
        MetersSquared(self.0 * other.0)
    }
}

fn main() {
    println!("{:?}", Meters(10).multiply(&Meters(20)));
}
```

- Асоціативні типи іноді також називають ”вихідними типами”. Ключовим зауваженням є те, що цей тип вибирає реалізатор, а не той, хто його викликає.
- Багато стандартних бібліотечних трейтів мають асоційовані типи, включаючи арифметичні оператори та `Iterator`.

13.3 Виведення

Підтримувані трейти можуть бути автоматично застосовані до ваших кастомних типів наступним чином:

```
struct Player {
    name: String,
    strength: u8,
    hit_points: u8,
}

fn main() {
    let p1 = Player::default(); // Трейт Default додає `default` конструктор .
    let mut p2 = p1.clone(); // Трейт Clone додає `clone` метод.
    p2.name = String::from("EldurScrollz");
    // Трейт Debug додає підтримку друку з `{:?}`.
    println!("{p1:?} проти {p2:?}");
}
```

Виведення реалізовано за допомогою макросів, і багато крейтів надають корисні макроси виведення для додавання корисної функціональності. Наприклад, `serde` може виводити підтримку серіалізації для структури за допомогою `#[derive(Serialize)]`.

13.4 Вправа: Трейт логгера

Давайте розробимо просту утиліту для ведення логів, використовуючи трейт `Logger` з методом `log`. Код, який може реєструвати свій прогрес, може отримати `&impl Logger`. Під час тестування це може призвести до запису повідомлень до тестового лог-файлу, тоді як у виробничій збірці повідомлення надсилатимуться до сервера логів.

Однак, наведений нижче `StdoutLogger` реєструє всі повідомлення, незалежно від їхньої докладності. Ваше завдання - написати тип `VerbosityFilter`, який ігноруватиме повідомлення з максимальною докладністю.

Це поширений патерн: структура, що обгортає реалізацію трейту і реалізує той самий трейт, додаючи поведінку в процесі. Які ще типи обгорток можуть бути корисними у утиліті для ведення логів?

```
pub trait Logger {
    /// Запишіть повідомлення із заданим рівнем докладності.
    fn log(&self, verbosity: u8, message: &str);
}

struct StdoutLogger;

impl Logger for StdoutLogger {
    fn log(&self, verbosity: u8, message: &str) {
        println!("verbosity={verbosity}: {message}");
    }
}

// TODO: Визначте та реалізуйте `VerbosityFilter`.

fn main() {
    let logger = VerbosityFilter { max_verbosity: 3, inner: StdoutLogger };
    logger.log(5, "FYI");
    logger.log(2, "Uhoh");
}
```

13.4.1 Рішення

```
pub trait Logger {
    /// Запишіть повідомлення із заданим рівнем докладності.
    fn log(&self, verbosity: u8, message: &str);
}

struct StdoutLogger;

impl Logger for StdoutLogger {
    fn log(&self, verbosity: u8, message: &str) {
        println!("verbosity={verbosity}: {message}");
    }
}

/// Записуйте повідомлення лише до заданого рівня докладності.
struct VerbosityFilter {
    max_verbosity: u8,
    inner: StdoutLogger,
}

impl Logger for VerbosityFilter {
    fn log(&self, verbosity: u8, message: &str) {
```

```
        if verbosity <= self.max_verbosity {
            self.inner.log(verbosity, message);
        }
    }
}

fn main() {
    let logger = VerbosityFilter { max_verbosity: 3, inner: StdoutLogger };
    logger.log(5, "FYI");
    logger.log(2, "Uhoh");
}
```

Частина IV

День 2: Полудень

Розділ 14

Ласкаво просимо назад

Враховуючи 10-хвилинні перерви, ця сесія має тривати близько 3 годин 15 хвилин. Вона містить:

Сегмент	Тривалість
Узагальнені типи	45 хвилин
Типи стандартної бібліотеки	1 година
Трейти стандартної бібліотеки	1 година та 10 хвилин

Розділ 15

Узагальнені типи

Цей сегмент повинен зайняти близько 45 хвилин. Він містить:

Слайд	Тривалість
Узагальнені функції	5 хвилин
Узагальнені типи даних	10 хвилин
Обмеження трейту	10 хвилин
impl Trait	5 хвилин
dyn Trait	5 хвилин
Вправа: узагальнена min	10 хвилин

15.1 Узагальнені функції

Rust підтримує узагальнені типи, що дозволяє абстрагувати алгоритми або структури даних (наприклад, сортування або бінарне дерево) від типів, що використовуються або зберігаються.

```
/// Виберіть `even` або `odd` в залежності від значення `n`.
fn pick<T>(n: i32, even: T, odd: T) -> T {
    if n % 2 == 0 {
        even
    } else {
        odd
    }
}

fn main() {
    println!("вибраний номер: {:?}", pick(97, 222, 333));
    println!("вибрав рядок: {:?}", pick(28, "собака", "кіт"));
}
```

- Rust визначає тип для T на основі типів аргументів та значення, що повертається.
- У цьому прикладі ми використовуємо лише примітивні типи `i32` та `&str` для T, але ми можемо використовувати будь-який тип, включаючи типи, визначені

користувачем:

```
struct Foo {
    val: u8,
}

pick(123, Foo { val: 7 }, Foo { val: 456 });
```

- Це схоже на шаблони C++, але Rust частково компілює узагальнену функцію одразу, тому ця функція має бути валідною для всіх типів, що відповідають обмеженням. Наприклад, спробуйте модифікувати `pick` так, щоб вона повертала `even + odd`, якщо `n == 0`. Навіть якщо використовується лише екземпляр `pick` з цілими числами, Rust все одно вважатиме його невірним. C++ дозволить вам зробити це.
- Узагальнений код перетворюється на не-узагальнений на основі сайтів виклику. Це абстракція з нульовою вартістю: ви отримуєте точно такий же результат, як якщо б ви написали структури даних власноруч без абстракції.

15.2 Узагальнені типи даних

Ви можете використовувати узагальнення для абстрагування від конкретного типу поля:

```
struct Point<T> {
    x: T,
    y: T,
}

impl<T> Point<T> {
    fn coords(&self) -> (&T, &T) {
        (&self.x, &self.y)
    }

    fn set_x(&mut self, x: T) {
        self.x = x;
    }
}

fn main() {
    let integer = Point { x: 5, y: 10 };
    let float = Point { x: 1.0, y: 4.0 };
    println!("{integer:?} та {float:?}");
    println!("координати: {:?}", integer.coords());
}
```

- *З:* Чому `T` вказаний двічі в `impl<T> Point<T> {}`? Хіба це не зайве?
  - Це пояснюється тим, що це частина узагальненої реалізації для узагальненого типу. Вони є узагальненими незалежно один від одного..
  - Це означає, що ці методи визначені для будь-якого `T`.
  - Можна написати `impl Point<u32> { .. }`.
    * `Point` все ще є узагальненим типом, і ви можете використовувати `Point<f64>`, але методи в цьому блоці будуть доступні лише для

`Point<u32>`.

- Спробуйте оголосити нову змінну `let p = Point { x: 5, y: 10.0 };`. Оновіть код, щоб дозволити створювати точки, які мають елементи різних типів, використовуючи дві змінні типу, наприклад, `T` і `U`.

15.3 Узагальнені трейти

Трейти також можуть бути загальними, так само як типи та функції. Параметри трейту отримують конкретні типи під час його використання.

```
struct Foo(String);

impl From<u32> for Foo {
    fn from(from: u32) -> Foo {
        Foo(format!("Перетворено з цілого числа: {from}"))
    }
}

impl From<bool> for Foo {
    fn from(from: bool) -> Foo {
        Foo(format!("Перетворено з булевого значення: {from}"))
    }
}

fn main() {
    let from_int = Foo::from(123);
    let from_bool = Foo::from(true);
    println!("{from_int:?}, {from_bool:?}");
}
```

- Трейт `From` буде розглянутий пізніше у курсі, але її визначення у документації `std` є простим.
- Реалізації трейту не обов'язково повинні охоплювати всі можливі параметри типів. У цьому випадку `Foo::from("hello")` не буде скомпільовано, оскільки для `Foo` не існує реалізації `From<&str>`.
- Узагальнені трейти приймають типи як "вхідні", тоді як асоціативні типи є своєрідним "вихідним" типом. Трейт може мати декілька реалізацій для різних вхідних типів.
- Ведеться робота над додаванням цієї підтримки, яка називається спеціалізація.

15.4 Обмеження трейту

При роботі з узагальненнями ви часто потребуєте, щоб типи реалізовували деякий трейт, щоб ви могли викликати методи цього трейту.

Ви можете зробити це за допомогою `T: Trait`:

```
fn duplicate<T: Clone>(a: T) -> (T, T) {
    (a.clone(), a.clone())
```

```
}

// struct NotCloneable;

fn main() {
    let foo = String::from("foo");
    let pair = duplicate(foo);
    println!("{pair:?}");
}
```

- Спробуйте зробити `NonCloneable` і передати його в `duplicate`.
- Якщо потрібно вказати декілька трейтів, використовуйте `+`, щоб об'єднати їх.
- Покажіть вираз `where`, студенти зустрінуться з ним під час читання коду.

  ```
  fn duplicate<T>(a: T) -> (T, T)
  where
      T: Clone,
  {
      (a.clone(), a.clone())
  }
  ```

  - Це розчищає сигнатуру функції, якщо у вас багато параметрів.
  - Він має додаткові функції, що робить його більш потужним.
    * Якщо хтось запитає, додаткова можливість полягає в тому, що тип ліворуч від ":" може бути довільним, наприклад `Option<T>`.
- Зауважте, що Rust (поки що) не підтримує спеціалізацію. Наприклад, за наявності оригінального `duplicate` додавання спеціалізованого `duplicate(a: u32)` є некоректним.

15.5 `impl Trait`

Подібно до меж трейтів, синтаксис `impl Trait` можна використовувати в аргументах функції та значеннях, що повертаються:

```
// Синтаксичний цукор для:
//   fn add_42_millions<T: Into<i32>>(x: T) -> i32 {
fn add_42_millions(x: impl Into<i32>) -> i32 {
    x.into() + 42_000_000
}

fn pair_of(x: u32) -> impl std::fmt::Debug {
    (x + 1, x - 1)
}

fn main() {
    let many = add_42_millions(42_i8);
    println!("{many}");
    let many_more = add_42_millions(10_000_000);
    println!("{many_more}");
    let debuggable = pair_of(27);
```

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

`impl Trait` дозволяє працювати з типами, які ви не можете назвати. Значення `impl Trait` дещо відрізняється у різних позиціях.

- У випадку параметра, `impl Trait` - це як анонімний загальний параметрр з обмеженням трейту.
- Для типу, що повертається, це означає, що тип, що повертається, є деяким конкретним типом, який реалізує трейт, без назви типу. Це може бути корисно, коли ви не хочете викривати конкретний тип у публічному API.

  У позиції повернення виведення є складним. Функція, що повертає `impl Foo`, вибирає конкретний тип, який вона повертає, не записуючи його у вихідному коді. Функція, що повертає узагальнений тип, наприклад, `collect<B>() -> B`, може повернути будь-який тип, що задовольняє `B`, і користувачеві може знадобитися вибрати один з них, наприклад, за допомогою `let x: Vec<_> = foo.collect()` або turbofish, `foo.collect::<Vec<_>>()`.

Який тип `debuggable`? Спробуйте `let debuggable: () = ..`, щоб побачити повідомлення про помилку.

15.6 `dyn Trait`

На додаток до використання трейтів для статичного пересилання за допомогою узагальнень, Rust також підтримує їх використання для динамічного пересилання зі стиранням типу за допомогою об'єктів трейтів:

```
struct Dog {
    name: String,
    age: i8,
}
struct Cat {
    lives: i8,
}

trait Pet {
    fn talk(&self) -> String;
}

impl Pet for Dog {
    fn talk(&self) -> String {
        format!("Гав, мене звуть {}!", self.name)
    }
}

impl Pet for Cat {
    fn talk(&self) -> String {
        String::from("Мяу!")
    }
}
```

```
// Використовує узагальнення та статичну диспетчеризацію.
fn generic(pet: &impl Pet) {
    println!("Привіт, ви хто? {}", pet.talk());
}

// Використовує стирання типів та динамічну диспетчеризацію.
fn dynamic(pet: &dyn Pet) {
    println!("Привіт, ви хто? {}", pet.talk());
}

fn main() {
    let cat = Cat { lives: 9 };
    let dog = Dog { name: String::from("Фідо"), age: 5 };

    generic(&cat);
    generic(&dog);

    dynamic(&cat);
    dynamic(&dog);
}
```

- Узагальнення, включаючи `impl Trait`, використовують мономорфізацію для створення спеціалізованого екземпляру функції для кожного окремого типу, який є екземпляром узагальнення. Це означає, що виклик методу трейта з узагальненої функції все ще використовує статичну диспетчеризацію, оскільки компілятор має повну інформацію про тип і може вирішити, яку саме реалізацію трейта типу слід використовувати.
- При використанні `dyn Trait` замість цього використовується динамічна диспетчеризація через віртуальну таблицю методів (vtable). Це означає, що існує єдина версія `fn dynamic`, яка використовується незалежно від того, який тип `Pet` передано.
- При використанні `dyn Trait` об'єкт трейта повинен знаходитися за якимось посередником. У цьому випадку це буде посилання, хоча також можна використовувати розумні типи вказівників, такі як `Box` (це буде продемонстровано у день 3).
- Під час виконання `&dyn Pet` представляється як "жирний вказівник", тобто пара з двох вказівників: Один вказівник вказує на конкретний об'єкт, який реалізує `Pet`, а інший вказує на таблицю vtable для реалізації трейту для цього типу. При виклику методу `talk` на `&dyn Pet` компілятор шукає вказівник на функцію `talk` у таблиці vtable, а потім викликає цю функцію, передаючи вказівник на `Dog` або `Cat` у цю функцію. Для цього компілятору не потрібно знати конкретний тип `Pet`.
- `dyn Trait` вважається "стертим типом", оскільки під час компіляції ми більше не знаємо, яким є конкретний тип.

15.7 Вправа: узагальнена `min`

У цій короткій вправі ви реалізуєте узагальнену функцію `min`, яка визначає мінімальне з двох значень, використовуючи трейт `Ord`.

```rust
use std::cmp::Ordering;

// TODO: реалізуйте функцію `min`, яка використовується в `main`.

fn main() {
    assert_eq!(min(0, 10), 0);
    assert_eq!(min(500, 123), 123);

    assert_eq!(min('a', 'z'), 'a');
    assert_eq!(min('7', '1'), '1');

    assert_eq!(min("привіт", "до побачення"), "до побачення");
    assert_eq!(min("кажан", "броненосець"), "броненосець");
}
```

- Покажіть учням трейт `Ord` та перелік `Ordering`.

15.7.1 Рішення

```rust
use std::cmp::Ordering;

fn min<T: Ord>(l: T, r: T) -> T {
    match l.cmp(&r) {
        Ordering::Less | Ordering::Equal => l,
        Ordering::Greater => r,
    }
}

fn main() {
    assert_eq!(min(0, 10), 0);
    assert_eq!(min(500, 123), 123);

    assert_eq!(min('a', 'z'), 'a');
    assert_eq!(min('7', '1'), '1');

    assert_eq!(min("привіт", "до побачення"), "до побачення");
    assert_eq!(min("кажан", "броненосець"), "броненосець");
}
```

Розділ 16

Типи стандартної бібліотеки

Цей сегмент повинен зайняти близько 1 години. Він містить:

Слайд	Тривалість
Стандартна бібліотека	3 хвилини
Документація	5 хвилин
Option	10 хвилин
Result	5 хвилин
String	5 хвилин
Vec	5 хвилин
HashMap	5 хвилин
Вправа: Лічильник	20 хвилин

Для кожного слайда в цьому розділі витратьте деякий час на перегляд сторінок документації, виділяючи деякі з найбільш поширених методів.

16.1 Стандартна бібліотека

Rust поставляється зі стандартною бібліотекою, яка допомагає встановити набір загальних типів, які використовуються бібліотекою та програмами Rust. Таким чином, дві бібліотеки можуть безперешкодно працювати разом, оскільки обидві використовують той самий тип `String`.

Насправді Rust містить кілька рівнів стандартної бібліотеки: `core`, `alloc` і `std`.

- `core` включає найпростіші типи та функції, які не залежать від `libc`, розподілювача чи навіть наявності операційної системи.
- `alloc` включає типи, для яких потрібен глобальний розподільник купи, наприклад `Vec`, `Box` і `Arc`.
- Вбудовані програми Rust часто використовують лише `core`, та іноді `alloc`.

16.2 Документація

Rust постачається з обширною документацією. Наприклад:

- Всі подробиці про цикли.
- Примітивні типи на зразок u8
- Типи стандартної бібліотеки, такі як `Option` або `BinaryHeap`.

Використовуйте `rustup doc --std` або https://std.rs для перегляду документації.

Фактично, ви можете документувати свій власний код:

```
/// Визначити, чи ділиться перший аргумент на другий.
///
/// Якщо другий аргумент дорівнює нулю, результат буде false.
fn is_divisible_by(lhs: u32, rhs: u32) -> bool {
    if rhs == 0 {
        return false;
    }
    lhs % rhs == 0
}
```

Контент розглядається як Markdown. Усі опубліковані крейти бібліотеки Rust автоматично документуються на `docs.rs` за допомогою rustdoc. Це ідіоматично документувати всі публічні елементи в API за допомогою цього шаблону.

Щоб задокументувати елемент із середини елемента (наприклад, всередині модуля), використовуйте `//!` або `/*! .. */`, які називаються "внутрішні коментарі до документу":

```
//! Цей модуль містить функціональність, пов'язану з подільністю цілих чисел.
```

- Покажіть студентам згенеровану документацію для крейта `rand` на https://docs.rs/rand.

16.3 Option

Ми вже бачили деяке використання `Option<T>`. Це зберігає або значення типу `T`, або нічого. Наприклад, `String::find` повертає `Option<usize>`.

```
fn main() {
    let name = "Löwe 老虎 Léopard Gepardi";
    let mut position: Option<usize> = name.find('é');
    println!("пошук повернув {position:?}");
    assert_eq!(position.unwrap(), 14);
    position = name.find('Z');
    println!("пошук повернув {position:?}");
    assert_eq!(position.expect("Символ не знайдено"), 0);
}
```

- `Option` широко використовуються, і не тільки в стандартній бібліотеці.
- `unwrap` поверне значення в `Option`, або паніку. `expect` працює аналогічно, але повертає повідомлення про помилку.

  - Ви можете панікувати на None, але ви не можете ”випадково” забути перевірити на None.
  - Під час швидкої експерементації зазвичай прийнято використовувати unwrap/expect повсюди, але у виробничому коді None зазвичай обробляється у більш зручному вигляді.
- ”Нішова оптимізація” означає, що Option<T> часто має той самий розмір у пам'яті, що й T, якщо є деяке представлення, яке не є допустимим значенням T. Наприклад, посилання не може бути NULL, тому Option<&T> автоматично використовує NULL для представлення варіанту None, і таким чином може зберігатися у тій самій пам'яті, що й &T.

16.4 Result

Result схожий на Option, але вказує на успіх або невдачу операції, кожен з яких має свій варіант переліку. Він має вигляд: Result<T, E>, де T використовується у варіанті Ok, а E з'являється у варіанті Err.

```
use std::fs::File;
use std::io::Read;

fn main() {
    let file: Result<File, std::io::Error> = File::open("diary.txt");
    match file {
        Ok(mut file) => {
            let mut contents = String::new();
            if let Ok(bytes) = file.read_to_string(&mut contents) {
                println!("Дорогий щоденник: {contents} ({bytes} байтів)");
            } else {
                println!("Не вдалося прочитати вміст файлу");
            }
        }
        Err(err) => {
            println!("Щоденник не вдалося відкрити: {err}");
        }
    }
}
```

- Як і у випадку з Option, успішне значення знаходиться всередині Result, змушуючи розробника явно витягти його. Це стимулює перевірку помилок. У випадку, коли помилка взагалі не очикується, можна викликати unwrap() або expect(), і це також є сигналом про наміри розробника.
- Порекомендуйте прочитати Result документацію. Не під час курсу, але варто згадати. Вона містить багато зручних методів і функцій, які допомагають програмувати у функціональному стилі.
- Result — це стандартний тип для реалізації обробки помилок, як ми побачимо у 4-му дні.

16.5 String

String — це розширюваний рядок у кодуванні UTF-8:

```
fn main() {
    let mut s1 = String::new();
    s1.push_str("Привіт");
    println!("s1: len = {}, capacity = {}", s1.len(), s1.capacity());

    let mut s2 = String::with_capacity(s1.len() + 1);
    s2.push_str(&s1);
    s2.push('!');
    println!("s2: len = {}, capacity = {}", s2.len(), s2.capacity());

    let s3 = String::from("🇨🇭");
    println!("s3: len = {}, number of chars = {}", s3.len(), s3.chars().count());
}
```

`String` реалізує `Deref<Target = str>`, що означає, що ви можете викликати усі методи `str` у `String`.

- `String::new` повертає новий порожній рядок. Використовуйте `String::with_capacity`, якщо ви знаєте, скільки даних ви хочете передати в рядок.
- `String::len` повертає розмір `String` у байтах (який може відрізнятися від його довжини в символах).
- `String::chars` повертає ітератор поверх фактичних символів. Зауважте, що `char` може відрізнятися від того, що людина вважатиме "символом" через кластери графем.
- Коли люди посилаються на рядки, вони можуть говорити про `&str` або `String`.
- Коли тип реалізує `Deref<Target = T>`, компілятор дозволить вам прозоро викликати методи з `T`.
  - Ми ще не обговорювали трейт `Deref`, тому на даний момент це здебільшого пояснює структуру бокової панелі у документації.
  - `String` реалізує `Deref<Target = str>`, що прозоро надає йому доступ до методів `str`.
  - Напишіть і порівняйте `let s3 = s1.deref();` and `let s3 = &*s1;`
- `String` реалізовано як оболонку навколо вектора байтів, багато операцій, які ви бачите, що підтримуються над векторами, також підтримуються `String`, але з деякими додатковими гарантіями.
- Порівняйте різні способи індексування `String`:
  - До символу за допомогою `s3.chars().nth(i).unwrap()`, де `i` є в межі, поза межами.
  - До підрядка за допомогою `s3[0..4]`, де цей фрагмент знаходиться на межах символів чи ні.
- Багато типів можна перетворити у рядок за допомогою методу `to_string`. Цей трейт автоматично реалізується для всіх типів, що реалізують `Display`, тому все, що може бути відформатовано, також може бути перетворено у рядок.

16.6 Vec

Vec — стандартний буфер із змінним розміром, виділений у купі:

```rust
fn main() {
    let mut v1 = Vec::new();
    v1.push(42);
    println!("v1: len = {}, capacity = {}", v1.len(), v1.capacity());

    let mut v2 = Vec::with_capacity(v1.len() + 1);
    v2.extend(v1.iter());
    v2.push(9999);
    println!("v2: len = {}, capacity = {}", v2.len(), v2.capacity());

    // Канонічний макрос для ініціалізації вектора з елементами.
    let mut v3 = vec![0, 0, 1, 2, 3, 4];

    // Зберігаємо тільки парні елементи.
    v3.retain(|x| x % 2 == 0);
    println!("{v3:?}");

    // Видаляємо дублікати, що йдуть підряд.
    v3.dedup();
    println!("{v3:?}");
}
```

Vec реалізує Deref<Target = [T]>, який означає, що ви можете викликати методи зрізу на Vec.

- Vec — це тип колекції разом із String і HashMap. Дані, які він містить, зберігаються в купі. Це означає, що кількість даних не потрібно знати під час компіляції. Він може рости або зменшуватися під час виконання.
- Зверніть увагу, що Vec<T> також є узагальненим типом, але вам не потрібно вказувати T явно. Як завжди з визначенням типу Rust, T було встановлено під час першого виклику push.
- vec![...] — це канонічний макрос для використання замість Vec::new(), який підтримує додавання початкових елементів до вектора.
- Щоб індексувати вектор, ви використовуєте [ ], але вони панікують, якщо вийдуть за межі. Крім того, використання get поверне Option. Функція pop видалить останній елемент.
- Зрізи розглядаються на 3-й день. Наразі студентам потрібно лише знати, що значення типу Vec дає доступ до всіх задокументованих методів зрізів.

16.7 HashMap

Стандартна хеш-карта із захистом від HashDoS-атак:

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

fn main() {
    let mut page_counts = HashMap::new();
    page_counts.insert("Adventures of Huckleberry Finn", 207);
    page_counts.insert("Grimms' Fairy Tales", 751);
    page_counts.insert("Pride and Prejudice", 303);
```

```
    if !page_counts.contains_key("Les Misérables") {
        println!(
            "Ми знаємо про {} книги, але не Les Misérables.",
            page_counts.len()
        );
    }

    for book in ["Pride and Prejudice", "Alice's Adventure in Wonderland"] {
        match page_counts.get(book) {
            Some(count) => println!("{book}: {count} сторінок"),
            None => println!("{book} невідома."),
        }
    }

    // Використовуйте метод .entry(), щоб вставити значення, якщо нічого не знайдено.
    for book in ["Pride and Prejudice", "Alice's Adventure in Wonderland"] {
        let page_count: &mut i32 = page_counts.entry(book).or_insert(0);
        *page_count += 1;
    }

    println!("{page_counts:#?}");
}
```

- `HashMap` не визначено в prelude, і її потрібно включити в область.
- Спробуйте наступні рядки коду. У першому рядку буде показано, чи є книга в хеш-мапі, і якщо ні, повернеться альтернативне значення. У другому рядку буде вставлено альтернативне значення в хеш-мапу, якщо книга не знайдена.

  ```
  let pc1 = page_counts
      .get("Harry Potter and the Sorcerer's Stone")
      .unwrap_or(&336);
  let pc2 = page_counts
      .entry("The Hunger Games")
      .or_insert(374);
  ```

- На відміну від `vec!`, на жаль, немає стандартного макросу `hashmap!`.
  - Хоча, починаючи з Rust 1.56, в HashMap реалізовано `From<[(K, V); N]>`, що дозволяє легко ініціалізувати хеш-карту з літерального масиву:

    ```
    let page_counts = HashMap::from([
      ("Harry Potter and the Sorcerer's Stone".to_string(), 336),
      ("The Hunger Games".to_string(), 374),
    ]);
    ```

- Крім того, HashMap можна створити з будь-якого `Iterator`, який видає кортежі ключ-значення.
- Цей тип має кілька "специфічних" типів повернення, таких як `std::collections::hash_map::Keys`. Ці типи часто з'являються під час пошуку в документації Rust. Покажіть учням документацію для цього типу та корисне посилання на метод `keys`.

16.8 Вправа: Лічильник

У цій вправі ви візьмете дуже просту структуру даних і зробите її узагальненою. Вона використовує `std::collections::HashMap` для відстеження того, які значення було переглянуто і скільки разів кожне з них з'являлося.

Початкова версія `Counter` жорстко налаштована на роботу лише зі значеннями `u32`. Зробіть структуру та її методи узагальненими щодо типу значення, яке відстежується, таким чином `Counter` зможе відстежувати будь-який тип значення.

Якщо ви закінчите раніше, спробуйте використати метод `entry`, щоб вдвічі зменшити кількість переглядів хешу, необхідних для реалізації методу `count`.

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

/// Counter підраховує кількість разів, коли кожне значення типу T було переглянуто.
struct Counter {
    values: HashMap<u32, u64>,
}

impl Counter {
    /// Створює новий Counter.
    fn new() -> Self {
        Counter {
            values: HashMap::new(),
        }
    }

    /// Підраховує входження заданого значення.
    fn count(&mut self, value: u32) {
        if self.values.contains_key(&value) {
            *self.values.get_mut(&value).unwrap() += 1;
        } else {
            self.values.insert(value, 1);
        }
    }

    /// Повертає кількість разів, коли було побачено задане значення.
    fn times_seen(&self, value: u32) -> u64 {
        self.values.get(&value).copied().unwrap_or_default()
    }
}

fn main() {
    let mut ctr = Counter::new();
    ctr.count(13);
    ctr.count(14);
    ctr.count(16);
    ctr.count(14);
    ctr.count(14);
    ctr.count(11);
```

```
    for i in 10..20 {
        println!("побачив {} значень рівних {}", ctr.times_seen(i), i);
    }

    let mut strctr = Counter::new();
    strctr.count("яблуко");
    strctr.count("апельсин");
    strctr.count("яблуко");
    println!("отримав {} яблук", strctr.times_seen("яблуко"));
}
```

16.8.1 Рішення

```
use std::collections::HashMap;
use std::hash::Hash;

/// Counter підраховує кількість разів, коли кожне значення типу T було переглянуто.
struct Counter<T> {
    values: HashMap<T, u64>,
}

impl<T: Eq + Hash> Counter<T> {
    /// Створює новий Counter.
    fn new() -> Self {
        Counter { values: HashMap::new() }
    }

    /// Підраховує входження заданого значення.
    fn count(&mut self, value: T) {
        *self.values.entry(value).or_default() += 1;
    }

    /// Повертає кількість разів, коли було побачено задане значення.
    fn times_seen(&self, value: T) -> u64 {
        self.values.get(&value).copied().unwrap_or_default()
    }
}

fn main() {
    let mut ctr = Counter::new();
    ctr.count(13);
    ctr.count(14);
    ctr.count(16);
    ctr.count(14);
    ctr.count(14);
    ctr.count(11);

    for i in 10..20 {
        println!("побачив {} значень рівних {}", ctr.times_seen(i), i);
    }
```

```
    let mut strctr = Counter::new();
    strctr.count("яблуко");
    strctr.count("апельсин");
    strctr.count("яблуко");
    println!("отримав {} яблук", strctr.times_seen("яблуко"));
}
```

Розділ 17

Трейти стандартної бібліотеки

Цей сегмент повинен зайняти близько 1 години 10 хвилин. Він містить:

Слайд	Тривалість
Порівняння	5 хвилин
Оператори	5 хвилин
From та Into	5 хвилин
Приведення	5 хвилин
Read та Write	5 хвилин
Default, синтаксис оновлення структури	5 хвилин
Закриття	10 хвилин
Вправа: ROT13	30 хвилин

Як і у випадку з типами стандартної бібліотеки, витратьте час на ознайомлення з документацією для кожного трейта.

Цей розділ довгий. Зробіть перерву на півдорозі.

17.1 Порівняння

Ці трейти підтримують порівняння між значеннями. Усі трейти можуть бути визначені для типів, що містять поля, які реалізують ці трейти

`PartialEq` та `Eq`

`PartialEq- це відношення часткової еквівалентності, з обов'язковим методомeqта наданим методомne. Оператори ==та!=‘ викликають ці методи.`

```
struct Key {
    id: u32,
    metadata: Option<String>,
}
```

```rust
impl PartialEq for Key {
    fn eq(&self, other: &Self) -> bool {
        self.id == other.id
    }
}
```

Eq - це відношення повної еквівалентності (рефлексивне, симетричне та транзитивне) і передбачає `PartialEq`. Функції, які вимагають повної еквівалентності, використовуватимуть Eq як обмеження трейту.

`PartialOrd` та `Ord`

`PartialOrd` визначає часткове впорядкування за допомогою методу `partial_cmp`. Він використовується для реалізації операторів <, <=, >= та >.

```rust
use std::cmp::Ordering;
struct Citation {
    author: String,
    year: u32,
}
impl PartialOrd for Citation {
    fn partial_cmp(&self, other: &Self) -> Option<Ordering> {
        match self.author.partial_cmp(&other.author) {
            Some(Ordering::Equal) => self.year.partial_cmp(&other.year),
            author_ord => author_ord,
        }
    }
}
```

`Ord` - це повне впорядкування, з `cmp` який повертає `Ordering`.

`PartialEq` може бути реалізовано між різними типами, але `Eq` не може, тому що є рефлексивним:

```rust
struct Key {
    id: u32,
    metadata: Option<String>,
}
impl PartialEq<u32> for Key {
    fn eq(&self, other: &u32) -> bool {
        self.id == *other
    }
}
```

На практиці ці трейти часто виводяться, але рідко реалізуються.

17.2 Оператори

Перевантаження операторів реалізовано за допомогою трейтів у `std::ops`:

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

```
}

impl std::ops::Add for Point {
    type Output = Self;

    fn add(self, other: Self) -> Self {
        Self { x: self.x + other.x, y: self.y + other.y }
    }
}

fn main() {
    let p1 = Point { x: 10, y: 20 };
    let p2 = Point { x: 100, y: 200 };
    println!("{p1:?} + {p2:?} = {:?}", p1 + p2);
}
```

Пункти обговорення:

- Ви можете реалізувати Add для &Point. У яких ситуаціях це може бути корисно?
  - Відповідь: Add::add споживає `self`. Якщо тип T, для якого ви перевантажуєте оператор, не є Copy, ви також повинні розглянути можливість перевантаження оператора &T. Це дозволяє уникнути непотрібного клонування на сайті виклику.
- Чому `Output` є асоційованим типом? Чи можна зробити це параметром типу методу?
  - Коротка відповідь: параметри типу функції контролюються тим, хто її викликає, а асоційовані типи (як `Output`) контролюються реалізатором трейту.
- Ви можете реалізувати Add для двох різних типів, напр. `impl Add<(i32, i32)> for Point` додасть кортеж до `Point`.

Трейт `Not` (оператор `!`) примітний тим, що він не "буліфікується", як той самий оператор у мовах сімейства C; натомість, для цілих типів він заперечує кожен біт числа, що арифметично еквівалентно відніманню від -1: `!5 == -6`.

17.3 `From` та `Into`

Типи реалізують `From` і [`Into`](https://doc.rust-lang.org/std /convert/trait.Into.html), щоб полегшити перетворення типів. На відміну від `as`, ці трейти відповідають безпомилковим перетворенням без втрат.

```
fn main() {
    let s = String::from("привіт");
    let addr = std::net::Ipv4Addr::from([127, 0, 0, 1]);
    let one = i16::from(true);
    let bigger = i32::from(123_i16);
    println!("{s}, {addr}, {one}, {bigger}");
}
```

`Into` реалізується автоматично, коли [`From`](https://doc.rust-lang.org/ std/convert/trait.From.html) реалізовано:

```
fn main() {
```

```
    let s: String = "привіт".into();
    let addr: std::net::Ipv4Addr = [127, 0, 0, 1].into();
    let one: i16 = true.into();
    let bigger: i32 = 123_i16.into();
    println!("{s}, {addr}, {one}, {bigger}");
}
```

- Ось чому прийнято реалізовувати лише `From`, оскільки ваш тип також отримає реалізацію `Into`.
- При оголошенні вхідного типу аргументу функції типу "все, що можна перетворити на `String`", правило протилежне, ви повинні використовувати `Into`. Ваша функція прийматиме типи, які реалізують `From`, і ті, які *лише* реалізують `Into`.

17.4 Приведення

Rust не має *неявних* перетворень типів, але підтримує явні приведення за допомогою `as`. Вони зазвичай відповідають семантиці C, де вони визначені.

```
fn main() {
    let value: i64 = 1000;
    println!("як u16: {}", value as u16);
    println!("як i16: {}", value as i16);
    println!("як u8: {}", value as u8);
}
```

Результати функції `as` *завжди* визначені у Rust і є стабільними на всіх платформах. Це може не збігатися з вашою інтуїцією щодо зміни знаку або приведення до меншого типу - зверніться до документації та коментарів для уточнення.

Приведення за допомогою `as` є відносно гнучким інструментом, який легко використовувати неправильно, і може бути джерелом малопомітних помилок, оскільки майбутні роботи супроводження змінюють типи, які використовуються, або діапазони значень у типах. Приведення до типу найкраще використовувати лише тоді, коли потрібно вказати безумовне усічення (наприклад, виділити молодші 32 біти `u64` за допомогою `as u32`, незалежно від того, що було у старших бітах).

Для безпомилкового приведення (наприклад, `u32` до `u64`) краще використовувати `From` або `Into` замість `as`, щоб переконатися, що приведення дійсно є безпомилковими. Для помилкових приведень доступні `TryFrom` і `TryInto`, якщо ви хочете обробити приведення, які відрізняються від тих, які не підходять.

Подумайте про перерву після цього слайда.

Оператор `as` подібний до статичного приведення у C++. Використання `as` у випадках, коли дані може бути втрачено, зазвичай не рекомендується або, принаймні, заслуговує на пояснювальний коментар.

Це типовий випадок приведення цілих чисел до `usize` для використання у якості індексу.

17.5 `Read` та `Write`

Використовуючи `Read` і `BufRead`, ви можете абстрагуватися над джерелами u8:

```
use std::io::{BufRead, BufReader, Read, Result};

fn count_lines<R: Read>(reader: R) -> usize {
    let buf_reader = BufReader::new(reader);
    buf_reader.lines().count()
}

fn main() -> Result<()> {
    let slice: &[u8] = b"foo\nbar\nbaz\n";
    println!("рядків у зрізі: {}", count_lines(slice));

    let file = std::fs::File::open(std::env::current_exe()?)?;
    println!("рядків у файлі: {}", count_lines(file));
    Ok(())
}
```

Подібним чином, `Write` дозволяє вам абстрагуватися над u8 прийомниками:

```
use std::io::{Result, Write};

fn log<W: Write>(writer: &mut W, msg: &str) -> Result<()> {
    writer.write_all(msg.as_bytes())?;
    writer.write_all("\n".as_bytes())
}

fn main() -> Result<()> {
    let mut buffer = Vec::new();
    log(&mut buffer, "Привіт")?;
    log(&mut buffer, "Світ")?;
    println!("Записано: {buffer:?}");
    Ok(())
}
```

17.6 Трейт `Default`

Трейт `Default` створює значення за замовчуванням для типу.

```
struct Derived {
    x: u32,
    y: String,
    z: Implemented,
}

struct Implemented(String);

impl Default for Implemented {
    fn default() -> Self {
        Self("John Smith".into())
```

```
    }
}

fn main() {
    let default_struct = Derived::default();
    println!("{default_struct:#?}");

    let almost_default_struct =
        Derived { y: "Y встановлено!".into(), ..Derived::default() };
    println!("{almost_default_struct:#?}");

    let nothing: Option<Derived> = None;
    println!("{:#?}", nothing.unwrap_or_default());
}
```

- Це може бути реалізовано безпосередньо або може бути отримано за допомогою `#[derive(Default)]`.
- Похідна реалізація створить значення, в якому всі поля мають значення за замовчуванням.
  - Це означає, що всі типи в структурі також мають реалізовувати `Default`.
- Стандартні типи Rust часто реалізують `Default` із прийнятними значеннями (наприклад, `0`, `""` тощо).
- Часткова ініціалізація структур чудово працює за замовчуванням.
- Стандартна бібліотека Rust усвідомлює, що типи можуть реалізовувати `Default` і надає зручні методи, які його використовують.
- Синтаксис `..` називається синтаксис оновлення структури.

17.7 Закриття

Замикання або лямбда-вирази мають типи, які не можна назвати. Однак вони реалізують спеціальні `Fn`, `FnMut` і `FnOnce` трейти:

```
fn apply_and_log(func: impl FnOnce(i32) -> i32, func_name: &str, input: i32) {
    println!("Викликаємо {func_name}({input}): {}", func(input))
}

fn main() {
    let n = 3;
    let add_3 = |x| x + n;
    apply_and_log(&add_3, "add_3", 10);
    apply_and_log(&add_3, "add_3", 20);

    let mut v = Vec::new();
    let mut accumulate = |x: i32| {
        v.push(x);
        v.iter().sum::<i32>()
    };
    apply_and_log(&mut accumulate, "accumulate", 4);
    apply_and_log(&mut accumulate, "accumulate", 5);

    let multiply_sum = |x| x * v.into_iter().sum::<i32>();
```

```
    apply_and_log(multiply_sum, "multiply_sum", 3);
}
```

Fn (наприклад, add_3) не споживає і не змінює захоплені значення. Вона може бути викликана, потребуючи лише спільного посилання на закриття, що означає, що замикання може бути виконано багаторазово і навіть одночасно.

FnMut (наприклад, accumulate) може змінити захоплені значення. Доступ до об'єкта замикання здійснюється за ексклюзивним посиланням, тому його можна викликати багаторазово, але не одночасно.

Якщо у вас є FnOnce (наприклад, multiply_sum), ви можете викликати її лише один раз. У такому випадку замикання поглинається разом з усіма значеннями, захопленими під час переміщення.

FnMut є підтипом FnOnce. Fn є підтипом FnMut і FnOnce. Тобто ви можете використовувати FnMut усюди, де викликається FnOnce, і ви можете використовувати Fn усюди, де викликається FnMut або FnOnce.

Коли ви визначаєте функцію, яка приймає закриття, вам слід використовувати FnOnce, якщо це можливо (тобто ви викликаєте її один раз), або FnMut в іншому випадку, і в останню чергу Fn. Це забезпечує найбільшу гнучкість для того, хто викликає функцію.

На противагу цьому, коли у вас є закриття, найбільш гнучким є Fn (яка може бути передана споживачеві будь-якої з 3 трейтів замикання), потім FnMut і, нарешті, FnOnce.

Компілятор також виводить Copy (наприклад, для add_3) і Clone (наприклад multiply_sum), залежно від того, що захоплює замикання. Покажчики функцій (посилання на елементи fn) реалізують Copy та Fn.

За замовчуванням замикання захоплюють кожну змінну із зовнішньої області видимості найменш вибагливою формою доступу (за спільним посиланням, якщо це можливо, потім за ексклюзивним посиланням, потім за переміщенням). Ключове слово move змушує захоплювати за значенням.

```
fn make_greeter(prefix: String) -> impl Fn(&str) {
    return move |name| println!("{} {}", prefix, name);
}

fn main() {
    let hi = make_greeter("Привіт".to_string());
    hi(" Грег");
}
```

17.8 Вправа: ROT13

У цьому прикладі ви будете реалізовувати класичний "ROT13" шифр. Скопіюйте цей код на ігровий майданчик і додайте біти, яких бракує. Перевертайте лише символи ASCII, щоб результат залишався дійсним UTF-8.

```
use std::io::Read;

struct RotDecoder<R: Read> {
    input: R,
    rot: u8,
```

```
}

// Реалізуйте трейт `Read` для `RotDecoder`.

fn main() {
    let mut rot =
        RotDecoder { input: "Gb trg gb gur bgure fvqr!".as_bytes(), rot: 13 };
    let mut result = String::new();
    rot.read_to_string(&mut result).unwrap();
    println!("{}", result);
}

mod test {
    use super::*;

    fn joke() {
        let mut rot =
            RotDecoder { input: "Gb trg gb gur bgure fvqr!".as_bytes(), rot: 13 };
        let mut result = String::new();
        rot.read_to_string(&mut result).unwrap();
        assert_eq!(&result, "To get to the other side!");
    }

    fn binary() {
        let input: Vec<u8> = (0..=255u8).collect();
        let mut rot = RotDecoder::<&[u8]> { input: input.as_ref(), rot: 13 };
        let mut buf = [0u8; 256];
        assert_eq!(rot.read(&mut buf).unwrap(), 256);
        for i in 0..=255 {
            if input[i] != buf[i] {
                assert!(input[i].is_ascii_alphabetic());
                assert!(buf[i].is_ascii_alphabetic());
            }
        }
    }
}
```

Що станеться, якщо зімкнути два екземпляри ‘RotDecoder'а разом, кожен з яких буде обертатися на 13 символів?

17.8.1 Рішення

```
use std::io::Read;

struct RotDecoder<R: Read> {
    input: R,
    rot: u8,
}

impl<R: Read> Read for RotDecoder<R> {
    fn read(&mut self, buf: &mut [u8]) -> std::io::Result<usize> {
```

```
        let size = self.input.read(buf)?;
        for b in &mut buf[..size] {
            if b.is_ascii_alphabetic() {
                let base = if b.is_ascii_uppercase() { 'A' } else { 'a' } as u8;
                *b = (*b - base + self.rot) % 26 + base;
            }
        }
        Ok(size)
    }
}

fn main() {
    let mut rot =
        RotDecoder { input: "Gb trg gb gur bgure fvqr!".as_bytes(), rot: 13 };
    let mut result = String::new();
    rot.read_to_string(&mut result).unwrap();
    println!("{}", result);
}

mod test {
    use super::*;

    fn joke() {
        let mut rot =
            RotDecoder { input: "Gb trg gb gur bgure fvqr!".as_bytes(), rot: 13 };
        let mut result = String::new();
        rot.read_to_string(&mut result).unwrap();
        assert_eq!(&result, "To get to the other side!");
    }

    fn binary() {
        let input: Vec<u8> = (0..=255u8).collect();
        let mut rot = RotDecoder::<&[u8]> { input: input.as_ref(), rot: 13 };
        let mut buf = [0u8; 256];
        assert_eq!(rot.read(&mut buf).unwrap(), 256);
        for i in 0..=255 {
            if input[i] != buf[i] {
                assert!(input[i].is_ascii_alphabetic());
                assert!(buf[i].is_ascii_alphabetic());
            }
        }
    }
}
```

Частина V

День 3: Ранок

Розділ 18

Ласкаво просимо до дня 3

Сьогодні ми розглянемо:

- Керування пам'яттю, час існування та перевірка запозичень: як Rust гарантує безпеку пам'яті.
- Розумні вказівники: стандартні бібліотечні типи вказівників.

Розклад

Враховуючи 10-хвилинні перерви, ця сесія має тривати близько 2 годин 20 хвилин. Вона містить:

Сегмент	Тривалість
Ласкаво просимо	3 хвилини
Управління пам'яттю	1 година
Розумні вказівники	55 хвилин

Розділ 19

Управління пам'яттю

Цей сегмент повинен зайняти близько 1 години. Він містить:

Слайд	Тривалість
Огляд пам'яті програми	5 хвилин
Підходи до управління пам'яттю	10 хвилин
Володіння	5 хвилин
Семантика переміщення	5 хвилин
Clone	2 хвилини
Типи які копіюються	5 хвилин
Drop	10 хвилин
Вправа: Тип будівельника	20 хвилин

19.1 Огляд пам'яті програми

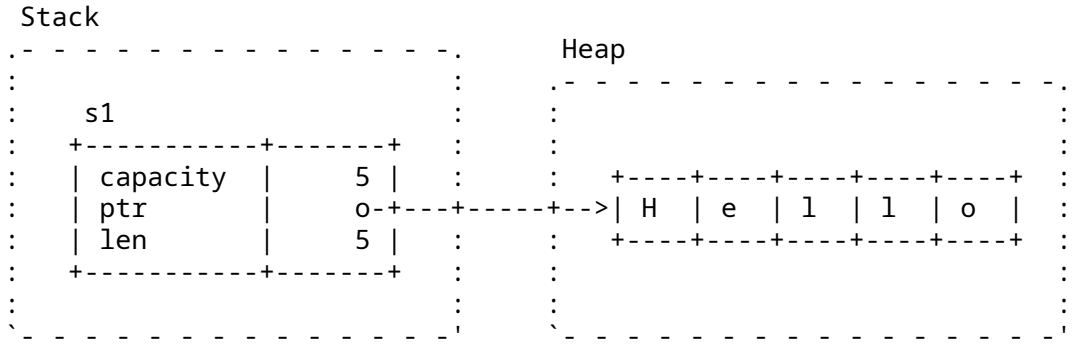
Програми виділяють пам'ять двома способами:

- Стек: безперервна область пам'яті для локальних змінних.
  - Значення мають фіксовані розміри, відомі під час компіляції.
  - Надзвичайно швидко: просто перемістіть вказівник стека.
  - Легко керувати: слідує за викликами функцій.
  - Чудова локальність пам'яті.
- Купа: Зберігання значень поза викликами функцій.
  - Значення мають динамічні розміри, визначені під час виконання.
  - Трохи повільніше, ніж стек: потрібн певний облік.
  - Немає гарантії локальності пам'яті.

Приклад

Створення `String` поміщає метадані фіксованого розміру в стек, а дані динамічного розміру, фактичний рядок, у купу:

```rust
fn main() {
    let s1 = String::from("Привіт");
}
```



- Нагадайте, що тип `String` підтримується `Vec`, тому має ємність і довжину та може зростати, якщо мутабельна, через перерозподіл у купі.
- Якщо студенти запитають про це, ви можете нагадати, що основна пам'ять розподіляється за допомогою System Allocator і користувальницькі розподільники можуть бути реалізовано за допомогою Allocator API

Більше інформації для вивчення

Ми можемо перевірити розташування пам'яті за допомогою `unsafe` Rust. Однак ви повинні зазначити, що це по праву небезпечно!

```rust
fn main() {
    let mut s1 = String::from("Привіт");
    s1.push(' ');
    s1.push_str("світ");
    // НЕ РОБІТЬ ЦЬОГО ВДОМА! Тільки в навчальних цілях.
    // String не надає жодних гарантій щодо своєї розмітки, тому це може призвести до
    // невизначеної поведінки.
    unsafe {
        let (capacity, ptr, len): (usize, usize, usize) = std::mem::transmute(s1);
        println!("capacity = {capacity}, ptr = {ptr:#x}, len = {len}");
    }
}
```

19.2 Підходи до управління пам'яттю

Традиційно мови поділяються на дві великі категорії:

- Повний контроль через ручне управління пам'яттю: C, C++, Pascal, ...
  - Програміст вирішує, коли виділяти або звільняти пам'ять купи.
  - Програміст повинен визначити, чи вказівник все ще вказує на дійсну пам'ять.
  - Дослідження показують, що програмісти роблять помилки.
- Повна безпека завдяки автоматичному управлінню пам'яттю під час виконання: Java, Python, Go, Haskell, ...

  - Система виконання гарантує, що пам'ять не звільняється доти, доки до неї не можна буде звертатися.
  - Зазвичай реалізується за допомогою підрахунку посилань або збору сміття.

Rust пропонує нову суміш:

> Повний контроль *та* безпека завдяки забезпеченню правильного керування пам'яттю під час компіляції.

Це робиться за допомогою чіткої концепції володіння.

Цей слайд має на меті допомогти студентам, які вивчають інші мови, помістити Rust у контекст.

- C має керувати купою вручну за допомогою `malloc` та `free`. Типові помилки включають забування виклику `free`, виклик `free` декілька разів для одного і того ж вказівника або розіменування вказівника після того, як пам'ять, на яку він вказує, було звільнено.
- C++ has tools like smart pointers (`unique_ptr`, `shared_ptr`) that take advantage of language guarantees about calling destructors to ensure memory is freed when a function returns. It is still quite easy to mis-use these tools and create similar bugs to C.
- У C++ є такі інструменти, як розумні вказівники (`unique_ptr`, `shared_ptr`), які використовують гарантії мови щодо виклику деструкторів для забезпечення звільнення пам'яті при завершенні функції. Але все одно досить легко неправильно використовувати ці інструменти і створювати помилки, подібні до помилок у мові C.

Модель володіння та запозичення Rust у багатьох випадках дозволяє отримати продуктивність C, з операціями alloc та free саме там, де вони потрібні - з нульовими витратами. Він також надає інструменти, подібні до розумних вказівників C++. За необхідності, доступні інші опції, такі як підрахунок посилань, і навіть є сторонні крейти для підтримки збирання сміття під час виконання (не розглядаються у цьому класі).

19.3 Володіння

Усі прив’язки змінних мають *область*, де вони дійсні, і використання змінної поза її областю є помилкою:

```
struct Point(i32, i32);

fn main() {
    {
        let p = Point(3, 4);
        println!("x: {}", p.0);
    }
    println!("y: {}", p.1);
}
```

Ми говоримо, що змінна *володіє* значенням. Кожне значення у Rust завжди має лише одного власника.

В кінці області видимості змінна *знищується* і дані звільняються. Тут може бути запущено деструктор, щоб звільнити ресурси.

Студенти, знайомі з реалізаціями збирачів сміття, знають, що збирач сміття починає роботу з набору ”коренів”, щоб виявити всю доступну пам'ять. Принцип ”єдиного власника” у Rust має схожу ідею.

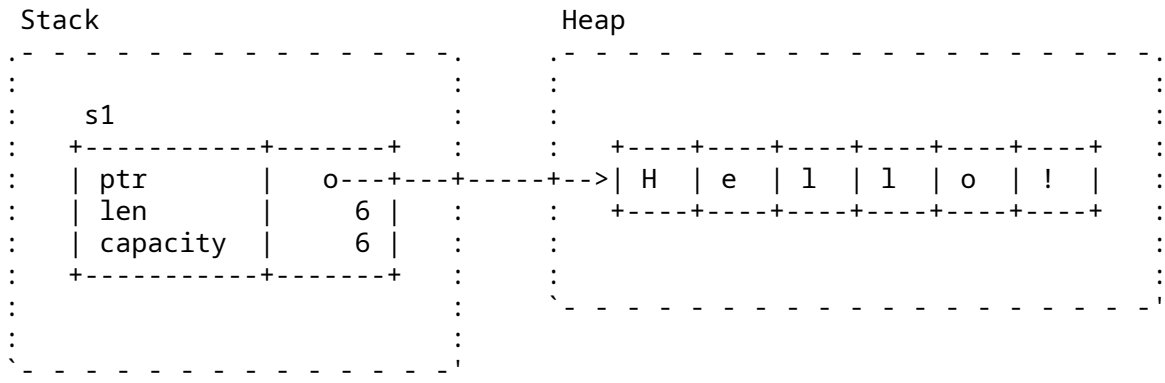
19.4 Семантика переміщення

Присвоєння переміщує *володіння* між змінними:

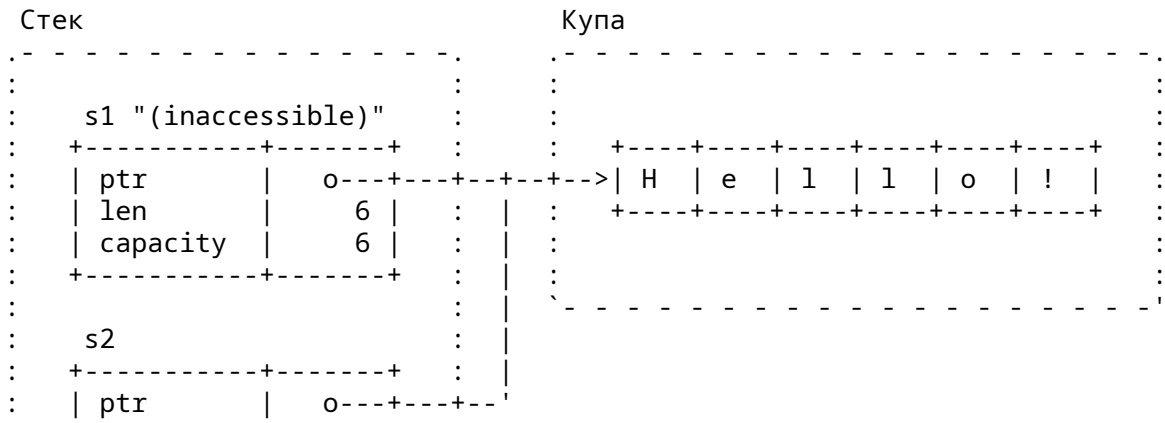
```
fn main() {
    let s1: String = String::from("Привіт!");
    let s2: String = s1;
    println!("s2: {s2}");
    // println!("s1: {s1}");
}
```

- Присвоєння `s1` до `s2` переміщує володіння.
- Коли `s1` виходить за межі області видимості, нічого не відбувається: вона нічим не володіє.
- Коли `s2` виходить за межі, дані рядка звільняються.

Перед переміщенням до `s2`:



Після переміщення до `s2`:



```
:   | len       |     6 |   :
:   | capacity  |     6 |   :
:   +-----------+-------+   :
:                           :
`- - - - - - - - - - - - - -'
```

Коли ви передаєте значення функції, це значення присвоюється параметру функції. Це переміщує володіння:

```
fn say_hello(name: String) {
    println!("Привіт {name}")
}

fn main() {
    let name = String::from("Alice");
    say_hello(name);
    // say_hello(name);
}
```

- Зауважте, що це протилежність поведінки за замовчуванням у C++, яка копіює за значенням, якщо ви не використовуєте `std::move` (і конструктор переміщення визначено!).
- Переміщується лише володіння. Чи генерується машинний код для маніпулювання самими даними - це питання оптимізації, і такі копії агресивно оптимізуються.
- Прості значення (наприклад, цілі числа) можна позначити `Copy` (див. наступні слайди).
- У Rust клони є явними (за допомогою `clone`).

У прикладі `say_hello`:

- З першим викликом `say_hello` `main` втрачає володіння `name`. Після цього `name` більше не можна використовувати в `main`.
- Пам'ять купи, виділена для `name`, буде звільнено в кінці функції `say_hello`.
- `main` може зберігти володіння, якщо передасть `name` як посилання (`&name`) і якщо `say_hello` приймає посилання як параметр.
- Крім того, `main` може передати клон `name` під час першого виклику (`name.clone()`).
- Rust ускладнює ненавмисне створення копій, на відмінну від C++, роблячи семантику переміщення за замовчуванням і змушуючи програмістів робити клони явними.

Більше інформації для вивчення

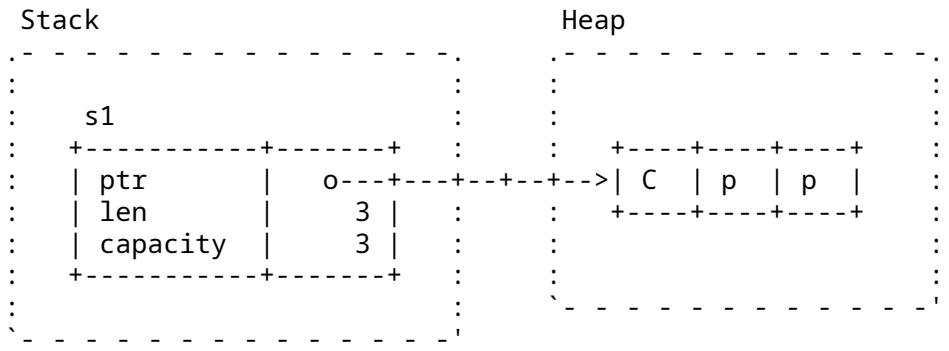
Захисні копії в сучасному C++

Сучасний C++ вирішує це інакше:

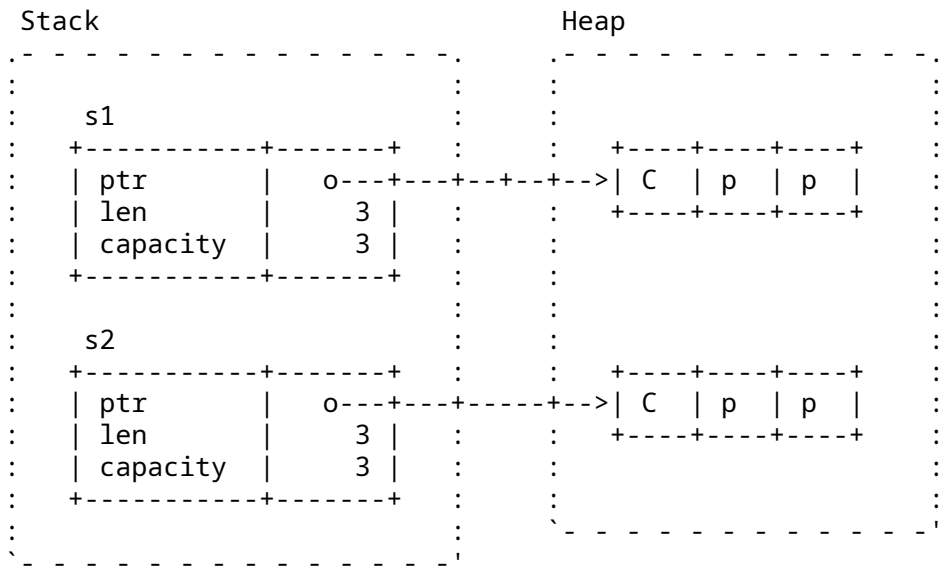
```
std::string s1 = "Cpp";
std::string s2 = s1;  // Дублювання даних в s1.
```

- Дані купи з `s1` дублюються, а `s2` отримує власну незалежну копію.
- Коли `s1` і `s2` виходять за межі видимості, кожен з них звільняє власну пам'ять.

Перед копіюванням:



Після копіювання:



Ключові моменти:

- C++ зробив дещо інший вибір, ніж Rust. Оскільки = копіює дані, дані рядка потрібно клонувати. Інакше ми отримаємо подвійне звільнення, коли будь-який рядок виходить за межі видимості.
- C++ також має `std::move`, який використовується щоб вказати коли значення можна перемістити. Якби приклад був `s2 = std::move(s1)`, розподілу купи не відбулося б. Після переміщення `s1` буде в діючому, але не визначеному стані. На відміну від Rust, програмісту дозволено використовувати `s1`.
- На відміну від Rust, = у C++ може виконувати довільний код, який визначається типом, який копіюється або переміщується.

19.5 Clone

Іноді вам *необхідно* зробити копію значення. Для цього призначений крейт `Clone`.

```
fn say_hello(name: String) {
    println!("Привіт {name}")
}

fn main() {
    let name = String::from("Alice");
    say_hello(name.clone());
    say_hello(name);
}
```

- Ідея `Clone` полягає у тому, щоб полегшити виявлення місць, де відбувається виділення пам'яті у купі. Шукайте `.clone()` та деякі інші, такі як `vec!` або `Box::new`.
- Зазвичай, ви "клонуєте свій вихід" з проблем з перевіркою позик, а потім повертаєтесь пізніше, щоб спробувати оптимізувати ці клони.
- `clone` зазвичай виконує глибоку копію значення, тобто якщо ви, наприклад, клонуєте масив, то всі елементи масиву також будуть клоновані.
- Поведінка функції `clone` визначається користувачем, тому вона може виконувати власну логіку клонування, якщо це необхідно.

19.6 Типи які копіюються

Хоча семантика переміщення використовується за замовчуванням, певні типи копіюються за замовчуванням:

```
fn main() {
    let x = 42;
    let y = x;
    println!("x: {x}"); // would not be accessible if not Copy
    println!("y: {y}");
}
```

Ці типи реалізують трейт `Copy`.

Ви можете вибрати власні типи для використання семантики копіювання:

```
struct Point(i32, i32);

fn main() {
    let p1 = Point(3, 4);
    let p2 = p1;
    println!("p1: {p1:?}");
    println!("p2: {p2:?}");
}
```

- Після присвоєння обидва `p1` і `p2` володіють власними даними.
- Ми також можемо використовувати `p1.clone()` для явного копіювання даних.

Копіювання та клонування – це не одне й те саме:

- Копіювання стосується побітових копій областей пам'яті та не працює з довільними об'єктами.

- Копіювання не допускає створювати власну логіку (на відміну від конструкторів копіювання в C++).
- Клонування — це більш загальна операція, яка також допускає нестандартну поведінку шляхом реалізації трейта `Clone`.
- Копіювання не працює з типами, які реалізують трейт `Drop`.

У наведеному вище прикладі спробуйте наступне:

- Додайте поле `String` до `struct Point`. Це не скомпілюється, оскільки `String` не є типом `Copy`.
- Видаліть `Copy` з атрибута `derive`. Помилка компілятора тепер у `println!` для `p1`.
- Покажіть, що це працює, якщо замість цього клонувати `p1`.

Більше інформації для вивчення

- Спільні посилання є `Copy`/`Clone`, змінні посилання - ні. Це пов'язано з тим, що Rust вимагає, щоб змінювані посилання були ексклюзивними, тому, хоча створення копії спільного посилання є допустимим, створення копії змінюваного посилання порушуватиме правила запозичення Rust.

19.7 Трейт `Drop`

Значення, які реалізують `Drop`, можуть вказувати код, який запускатиметься, коли вони виходять за межі області видимості:

```
struct Droppable {
    name: &'static str,
}

impl Drop for Droppable {
    fn drop(&mut self) {
        println!("Відкидаємо {}", self.name);
    }
}

fn main() {
    let a = Droppable { name: "a" };
    {
        let b = Droppable { name: "b" };
        {
            let c = Droppable { name: "c" };
            let d = Droppable { name: "d" };
            println!("Виходимо з блоку B");
        }
        println!("Виходимо з блоку A");
    }
    drop(a);
    println!("Виходимо з main");
}
```

- Зверніть увагу, що `std::mem::drop` не те саме, що `std::ops::Drop::drop`.

- Значення автоматично відкидаються, коли вони виходять за межі області видимості.
- Коли значення відкидається, якщо воно реалізує `std::ops::Drop`, то буде викликано його реалізацію `Drop::drop`.
- Всі його поля також будуть видалені, незалежно від того, чи реалізовано в ньому `Drop` чи ні.
- `std::mem::drop` - це просто порожня функція, яка приймає будь-яке значення. Важливо те, що вона отримує володіння значенням, тому в кінці своєї області видимості вона його відкидає. Це робить її зручним способом явного відкидання значень раніше, ніж вони вийдуть за межі області видимості.
  - Це може бути корисно для об'єктів, які виконують деяку роботу на `drop`: зняття блокування, закриття файлів тощо.

Пункти обговорення:

- Чому `Drop::drop` не приймає `self`?
  - Коротка відповідь: якби це було так, `std::mem::drop` викликався б у кінці блоку, що призвело б до ще одного виклику `Drop::drop` і переповнення стеку!
- Спробуйте замінити `drop(a)` на `a.drop()`.

19.8 Вправа: Тип будівельника

У цьому прикладі ми реалізуємо складний тип даних, який володіє всіма своїми даними. Ми використаємо патерн "конструктор" для підтримки побудови нового значення по частинах за допомогою зручних функцій.

Заповніть пропущені частини.

```
enum Language {
    Rust,
    Java,
    Perl,
}

struct Dependency {
    name: String,
    version_expression: String,
}

/// Представлення програмного пакету.
struct Package {
    name: String,
    version: String,
    authors: Vec<String>,
    dependencies: Vec<Dependency>,
    language: Option<Language>,
}

impl Package {
    /// Повертає представлення цього пакунка у вигляді залежності для використання у
    /// збірці інших пакетів.
    fn as_dependency(&self) -> Dependency {
```

```
        todo!("1")
    }
}

/// Конструктор для Package. Використовуйте `build()` для створення самого `Package`.
struct PackageBuilder(Package);

impl PackageBuilder {
    fn new(name: impl Into<String>) -> Self {
        todo!("2")
    }

    /// Задає версію пакета.
    fn version(mut self, version: impl Into<String>) -> Self {
        self.0.version = version.into();
        self
    }

    /// Задає автора пакета.
    fn authors(mut self, authors: Vec<String>) -> Self {
        todo!("3")
    }

    /// Додає додаткову залежність.
    fn dependency(mut self, dependency: Dependency) -> Self {
        todo!("4")
    }

    /// Задає мову. Якщо не вказано, за замовчуванням буде встановлено значення None.
    fn language(mut self, language: Language) -> Self {
        todo!("5")
    }

    fn build(self) -> Package {
        self.0
    }
}

fn main() {
    let base64 = PackageBuilder::new("base64").version("0.13").build();
    println!("base64: {base64:?}");
    let log =
        PackageBuilder::new("log").version("0.4").language(Language::Rust).build();
    println!("log: {log:?}");
    let serde = PackageBuilder::new("serde")
        .authors(vec!["djmitche".into()])
        .version(String::from("4.0"))
        .dependency(base64.as_dependency())
        .dependency(log.as_dependency())
        .build();
    println!("serde: {serde:?}");
```

```
}
```

19.8.1 Рішення

```
enum Language {
    Rust,
    Java,
    Perl,
}

struct Dependency {
    name: String,
    version_expression: String,
}

/// Представлення програмного пакету.
struct Package {
    name: String,
    version: String,
    authors: Vec<String>,
    dependencies: Vec<Dependency>,
    language: Option<Language>,
}

impl Package {
    /// Повертає представлення цього пакунка у вигляді залежності для використання у
    /// збірці інших пакетів.
    fn as_dependency(&self) -> Dependency {
        Dependency {
            name: self.name.clone(),
            version_expression: self.version.clone(),
        }
    }
}

/// Конструктор для Package. Використовуйте `build()` для створення самого `Package`.
struct PackageBuilder(Package);

impl PackageBuilder {
    fn new(name: impl Into<String>) -> Self {
        Self(Package {
            name: name.into(),
            version: "0.1".into(),
            authors: vec![],
            dependencies: vec![],
            language: None,
        })
    }

    /// Задає версію пакета.
    fn version(mut self, version: impl Into<String>) -> Self {
```

```
        self.0.version = version.into();
        self
    }

    /// Задає автора пакета.
    fn authors(mut self, authors: Vec<String>) -> Self {
        self.0.authors = authors;
        self
    }

    /// Додає додаткову залежність.
    fn dependency(mut self, dependency: Dependency) -> Self {
        self.0.dependencies.push(dependency);
        self
    }

    /// Задає мову. Якщо не вказано, за замовчуванням буде встановлено значення None.
    fn language(mut self, language: Language) -> Self {
        self.0.language = Some(language);
        self
    }

    fn build(self) -> Package {
        self.0
    }
}

fn main() {
    let base64 = PackageBuilder::new("base64").version("0.13").build();
    println!("base64: {base64:?}");
    let log =
        PackageBuilder::new("log").version("0.4").language(Language::Rust).build();
    println!("log: {log:?}");
    let serde = PackageBuilder::new("serde")
        .authors(vec!["djmitche".into()])
        .version(String::from("4.0"))
        .dependency(base64.as_dependency())
        .dependency(log.as_dependency())
        .build();
    println!("serde: {serde:?}");
}
```

Розділ 20

Розумні вказівники

Цей сегмент повинен зайняти близько 55 хвилин. Він містить:

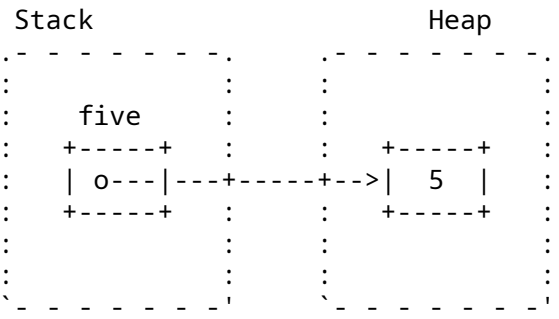
Слайд	Тривалість
Box	

|10 хвилин| |Rc|5 хвилин| |Принадлежні об'єкти трейтів|10 хвилин| |Вправа: Бінарне дерево|30 хвилин|

20.1 `Box<T>`

`Box` — це вказівник на дані в купі:

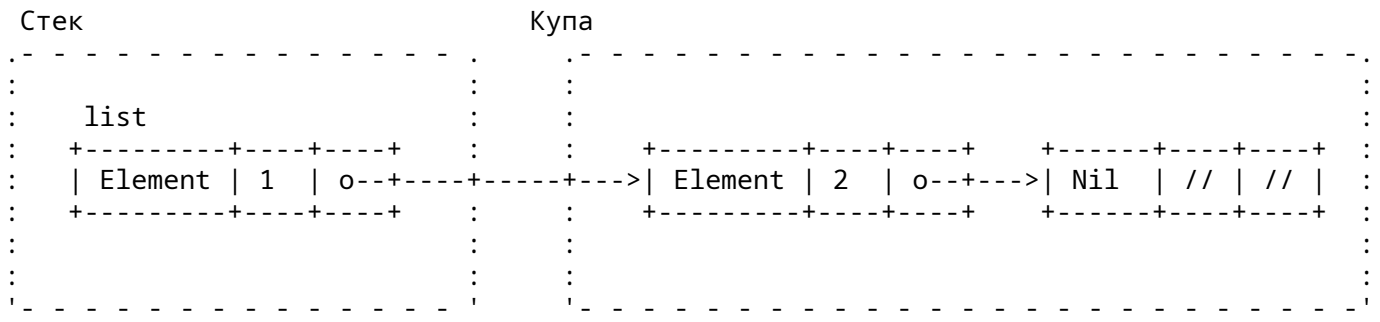
```
fn main() {
    let five = Box::new(5);
    println!("five: {}", *five);
}
```



`Box<T>` реалізує `Deref<Target = T>`, що означає, що ви можете викликати методи з `T` безпосередньо на `Box<T>`.

Рекурсивні типи даних або типи даних з динамічним розміром не можуть зберігатися вбудованими без перенаправлення вказівника. `Box` здійснює цю опосередкованість:

```rust
enum List<T> {
    /// Непорожній список: перший елемент та решта списку.
    Element(T, Box<List<T>>),
    /// Порожній список.
    Nil,
}

fn main() {
    let list: List<i32> =
        List::Element(1, Box::new(List::Element(2, Box::new(List::Nil))));
    println!("{list:?}");
}
```



- Box схожий на `std::unique_ptr` у C++, за винятком того, що він гарантовано не буде null.
- Box може бути корисним, коли ви:
  - маєте тип, розмір якого не може бути відомий під час компіляції, але компілятор Rust хоче знати точний розмір.
  - хочете передати володіння на великий обсяг даних. Щоб уникнути копіювання великих обсягів даних у стеку, натомість зберігайте дані в купі в Box, щоб переміщувався лише вказівник.
- Якщо би Box не використовувався, і ми намагалися вставити `List` безпосередньо в `List`, компілятор не зміг би обчислити фіксований розмір структури в пам'яті (`List` мав би нескінченний розмір).
- Box вирішує цю проблему, оскільки має той самий розмір, що й звичайний вказівник, і лише вказує на наступний елемент `List` у купі.
- Видаліть Box у визначенні списку та відобразіть помилку компілятора. Ми отримаємо повідомлення "recursive without indirection", тому що для рекурсії даних ми повинні використовувати посередництво, Box або якесь посилання, замість того, щоб зберігати значення безпосередньо.
- Хоча Box виглядає як `std::unique_ptr` у C++, він не може бути порожнім/нульовим. Це робить Box одним з типів, які дозволяють компілятору оптимізувати зберігання деяких переліків ("нішова оптимізація").

20.2 Rc

Rc — це спільний вказівник із підрахунком посилань. Використовуйте це, коли вам

потрібно звернутися до тих самих даних з кількох місць:

```
use std::rc::Rc;

fn main() {
    let a = Rc::new(10);
    let b = Rc::clone(&a);

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

- Дивіться `Arc` та `Mutex`, якщо ви працюєте у багатопотоковому контексті.
- Ви можете *понизити* спільний вказівник на `Weak` вказівник, щоб створити цикли, які буде відкинуті.

- Рахунок `Rc` гарантує, що значення, яке міститься в ньому, буде дійсним до тих пір, поки існують посилання.
- `Rc` у Rust схожий на `std::shared_ptr` у C++.
- `Rc::clone` дешевий: він створює вказівник на той самий розділ пам'яті і збільшує кількість посилань. Він не робить глибокого клонування, і, як правило, його можна ігнорувати, шукаючи проблеми з продуктивністю в коді.
- `make_mut` насправді клонує внутрішнє значення, якщо необхідно ("clone-on-write") і повертає мутабельне посилання.
- Використовуйте `Rc::strong_count`, щоб перевірити кількість посилань.
- `Rc::downgrade` дає вам *слабкий об'єкт з підрахунком посилань* для створення циклів, які будуть відкинуті належним чином (ймовірно, у поєднанні з `RefCell`).

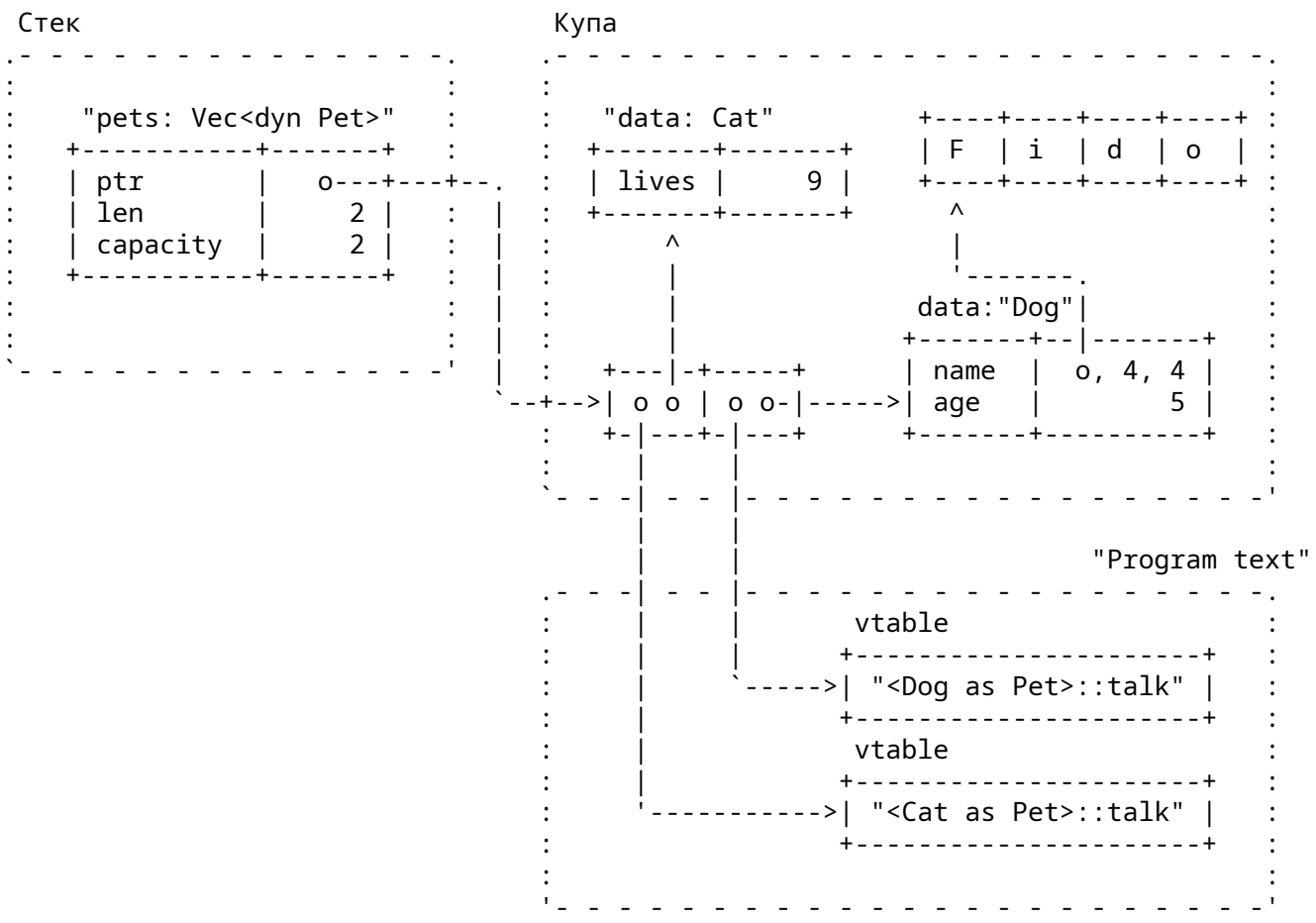
20.3 Принадлежні об'єкти трейтів

Раніше ми бачили, як об'єкти трейтів можна використовувати з посиланнями, наприклад, `&dyn Pet`. Однак, ми також можемо використовувати об'єкти трейтів з розумними вказівниками, такими як `Box`, щоб створити власний об'єкт трейту: `Box<dyn Pet>`.

```
struct Dog {
    name: String,
    age: i8,
}
struct Cat {
    lives: i8,
}

trait Pet {
    fn talk(&self) -> String;
}

impl Pet for Dog {
    fn talk(&self) -> String {
        format!("Гав, мене звуть {}!", self.name)
    }
}
```

```rust
impl Pet for Cat {
    fn talk(&self) -> String {
        String::from("Мяу!")
    }
}

fn main() {
    let pets: Vec<Box<dyn Pet>> = vec![
        Box::new(Cat { lives: 9 }),
        Box::new(Dog { name: String::from("Фідо"), age: 5 }),
    ];
    for pet in pets {
        println!("Привіт, ви хто? {}", pet.talk());
    }
}
```

Розташування пам'яті після виділення `pets`:



- Типи, що реалізують певний трейт, можуть бути різних розмірів. Це унеможливлює створення таких типів, як `Vec<dyn Pet>` у наведеному вище прикладі.

- `dyn Pet` — це спосіб повідомити компілятору про тип динамічного розміру, який реалізує `Pet`.
- У прикладі `pets` розміщується у стеку, а векторні дані - у купі. Два векторні елементи є *жирними вказівниками*:
  - Жирний вказівник - це вказівник подвійної ширини. Він складається з двох компонентів: вказівника на власне об'єкт і вказівника на таблицю віртуальних методів (vtable) для реалізації `Pet` цього конкретного об'єкта.
  - Дані для `Dog` на ім'я Фідо - це поля `name` та `age`. Для `Cat` є поле `lives`.
- Порівняйте ці результати в наведеному вище прикладі:

```
println!("{} {}", std::mem::size_of::<Dog>(), std::mem::size_of::<Cat>());
println!("{} {}", std::mem::size_of::<&Dog>(), std::mem::size_of::<&Cat>());
println!("{}", std::mem::size_of::<&dyn Pet>());
println!("{}", std::mem::size_of::<Box<dyn Pet>>());
```

20.4 Вправа: Бінарне дерево

Бінарне дерево - це структура даних деревовидного типу, де кожен вузол має двох нащадків (лівого і правого). Ми створимо дерево, у якому кожна вершина зберігає значення. Для заданого вузла N всі вузли лівого піддерева N містять менші значення, а всі вузли правого піддерева N будуть містити більші значення.

Реалізуйте наступні типи так, щоб задані тести пройшли.

Додаткове завдання: реалізувати ітератор над бінарним деревом, який повертає значення відповідно до порядку.

```
/// Вузол у бінарному дереві.
struct Node<T: Ord> {
    value: T,
    left: Subtree<T>,
    right: Subtree<T>,
}

/// Можливо-порожнє піддерево.
struct Subtree<T: Ord>(Option<Box<Node<T>>>);

/// Контейнер, що зберігає набір значень, використовуючи бінарне дерево.
///
/// Якщо одне й те саме значення додається кілька разів, воно зберігається лише один раз
pub struct BinaryTree<T: Ord> {
    root: Subtree<T>,
}

impl<T: Ord> BinaryTree<T> {
    fn new() -> Self {
        Self { root: Subtree::new() }
    }

    fn insert(&mut self, value: T) {
        self.root.insert(value);
    }
```

```rust
    fn has(&self, value: &T) -> bool {
        self.root.has(value)
    }

    fn len(&self) -> usize {
        self.root.len()
    }
}

// Реалізуйте `new`, `insert`, `len` та `has` для `Subtree`.

mod tests {
    use super::*;

    fn len() {
        let mut tree = BinaryTree::new();
        assert_eq!(tree.len(), 0);
        tree.insert(2);
        assert_eq!(tree.len(), 1);
        tree.insert(1);
        assert_eq!(tree.len(), 2);
        tree.insert(2); // не унікальний елемент
        assert_eq!(tree.len(), 2);
    }

    fn has() {
        let mut tree = BinaryTree::new();
        fn check_has(tree: &BinaryTree<i32>, exp: &[bool]) {
            let got: Vec<bool> =
                (0..exp.len()).map(|i| tree.has(&(i as i32))).collect();
            assert_eq!(&got, exp);
        }

        check_has(&tree, &[false, false, false, false, false]);
        tree.insert(0);
        check_has(&tree, &[true, false, false, false, false]);
        tree.insert(4);
        check_has(&tree, &[true, false, false, false, true]);
        tree.insert(4);
        check_has(&tree, &[true, false, false, false, true]);
        tree.insert(3);
        check_has(&tree, &[true, false, false, true, true]);
    }

    fn unbalanced() {
        let mut tree = BinaryTree::new();
        for i in 0..100 {
            tree.insert(i);
        }
        assert_eq!(tree.len(), 100);
```

```
        assert!(tree.has(&50));
    }
}
```

20.4.1 Рішення

```
use std::cmp::Ordering;

/// Вузол у бінарному дереві.
struct Node<T: Ord> {
    value: T,
    left: Subtree<T>,
    right: Subtree<T>,
}

/// Можливо-порожнє піддерево.
struct Subtree<T: Ord>(Option<Box<Node<T>>>);

/// Контейнер, що зберігає набір значень, використовуючи бінарне дерево.
///
/// Якщо одне й те саме значення додається кілька разів, воно зберігається лише один ра
pub struct BinaryTree<T: Ord> {
    root: Subtree<T>,
}

impl<T: Ord> BinaryTree<T> {
    fn new() -> Self {
        Self { root: Subtree::new() }
    }

    fn insert(&mut self, value: T) {
        self.root.insert(value);
    }

    fn has(&self, value: &T) -> bool {
        self.root.has(value)
    }

    fn len(&self) -> usize {
        self.root.len()
    }
}

impl<T: Ord> Subtree<T> {
    fn new() -> Self {
        Self(None)
    }

    fn insert(&mut self, value: T) {
        match &mut self.0 {
            None => self.0 = Some(Box::new(Node::new(value))),
```

```
            Some(n) => match value.cmp(&n.value) {
                Ordering::Less => n.left.insert(value),
                Ordering::Equal => {}
                Ordering::Greater => n.right.insert(value),
            },
        }
    }

    fn has(&self, value: &T) -> bool {
        match &self.0 {
            None => false,
            Some(n) => match value.cmp(&n.value) {
                Ordering::Less => n.left.has(value),
                Ordering::Equal => true,
                Ordering::Greater => n.right.has(value),
            },
        }
    }

    fn len(&self) -> usize {
        match &self.0 {
            None => 0,
            Some(n) => 1 + n.left.len() + n.right.len(),
        }
    }
}

impl<T: Ord> Node<T> {
    fn new(value: T) -> Self {
        Self { value, left: Subtree::new(), right: Subtree::new() }
    }
}

fn main() {
    let mut tree = BinaryTree::new();
    tree.insert("foo");
    assert_eq!(tree.len(), 1);
    tree.insert("bar");
    assert!(tree.has(&"foo"));
}

mod tests {
    use super::*;

    fn len() {
        let mut tree = BinaryTree::new();
        assert_eq!(tree.len(), 0);
        tree.insert(2);
        assert_eq!(tree.len(), 1);
        tree.insert(1);
        assert_eq!(tree.len(), 2);
```

```
        tree.insert(2); // не унікальний елемент
        assert_eq!(tree.len(), 2);
    }

    fn has() {
        let mut tree = BinaryTree::new();
        fn check_has(tree: &BinaryTree<i32>, exp: &[bool]) {
            let got: Vec<bool> =
                (0..exp.len()).map(|i| tree.has(&(i as i32))).collect();
            assert_eq!(&got, exp);
        }

        check_has(&tree, &[false, false, false, false, false]);
        tree.insert(0);
        check_has(&tree, &[true, false, false, false, false]);
        tree.insert(4);
        check_has(&tree, &[true, false, false, false, true]);
        tree.insert(4);
        check_has(&tree, &[true, false, false, false, true]);
        tree.insert(3);
        check_has(&tree, &[true, false, false, true, true]);
    }

    fn unbalanced() {
        let mut tree = BinaryTree::new();
        for i in 0..100 {
            tree.insert(i);
        }
        assert_eq!(tree.len(), 100);
        assert!(tree.has(&50));
    }
}
```

Частина VI

День 3: Полудень

Розділ 21

Ласкаво просимо назад

Враховуючи 10-хвилинні перерви, цея сесія має тривати близько 1 годин 55 хвилин. Вона містить:

Сегмент	Тривалість
Запозичення	55 хвилин
Тривалість життя	50 хвилин

Розділ 22

Запозичення

Цей сегмент повинен зайняти близько 55 хвилин. Він містить:

Слайд	Тривалість
Запозичення значення	10 хвилин
Перевірка запозичення	10 хвилин
Помилки запозичення	3 хвилини
Внутрішня мутабельність	10 хвилин
Вправа: Статистика здоров'я	20 хвилин

22.1 Запозичення значення

Як ми бачили раніше, замість того, щоб передавати право власності при виклику функції, ви можете дозволити функції *позичити* значення:

```
struct Point(i32, i32);

fn add(p1: &Point, p2: &Point) -> Point {
    Point(p1.0 + p2.0, p1.1 + p2.1)
}

fn main() {
    let p1 = Point(3, 4);
    let p2 = Point(10, 20);
    let p3 = add(&p1, &p2);
    println!("{p1:?} + {p2:?} = {p3:?}");
}
```

- Функція add *позичає* дві точки та повертає нову точку.
- Викликач зберігає право власності на вхідні дані.

Цей слайд є оглядом матеріалу про посилання з першого дня, дещо розширеного за рахунок включення аргументів функцій та значень, що повертаються.

Більше інформації для вивчення

Примітки щодо повернення стеку та вбудовування:

- Продемонструйте, що повернення з add є дешевим, оскільки компілятор може виключити операцію копіювання, вбудовуючи виклик додавання в main. Змініть наведений вище код так, щоб він виводив адреси стеку, і запустіть його на Playground або перегляньте збірку в Godbolt. На рівні оптимізації ”DEBUG” адреси мають змінитися, але вони залишаються незмінними під час переходу до налаштування ”RELEASE”:

```
struct Point(i32, i32);

fn add(p1: &Point, p2: &Point) -> Point {
    let p = Point(p1.0 + p2.0, p1.1 + p2.1);
    println!("&p.0: {:p}", &p.0);
    p
}

pub fn main() {
    let p1 = Point(3, 4);
    let p2 = Point(10, 20);
    let p3 = add(&p1, &p2);
    println!("&p3.0: {:p}", &p3.0);
    println!("{p1:?} + {p2:?} = {p3:?}");
}
```

- Компілятор Rust може виконувати автоматичне вбудовування, яке можна вимкнути на рівні функції за допомогою `#[inline(never)]`.
- Якщо вимкнено, друкована адреса зміниться на всіх рівнях оптимізації. Дивлячись на Godbolt або Playground, можна побачити, що в цьому випадку повернення значення залежить від ABI, наприклад, на amd64 два i32, що складають точку, будуть повернуті у 2x регістрах (eax і edx).

22.2 Перевірка запозичення

Перевірка запозичень у Rust'і накладає обмеження на способи, якими ви можете запозичувати значення. Для певного значення, у будь-який час:

- Ви можете мати одне або декілька спільних посилань на значення, *або*
- Ви можете мати лише одне ексклюзивне посилання на значення.

```
fn main() {
    let mut a: i32 = 10;
    let b: &i32 = &a;

    {
        let c: &mut i32 = &mut a;
        *c = 20;
    }

    println!("a: {a}");
```

```
    println!("b: {b}");
}
```

- Зверніть увагу, що вимога полягає в тому, що конфліктуючі посилання не повинні *існувати* в тій самій момент часу. Не має значення, де посилання буде розіменовано.
- Наведений вище код не компілюється, оскільки `a` запозичено як мутабельну змінну (через `c`) і як немутабельну (через `b`) одночасно.
- Перемістіть інструкцію `println!` для `b` перед областю видимості, яка вводить `c`, щоб забезпечити компіляцію коду.
- Після цієї зміни компілятор розуміє, що `b` використовується тільки перед новим мутабельним запозиченням `a` через `c`. Це функція перевірки запозичень під назвою ”нелексичні терміни життя”.
- Обмеження ексклюзивного посилання є досить сильним. Rust використовує його для запобігання гонці даних. Rust також *покладається* на це обмеження для оптимізації коду. Наприклад, значення за спільним посиланням можна безпечно кешувати у регістрі на весь час існування цього посилання.
- Перевірку запозичень розроблено з урахуванням багатьох поширених шаблонів, таких як одночасне отримання ексклюзивних посилань на різні поля у структурі. Але бувають ситуації, коли вона не зовсім ”розуміє що відбувається”, і це часто призводить до ”боротьби з перевіряльником запозичень”.

22.3 Помилки запозичення

Як конкретний приклад того, як ці правила запозичення запобігають помилкам пам'яті, розглянемо випадок модифікації колекції, коли на її елементи є посилання:

```
fn main() {
    let mut vec = vec![1, 2, 3, 4, 5];
    let elem = &vec[2];
    vec.push(6);
    println!("{elem}");
}
```

Аналогічно, розглянемо випадок оголошення ітератора недійсним:

```
fn main() {
    let mut vec = vec![1, 2, 3, 4, 5];
    for elem in &vec {
        vec.push(elem * 2);
    }
}
```

- В обох випадках модифікація колекції шляхом додавання до неї нових елементів може потенційно зробити недійсними наявні посилання на елементи колекції, якщо колекція буде перерозподілена.

22.4 Внутрішня мутабельність

У деяких ситуаціях необхідно модифікувати дані за спільним (доступним лише для читання) посиланням. Наприклад, структура даних зі спільним доступом може мати

внутрішній кеш, і ви хочете оновити цей кеш методами, доступними лише для читання.

Паттерн "внутрішня мутабельність" дозволяє ексклюзивний (мутабельний) доступ за спільним посиланням. Стандартна бібліотека надає декілька способів зробити це, забезпечуючи при цьому безпеку, як правило, шляхом виконання перевірки під час виконання.

`Cell`

`Cell` обгортає значення і дозволяє отримати або встановити значення, використовуючи лише спільне посилання на `Cell`. Однак, вона не дозволяє жодних посилань на внутрішнє значення. Оскільки посилань немає, правила запозичення не можуть бути порушені.

```
use std::cell::Cell;

fn main() {
    // Зауважте, що `cell` НЕ оголошено як мутабельну.
    let cell = Cell::new(5);

    cell.set(123);
    println!("{}", cell.get());
}
```

`RefCell`

`RefCell` дозволяє отримувати доступ до обгорнутого значення та змінювати його, надаючи альтернативні типи `Ref` та `RefMut`, які імітують `&T`/`&mut T`, не будучи насправді Rust посиланнями.

Ці типи виконують динамічні перевірки за допомогою лічильника в `RefCell`, щоб запобігти існуванню `RefMut` поряд з іншим `Ref`/`RefMut`.

Завдяки реалізації `Deref` (і `DerefMut` для `RefMut`), ці типи дозволяють викликати методи за внутрішнім значенням, не дозволяючи посиланням втекти.

```
use std::cell::RefCell;

fn main() {
    // Зауважте, що `cell` НЕ оголошено як мутабельну.
    let cell = RefCell::new(5);

    {
        let mut cell_ref = cell.borrow_mut();
        *cell_ref = 123;

        // Це спричиняє помилку під час виконання.
        // let other = cell.borrow();
        // println!("{}", *other);
    }
```

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

Основне, що можна винести з цього слайду, це те, що Rust надає *безпечні* способи модифікації даних за спільним посиланням. Існує безліч способів забезпечити цю захищеність, і `RefCell` та `Cell` - два з них.

- `RefCell`застосовує звичайні правила запозичень Rust (або декілька спільних посилань, або одне ексклюзивне посилання) з перевіркою під час виконання. У цьому випадку всі запозичення дуже короткі і ніколи не перетинаються, тому перевірки завжди проходять успішно.
  - Додатковий блок у прикладі `RefCell` призначений для завершення запозичення, створеного викликом `borrow_mut`, до того, як ми надрукуємо комірку. Спроба надрукувати запозичену комірку `RefCell` просто покаже повідомлення `"{borrowed}"`.
- `Cell` є найпростішим засобом гарантування безпеки: він має метод `set`, який приймає значення `&self`. Це не потребує перевірки під час виконання, але вимагає переміщення значень, що може мати свою ціну.
- І `RefCell`, і `Cell` є `!Sync`, що означає, що `&RefCell` і `&Cell` не можна передавати між потоками. Це запобігає спробам двох потоків одночасно отримати доступ до комірки.

22.5 Вправа: Статистика здоров'я

Ви працюєте над впровадженням системи моніторингу здоров'я. У рамках цього вам потрібно відстежувати статистику здоров'я користувачів.

Ви почнете із заглушки функції у блоці `impl`, а також з визначення структури `User`. Ваша мета — рреалізувати заглушений метод в структурі `User`, визначений у блоці `impl`.

Скопіюйте код нижче в https://play.rust-lang.org/ та заповніть відсутній метод:

```
// TODO: видаліть це, коли закінчите реалізацію.


pub struct User {
    name: String,
    age: u32,
    height: f32,
    visit_count: usize,
    last_blood_pressure: Option<(u32, u32)>,
}

pub struct Measurements {
    height: f32,
    blood_pressure: (u32, u32),
}

pub struct HealthReport<'a> {
    patient_name: &'a str,
```

```
    visit_count: u32,
    height_change: f32,
    blood_pressure_change: Option<(i32, i32)>,
}

impl User {
    pub fn new(name: String, age: u32, height: f32) -> Self {
        Self { name, age, height, visit_count: 0, last_blood_pressure: None }
    }

    pub fn visit_doctor(&mut self, measurements: Measurements) -> HealthReport {
        todo!(" Оновити статистику користувача на основі вимірювань під час візиту до л
    }
}

fn main() {
    let bob = User::new(String::from("Bob"), 32, 155.2);
    println!("Мене звуть {}, а мій вік - {}", bob.name, bob.age);
}

fn test_visit() {
    let mut bob = User::new(String::from("Bob"), 32, 155.2);
    assert_eq!(bob.visit_count, 0);
    let report =
        bob.visit_doctor(Measurements { height: 156.1, blood_pressure: (120, 80) });
    assert_eq!(report.patient_name, "Bob");
    assert_eq!(report.visit_count, 1);
    assert_eq!(report.blood_pressure_change, None);
    assert!((report.height_change - 0.9).abs() < 0.00001);

    let report =
        bob.visit_doctor(Measurements { height: 156.1, blood_pressure: (115, 76) });

    assert_eq!(report.visit_count, 2);
    assert_eq!(report.blood_pressure_change, Some((-5, -4)));
    assert_eq!(report.height_change, 0.0);
}
```

22.5.1 Рішення

```
pub struct User {
    name: String,
    age: u32,
    height: f32,
    visit_count: usize,
    last_blood_pressure: Option<(u32, u32)>,
}

pub struct Measurements {
    height: f32,
```

```
    blood_pressure: (u32, u32),
}

pub struct HealthReport<'a> {
    patient_name: &'a str,
    visit_count: u32,
    height_change: f32,
    blood_pressure_change: Option<(i32, i32)>,
}

impl User {
    pub fn new(name: String, age: u32, height: f32) -> Self {
        Self { name, age, height, visit_count: 0, last_blood_pressure: None }
    }

    pub fn visit_doctor(&mut self, measurements: Measurements) -> HealthReport {
        self.visit_count += 1;
        let bp = measurements.blood_pressure;
        let report = HealthReport {
            patient_name: &self.name,
            visit_count: self.visit_count as u32,
            height_change: measurements.height - self.height,
            blood_pressure_change: match self.last_blood_pressure {
                Some(lbp) => {
                    Some((bp.0 as i32 - lbp.0 as i32, bp.1 as i32 - lbp.1 as i32))
                }
                None => None,
            },
        };
        self.height = measurements.height;
        self.last_blood_pressure = Some(bp);
        report
    }
}

fn main() {
    let bob = User::new(String::from("Bob"), 32, 155.2);
    println!("Мене звуть {}, а мій вік - {}", bob.name, bob.age);
}

fn test_visit() {
    let mut bob = User::new(String::from("Bob"), 32, 155.2);
    assert_eq!(bob.visit_count, 0);
    let report =
        bob.visit_doctor(Measurements { height: 156.1, blood_pressure: (120, 80) });
    assert_eq!(report.patient_name, "Bob");
    assert_eq!(report.visit_count, 1);
    assert_eq!(report.blood_pressure_change, None);
    assert!((report.height_change - 0.9).abs() < 0.00001);

    let report =
```

```
        bob.visit_doctor(Measurements { height: 156.1, blood_pressure: (115, 76) });

    assert_eq!(report.visit_count, 2);
    assert_eq!(report.blood_pressure_change, Some((-5, -4)));
    assert_eq!(report.height_change, 0.0);
}
```

Розділ 23

Тривалість життя

Цей сегмент повинен зайняти близько 50 хвилин. Він містить:

Слайд	Тривалість
Анотації тривалісті життя	10 хвилин
Упущення тривалісті життя	5 хвилин
Тривалість життя структур	5 хвилин
Вправа: Розбір Protobuf	30 хвилин

23.1 Анотації тривалісті життя

Посилання має *тривалість життя*, яка не повинна "пережити" значення, на яке воно посилається. Це перевіряється чекером запозичень.

Тривалість життя може бути неявною - це те, що ми бачили досі. Часи життя також можуть бути явними: `&'a Point''', &'document str''`. Тривалість життя починається з `'`, а `'a` є типовим іменем за замовчуванням. Читати `&'a Point` як "запозичений `Point`, який є дійсним принаймні протягом тривалості життя `a`".

Тривалість життя завжди визначається компілятором: ви не можете призначити час життя самостійно. Явні анотації часу життя створюють обмеження там, де існує неоднозначність; компілятор перевіряє, чи існує правильний розв'язок.

Часи життя ускладнюються, якщо врахувати передачу значень у функції та повернення значень з них.

```
struct Point(i32, i32);

fn left_most(p1: &Point, p2: &Point) -> &Point {
    if p1.0 < p2.0 {
        p1
    } else {
        p2
    }
}
```

```rust
fn main() {
    let p1: Point = Point(10, 10);
    let p2: Point = Point(20, 20);
    let p3 = left_most(&p1, &p2); // Яка тривалість життя p3?
    println!("p3: {p3:?}");
}
```

У цьому прикладі компілятор не знає, яку тривалість життя виводити для p3. Якщо зазирнути у тіло функції, то можна з упевненістю припустити, що час життя p3 є меншим з двох: p1 та p2. Але так само, як і типи, Rust вимагає явних анотацій тривалості життя для аргументів функції та значень, що повертаються.

Додає `'a` відповідним чином до `left_most`:

```rust
fn left_most<'a>(p1: &'a Point, p2: &'a Point) -> &'a Point {
```

Це говорить, що "за умови, що p1 і p2 живуть довше за `'a`, значення, що повертається, живе принаймні `'a`.

У поширених випадках час життя можна опустити, як описано на наступному слайді.

23.2 Тривалість життя у викликах функцій

Тривалість життя аргументів функції та значень, що повертаються, має бути повністю вказана, але Rust дозволяє у більшості випадків не вказувати тривалість життя за допомогою кількох простих правил. Це не виведення - це просто синтаксичне скорочення.

- Кожному аргументу, який не має анотації тривалості життя, присвоюється одна.
- Якщо існує лише одна тривалість життя аргументу, то вона надається всім неанотованим значенням, що повертаються.
- Якщо існує декілька тривалостей життя аргументів, але перша з них призначена для `self`, ця тривалість надається усім неанотованим значенням повернення.

```rust
struct Point(i32, i32);

fn cab_distance(p1: &Point, p2: &Point) -> i32 {
    (p1.0 - p2.0).abs() + (p1.1 - p2.1).abs()
}

fn nearest<'a>(points: &'a [Point], query: &Point) -> Option<&'a Point> {
    let mut nearest = None;
    for p in points {
        if let Some((_, nearest_dist)) = nearest {
            let dist = cab_distance(p, query);
            if dist < nearest_dist {
                nearest = Some((p, dist));
            }
        } else {
            nearest = Some((p, cab_distance(p, query)));
        };
    }
```

```
    nearest.map(|(p, _)| p)
}

fn main() {
    let points = &[Point(1, 0), Point(1, 0), Point(-1, 0), Point(0, -1)];
    println!("{:?}", nearest(points, &Point(0, 2)));
}
```

У цьому прикладі `cab_distance` тривіально вилучено.

Функція `nearest` є ще одним прикладом функції з декількома посиланнями в аргументах, яка потребує явної анотації.

Спробуйте налаштувати сигнатуру на "брехню" про повернуту тривалість життя:

```
fn nearest<'a, 'q>(points: &'a [Point], query: &'q Point) -> Option<&'q Point> {
```

Це не скомпілюється, демонструючи, що компілятор перевіряє анотації на валідність. Зауважте, що це не стосується сирих вказівників (небезпечних), і це є поширеним джерелом помилок у небезпечному Rust.

Учні можуть запитати, коли слід використовувати тривалість життя. Rust запозичення *завжди* мають тривалість життя. Здебільшого, опускання та виведення типу означають, що їх не потрібно прописувати. У більш складних випадках, анотації тривалості життя можуть допомогти вирішити неоднозначність. Часто, особливо при створенні прототипів, простіше просто працювати з данними якими володіють, клонуючи значення там, де це необхідно.

23.3 Тривалість життя в структурах даних

Якщо тип даних зберігає запозичені дані, він повинен мати анотацію із зазначенням тривалості життя:

```
struct Highlight<'doc>(&'doc str);

fn erase(text: String) {
    println!("Bye {text}!");
}

fn main() {
    let text = String::from("The quick brown fox jumps over the lazy dog.");
    let fox = Highlight(&text[4..19]);
    let dog = Highlight(&text[35..43]);
    // erase(text);
    println!("{fox:?}");
    println!("{dog:?}");
}
```

- У наведеному вище прикладі анотація до `Highlight` гарантує, що дані, які лежать в основі `&str`, існують принаймні стільки, скільки існує будь-який екземпляр `Highlight`, який використовує ці дані.
- Якщо `text` буде спожито до закінчення життя `fox` (або dog), перевірка запозичень видасть помилку.

- Типи з запозиченими даними змушують користувачів зберігати оригінальні дані. Це може бути корисно для створення полегшених представлень, але загалом робить їх дещо складнішими у використанні.
- Якщо це можливо, зробіть так, щоб структури даних безпосередньо володіли своїми даними.
- Деякі структури з декількома посиланнями всередині можуть мати більше ніж одну анотацію про тривалість життя. Це може знадобитися, якщо потрібно описати зв'язки між самими посиланнями впродовж тривалості життя, на додачу до тривалості життя самої структури. Це дуже просунуті випадки використання.

23.4 Вправа: Розбір Protobuf

У цій вправі ви створите синтаксичний аналізатор для бінарного кодування protobuf. Не хвилюйтеся, це простіше, ніж здається! Це ілюструє загальну схему синтаксичного аналізу, передаючи зрізи даних. Самі дані ніколи не копіюються.

Повноцінний розбір повідомлення protobuf вимагає знання типів полів, проіндексованих за номерами полів. Зазвичай ця інформація міститься у файлі `proto`. У цій вправі ми закодуємо цю інформацію у оператори `match` у функціях, які викликаються для кожного поля.

Ми використаємо наступний proto:

```
message PhoneNumber {
  optional string number = 1;
  optional string type = 2;
}

message Person {
  optional string name = 1;
  optional int32 id = 2;
  repeated PhoneNumber phones = 3;
}
```

Messages

A proto message is encoded as a series of fields, one after the next. Each is implemented as a ”tag” followed by the value. The tag contains a field number (e.g., 2 for the `id` field of a `Person` message) and a wire type defining how the payload should be determined from the byte stream. These are combined into a single integer, as decoded in `unpack_tag` below.

Varint

Integers, including the tag, are represented with a variable-length encoding called VARINT. Luckily, `parse_varint` is defined for you below.

Wire Types

Proto defines several wire types, only two of which are used in this exercise.

The `Varint` wire type contains a single varint, and is used to encode proto values of type `int32` such as `Person.id`.

The `Len` wire type contains a length expressed as a varint, followed by a payload of that number of bytes. This is used to encode proto values of type `string` such as `Person.name`. It is also used to encode proto values containing sub-messages such as `Person.phones`, where the payload contains an encoding of the sub-message.

Вправа

The given code also defines callbacks to handle `Person` and `PhoneNumber` fields, and to parse a message into a series of calls to those callbacks.

Вам залишається реалізувати функцію `parse_field` та трейт `ProtoMessage` для `Person` та `PhoneNumber`.

```
/// wire type як він приходить по дроту.
enum WireType {
    /// Varint WireType вказує на те, що значення є одним VARINT.
    Varint,
    // The I64 WireType indicates that the value is precisely 8 bytes in
    // little-endian order containing a 64-bit signed integer or double type.
    //I64,  -- not needed for this exercise
    /// The Len WireType indicates that the value is a length represented as a
    /// VARINT followed by exactly that number of bytes.
    Len,
    // Тип WireType I32 вказує на те, що значення - це рівно 4 байти в
    // little-endian порядку, що містять 32-бітне ціле число зі знаком або тип з плаваю
    //I32,  -- не потрібно для цієї вправи
}

/// Значення поля, введене на основі wire type.
enum FieldValue<'a> {
    Varint(u64),
    //I64(i64),  -- не потрібно для цієї вправи
    Len(&'a [u8]),
    //I32(i32),  -- не потрібно для цієї вправи
}

/// Поле, що містить номер поля та його значення.
struct Field<'a> {
    field_num: u64,
    value: FieldValue<'a>,
}

trait ProtoMessage<'a>: Default {
    fn add_field(&mut self, field: Field<'a>);
}

impl From<u64> for WireType {
    fn from(value: u64) -> Self {
        match value {
```

```
            0 => WireType::Varint,
            //1 => WireType::I64,  -- не потрібно для цієї вправи
            2 => WireType::Len,
            //5 => WireType::I32,  -- не потрібно для цієї вправи
            _ => panic!("Неправильний wire type: {value}"),
        }
    }
}

impl<'a> FieldValue<'a> {
    fn as_str(&self) -> &'a str {
        let FieldValue::Len(data) = self else {
            panic!("Очікуваний рядок має бути полем `Len`");
        };
        std::str::from_utf8(data).expect("Неправильний рядок")
    }

    fn as_bytes(&self) -> &'a [u8] {
        let FieldValue::Len(data) = self else {
            panic!("Очікувані байти мають бути полем `Len`");
        };
        data
    }

    fn as_u64(&self) -> u64 {
        let FieldValue::Varint(value) = self else {
            panic!("Очікувалося, що `u64` буде полем `Varint`");
        };
        *value
    }
}

/// Розбір VARINT з поверненням розібраного значення та решти байтів.
fn parse_varint(data: &[u8]) -> (u64, &[u8]) {
    for i in 0..7 {
        let Some(b) = data.get(i) else {
            panic!("Недостатньо байт для varint");
        };
        if b & 0x80 == 0 {
            // Це останній байт VARINT, тому перетворюємо його
            // в u64 і повертаємо.
            let mut value = 0u64;
            for b in data[..=i].iter().rev() {
                value = (value << 7) | (b & 0x7f) as u64;
            }
            return (value, &data[i + 1..]);
        }
    }

    // Більше 7 байт є неприпустимим.
    panic!("Забагато байт для varint");
```

```rust
}

/// Перетворити тег у номер поля та WireType.
fn unpack_tag(tag: u64) -> (u64, WireType) {
    let field_num = tag >> 3;
    let wire_type = WireType::from(tag & 0x7);
    (field_num, wire_type)
}


/// Розбір поля з поверненням залишку байтів
fn parse_field(data: &[u8]) -> (Field, &[u8]) {
    let (tag, remainder) = parse_varint(data);
    let (field_num, wire_type) = unpack_tag(tag);
    let (fieldvalue, remainder) = match wire_type {
        _ => todo!("На основі wire type побудуйте Field, використовуючи стільки байт, с
    };
    todo!("Повернути поле та всі невикористані байти.")
}

/// Розбір повідомлення за заданими даними, викликаючи `T::add_field` для кожного поля
/// повідомленні.
///
/// Споживаються всі вхідні дані.
fn parse_message<'a, T: ProtoMessage<'a>>(mut data: &'a [u8]) -> T {
    let mut result = T::default();
    while !data.is_empty() {
        let parsed = parse_field(data);
        result.add_field(parsed.0);
        data = parsed.1;
    }
    result
}

struct PhoneNumber<'a> {
    number: &'a str,
    type_: &'a str,
}

struct Person<'a> {
    name: &'a str,
    id: u64,
    phone: Vec<PhoneNumber<'a>>,
}

// TODO: Реалізувати ProtoMessage для Person та PhoneNumber.

fn main() {
    let person_id: Person = parse_message(&[0x10, 0x2a]);
    assert_eq!(person_id, Person { name: "", id: 42, phone: vec![] });
```

```
    let person_name: Person = parse_message(&[
        0x0a, 0x0e, 0x62, 0x65, 0x61, 0x75, 0x74, 0x69, 0x66, 0x75, 0x6c, 0x20,
        0x6e, 0x61, 0x6d, 0x65,
    ]);
    assert_eq!(person_name, Person { name: "beautiful name", id: 0, phone: vec![] });

    let person_name_id: Person =
        parse_message(&[0x0a, 0x04, 0x45, 0x76, 0x61, 0x6e, 0x10, 0x16]);
    assert_eq!(person_name_id, Person { name: "Evan", id: 22, phone: vec![] });

    let phone: Person = parse_message(&[
        0x0a, 0x00, 0x10, 0x00, 0x1a, 0x16, 0x0a, 0x0e, 0x2b, 0x31, 0x32, 0x33,
        0x34, 0x2d, 0x37, 0x37, 0x37, 0x2d, 0x39, 0x30, 0x39, 0x30, 0x12, 0x04,
        0x68, 0x6f, 0x6d, 0x65,
    ]);
    assert_eq!(
        phone,
        Person {
            name: "",
            id: 0,
            phone: vec![PhoneNumber { number: "+1234-777-9090", type_: "home" },],
        }
    );

    // Put that all together into a single parse.
    let person: Person = parse_message(&[
        0x0a, 0x07, 0x6d, 0x61, 0x78, 0x77, 0x65, 0x6c, 0x6c, 0x10, 0x2a, 0x1a,
        0x16, 0x0a, 0x0e, 0x2b, 0x31, 0x32, 0x30, 0x32, 0x2d, 0x35, 0x35, 0x35,
        0x2d, 0x31, 0x32, 0x31, 0x32, 0x12, 0x04, 0x68, 0x6f, 0x6d, 0x65, 0x1a,
        0x18, 0x0a, 0x0e, 0x2b, 0x31, 0x38, 0x30, 0x30, 0x2d, 0x38, 0x36, 0x37,
        0x2d, 0x35, 0x33, 0x30, 0x38, 0x12, 0x06, 0x6d, 0x6f, 0x62, 0x69, 0x6c,
        0x65,
    ]);
    assert_eq!(
        person,
        Person {
            name: "maxwell",
            id: 42,
            phone: vec![
                PhoneNumber { number: "+1202-555-1212", type_: "home" },
                PhoneNumber { number: "+1800-867-5308", type_: "mobile" },
            ]
        }
    );
}
```

- У цій вправі існують різні випадки, коли розбір protobuf може не спрацювати, наприклад, якщо ви спробуєте розібрати `i32`, коли у буфері даних залишилося менше 4 байт. У звичайному Rust-коді ми б впоралися з цим за допомогою переліку `Result`, але для простоти у цій вправі ми панікуємо, якщо виникають помилки. На четвертий день ми розглянемо обробку помилок у Rust більш детально.

23.4.1 Рішення

```
/// wire type як він приходить по дроту.
enum WireType {
    /// Varint WireType вказує на те, що значення є одним VARINT.
    Varint,
    // The I64 WireType indicates that the value is precisely 8 bytes in
    // little-endian order containing a 64-bit signed integer or double type.
    //I64,  -- not needed for this exercise
    /// The Len WireType indicates that the value is a length represented as a
    /// VARINT followed by exactly that number of bytes.
    Len,
    // Тип WireType I32 вказує на те, що значення - це рівно 4 байти в
    // little-endian порядку, що містять 32-бітне ціле число зі знаком або тип з плаваю
    //I32,  -- не потрібно для цієї вправи
}

/// Значення поля, введене на основі wire type.
enum FieldValue<'a> {
    Varint(u64),
    //I64(i64),  -- не потрібно для цієї вправи
    Len(&'a [u8]),
    //I32(i32),  -- не потрібно для цієї вправи
}

/// Поле, що містить номер поля та його значення.
struct Field<'a> {
    field_num: u64,
    value: FieldValue<'a>,
}

trait ProtoMessage<'a>: Default {
    fn add_field(&mut self, field: Field<'a>);
}

impl From<u64> for WireType {
    fn from(value: u64) -> Self {
        match value {
            0 => WireType::Varint,
            //1 => WireType::I64,  -- не потрібно для цієї вправи
            2 => WireType::Len,
            //5 => WireType::I32,  -- не потрібно для цієї вправи
            _ => panic!("Неправильний wire type: {value}"),
        }
    }
}

impl<'a> FieldValue<'a> {
    fn as_str(&self) -> &'a str {
        let FieldValue::Len(data) = self else {
            panic!("Очікуваний рядок має бути полем `Len`");
```

```
        };
        std::str::from_utf8(data).expect("Неправильний рядок")
    }

    fn as_bytes(&self) -> &'a [u8] {
        let FieldValue::Len(data) = self else {
            panic!("Очікувані байти мають бути полем `Len`");
        };
        data
    }

    fn as_u64(&self) -> u64 {
        let FieldValue::Varint(value) = self else {
            panic!("Очікувалося, що `u64` буде полем `Varint`");
        };
        *value
    }
}

/// Розбір VARINT з поверненням розібраного значення та решти байтів.
fn parse_varint(data: &[u8]) -> (u64, &[u8]) {
    for i in 0..7 {
        let Some(b) = data.get(i) else {
            panic!("Недостатньо байт для varint");
        };
        if b & 0x80 == 0 {
            // Це останній байт VARINT, тому перетворюємо його
            // в u64 і повертаємо.
            let mut value = 0u64;
            for b in data[..=i].iter().rev() {
                value = (value << 7) | (b & 0x7f) as u64;
            }
            return (value, &data[i + 1..]);
        }
    }

    // Більше 7 байт є неприпустимим.
    panic!("Забагато байт для varint");
}

/// Перетворити тег у номер поля та WireType.
fn unpack_tag(tag: u64) -> (u64, WireType) {
    let field_num = tag >> 3;
    let wire_type = WireType::from(tag & 0x7);
    (field_num, wire_type)
}

/// Розбір поля з поверненням залишку байтів
fn parse_field(data: &[u8]) -> (Field, &[u8]) {
    let (tag, remainder) = parse_varint(data);
    let (field_num, wire_type) = unpack_tag(tag);
```

```
    let (fieldvalue, remainder) = match wire_type {
        WireType::Varint => {
            let (value, remainder) = parse_varint(remainder);
            (FieldValue::Varint(value), remainder)
        }
        WireType::Len => {
            let (len, remainder) = parse_varint(remainder);
            let len: usize = len.try_into().expect("len не є допустимим `usize`");
            if remainder.len() < len {
                panic!("Несподіваний EOF");
            }
            let (value, remainder) = remainder.split_at(len);
            (FieldValue::Len(value), remainder)
        }
    };
    (Field { field_num, value: fieldvalue }, remainder)
}

/// Розбір повідомлення за заданими даними, викликаючи `T::add_field` для кожного поля в
/// повідомленні.
///
/// Споживаються всі вхідні дані.
fn parse_message<'a, T: ProtoMessage<'a>>(mut data: &'a [u8]) -> T {
    let mut result = T::default();
    while !data.is_empty() {
        let parsed = parse_field(data);
        result.add_field(parsed.0);
        data = parsed.1;
    }
    result
}

struct PhoneNumber<'a> {
    number: &'a str,
    type_: &'a str,
}

struct Person<'a> {
    name: &'a str,
    id: u64,
    phone: Vec<PhoneNumber<'a>>,
}

impl<'a> ProtoMessage<'a> for Person<'a> {
    fn add_field(&mut self, field: Field<'a>) {
        match field.field_num {
            1 => self.name = field.value.as_str(),
            2 => self.id = field.value.as_u64(),
            3 => self.phone.push(parse_message(field.value.as_bytes())),
            _ => {} // пропустити все інше
        }
```

```
    }
}

impl<'a> ProtoMessage<'a> for PhoneNumber<'a> {
    fn add_field(&mut self, field: Field<'a>) {
        match field.field_num {
            1 => self.number = field.value.as_str(),
            2 => self.type_ = field.value.as_str(),
            _ => {} // пропустити все інше
        }
    }
}

fn main() {
    let person_id: Person = parse_message(&[0x10, 0x2a]);
    assert_eq!(person_id, Person { name: "", id: 42, phone: vec![] });

    let person_name: Person = parse_message(&[
        0x0a, 0x0e, 0x62, 0x65, 0x61, 0x75, 0x74, 0x69, 0x66, 0x75, 0x6c, 0x20,
        0x6e, 0x61, 0x6d, 0x65,
    ]);
    assert_eq!(person_name, Person { name: "beautiful name", id: 0, phone: vec![] });

    let person_name_id: Person =
        parse_message(&[0x0a, 0x04, 0x45, 0x76, 0x61, 0x6e, 0x10, 0x16]);
    assert_eq!(person_name_id, Person { name: "Evan", id: 22, phone: vec![] });

    let phone: Person = parse_message(&[
        0x0a, 0x00, 0x10, 0x00, 0x1a, 0x16, 0x0a, 0x0e, 0x2b, 0x31, 0x32, 0x33,
        0x34, 0x2d, 0x37, 0x37, 0x37, 0x2d, 0x39, 0x30, 0x39, 0x30, 0x12, 0x04,
        0x68, 0x6f, 0x6d, 0x65,
    ]);
    assert_eq!(
        phone,
        Person {
            name: "",
            id: 0,
            phone: vec![PhoneNumber { number: "+1234-777-9090", type_: "home" },],
        }
    );

    // Put that all together into a single parse.
    let person: Person = parse_message(&[
        0x0a, 0x07, 0x6d, 0x61, 0x78, 0x77, 0x65, 0x6c, 0x6c, 0x10, 0x2a, 0x1a,
        0x16, 0x0a, 0x0e, 0x2b, 0x31, 0x32, 0x30, 0x32, 0x2d, 0x35, 0x35, 0x35,
        0x2d, 0x31, 0x32, 0x31, 0x32, 0x12, 0x04, 0x68, 0x6f, 0x6d, 0x65, 0x1a,
        0x18, 0x0a, 0x0e, 0x2b, 0x31, 0x38, 0x30, 0x30, 0x2d, 0x38, 0x36, 0x37,
        0x2d, 0x35, 0x33, 0x30, 0x38, 0x12, 0x06, 0x6d, 0x6f, 0x62, 0x69, 0x6c,
        0x65,
    ]);
    assert_eq!(
```

```
        person,
        Person {
            name: "maxwell",
            id: 42,
            phone: vec![
                PhoneNumber { number: "+1202-555-1212", type_: "home" },
                PhoneNumber { number: "+1800-867-5308", type_: "mobile" },
            ]
        }
    );
}
```

Частина VII

День 4: Ранок

Розділ 24

Ласкаво просимо до Дня 4

Сьогодні ми розглянемо теми, що стосуються створення великомасштабного програмного забезпечення на Rust:

- Ітератори: глибоке занурення в трейт `Iterator`.
- Модулі та видимість.
- Тестування
- Обробка помилок: паніка, `Result` і оператор спроби `?`.
- Небезпечний Rust: рятувальний отвір, коли ви не можете виразити себе в безпечному Rust.

Розклад

Враховуючи 10-хвилинні перерви, ця сесія має тривати близько 2 годин 40 хвилин. Вона містить:

Сегмент	Тривалість
Ласкаво просимо	3 хвилини
Ітератори	45 хвилин
Модулі	40 хвилин
Тестування	45 хвилин

Розділ 25

Ітератори

Цей сегмент повинен зайняти близько 45 хвилин. Він містить:

Слайд	Тривалість
Ітератор	5 хвилин
IntoIterator	5 хвилин
FromIterator	5 хвилин
Вправа: ланцюжок методів ітератора	30 хвилин

25.1 `Ітератор`

Трейт `Iterator` підтримує ітерацію над значеннями у колекції. Він вимагає наявності методу `next` і надає багато методів. Багато стандартних бібліотечних типів реалізують `Iterator`, і ви також можете реалізувати його самостійно:

```
struct Fibonacci {
    curr: u32,
    next: u32,
}

impl Iterator for Fibonacci {
    type Item = u32;

    fn next(&mut self) -> Option<Self::Item> {
        let new_next = self.curr + self.next;
        self.curr = self.next;
        self.next = new_next;
        Some(self.curr)
    }
}

fn main() {
    let fib = Fibonacci { curr: 0, next: 1 };
```

```
    for (i, n) in fib.enumerate().take(5) {
        println!("fib({i}): {n}");
    }
}
```

- Трейт `Iterator` реалізує багато поширених функціональних операцій програмування над колекціями (наприклад, `map`, `filter`, `reduce` і т.д.). Це цій трейт, де ви можете знайти всю документацію про них. У Rust ці функції мають створювати код, який є настільки ж ефективним, як і еквівалентні імперативні реалізації.
- `IntoIterator` — це трейт, яка забезпечує роботу циклів for. Він реалізований такими типами колекцій, як `Vec<T>`, і посиланнями на них, наприклад `&Vec<T>` і `&[T]`. Діапазони також реалізують його. Ось чому ви можете перебирати вектор з `for i in some_vec { .. }`, але `some_vec.next()` не існує.

25.2 IntoIterator

Трейт `Iterator` описує, як виконувати *ітерацію* після того, як ви створили ітератор. Пов'язаний з нею трейт `IntoIterator` визначає, як створити ітератор для типу. Він автоматично використовується циклом `for`.

```
struct Grid {
    x_coords: Vec<u32>,
    y_coords: Vec<u32>,
}

impl IntoIterator for Grid {
    type Item = (u32, u32);
    type IntoIter = GridIter;
    fn into_iter(self) -> GridIter {
        GridIter { grid: self, i: 0, j: 0 }
    }
}

struct GridIter {
    grid: Grid,
    i: usize,
    j: usize,
}

impl Iterator for GridIter {
    type Item = (u32, u32);

    fn next(&mut self) -> Option<(u32, u32)> {
        if self.i >= self.grid.x_coords.len() {
            self.i = 0;
            self.j += 1;
            if self.j >= self.grid.y_coords.len() {
                return None;
            }
```

```
        }
        let res = Some((self.grid.x_coords[self.i], self.grid.y_coords[self.j]));
        self.i += 1;
        res
    }
}

fn main() {
    let grid = Grid { x_coords: vec![3, 5, 7, 9], y_coords: vec![10, 20, 30, 40] };
    for (x, y) in grid {
        println!("точка = {x}, {y}");
    }
}
```

Перейдіть до документації по `IntoIterator`. Кожна реалізація `IntoIterator` повинна декларувати два типи:

- `Item`: тип для ітерації, наприклад, `i8`,
- `IntoIter`: тип `Iterator`, що повертається методом `into_iter`.

Зауважте, що `IntoIter` і `Item` пов'язані: ітератор повинен мати той самий тип `Item`, що означає, що він повертає `Option<Item>`

У прикладі перебираються всі комбінації координат x та y.

Спробуйте виконати ітерацію над сіткою двічі в `main`. Чому це не спрацьовує? Зверніть увагу, що `IntoIterator::into_iter` отримує право власності на `self`.

Виправте цю проблему, реалізувавши `IntoIterator` для `&Grid` і зберігаючи посилання на `Grid` у `GridIter`.

Така сама проблема може виникнути для стандартних бібліотечних типів: `for e in some_vector` отримає право власності на `some_vector` і буде перебирати елементи з цього вектора, які йому належать. Натомість використовуйте `for e in &some_vector` для перебору посилань на елементи `some_vector`.

25.3 FromIterator

`FromIterator` дозволяє створювати колекцію з `Iterator`.

```
fn main() {
    let primes = vec![2, 3, 5, 7];
    let prime_squares = primes.into_iter().map(|p| p * p).collect::<Vec<_>>();
    println!("prime_squares: {prime_squares:?}");
}
```

`Iterator` реалізує

```
fn collect<B>(self) -> B
where
    B: FromIterator<Self::Item>,
    Self: Sized
```

Існує два способи вказати `B` для цього методу:

- З ” turbofish”: `some_iterator.collect::<COLLECTION_TYPE>()`, як показано. Скорочення _, використане тут, дозволяє Rust визначити тип елементів `Vec`.
- З виведенням типу: `let prime_squares: Vec<_> = some_iterator.collect()`. Перепишіть приклад так, щоб він мав такий вигляд.

Існують базові реалізації `FromIterator` для `Vec`, `HashMap` тощо. Існують також більш спеціалізовані реалізації, які дозволяють робити цікаві речі, наприклад, перетворювати `Iterator<Item = Result<V, E>>` у `Result<Vec<V>, E>`.

25.4 Вправа: ланцюжок методів ітератора

У цій вправі вам потрібно буде знайти і використати деякі з методів, наданих у трейті `Iterator`, для реалізації складних обчислень.

Скопіюйте наступний код до https://play.rust-lang.org/ і запустіть тести. Для побудови значення, що повертається, використовуйте вираз ітератору та `collect` результат.

```
/// Обчислює різницю між елементами `values`, зміщеними на  `offset`,
/// обгортаючи навколо від кінця `values` до початку.
///
/// Елемент `n` результату має вигляд `values[(n+offset)%len] - values[n]`.
fn offset_differences(offset: usize, values: Vec<i32>) -> Vec<i32> {
    unimplemented!()
}

fn test_offset_one() {
    assert_eq!(offset_differences(1, vec![1, 3, 5, 7]), vec![2, 2, 2, -6]);
    assert_eq!(offset_differences(1, vec![1, 3, 5]), vec![2, 2, -4]);
    assert_eq!(offset_differences(1, vec![1, 3]), vec![2, -2]);
}

fn test_larger_offsets() {
    assert_eq!(offset_differences(2, vec![1, 3, 5, 7]), vec![4, 4, -4, -4]);
    assert_eq!(offset_differences(3, vec![1, 3, 5, 7]), vec![6, -2, -2, -2]);
    assert_eq!(offset_differences(4, vec![1, 3, 5, 7]), vec![0, 0, 0, 0]);
    assert_eq!(offset_differences(5, vec![1, 3, 5, 7]), vec![2, 2, 2, -6]);
}

fn test_degenerate_cases() {
    assert_eq!(offset_differences(1, vec![0]), vec![0]);
    assert_eq!(offset_differences(1, vec![1]), vec![0]);
    let empty: Vec<i32> = vec![];
    assert_eq!(offset_differences(1, empty), vec![]);
}
```

25.4.1 Рішення

```
/// Обчислює різницю між елементами `values`, зміщеними на  `offset`,
/// обгортаючи навколо від кінця `values` до початку.
///
/// Елемент `n` результату має вигляд `values[(n+offset)%len] - values[n]`.
```

```rust
fn offset_differences(offset: usize, values: Vec<i32>) -> Vec<i32> {
    let a = (&values).into_iter();
    let b = (&values).into_iter().cycle().skip(offset);
    a.zip(b).map(|(a, b)| *b - *a).collect()
}

fn test_offset_one() {
    assert_eq!(offset_differences(1, vec![1, 3, 5, 7]), vec![2, 2, 2, -6]);
    assert_eq!(offset_differences(1, vec![1, 3, 5]), vec![2, 2, -4]);
    assert_eq!(offset_differences(1, vec![1, 3]), vec![2, -2]);
}

fn test_larger_offsets() {
    assert_eq!(offset_differences(2, vec![1, 3, 5, 7]), vec![4, 4, -4, -4]);
    assert_eq!(offset_differences(3, vec![1, 3, 5, 7]), vec![6, -2, -2, -2]);
    assert_eq!(offset_differences(4, vec![1, 3, 5, 7]), vec![0, 0, 0, 0]);
    assert_eq!(offset_differences(5, vec![1, 3, 5, 7]), vec![2, 2, 2, -6]);
}

fn test_degenerate_cases() {
    assert_eq!(offset_differences(1, vec![0]), vec![0]);
    assert_eq!(offset_differences(1, vec![1]), vec![0]);
    let empty: Vec<i32> = vec![];
    assert_eq!(offset_differences(1, empty), vec![]);
}

fn main() {}
```

Розділ 26

Модулі

Цей сегмент повинен зайняти близько 40 хвилин. Він містить:

Слайд	Тривалість
Модулі	3 хвилини
Ієрархія файлової системи	5 хвилин
Видимість	5 хвилин
use, super, self	10 хвилин
Вправа: Модулі для бібліотеки графічного інтерфейсу користувача	15 хвилин

26.1 Модулі

Ми бачили, як блоки `impl` дозволяють нам співвідносити функції з типом.

Аналогічно, `mod` надає нам можливості співвідносити типи та функції:

```
mod foo {
    pub fn do_something() {
        println!("У модулі foo");
    }
}

mod bar {
    pub fn do_something() {
        println!("У модулі bar");
    }
}

fn main() {
    foo::do_something();
    bar::do_something();
}
```

- Пакети забезпечують функціональність і включають файл `Cargo.toml`, який описує, як створити пакет із 1+ крейтів.
- Крейти — це дерево модулів, у якому бінарний крейт створює виконуваний файл, а бібліотечний крейт компілюється в бібліотеку.
- Модулі визначають організацію, обсяг і є темою цього розділу.

26.2 Ієрархія файлової системи

Пропущення вмісту модуля призведе до того, що Rust шукатиме його в іншому файлі:

```
mod garden;
```

Це повідомляє Rust, що вміст модуля `garden` знаходиться в `src/garden.rs`. Так само модуль `garden::vegetables` можна знайти на `src/garden/vegetables.rs`.

Корінь `crate` знаходиться в:

- `src/lib.rs` (для крейта бібліотеки)
- `src/main.rs` (для крейта виконуваного файлу)

Модулі, визначені у файлах, також можна документувати за допомогою "внутрішніх коментарів документа". Вони документують елемент, який їх містить – у цьому випадку це модуль.

```
//! Цей модуль реалізує сад, включаючи високоефективну реалізацію
//! пророщування.

// Ре-експорт типів з цього модуля.
pub use garden::Garden;
pub use seeds::SeedPacket;

/// Посіяти задані пакети насіння.
pub fn sow(seeds: Vec<SeedPacket>) {
    todo!()
}

/// Збір врожаю в саду, який вже готовий.
pub fn harvest(garden: &mut Garden) {
    todo!()
}
```

- До Rust 2018 модулі мали розташовуватися в `module/mod.rs` замість `module.rs`, і це все ще робоча альтернатива для випусків після 2018 року.
- Основною причиною введення `filename.rs` як альтернативи `filename/mod.rs` було те, що багато файлів з назвами `mod.rs` важко розрізнити в IDE.
- Більш глибоке вкладення може використовувати папки, навіть якщо основним модулем є файл:

```
src/
├── main.rs
├── top_module.rs
└── top_module/
    └── sub_module.rs
```

- Місце, де Rust шукатиме модулі, можна змінити за допомогою директиви компілятора:

  ```
  mod some_module;
  ```

  Це корисно, наприклад, якщо ви хочете розмістити тести для модуля у файлі з іменем `some_module_test.rs`, подібно до конвенції у Go.

26.3 Видимість

Модулі є межею конфіденційності:

- Елементи модуля є приватними за замовчуванням (приховує деталі реалізації).
- Батьківські та споріднені елементи завжди видно.
- Іншими словами, якщо елемент видимий у модулі `foo`, він видимий у всіх нащадках `foo`.

```
mod outer {
    fn private() {
        println!("outer::private");
    }

    pub fn public() {
        println!("outer::public");
    }

    mod inner {
        fn private() {
            println!("outer::inner::private");
        }

        pub fn public() {
            println!("outer::inner::public");
            super::private();
        }
    }
}

fn main() {
    outer::public();
}
```

- Використовуйте ключове слово pub, щоб зробити модулі загальнодоступними.

Крім того, існують розширені специфікатори `pub(...)` для обмеження обсягу публічної видимості.

- Перегляньте довідник Rust.
- Налаштування видимості `pub(crate)` є типовим шаблоном.
- Рідше ви можете надати видимість певному шляху.
- У будь-якому випадку видимість повинна бути надана модулю-предпопереднику (і всім його нащадкам).

26.4 use, super, self

Модуль може залучати символи з іншого модуля до області видимості за допомогою `use`. Зазвичай ви бачите щось подібне у верхній частині кожного модуля:

```
use std::collections::HashSet;
use std::process::abort;
```

Шляхи

Шляхи вирішуються таким чином:

1. Як відносний шлях:
    - `foo` або `self::foo` посилається на `foo` в поточному модулі,
    - `super::foo` посилається на `foo` у батьківському модулі.
2. Як абсолютний шлях:
    - `crate::foo` посилається на `foo` в корені поточного крейту,
    - `bar::foo` посилається на `foo` в крейті `bar`.

- Зазвичай символи ”реекспортуються” коротшим шляхом. Наприклад, файл `lib.rs` верхнього рівня у крейті може мати

  ```
  mod storage;

  pub use storage::disk::DiskStorage;
  pub use storage::network::NetworkStorage;
  ```

  зробити `DiskStorage` і `NetworkStorage` доступними для інших крейтів зручним і коротким шляхом.

- Здебільшого використовувати `use` потрібно лише з тими елементами, які з'являються в модулі. Однак, щоб викликати будь-які методи, трейт повинен бути в області видимості, навіть якщо тип, що реалізує цей трейт, вже знаходиться в області видимості. Наприклад, для використання методу `read_to_string` на типі, що реалізує трейт`Read`, вам потрібно `use std::io::Read`.
- Оператор `use` може мати символ підстановки: `use std::io::*`. Це не рекомендується, оскільки незрозуміло, які саме елементи імпортуються, а вони можуть змінюватися з часом.

26.5 Вправа: Модулі для бібліотеки графічного інтерфейсу користувача

У цій вправі ви реорганізуєте невелику реалізацію бібліотеки графічного інтерфейсу. У цій бібліотеці визначено трейт `Widget` та декілька реалізацій цього трейту, а також функцію `main`.

Зазвичай кожен тип або групу тісно пов'язаних між собою типів розміщують у власному модулі, тому кожен тип віджету має отримати свій власний модуль.

Установка Cargo

Ігрове середовище Rust підтримує лише один файл, тому вам потрібно створити проект Cargo у вашій локальній файловій системі:

```
cargo init gui-modules
cd gui-modules
cargo run
```

Відредагуйте отриманий файл `src/main.rs`, додавши оператори `mod`, та додайте додаткові файли до каталогу `src`.

Джерело

Ось одномодульна реалізація бібліотеки графічного інтерфейсу:

```
pub trait Widget {
    /// Натуральна ширина `self`.
    fn width(&self) -> usize;

    /// Малює віджету у буфер.
    fn draw_into(&self, buffer: &mut dyn std::fmt::Write);

    /// Малює віджет на стандартному виводі.
    fn draw(&self) {
        let mut buffer = String::new();
        self.draw_into(&mut buffer);
        println!("{buffer}");
    }
}

pub struct Label {
    label: String,
}

impl Label {
    fn new(label: &str) -> Label {
        Label { label: label.to_owned() }
    }
}

pub struct Button {
    label: Label,
}

impl Button {
    fn new(label: &str) -> Button {
        Button { label: Label::new(label) }
    }
}

pub struct Window {
```

```
    title: String,
    widgets: Vec<Box<dyn Widget>>,
}

impl Window {
    fn new(title: &str) -> Window {
        Window { title: title.to_owned(), widgets: Vec::new() }
    }

    fn add_widget(&mut self, widget: Box<dyn Widget>) {
        self.widgets.push(widget);
    }

    fn inner_width(&self) -> usize {
        std::cmp::max(
            self.title.chars().count(),
            self.widgets.iter().map(|w| w.width()).max().unwrap_or(0),
        )
    }
}

impl Widget for Window {
    fn width(&self) -> usize {
        // Додати 4 відступи для країв
        self.inner_width() + 4
    }

    fn draw_into(&self, buffer: &mut dyn std::fmt::Write) {
        let mut inner = String::new();
        for widget in &self.widgets {
            widget.draw_into(&mut inner);
        }

        let inner_width = self.inner_width();

        // TODO: Змініть draw_into на return Result<(), std::fmt::Error>. Тоді використо
        // ?-оператор тут замість .unwrap().
        writeln!(buffer, "+-{:-<inner_width$}-+", "").unwrap();
        writeln!(buffer, "| {:^inner_width$} |", &self.title).unwrap();
        writeln!(buffer, "+={:=<inner_width$}=+", "").unwrap();
        for line in inner.lines() {
            writeln!(buffer, "| {:inner_width$} |", line).unwrap();
        }
        writeln!(buffer, "+-{:-<inner_width$}-+", "").unwrap();
    }
}

impl Widget for Button {
    fn width(&self) -> usize {
        self.label.width() + 8 // додати трохи відступів
    }
```

```
    fn draw_into(&self, buffer: &mut dyn std::fmt::Write) {
        let width = self.width();
        let mut label = String::new();
        self.label.draw_into(&mut label);

        writeln!(buffer, "+{:-<width$}+", "").unwrap();
        for line in label.lines() {
            writeln!(buffer, "|{:^width$}|", &line).unwrap();
        }
        writeln!(buffer, "+{:-<width$}+", "").unwrap();
    }
}

impl Widget for Label {
    fn width(&self) -> usize {
        self.label.lines().map(|line| line.chars().count()).max().unwrap_or(0)
    }

    fn draw_into(&self, buffer: &mut dyn std::fmt::Write) {
        writeln!(buffer, "{}", &self.label).unwrap();
    }
}

fn main() {
    let mut window = Window::new("Rust GUI Demo 1.23");
    window.add_widget(Box::new(Label::new("Це невелика текстова демонстрація графічного
    window.add_widget(Box::new(Button::new("Клацни на мене!")));
    window.draw();
}
```

Заохочуйте студентів розділити код так, як вони вважають за потрібне, і звикнути до необхідних декларацій mod, use і pub. Після цього обговоріть, які організації є найбільш ідіоматичними.

26.5.1 Рішення

```
src
├── main.rs
├── widgets
│   ├── button.rs
│   ├── label.rs
│   └── window.rs
└── widgets.rs
```

```
// ---- src/widgets.rs ----
mod button;
mod label;
mod window;

pub trait Widget {
```

```
    /// Натуральна ширина `self`.
    fn width(&self) -> usize;

    /// Малює віджету у буфер.
    fn draw_into(&self, buffer: &mut dyn std::fmt::Write);

    /// Малює віджет на стандартному виводі.
    fn draw(&self) {
        let mut buffer = String::new();
        self.draw_into(&mut buffer);
        println!("{buffer}");
    }
}

pub use button::Button;
pub use label::Label;
pub use window::Window;
```

```
// ---- src/widgets/label.rs ----
use super::Widget;

pub struct Label {
    label: String,
}

impl Label {
    pub fn new(label: &str) -> Label {
        Label { label: label.to_owned() }
    }
}

impl Widget for Label {
    fn width(&self) -> usize {
        // ANCHOR_END: Label-width
        self.label.lines().map(|line| line.chars().count()).max().unwrap_or(0)
    }

    // ANCHOR: Label-draw_into
    fn draw_into(&self, buffer: &mut dyn std::fmt::Write) {
        // ANCHOR_END: Label-draw_into
        writeln!(buffer, "{}", &self.label).unwrap();
    }
}
```

```
// ---- src/widgets/button.rs ----
use super::{Label, Widget};

pub struct Button {
    label: Label,
}

impl Button {
```

```
    pub fn new(label: &str) -> Button {
        Button { label: Label::new(label) }
    }
}

impl Widget for Button {
    fn width(&self) -> usize {
        // ANCHOR_END: Button-width
        self.label.width() + 8 // додати трохи відступів
    }

    // ANCHOR: Button-draw_into
    fn draw_into(&self, buffer: &mut dyn std::fmt::Write) {
        // ANCHOR_END: Button-draw_into
        let width = self.width();
        let mut label = String::new();
        self.label.draw_into(&mut label);

        writeln!(buffer, "+{:-<width$}+", "").unwrap();
        for line in label.lines() {
            writeln!(buffer, "|{:^width$}|", &line).unwrap();
        }
        writeln!(buffer, "+{:-<width$}+", "").unwrap();
    }
}

// ---- src/widgets/window.rs ----
use super::Widget;

pub struct Window {
    title: String,
    widgets: Vec<Box<dyn Widget>>,
}

impl Window {
    pub fn new(title: &str) -> Window {
        Window { title: title.to_owned(), widgets: Vec::new() }
    }

    pub fn add_widget(&mut self, widget: Box<dyn Widget>) {
        self.widgets.push(widget);
    }

    fn inner_width(&self) -> usize {
        std::cmp::max(
            self.title.chars().count(),
            self.widgets.iter().map(|w| w.width()).max().unwrap_or(0),
        )
    }
}
```

```
impl Widget for Window {
    fn width(&self) -> usize {
        // ANCHOR_END: Window-width
        // Add 4 paddings for borders
        self.inner_width() + 4
    }

    // ANCHOR: Window-draw_into
    fn draw_into(&self, buffer: &mut dyn std::fmt::Write) {
        // ANCHOR_END: Window-draw_into
        let mut inner = String::new();
        for widget in &self.widgets {
            widget.draw_into(&mut inner);
        }

        let inner_width = self.inner_width();

        // TODO: після вивчення обробки помилок можна змінити
        // draw_into щоб повертати Result<(), std::fmt::Error>. Тоді
        // використовуйте тут оператор ? замість .unwrap().
        writeln!(buffer, "+-{:-<inner_width$}-+", "").unwrap();
        writeln!(buffer, "| {:^inner_width$} |", &self.title).unwrap();
        writeln!(buffer, "+={:=<inner_width$}=+", "").unwrap();
        for line in inner.lines() {
            writeln!(buffer, "| {:inner_width$} |", line).unwrap();
        }
        writeln!(buffer, "+-{:-<inner_width$}-+", "").unwrap();
    }
}

// ---- src/main.rs ----
mod widgets;

use widgets::Widget;

fn main() {
    let mut window = widgets::Window::new("Rust GUI Demo 1.23");
    window
        .add_widget(Box::new(widgets::Label::new("Це невелика текстова демонстрація гра
    window.add_widget(Box::new(widgets::Button::new("Клацни на мене!")));
    window.draw();
}
```

Розділ 27

Тестування

Цей сегмент повинен зайняти близько 45 хвилин. Він містить:

Слайд	Тривалість
Тестові модулі	5 хвилин
Інші типи тестів	5 хвилин
Лінти компілятора та Clippy	3 хвилини
Вправа: Алгоритм Луна	30 хвилин

27.1 Модульні тести

Rust і Cargo постачаються з простим фреймворком для модульного тестування:

- Модульні тести підтримуються у всьому коді.
- Тести інтеграції підтримуються через каталог `tests/`.

Тести позначаються `#[test]`. Модульні тести часто розміщують у вкладеному модулі `tests`, використовуючи `#[cfg(test)]` для їх умовної компіляції лише під час збирання тестів.

```
fn first_word(text: &str) -> &str {
    match text.find(' ') {
        Some(idx) => &text[..idx],
        None => &text,
    }
}

mod tests {
    use super::*;

    fn test_empty() {
        assert_eq!(first_word(""), "");
    }
```

```
    fn test_single_word() {
        assert_eq!(first_word("Привіт"), "Привіт");
    }

    fn test_multiple_words() {
        assert_eq!(first_word("Привіт, світ!"), "Привіт");
    }
}
```

- Це дозволяє тестувати приватних помічників.
- Атрибут `#[cfg(test)]` активний лише тоді, коли ви запускаєте `cargo test`.

Запустіть тести на майданчику, щоб показати їхні результати.

27.2 Інші типи тестів

Інтеграційні тести

Якщо ви хочете перевірити свою бібліотеку як клієнт, скористайтеся інтеграційним тестом.

Створіть файл `.rs` у `tests/`:

```
// tests/my_library.rs
use my_library::init;

fn test_init() {
    assert!(init().is_ok());
}
```

Ці тести мають доступ лише до публічного API вашого ящика.

Тести документації

Rust має вбудовану підтримку для тестування документації:

```
/// Скорочує рядок до заданої довжини.
///
/// ```
/// # use playground::shorten_string;
/// assert_eq!(shorten_string("Привіт, світ", 5), "Привіт");
/// assert_eq!(shorten_string("Привіт, світ", 20), "Привіт, світ");
/// ```
pub fn shorten_string(s: &str, length: usize) -> &str {
    &s[..std::cmp::min(length, s.len())]
}
```

- Блоки коду в коментарях `///` автоматично сприймаються як код Rust.
- Код буде скомпільовано та виконано як частину `cargo test`.
- Додавання `#` до коду приховає його з документації, але все одно скомпілює/запустить.
- Перевірте наведений вище код на Rust Playground.

27.3 Лінти компілятора та Clippy

Компілятор Rust видає фантастичні повідомлення про помилки, а також корисні вбудовані лінти. Clippy надає ще більше лінтів, організованих у групи, які можна вмикати для кожного проекту.

```
fn main() {
    let x = 3;
    while (x < 70000) {
        x *= 2;
    }
    println!("X, напевно, поміститься в u16, так? {}", x as u16);
}
```

Запустіть приклад коду і вивчіть повідомлення про помилку. Тут також видно лінти, але вони не будуть показані після компіляції коду. Перейдіть на сайт майданчика, щоб показати ці лінти.

Після усунення лінтів запустіть `clippy` на сайті майданчика, щоб показати попередження clippy. Clippy має вичерпну документацію щодо своїх лінтів і постійно додає нові лінти (включно з лінтами, які заборонено за замовчуванням).

Зауважте, що помилки або попередження з `help: ...` можна виправити за допомогою `cargo fix` або за допомогою вашого редактора.

27.4 Вправа: Алгоритм Луна

Алгоритм Луна використовується для перевірки номерів кредитних карток. Алгоритм приймає рядок як вхідні дані та виконує наступне, щоб перевірити номер кредитної картки:

- Ігноруємо всі пробіли. Відхиляємо числа із менш ніж двома цифрами.
- Рухаючись **справа наліво**, подвоює кожну другу цифру: для числа `1234` ми подвоюємо 3 і 1. Для числа `98765` ми подвоюємо 6 і 8.
- Після подвоєння цифри підсумовує цифри, якщо результат більший за 9. Таким чином, подвоєння 7 перетворюється на 14, яке стає `1 + 4 = 5`.
- Підсумовує всі неподвоєні та подвоєні цифри.
- Номер кредитної картки дійсний, якщо сума закінчується на `0`.

Наданий код містить реалізацію алгоритму Луна з помилками, разом з двома базовими модульними тестами, які підтверджують, що більша частина алгоритму реалізована коректно.

Скопіюйте наведений нижче код на https://play.rust-lang.org/ і напишіть додаткові тести для виявлення помилок у наданій реалізації, виправивши всі знайдені помилки.

```
pub fn luhn(cc_number: &str) -> bool {
    let mut sum = 0;
    let mut double = false;

    for c in cc_number.chars().rev() {
        if let Some(digit) = c.to_digit(10) {
```

```
            if double {
                let double_digit = digit * 2;
                sum +=
                    if double_digit > 9 { double_digit - 9 } else { double_digit };
            } else {
                sum += digit;
            }
            double = !double;
        } else {
            continue;
        }
    }

    sum % 10 == 0
}

mod test {
    use super::*;

    fn test_valid_cc_number() {
        assert!(luhn("4263 9826 4026 9299"));
        assert!(luhn("4539 3195 0343 6467"));
        assert!(luhn("7992 7398 713"));
    }

    fn test_invalid_cc_number() {
        assert!(!luhn("4223 9826 4026 9299"));
        assert!(!luhn("4539 3195 0343 6476"));
        assert!(!luhn("8273 1232 7352 0569"));
    }
}
```

27.4.1 Рішення

```
// Це версія з помилками, яка з'являється у проблемі.
pub fn luhn(cc_number: &str) -> bool {
    let mut sum = 0;
    let mut double = false;

    for c in cc_number.chars().rev() {
        if let Some(digit) = c.to_digit(10) {
            if double {
                let double_digit = digit * 2;
                sum +=
                    if double_digit > 9 { double_digit - 9 } else { double_digit };
            } else {
                sum += digit;
            }
            double = !double;
        } else {
            continue;
```

```
        }
    }

    sum % 10 == 0
}

// Це рішення, яке проходить усі наведені нижче тести.
pub fn luhn(cc_number: &str) -> bool {
    let mut sum = 0;
    let mut double = false;
    let mut digits = 0;

    for c in cc_number.chars().rev() {
        if let Some(digit) = c.to_digit(10) {
            digits += 1;
            if double {
                let double_digit = digit * 2;
                sum +=
                    if double_digit > 9 { double_digit - 9 } else { double_digit };
            } else {
                sum += digit;
            }
            double = !double;
        } else if c.is_whitespace() {
            // Нове: прийняти пробіли.
            continue;
        } else {
            // Нове: відкинути всі інші символи.
            return false;
        }
    }

    // Нове: перевіряємо, чи є хоча б дві цифри
    digits >= 2 && sum % 10 == 0
}

fn main() {
    let cc_number = "1234 5678 1234 5670";
    println!(
        "Чи є {cc_number} дійсним номером кредитної картки? {}",
        if luhn(cc_number) { "так" } else { "ні" }
    );
}

mod test {
    use super::*;

    fn test_valid_cc_number() {
        assert!(luhn("4263 9826 4026 9299"));
        assert!(luhn("4539 3195 0343 6467"));
        assert!(luhn("7992 7398 713"));
```

```
    }

    fn test_invalid_cc_number() {
        assert!(!luhn("4223 9826 4026 9299"));
        assert!(!luhn("4539 3195 0343 6476"));
        assert!(!luhn("8273 1232 7352 0569"));
    }

    fn test_non_digit_cc_number() {
        assert!(!luhn("foo"));
        assert!(!luhn("foo 0 0"));
    }

    fn test_empty_cc_number() {
        assert!(!luhn(""));
        assert!(!luhn(" "));
        assert!(!luhn("  "));
        assert!(!luhn("    "));
    }

    fn test_single_digit_cc_number() {
        assert!(!luhn("0"));
    }

    fn test_two_digit_cc_number() {
        assert!(luhn(" 0 0 "));
    }
}
```

Частина VIII

День 4: Полудень

Розділ 28

Ласкаво просимо назад

Враховуючи 10-хвилинні перерви, ця сесія має тривати близько 2 годин 20 хвилин. Вона містить:

Сегмент	Тривалість
Обробка помилок	1 година та 5 хвилин
Небезпечний Rust	1 година та 5 хвилин

Розділ 29

Обробка помилок

Цей сегмент повинен зайняти близько 1 години 5 хвилин. Він містить:

Слайд	Тривалість
Паніки	3 хвилини
Result	5 хвилин
Оператор спроб	5 хвилин
Перетворення спроб	5 хвилин
Error трейт	5 хвилин
thiserror	5 хвилин
anyhow	5 хвилин
Вправа: Переписування с Result	30 хвилин

29.1 Паніки

Rust обробляє фатальні помилки з ”panic”.

Rust викличе паніку, якщо під час виконання станеться фатальна помилка:

```
fn main() {
    let v = vec![10, 20, 30];
    println!("v[100]: {}", v[100]);
}
```

- Паніки – це невиправні та несподівані помилки.
  - Паніки є ознакою помилок у програмі.
  - Збої у виконанні, такі як невдалі перевірки меж, можуть викликати паніку
  - Твердження (наприклад, `assert!`) панікують у разі невдачі
  - Для спеціальних панік можна використовувати макрос `panic!`.
- Паніка ”розмотує” стек, відкидаючи значення так, як якщо б функції повернулися.
- Використовуйте API, що не викликають паніки (такі як `Vec::get`), якщо збій неприйнятний.

За замовчуванням паніка призведе до розмотування стека. Розмотування можна зловити:

```
use std::panic;

fn main() {
    let result = panic::catch_unwind(|| "Ніяких проблем!");
    println!("{result:?}");

    let result = panic::catch_unwind(|| {
        panic!("о, ні!");
    });
    println!("{result:?}");
}
```

- Перехоплення є незвичайним; не намагайтеся реалізувати виключення за допомогою `catch_unwind`!
- Це може бути корисним на серверах, які повинні продовжувати працювати навіть у разі збою одного запиту.
- Це не працює, якщо у вашому `Cargo.toml` встановлено `panic = 'abort'`.

29.2 `Result`

Основним механізмом обробки помилок у Rust є перелік `Result`, який ми коротко розглядали під час обговорення стандартних бібліотечних типів.

```
use std::fs::File;
use std::io::Read;

fn main() {
    let file: Result<File, std::io::Error> = File::open("diary.txt");
    match file {
        Ok(mut file) => {
            let mut contents = String::new();
            if let Ok(bytes) = file.read_to_string(&mut contents) {
                println!("Дорогий щоденник: {contents} ({bytes} байтів)");
            } else {
                println!("Не вдалося прочитати вміст файлу");
            }
        }
        Err(err) => {
            println!("Щоденник не вдалося відкрити: {err}");
        }
    }
}
```

- `Result` має два варіанти: `Ok`, який містить значення успіху, і `Err`, який містить деяке значення помилки.
- Чи може функція спричинити помилку, кодується у сигнатурі типу функції, яка повертає значення `Result`.
- Як і у випадку з `Option`, ви не можете забути обробити помилку: Ви не можете отримати доступ ні до значення успіху, ні до значення помилки без попередньої обробки шаблону на `Result`, щоб перевірити, який саме варіант ви отримали.

Методи на кшталт `unwrap` полегшують написання швидкого і брудного коду, який не забезпечує надійну обробку помилок, але означає, що ви завжди можете побачити у вихідному коді, де було пропущено належну обробку помилок.

Більше інформації для вивчення

Може бути корисно порівняти обробку помилок у Rust зі стандартами обробки помилок, з якими студенти можуть бути знайомі з інших мов програмування.

Виключення

- Багато мов використовують виключення, наприклад, C++, Java, Python.
- У більшості мов з виключеннями інформація про те, чи може функція згенерувати виключення, не відображається у сигнатурі її типу. Це зазвичай означає, що при виклику функції ви не можете визначити, чи може вона згенерувати виключення.
- Виключення, як правило, розмотують стек викликів, поширюючись вгору, поки не буде досягнуто блоку `try`. Помилка, що виникла глибоко у стеку викликів, може вплинути на не пов'язану з нею функцію, розташовану вище.

Коди помилок

- У деяких мовах функції повертають код помилки (або інше значення помилки) окремо від успішного значення, яке повертає функція. Приклади включають C та Go.
- Залежно від мови можна забути перевірити значення помилки, і в цьому випадку ви можете отримати доступ до неініціалізованого або іншим чином недійсного значення успішного завершення.

29.3 Оператор спроб

Помилки виконання, такі як відмова у з'єднанні або не знайдено файл, обробляються за допомогою типу `Result`, але зіставлення цього типу для кожного виклику може бути громіздким. Оператор спроби `?` використовується для повернення помилок користувачеві. Він дозволяє перетворити звичайний оператор

```
match some_expression {
    Ok(value) => value,
    Err(err) => return Err(err),
}
```

у набагато простіше

```
some_expression?
```

Ми можемо використовувати це, щоб спростити наш код обробки помилок:

```
use std::io::Read;
use std::{fs, io};
```

```rust
fn read_username(path: &str) -> Result<String, io::Error> {
    let username_file_result = fs::File::open(path);
    let mut username_file = match username_file_result {
        Ok(file) => file,
        Err(err) => return Err(err),
    };

    let mut username = String::new();
    match username_file.read_to_string(&mut username) {
        Ok(_) => Ok(username),
        Err(err) => Err(err),
    }
}

fn main() {
    //fs::write("config.dat", "alice").unwrap();
    let username = read_username("config.dat");
    println!("ім'я користувача або помилка: {username:?}");
}
```

Спростіть функцію `read_username` до використання `?`.

Ключові моменти:

- Змінна `username` може мати значення `Ok(string)` або `Err(error)`.
- Використовуйте виклик `fs::write`, щоб перевірити різні сценарії: відсутність файлу, порожній файл, файл з іменем користувача.
- Зверніть увагу, що `main` може повертати `Result<(), E>`, якщо вона реалізує `std::process:Termination`. На практиці це означає, що `E` реалізує `Debug`. Виконуваний файл виведе варіант `Err` і поверне ненульовий статус виходу у разі помилки.

29.4 Перетворення спроб

Ефективне розширення `?` є трохи складнішим, ніж було зазначено раніше:

`expression?`

працює так само, як

```rust
match expression {
    Ok(value) => value,
    Err(err)  => return Err(From::from(err)),
}
```

Виклик `From::from` тут означає, що ми намагаємося перетворити тип помилки на тип, який повертає функція. Це дозволяє легко інкапсулювати помилки у помилки вищого рівня.

Приклад

```rust
use std::error::Error;
use std::io::Read;
```

```rust
use std::{fmt, fs, io};

enum ReadUsernameError {
    IoError(io::Error),
    EmptyUsername(String),
}

impl Error for ReadUsernameError {}

impl fmt::Display for ReadUsernameError {
    fn fmt(&self, f: &mut fmt::Formatter) -> fmt::Result {
        match self {
            Self::IoError(e) => write!(f, "I/O помилка: {e}"),
            Self::EmptyUsername(path) => write!(f, "Не знайдено імені користувача в {pa
        }
    }
}

impl From<io::Error> for ReadUsernameError {
    fn from(err: io::Error) -> Self {
        Self::IoError(err)
    }
}

fn read_username(path: &str) -> Result<String, ReadUsernameError> {
    let mut username = String::with_capacity(100);
    fs::File::open(path)?.read_to_string(&mut username)?;
    if username.is_empty() {
        return Err(ReadUsernameError::EmptyUsername(String::from(path)));
    }
    Ok(username)
}

fn main() {
    //std::fs::write("config.dat", "").unwrap();
    let username = read_username("config.dat");
    println!("ім'я користувача або помилка: {username:?}");
}
```

Оператор ? повинен повертати значення, сумісне з типом повернення функції. Для `Result` це означає, що типи помилок мають бути сумісними. Функція, яка повертає `Result<T, ErrorOuter>`, може використовувати ? на значенні типу `Result<U, ErrorInner>` тільки якщо `ErrorOuter` і `ErrorInner` мають однаковий тип або якщо `ErrorOuter` реалізує `From<ErrorInner>`.

Поширеною альтернативою реалізації `From` є `Result::map_err`, особливо коли перетворення відбувається лише в одному місці.

Для `Option` немає вимог щодо сумісності. Функція, що повертає `Option<T>`, може використовувати оператор ? на `Option<U>` для довільних типів `T` та `U`.

Функція, яка повертає `Result`, не може використовувати ? в `Option` і навпаки. Однак, `Option::ok_or` перетворює `Option` в `Result`, тоді як `Result::ok` перетворює `Result`

в `Option`.

29.5 Динамічні типи помилок

Іноді ми хочемо дозволити повертати будь-який тип помилки без написання власного переліку, що охоплює всі різні можливості. Трейт `std::error::Error` дозволяє легко створити об'єкт трейту, який може містити будь-яку помилку.

```
use std::error::Error;
use std::fs;
use std::io::Read;

fn read_count(path: &str) -> Result<i32, Box<dyn Error>> {
    let mut count_str = String::new();
    fs::File::open(path)?.read_to_string(&mut count_str)?;
    let count: i32 = count_str.parse()?;
    Ok(count)
}

fn main() {
    fs::write("count.dat", "1i3").unwrap();
    match read_count("count.dat") {
        Ok(count) => println!("Підрахунок: {count}"),
        Err(err) => println!("Помилка: {err}"),
    }
}
```

Функція `read_count` може повернути `std::io::Error` (з файлових операцій) або `std::num::ParseIntError` (з `String::parse`).

Пакування помилок економить код, але позбавляє можливості чисто обробляти різні випадки помилок по-різному у програмі. Загалом, це не дуже гарна ідея використовувати `Box<dyn Error>` у публічному API бібліотеки, але це може бути гарним варіантом у програмі, де ви просто хочете десь вивести повідомлення про помилку.

Переконайтеся, що ви використовуєте трейт`std::error::Error` під час визначення користувацького типу помилки, щоб її можна було упакувати.

29.6 `thiserror`

Крейт `thiserror` містить макроси, які допомагають уникнути повторювань при визначенні типів помилок. Він містить похідні макроси, які допомагають реалізувати `From<T>`, `Display` та трейт`Error`.

```
use std::io::Read;
use std::{fs, io};
use thiserror::Error;

enum ReadUsernameError {
    IoError(#[from] io::Error),
```

```
    EmptyUsername(String),
}

fn read_username(path: &str) -> Result<String, ReadUsernameError> {
    let mut username = String::with_capacity(100);
    fs::File::open(path)?.read_to_string(&mut username)?;
    if username.is_empty() {
        return Err(ReadUsernameError::EmptyUsername(String::from(path)));
    }
    Ok(username)
}

fn main() {
    //fs::write("config.dat", "").unwrap();
    match read_username("config.dat") {
        Ok(username) => println!("Ім'я користувача: {username}"),
        Err(err) => println!("Помилка: {err:?}"),
    }
}
```

- Похідний макрос `Error` надається `thiserror` і має багато корисних атрибутів для компактного визначення типів помилок.
- Повідомлення з `#[error]` використовується для отримання трейту `Display`.
- Зауважте, що похідний макрос `(thiserror::)Error`, хоча і має ефект реалізації трейту `(std::error::)Error`, не є тим самим; трейти та макроси не мають спільного простору імен.

29.7 anyhow

Крейт anyhow надає багатий тип помилок з підтримкою передачі додаткової контекстної інформації, яка може бути використана для семантичного відстеження дій програми, що призвели до виникнення помилки.

Це можна поєднати зі зручними макросами з thiserror, щоб уникнути написання реалізацій трейтів явно для користувацьких типів помилок.

```
use anyhow::{bail, Context, Result};
use std::fs;
use std::io::Read;
use thiserror::Error;

struct EmptyUsernameError(String);

fn read_username(path: &str) -> Result<String> {
    let mut username = String::with_capacity(100);
    fs::File::open(path)
        .with_context(|| format!("Не вдалося відкрити {path}"))?
        .read_to_string(&mut username)
        .context("Не вдалося прочитати")?;
    if username.is_empty() {
        bail!(EmptyUsernameError(path.to_string()));
```

```
    }
    Ok(username)
}

fn main() {
    //fs::write("config.dat", "").unwrap();
    match read_username("config.dat") {
        Ok(username) => println!("Ім'я користувача: {username}"),
        Err(err) => println!("Помилка: {err:?}"),
    }
}
```

- `anyhow::Error` по суті є обгорткою навколо `Box<dyn Error>`. Таким чином, це знову ж таки, як правило, не є хорошим вибором для загальнодоступного API бібліотеки, але широко використовується в програмах.
- `anyhow::Result<V>` — це псевдонім типу для `Result<V, anyhow::Error>`.
- Функціональність, яку надає `anyhow::Error`, може бути знайома розробникам Go, оскільки вона забезпечує поведінку, подібну до типу Go `error`, а `Result<T, anyhow::Error>` дуже схожа на Go `(T, error)` (з умовою, що тільки один елемент пари є значущим).
- `anyhow::Context` - це трейт, реалізований для стандартних типів `Result` та `Option`. Використання `anyhow::Context` необхідне для того, щоб дозволити використання `.context()` та `.with_context()` для цих типів.

Більше інформації для вивчення

- `anyhow::Error` має підтримку даункастингу, подібно до `std::any::Any`; конкретний тип помилки, що зберігається всередині, може бути витягнутий для вивчення за допомогою `Error::downcast`.

29.8 Вправа: Переписування с Result

Нижче реалізовано дуже простий синтаксичний аналізатор для мови виразів. Однак, він обробляє помилки панічно. Перепишіть його так, щоб він використовував ідіоматичну обробку помилок і поширював помилки на повернення з `main`. Сміливо використовуйте `thiserror` і `anyhow`.

> **Підказка:** почніть з виправлення обробки помилок у функції `parse`. Після того, як вона буде працювати коректно, оновіть `Tokenizer` для реалізації `Iterator<Item=Result<Token, TokenizerError>>` і обробіть це у парсері.

```
use std::iter::Peekable;
use std::str::Chars;

/// Арифметичний оператор.
enum Op {
    Add,
    Sub,
}
```

```rust
/// Токен у мові виразів.
enum Token {
    Number(String),
    Identifier(String),
    Operator(Op),
}

/// Вираз у мові виразів.
enum Expression {
    /// Посилання на змінну.
    Var(String),
    /// Буквальне число.
    Number(u32),
    /// Бінарна операція.
    Operation(Box<Expression>, Op, Box<Expression>),
}

fn tokenize(input: &str) -> Tokenizer {
    return Tokenizer(input.chars().peekable());
}

struct Tokenizer<'a>(Peekable<Chars<'a>>);

impl<'a> Tokenizer<'a> {
    fn collect_number(&mut self, first_char: char) -> Token {
        let mut num = String::from(first_char);
        while let Some(&c @ '0'..='9') = self.0.peek() {
            num.push(c);
            self.0.next();
        }
        Token::Number(num)
    }

    fn collect_identifier(&mut self, first_char: char) -> Token {
        let mut ident = String::from(first_char);
        while let Some(&c @ ('a'..='z' | '_' | '0'..='9')) = self.0.peek() {
            ident.push(c);
            self.0.next();
        }
        Token::Identifier(ident)
    }
}

impl<'a> Iterator for Tokenizer<'a> {
    type Item = Token;

    fn next(&mut self) -> Option<Token> {
        let c = self.0.next()?;
        match c {
            '0'..='9' => Some(self.collect_number(c)),
            'a'..='z' => Some(self.collect_identifier(c)),
```

```
            '+' => Some(Token::Operator(Op::Add)),
            '-' => Some(Token::Operator(Op::Sub)),
            _ => panic!("Неочікуваний символ {c}"),
        }
    }
}

fn parse(input: &str) -> Expression {
    let mut tokens = tokenize(input);

    fn parse_expr<'a>(tokens: &mut Tokenizer<'a>) -> Expression {
        let Some(tok) = tokens.next() else {
            panic!("Неочікуваний кінець вводу");
        };
        let expr = match tok {
            Token::Number(num) => {
                let v = num.parse().expect("Неправильне 32-бітне ціле число");
                Expression::Number(v)
            }
            Token::Identifier(ident) => Expression::Var(ident),
            Token::Operator(_) => panic!("Неочікуваний токен {tok:?}"),
        };
        // Заглянути наперед, щоб розібрати бінарну операцію, якщо вона присутня.
        match tokens.next() {
            None => expr,
            Some(Token::Operator(op)) => Expression::Operation(
                Box::new(expr),
                op,
                Box::new(parse_expr(tokens)),
            ),
            Some(tok) => panic!("Неочікуваний токен {tok:?}"),
        }
    }

    parse_expr(&mut tokens)
}

fn main() {
    let expr = parse("10+foo+20-30");
    println!("{expr:?}");
}
```

29.8.1 Рішення

```
use thiserror::Error;
use std::iter::Peekable;
use std::str::Chars;

/// Арифметичний оператор.
enum Op {
    Add,
```

```
    Sub,
}

/// Токен у мові виразів.
enum Token {
    Number(String),
    Identifier(String),
    Operator(Op),
}

/// Вираз у мові виразів.
enum Expression {
    /// Посилання на змінну.
    Var(String),
    /// Буквальне число.
    Number(u32),
    /// Бінарна операція.
    Operation(Box<Expression>, Op, Box<Expression>),
}

fn tokenize(input: &str) -> Tokenizer {
    return Tokenizer(input.chars().peekable());
}

enum TokenizerError {
    UnexpectedCharacter(char),
}

struct Tokenizer<'a>(Peekable<Chars<'a>>);

impl<'a> Tokenizer<'a> {
    fn collect_number(&mut self, first_char: char) -> Token {
        let mut num = String::from(first_char);
        while let Some(&c @ '0'..='9') = self.0.peek() {
            num.push(c);
            self.0.next();
        }
        Token::Number(num)
    }

    fn collect_identifier(&mut self, first_char: char) -> Token {
        let mut ident = String::from(first_char);
        while let Some(&c @ ('a'..='z' | '_' | '0'..='9')) = self.0.peek() {
            ident.push(c);
            self.0.next();
        }
        Token::Identifier(ident)
    }
}

impl<'a> Iterator for Tokenizer<'a> {
```

```
    type Item = Result<Token, TokenizerError>;

    fn next(&mut self) -> Option<Result<Token, TokenizerError>> {
        let c = self.0.next()?;
        match c {
            '0'..='9' => Some(Ok(self.collect_number(c))),
            'a'..='z' | '_' => Some(Ok(self.collect_identifier(c))),
            '+' => Some(Ok(Token::Operator(Op::Add))),
            '-' => Some(Ok(Token::Operator(Op::Sub))),
            _ => Some(Err(TokenizerError::UnexpectedCharacter(c))),
        }
    }
}

enum ParserError {
    TokenizerError(#[from] TokenizerError),
    UnexpectedEOF,
    UnexpectedToken(Token),
    InvalidNumber(#[from] std::num::ParseIntError),
}

fn parse(input: &str) -> Result<Expression, ParserError> {
    let mut tokens = tokenize(input);

    fn parse_expr<'a>(
        tokens: &mut Tokenizer<'a>,
    ) -> Result<Expression, ParserError> {
        let tok = tokens.next().ok_or(ParserError::UnexpectedEOF)??;
        let expr = match tok {
            Token::Number(num) => {
                let v = num.parse()?;
                Expression::Number(v)
            }
            Token::Identifier(ident) => Expression::Var(ident),
            Token::Operator(_) => return Err(ParserError::UnexpectedToken(tok)),
        };
        // Заглянути наперед, щоб розібрати бінарну операцію, якщо вона присутня.
        Ok(match tokens.next() {
            None => expr,
            Some(Ok(Token::Operator(op))) => Expression::Operation(
                Box::new(expr),
                op,
                Box::new(parse_expr(tokens)?),
            ),
            Some(Err(e)) => return Err(e.into()),
            Some(Ok(tok)) => return Err(ParserError::UnexpectedToken(tok)),
        })
    }

    parse_expr(&mut tokens)
}
```

```rust
fn main() -> anyhow::Result<()> {
    let expr = parse("10+foo+20-30")?;
    println!("{expr:?}");
    Ok(())
}
```

Розділ 30

Небезпечний Rust

Цей сегмент повинен зайняти близько 1 години 5 хвилин. Він містить:

Слайд	Тривалість
Unsafe	5 хвилин
Розіменування ”сирих” вказівників	10 хвилин
Несталі статичні змінні	5 хвилин
Об'єднання	5 хвилин
Небезпечні функції	5 хвилин
Небезпечні трейти	5 хвилин
Вправа: обгортка FFI	30 хвилин

30.1 Небезпечний Rust

Мова Rust складається з двох частин:

- **Safe Rust:** безпека пам’яті, невизначена поведінка неможлива.
- **Небезпечний Rust:** може викликати невизначену поведінку, якщо порушуються попередні умови.

У цьому курсі ми розглянули переважно безпечний Rust, але важливо знати, що таке небезпечний Rust.

Небезпечний код зазвичай невеликий та ізольований, і його правильність слід ретельно задокументувати. Зазвичай він загорнутий у безпечний рівень абстракції.

Небезпечний Rust дає вам доступ до п’яти нових можливостей:

- Розіменування необроблених вказівників.
- Доступ або зміна мутабельних статичних змінних.
- Доступ до полів `union`.
- Викликати `unsafe` функції, включаючи `extern` функції.
- Реалізація `unsafe` трейтів.

Далі ми коротко розглянемо небезпечні можливості. Щоб отримати повну інформацію, перегляньте розділ 19.1 у книзі Rust та [Rustonomicon](https://doc .rust-lang.org/nomicon/).

Небезпечний Rust не означає, що код неправильний. Це означає, що розробники вимкнули деякі функції безпеки компілятора і змушені писати коректний код самостійно. Це означає, що компілятор більше не забезпечує дотримання правил безпеки пам'яті Rust.

30.2 Розіменування "сирих" вказівників

Створення вказівників є безпечним, але для їх розіменування потрібно `unsafe`:

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

    let r1 = &raw mut s;
    let r2 = r1 as *const String;

    // БЕЗПЕКА: r1 та r2 були отримані з посилань і тому
    // гарантовано є ненульовими та правильно вирівняними, об'єкти, що лежать в основі
    // посилань, з яких вони були отримані, є дійсними на протязі
    // всього небезпечного блоку, і до них немає доступу ні через
    // посилання, ні одночасно через будь-які інші покажчики.
    unsafe {
        println!("r1 є: {}", *r1);
        *r1 = String::from("yyyxoooox");
        println!("r2 є: {}", *r2);
    }

    // НЕБЕЗПЕЧНО. НЕ РОБІТЬ ЦЬОГО.
    /*
    let r3: &String = unsafe { &*r1 };
    drop(s);
    println!("r3 is: {}", *r3);
    */
}
```

Хорошою практикою є (і вимагається посібником зі стилю Android Rust) писати коментар до кожного `unsafe` блоку, пояснюючи, наскільки код у ньому відповідає вимогам безпеки небезпечних операцій, які він виконує.

У випадку розіменувань покажчиків це означає, що покажчики мають бути *дійсними*, тобто:

- Покажчик має бути ненульовим.
- Покажчик має бути *розіменоваючим* (у межах одного виділеного об'єкта).
- Об'єкт не повинен бути звільнений.
- Не повинно бути одночасних доступів до того самого розташування.
- Якщо вказівник було отримано шляхом приведення посилання, базовий об'єкт має бути дійсним, і жодне посилання не може використовуватися для доступу до пам'яті.

У більшості випадків вказівник також має бути правильно вирівняний.

У розділі "НЕ БЕЗПЕЧНО" наведено приклад поширеної помилки UB: `*r1` має `'static` час життя, тому `r3` має тип `&'static String`, і таким чином переживає `s`. Створення

посилання з покажчика вимагає *великої обережності*.

30.3 Несталі статичні змінні

Читати незмінну статичну змінну безпечно:

```
static HELLO_WORLD: &str = "Привіт, світ!";

fn main() {
    println!("HELLO_WORLD: {HELLO_WORLD}");
}
```

Однак, оскільки можуть відбуватися перегони даних, небезпечно читати і записувати статичні змінні, що мутуються:

```
static mut COUNTER: u32 = 0;

fn add_to_counter(inc: u32) {
    // БЕЗПЕКА: Немає інших потоків, які могли б отримати доступ до `COUNTER`.
    unsafe {
        COUNTER += inc;
    }
}

fn main() {
    add_to_counter(42);

    // БЕЗПЕКА: Немає інших потоків, які могли б отримати доступ до `COUNTER`.
    unsafe {
        println!("COUNTER: {COUNTER}");
    }
}
```

- Програма тут безпечна, оскільки вона однопотокова. Однак компілятор Rust є консервативним і припускає найгірше. Спробуйте видалити `unsafe` і подивіться, як компілятор пояснює, що мутація статики з кількох потоків є невизначеною поведінкою.
- Використання мутабельної статики, як правило, погана ідея, але є деякі випадки, коли це може мати сенс у низькорівневому коді `no_std`, наприклад реалізація розподілювача купи або робота з деякими API C.

30.4 Об'єднання

Об’єднання подібні до переліків, але вам потрібно самостійно відстежувати активне поле:

```
union MyUnion {
    i: u8,
    b: bool,
}
```

```
fn main() {
    let u = MyUnion { i: 42 };
    println!("int: {}", unsafe { u.i });
    println!("bool: {}", unsafe { u.b }); // Невизначена поведінка!
}
```

Об'єднання дуже рідко потрібні в Rust, оскільки зазвичай можна використовувати перелік. Іноді вони потрібні для взаємодії з API бібліотек C.

Якщо ви просто хочете по-новому інтерпретувати байти як інший тип, вам, мабуть, знадобиться `std::mem::transmute` або безпечна оболонка, як-от крейт `zerocopy`.

30.5 Небезпечні функції

Виклик небезпечних функцій

Функцію або метод можна позначити як `unsafe`, якщо вони мають додаткові передумови, які ви повинні підтримувати, щоб уникнути невизначеної поведінки:

```
extern "C" {
    fn abs(input: i32) -> i32;
}

fn main() {
    let emojis = "🗻∈🌏";

    // БЕЗПЕКА: Індекси розташовані в правильному порядку в межах
    // фрагмента рядка та лежать на межах послідовності UTF-8.
    unsafe {
        println!("смайлик: {}", emojis.get_unchecked(0..4));
        println!("смайлик: {}", emojis.get_unchecked(4..7));
        println!("смайлик: {}", emojis.get_unchecked(7..11));
    }

    println!("кількість символів: {}", count_chars(unsafe { emojis.get_unchecked(0..7) }

    // БЕЗПЕКА: `abs` не працює з покажчиками і не має жодних вимог до
    // безпеки.
    unsafe {
        println!("Абсолютне значення -3 згідно з C: {}", abs(-3));
    }

    // Недотримання вимог кодування UTF-8 порушує безпеку пам'яті!
    // println!("смайлик: {}", unsafe { emojis.get_unchecked(0..3) });
    // println!("кількість символів: {}", count_chars(unsafe {
    // emojis.get_unchecked(0..3) }));
}

fn count_chars(s: &str) -> usize {
    s.chars().count()
}
```

Написання небезпечних функцій

Ви можете позначити власні функції як `unsafe`, якщо вони вимагають певних умов, щоб уникнути невизначеної поведінки.

```
/// Міняє місцями значення, на які вказують задані покажчики.
///
/// # Безпека
///
/// Покажчики повинні бути дійсними і правильно вирівняними.
unsafe fn swap(a: *mut u8, b: *mut u8) {
    let temp = *a;
    *a = *b;
    *b = temp;
}

fn main() {
    let mut a = 42;
    let mut b = 66;

    // БЕЗПЕКА: ...
    unsafe {
        swap(&mut a, &mut b);
    }

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

Виклик небезпечних функцій

`get_unchecked`, як і більшість функцій `_unchecked`, небезпечна, оскільки може створити UB, якщо діапазон невірний. Функція `abs` небезпечна з іншої причини: вона є зовнішньою функцією (FFI). Виклик зовнішніх функцій зазвичай є проблемою лише тоді, коли ці функції роблять щось із вказівниками, що може порушити модель пам'яті Rust, але загалом будь-яка функція C може мати невизначену поведінку за довільних обставин.

У цьому прикладі `"C"` - це ABI; інші ABI також доступні.

Написання небезпечних функцій

Насправді ми не будемо використовувати вказівники для функції `swap` - це можна безпечно зробити за допомогою посилань.

Зверніть увагу, що небезпечний код дозволяється всередині небезпечної функції без блоку `unsafe`. Ми можемо заборонити це за допомогою `#[deny(unsafe_op_in_unsafe_fn)]`. Спробуйте додати його і подивіться, що станеться. Ймовірно, це буде змінено у майбутньому виданні Rust..

30.6 Реалізація небезпечних трейтів

Як і у випадку з функціями, ви можете позначити трейт `unsafe`, якщо реалізація повинна гарантувати певні умови, щоб уникнути невизначеної поведінки.

Наприклад, крейт `zerocopy` має небезпечний трейт, який виглядає приблизно так:

```
use std::{mem, slice};

/// ...
/// # Безпека
/// Тип повинен мати визначене представлення і не мати заповнень.
pub unsafe trait IntoBytes {
    fn as_bytes(&self) -> &[u8] {
        let len = mem::size_of_val(self);
        unsafe { slice::from_raw_parts((&raw const self).cast::<u8>(), len) }
    }
}

// БЕЗПЕКА: `u32` має визначене представлення і не має заповнення.
unsafe impl IntoBytes for u32 {}
```

У Rustdoc має бути розділ `# Safety` для трейту, що пояснює вимоги до безпечної реалізації функції.

Фактичний розділ безпеки для `IntoBytes` довший і складніший.

Вбудовані Send та Sync трейти є небезпечними.

30.7 Безпечна обгортка інтерфейсу зовнішньої функції (FFI)

Rust має чудову підтримку виклику функцій через *інтерфейс зовнішніх функцій* (FFI). Ми скористаємося цим, щоб створити безпечну оболонку для функцій `libc`, які ви використовуєте у C для читання імен файлів у директорії.

Ви захочете переглянути сторінки посібника:

- `opendir(3)`
- `readdir(3)`
- `closedir(3)`

Ви також захочете переглянути модуль `std::ffi`. Там ви знайдете ряд типів рядків, які вам знадобляться для вправи:

Типи	Кодування	Використання
`str` і `String`	UTF-8	Обробка тексту в Rust
`CStr` і `CString`	NUL-термінований	Спілкування з функціями C
`OsStr` і `OsString`	Специфічні для ОС	Спілкування з ОС

Ви будете конвертувати між усіма цими типами:

- `&str` до `CString`: вам потрібно виділити місце для кінцевого символу `\0`,
- `CString` до `*const i8`: вам потрібен покажчик для виклику функцій C,
- `*const i8` до `&CStr`: вам потрібно щось, що може знайти кінцевий символ `\0`,
- `&CStr` до `&[u8]`: зріз байт є універсальним інтерфейсом для "деяких невідомих даних",
- `&[u8]` до `&OsStr`: `&OsStr` є кроком до `OsString`, використовуйте `OsStrExt`, щоб створити його,
- `&OsStr` до `OsString`: вам потрібно клонувати дані в `&OsStr`, щоб мати можливість повернути їх і знову викликати `readdir`.

У Nomicon також є дуже корисний розділ про FFI.

Скопіюйте код нижче на https://play.rust-lang.org/ і заповніть відсутні функції та методи:

```
// TODO: видаліть це, коли закінчите реалізацію.

mod ffi {
    use std::os::raw::{c_char, c_int};
    use std::os::raw::{c_long, c_uchar, c_ulong, c_ushort};

    //Прозорий тип. Дивіться https://doc.rust-lang.org/nomicon/ffi.html.
    pub struct DIR {
        _data: [u8; 0],
        _marker: core::marker::PhantomData<(*mut u8, core::marker::PhantomPinned)>,
    }

    // Розміщення відповідно до man-сторінки Linux для readdir(3), де ino_t та
    // off_t розгорнуто відповідно до визначень у
    // /usr/include/x86_64-linux-gnu/{sys/types.h, bits/typesizes.h}.
    pub struct dirent {
        pub d_ino: c_ulong,
        pub d_off: c_long,
        pub d_reclen: c_ushort,
        pub d_type: c_uchar,
        pub d_name: [c_char; 256],
    }

    // Розміщення відповідно до man-сторінки macOS для dir(5).
    pub struct dirent {
        pub d_fileno: u64,
        pub d_seekoff: u64,
        pub d_reclen: u16,
        pub d_namlen: u16,
        pub d_type: u8,
        pub d_name: [c_char; 1024],
    }

    unsafe extern "C" {
        pub unsafe fn opendir(s: *const c_char) -> *mut DIR;

        pub unsafe fn readdir(s: *mut DIR) -> *const dirent;
```

```
        // Дивіться https://github.com/rust-lang/libc/issues/414 та розділ про
        // _DARWIN_FEATURE_64_BIT_INODE у man-сторінці macOS про stat(2).
        //
        // "Платформи, які існували до виходу цих оновлень" відносяться
        // до macOS (на відміну від iOS / wearOS / тощо) на Intel і PowerPC.
        pub unsafe fn readdir(s: *mut DIR) -> *const dirent;

        pub unsafe fn closedir(s: *mut DIR) -> c_int;
    }
}

use std::ffi::{CStr, CString, OsStr, OsString};
use std::os::unix::ffi::OsStrExt;

struct DirectoryIterator {
    path: CString,
    dir: *mut ffi::DIR,
}

impl DirectoryIterator {
    fn new(path: &str) -> Result<DirectoryIterator, String> {
        // Викликати opendir і повернути значення Ok, якщо це спрацювало,
        // інакше повернути Err з повідомленням.
        unimplemented!()
    }
}

impl Iterator for DirectoryIterator {
    type Item = OsString;
    fn next(&mut self) -> Option<OsString> {
        // Продовжуємо викликати readdir до тих пір, поки не отримаємо назад вказівник N
        unimplemented!()
    }
}

impl Drop for DirectoryIterator {
    fn drop(&mut self) {
        // Викликати closedir за необхідністю.
        unimplemented!()
    }
}

fn main() -> Result<(), String> {
    let iter = DirectoryIterator::new(".")?;
    println!("файли: {:#?}", iter.collect::<Vec<_>>());
    Ok(())
}
```

30.7.1 Рішення

```
mod ffi {
    use std::os::raw::{c_char, c_int};
    use std::os::raw::{c_long, c_uchar, c_ulong, c_ushort};

    //Прозорий тип. Дивіться https://doc.rust-lang.org/nomicon/ffi.html.
    pub struct DIR {
        _data: [u8; 0],
        _marker: core::marker::PhantomData<(*mut u8, core::marker::PhantomPinned)>,
    }

    // Розміщення відповідно до man-сторінки Linux для readdir(3), де ino_t та
    // off_t розгорнуто відповідно до визначень у
    // /usr/include/x86_64-linux-gnu/{sys/types.h, bits/typesizes.h}.
    pub struct dirent {
        pub d_ino: c_ulong,
        pub d_off: c_long,
        pub d_reclen: c_ushort,
        pub d_type: c_uchar,
        pub d_name: [c_char; 256],
    }

    // Розміщення відповідно до man-сторінки macOS для dir(5).
    pub struct dirent {
        pub d_fileno: u64,
        pub d_seekoff: u64,
        pub d_reclen: u16,
        pub d_namlen: u16,
        pub d_type: u8,
        pub d_name: [c_char; 1024],
    }

    unsafe extern "C" {
        pub unsafe fn opendir(s: *const c_char) -> *mut DIR;

        pub unsafe fn readdir(s: *mut DIR) -> *const dirent;

        // Дивіться https://github.com/rust-lang/libc/issues/414 та розділ про
        // _DARWIN_FEATURE_64_BIT_INODE у man-сторінці macOS про stat(2).
        //
        // "Платформи, які існували до виходу цих оновлень" відносяться
        // до macOS (на відміну від iOS / wearOS / тощо) на Intel і PowerPC.
        pub unsafe fn readdir(s: *mut DIR) -> *const dirent;

        pub unsafe fn closedir(s: *mut DIR) -> c_int;
    }
}

use std::ffi::{CStr, CString, OsStr, OsString};
use std::os::unix::ffi::OsStrExt;
```

```rust
struct DirectoryIterator {
    path: CString,
    dir: *mut ffi::DIR,
}

impl DirectoryIterator {
    fn new(path: &str) -> Result<DirectoryIterator, String> {
        // Викликати opendir і повернути значення Ok, якщо це спрацювало,
        // інакше повернути Err з повідомленням.
        let path =
            CString::new(path).map_err(|err| format!("Неправильний путь: {err}"))?;
        // БЕЗПЕКА: path.as_ptr() не може бути NULL.
        let dir = unsafe { ffi::opendir(path.as_ptr()) };
        if dir.is_null() {
            Err(format!("Не вдалося відкрити {path:?}"))
        } else {
            Ok(DirectoryIterator { path, dir })
        }
    }
}

impl Iterator for DirectoryIterator {
    type Item = OsString;
    fn next(&mut self) -> Option<OsString> {
        // Продовжуємо викликати readdir, доки не отримаємо назад вказівник NULL.
        // БЕЗПЕКА: self.dir ніколи не є NULL.
        let dirent = unsafe { ffi::readdir(self.dir) };
        if dirent.is_null() {
            // Ми досягли кінця директорії.
            return None;
        }
        // БЕЗПЕКА: dirent не є NULL і dirent.d_name є NULL
        // завершено.
        let d_name = unsafe { CStr::from_ptr((*dirent).d_name.as_ptr()) };
        let os_str = OsStr::from_bytes(d_name.to_bytes());
        Some(os_str.to_owned())
    }
}

impl Drop for DirectoryIterator {
    fn drop(&mut self) {
        // Викликаємо closedir за потреби.
        // БЕЗПЕКА: self.dir ніколи не є NULL.
        if unsafe { ffi::closedir(self.dir) } != 0 {
            panic!("Не вдалося закрити {:?}", self.path);
        }
    }
}

fn main() -> Result<(), String> {
```

```
    let iter = DirectoryIterator::new(".")?;
    println!("файли: {:#?}", iter.collect::<Vec<_>>());
    Ok(())
}

mod tests {
    use super::*;
    use std::error::Error;

    fn test_nonexisting_directory() {
        let iter = DirectoryIterator::new("no-such-directory");
        assert!(iter.is_err());
    }

    fn test_empty_directory() -> Result<(), Box<dyn Error>> {
        let tmp = tempfile::TempDir::new()?;
        let iter = DirectoryIterator::new(
            tmp.path().to_str().ok_or("Не UTF-8 символ у шляху")?,
        )?;
        let mut entries = iter.collect::<Vec<_>>();
        entries.sort();
        assert_eq!(entries, &[".", ".."]);
        Ok(())
    }

    fn test_nonempty_directory() -> Result<(), Box<dyn Error>> {
        let tmp = tempfile::TempDir::new()?;
        std::fs::write(tmp.path().join("foo.txt"), "The Foo Diaries\n")?;
        std::fs::write(tmp.path().join("bar.png"), "<PNG>\n")?;
        std::fs::write(tmp.path().join("crab.rs"), "//! Crab\n")?;
        let iter = DirectoryIterator::new(
            tmp.path().to_str().ok_or("Не UTF-8 символ у шляху")?,
        )?;
        let mut entries = iter.collect::<Vec<_>>();
        entries.sort();
        assert_eq!(entries, &[".", "..", "bar.png", "crab.rs", "foo.txt"]);
        Ok(())
    }
}
```

Частина IX

Android

Розділ 31

Ласкаво просимо до Rust в Android

Rust підтримується для системного програмного забезпечення на Android. Це означає, що ви можете писати нові сервіси, бібліотеки, драйвери або навіть прошивки на Rust (або покращувати існуючий код за потреби).

Доповідач може згадати будь-яку з наступних тем, враховуючи зростаюче використання Rust в Android:

- Приклад сервісу: DNS через HTTP.
- Бібліотеки: Rutabaga Virtual Graphics Interface.
- Драйвери ядра: Binder.
- Прошивка: прошивка pKVM.

Розділ 32

Установка

Для тестування нашого коду ми будемо використовувати Cuttlefish Android Virtual Device. Переконайтеся, що у вас є доступ до нього або створіть новий:

```
source build/envsetup.sh
lunch aosp_cf_x86_64_phone-trunk_staging-userdebug
acloud create
```

Докладнішу інформацію можна знайти в Android Developer Codelab.

Код на наступних сторінках можна знайти в `src/android/` директорії матеріалів курсу. Будь ласка, `git clone` репозиторій, щоб продовжити роботу.

Ключові моменти:

- Cuttlefish - це еталонний пристрій Android, призначений для роботи на типових робочих столах Linux. Також планується підтримка MacOS.
- Образ системи Cuttlefish зберігає високу точність до реальних пристроїв і є ідеальним емулятором для запуску багатьох сценаріїв використання Rust.

Розділ 33

Правила побудови

Система збірки Android (Soong) підтримує Rust за допомогою кількох модулів:

Тип модуля	Опис
`rust_binary`	Створює бінарний файл Rust.
`rust_library`	Створює бібліотеку Rust і надає варіанти `rlib` та `dylib`.
`rust_ffi`	Створює бібліотеку Rust C, яку використовують модулі `cc`, і надає як статичні, так і спільні варіанти.
`rust_proc_macro`	Створює бібліотеку `proc-macro` Rust. Вони аналогічні плагінам компілятора.
`rust_test`	Створює бінарний файл тесту Rust, який використовує стандартну систему тестування Rust.
`rust_fuzz`	Створює бінарний файл Rust fuzz, використовуючи `libfuzzer`.
`rust_protobuf`	Генерує вихідний код і створює бібліотеку Rust, яка надає інтерфейс для певного protobuf.

`rust_bindgen`	Генерує вихідний код і створює бібліотеку Rust, яка містить прив'язки Rust до бібліотек C.

Далі ми розглянемо `rust_binary` і `rust_library`.

Спікер може згадати додаткові деталі:

- Cargo не оптимізований для багатомовних репозиторіїв, а також завантажує пакети з інтернету.
- Для сумісності та продуктивності, Android повинен мати крейти в межах дерева. Він також повинен взаємодіяти з кодом C/C++/Java. Soong заповнює цю прогалину.
- Soong має багато спільного з Bazel, який є варіантом Blaze з відкритим вихідним кодом (використовується в google3).
- Цікавий факт: Дані із "Зоряного шляху" - це Android типу Soong

33.1 Бінарні файли Rust

Почнемо з простої програми. У корені AOSP-каси створіть наступні файли:

hello_rust/Android.bp:

```
rust_binary {
    name: "hello_rust",
    crate_name: "hello_rust",
    srcs: ["src/main.rs"],
}
```

hello_rust/src/main.rs:

```
//! Демонстрація Rust.

/// Виводить привітання у стандартний вивід.
fn main() {
    println!("Привіт від Rust!");
}
```

Тепер ви можете створювати, завантажувати та запускати бінарний файл:

```
m hello_rust
adb push "$ANDROID_PRODUCT_OUT/system/bin/hello_rust" /data/local/tmp
adb shell /data/local/tmp/hello_rust
```

```
Hello from Rust!
```

33.2 Бібліотеки Rust

Ви використовуєте `rust_library`, щоб створити нову бібліотеку Rust для Android.

Тут ми оголошуємо залежність від двох бібліотек:

- `libgreeting`, який ми визначаємо нижче,
- `libtextwrap`, який є крейтом, який уже поставляється в `external/rust/crates/`.

hello_rust/Android.bp:

```
rust_binary {
    name: "hello_rust_with_dep",
    crate_name: "hello_rust_with_dep",
    srcs: ["src/main.rs"],
    rustlibs: [
        "libgreetings",
        "libtextwrap",
    ],
    prefer_rlib: true, // Це потрібно, щоб уникнути помилки динамічного лінкування.
}

rust_library {
    name: "libgreetings",
    crate_name: "привіт",
    srcs: ["src/lib.rs"],
}
```

hello_rust/src/main.rs:

```
//! Демонстрація Rust.

use greetings::greeting;
use textwrap::fill;

/// Виводить привітання у стандартний вивід.
fn main() {
    println!("{}", fill(&greeting("Bob"), 24));
}
```

hello_rust/src/lib.rs:

```
//! Бібліотека привітання.

/// Привітати `name`.
pub fn greeting(name: &str) -> String {
    format!("Привіт, {name}, дуже приємно познайомитися з вами!")
}
```

Ви створюєте, завантажуєте та запускаєте бінарний файл, як і раніше:

```
m hello_rust_with_dep
adb push "$ANDROID_PRODUCT_OUT/system/bin/hello_rust_with_dep" /data/local/tmp
adb shell /data/local/tmp/hello_rust_with_dep
```

```
Hello Bob, it is very
nice to meet you!
```

Розділ 34

AIDL

Android Interface Definition Language (AIDL) підтримується в Rust:

- Код Rust може викликати існуючі сервери AIDL,
- Ви можете створювати нові сервери AIDL у Rust.
- AIDL is what enables Android apps to interact with each other.
- Since Rust is supported as a first-class citizen in this ecosystem, Rust services can be called by any other process on the phone.

34.1 Посібник із сервісу Birthday

Щоб проілюструвати, як використовувати Rust з Binder, ми розглянемо процес створення інтерфейсу Binder. Потім ми реалізуємо описаний сервіс і напишемо клієнтський код, який взаємодіє з цим сервісом.

34.1.1 Інтерфейси AIDL

Ви оголошуєте API свого сервісу за допомогою інтерфейсу AIDL:

birthday_service/aidl/com/example/birthdayservice/IBirthdayService.aidl:

```
package com.example.birthdayservice;

/** Інтерфейс сервісу Birthday. */
interface IBirthdayService {
    /** Генерує привітання з днем народження. */
    String wishHappyBirthday(String name, int years);
}
```

birthday_service/aidl/Android.bp:

```
aidl_interface {
    name: "com.example.birthdayservice",
    srcs: ["com/example/birthdayservice/*.aidl"],
    unstable: true,
    backend: {
```

```
        rust: { // Rust не увімкнено за замовчуванням
            enabled: true,
        },
    },
}
```

- Зверніть увагу, що структура каталогів у каталозі `aidl/` має відповідати назві пакета, що використовується у файлі AIDL, тобто пакетом є `com.example.birthdayservice`, а файл знаходиться за адресою `aidl/com/example/IBirthdayService.aidl`.

34.1.2 Згенерований API сервісу

Binder generates a trait for each interface definition.

birthday_service/aidl/com/example/birthdayservice/IBirthdayService.aidl:

```
/** Інтерфейс сервісу Birthday. */
interface IBirthdayService {
    /** Генерує привітання з днем народження. */
    String wishHappyBirthday(String name, int years);
}
```

out/soong/.intermediates/.../birthdayservice/IBirthdayService.rs:

```
trait IBirthdayService {
    fn wishHappyBirthday(&self, name: &str, years: i32) -> binder::Result<String>;
}
```

Ваш сервіс повинен реалізувати цей трейт, а ваш клієнт використовуватиме цей трейт для спілкування зі сервісом.

- Вкажіть, як сигнатура згенерованої функції, зокрема, типи аргументів та повернення, відповідають визначенню інтерфейсу.
  - `String` як аргумент призводить до іншого типу Rust, ніж `String` як тип повернення.

34.1.3 Реалізація сервісу

Тепер ми можемо реалізувати сервіс AIDL:

birthday_service/src/lib.rs:

```
use com_example_birthdayservice::aidl::com::example::birthdayservice::IBirthdayService:
use com_example_birthdayservice::binder;

/// Реалізація `IBirthdayService`.
pub struct BirthdayService;

impl binder::Interface for BirthdayService {}

impl IBirthdayService for BirthdayService {
    fn wishHappyBirthday(&self, name: &str, years: i32) -> binder::Result<String> {
        Ok(format!("З днем народження {name}, вітаємо з {years} роками!"))
    }
}
```

birthday_service/Android.bp:

```
rust_library {
    name: "libbirthdayservice",
    srcs: ["src/lib.rs"],
    crate_name: "birthdayservice",
    rustlibs: [
        "com.example.birthdayservice-rust",
        "libbinder_rs",
    ],
}
```

- Вкажіть шлях до створеного трейту `IBirthdayService` і поясніть, навіщо потрібен кожен з сегментів.
- TODO: Що робить трейт `binder::Interface`? Чи є методи для перевизначення? Де знаходиться вхідний код?

34.1.4 Сервер AIDL

Нарешті, ми можемо створити сервер, який надаватиме сервіс:

birthday_service/src/server.rs:

```
//! Сервіс привітання з днем народження.
use birthdayservice::BirthdayService;
use com_example_birthdayservice::aidl::com::example::birthdayservice::IBirthdayService:
use com_example_birthdayservice::binder;

const SERVICE_IDENTIFIER: &str = "birthdayservice";

/// Точка входу для сервісу дня народження.
fn main() {
    let birthday_service = BirthdayService;
    let birthday_service_binder = BnBirthdayService::new_binder(
        birthday_service,
        binder::BinderFeatures::default(),
    );
    binder::add_service(SERVICE_IDENTIFIER, birthday_service_binder.as_binder())
        .expect("Не вдалося зареєструвати сервіс");
    binder::ProcessState::join_thread_pool();
}
```

birthday_service/Android.bp:

```
rust_binary {
    name: "birthday_server",
    crate_name: "birthday_server",
    srcs: ["src/server.rs"],
    rustlibs: [
        "com.example.birthdayservice-rust",
        "libbinder_rs",
        "libbirthdayservice",
    ],
```

```
    prefer_rlib: true, // Щоб уникнути помилки динамічного лінкування.
}
```

Процес створення користувацької реалізації сервісу (у цьому випадку типу `BirthdayService`, який реалізує `IBirthdayService`) і запуску його як сервісу Binder складається з кількох кроків і може здатися складнішим, ніж ті, хто звик до Binder з C++ або іншої мови. Поясніть учням, чому кожен крок є необхідним.

1. Створіть екземпляр вашого типу сервісу (`BirthdayService`).
2. Оберніть об'єкт сервісу у відповідний тип `Bn*` (у цьому випадку `BnBirthdayService`). Цей тип генерується Binder і надає загальну функціональність Binder, яку надавав би базовий клас `BnBinder` у C++. У Rust немає успадкування, тому замість нього ми використаємо композицію, помістивши наш `BirthdayService` всередину згенерованого `BnBinderService`.
3. Викликаємо `add_service`, передавши йому ідентифікатор сервісу і ваш об'єкт сервісу (у прикладі - об'єкт `BnBirthdayService`).
4. Викликаємо `join_thread_pool` щоб додати поточний потік до пулу потоків Binder'а і починаємо чекати на з'єднання.

34.1.5 Розгортка

Тепер ми можемо створювати, надсилати та запускати службу:

```
m birthday_server
adb push "$ANDROID_PRODUCT_OUT/system/bin/birthday_server" /data/local/tmp
adb root
adb shell /data/local/tmp/birthday_server
```

В іншому терміналі перевірте, чи працює сервіс:

```
adb shell service check birthdayservice
```

```
Service birthdayservice: found
```

Ви також можете викликати сервіс за допомогою `service call`:

```
adb shell service call birthdayservice 1 s16 Bob i32 24
```

```
Result: Parcel(
  0x00000000: 00000000 00000036 00610048 00700070 '....6...H.a.p.p.'
  0x00000010: 00200079 00690042 00740072 00640068 'y. .B.i.r.t.h.d.'
  0x00000020: 00790061 00420020 0062006f 0020002c 'a.y. .B.o.b.,. .'
  0x00000030: 006f0063 0067006e 00610072 00750074 'c.o.n.g.r.a.t.u.'
  0x00000040: 0061006c 00690074 006e006f 00200073 'l.a.t.i.o.n.s. .'
  0x00000050: 00690077 00680074 00740020 00650068 'w.i.t.h. .t.h.e.'
  0x00000060: 00320020 00200034 00650079 00720061 ' .2.4. .y.e.a.r.'
  0x00000070: 00210073 00000000                   's.!.....       ')
```

34.1.6 Клієнт AIDL

Нарешті ми можемо створити клієнт Rust для нашого нового сервісу.

birthday_service/src/client.rs:

```
use com_example_birthdayservice::aidl::com::example::birthdayservice::IBirthdayService:
use com_example_birthdayservice::binder;
```

```rust
const SERVICE_IDENTIFIER: &str = "birthdayservice";

/// Виклик сервісу привітання з днем народження.
fn main() -> Result<(), Box<dyn Error>> {
    let name = std::env::args().nth(1).unwrap_or_else(|| String::from("Bob"));
    let years = std::env::args()
        .nth(2)
        .and_then(|arg| arg.parse::<i32>().ok())
        .unwrap_or(42);

    binder::ProcessState::start_thread_pool();
    let service = binder::get_interface::<dyn IBirthdayService>(SERVICE_IDENTIFIER)
        .map_err(|_| "Не вдалося підключитися до BirthdayService")?;

    // Викликаемо сервіс.
    let msg = service.wishHappyBirthday(&name, years)?;
    println!("{msg}");
}
```

birthday_service/Android.bp:

```
rust_binary {
    name: "birthday_client",
    crate_name: "birthday_client",
    srcs: ["src/client.rs"],
    rustlibs: [
        "com.example.birthdayservice-rust",
        "libbinder_rs",
    ],
    prefer_rlib: true, // Щоб уникнути помилки динамічного лінкування.
}
```

Зауважте, що клієнт не залежить від `libbirthdayservice`.

Створіть, завантажте та запустіть клієнт на своєму пристрої:

```
m birthday_client
adb push "$ANDROID_PRODUCT_OUT/system/bin/birthday_client" /data/local/tmp
adb shell /data/local/tmp/birthday_client Charlie 60

Happy Birthday Charlie, congratulations with the 60 years!
```

- `Strong<dyn IBirthdayService>` - це об'єкт трейту, що представляє сервіс, до якого підключився клієнт.
  - `Strong` - це спеціальний тип розумного вказівника для Binder. Він обробляє як внутрішньопроцесний лічильник посилань на об'єкт сервісного трейту, так і глобальний лічильник посилань Binder, який відстежує, скільки процесів мають посилання на об'єкт.
  - Зверніть увагу, що об'єкт трейта, який клієнт використовує для спілкування з сервісом, використовує той самий трейт, що реалізований на сервері. Для певного інтерфейсу Binder генерується єдиний трейт Rust, який використовується як клієнтом, так і сервером.
- Використовуйте той самий ідентифікатор сервісу, який використовувався при

реєстрації сервісу. В ідеалі він має бути визначений у спільному крейті, на який можуть покладатися як клієнт, так і сервер.

34.1.7 Зміна API

Давайте розширимо API, додавши більше функціональних можливостей: ми хочемо дозволити клієнтам вказувати список рядків для листівки з днем народження:

```
package com.example.birthdayservice;

/** Інтерфейс сервісу Birthday. */
interface IBirthdayService {
    /** Генерує привітання з днем народження. */
    String wishHappyBirthday(String name, int years, in String[] text);
}
```

У результаті буде оновлено визначення трейту для `IBirthdayService`:

```
trait IBirthdayService {
    fn wishHappyBirthday(
        &self,
        name: &str,
        years: i32,
        text: &[String],
    ) -> binder::Result<String>;
}
```

- Зверніть увагу, що `String[]` у визначенні AIDL перекладається як `&[String]` у Rust, тобто ідіоматичні типи Rust використовуються у згенерованих зв'язках скрізь, де це можливо:
  - Аргументи масиву `in` переводяться у зрізи.
  - Аргументи `out` та `inout` транслюються у `&mut Vec<T>`.
  - Значення, що повертаються, перетворюються на `Vec<T>`.

34.1.8 Оновлення клієнта та сервісу

Оновіть клієнтський та серверний код, щоб врахувати новий API.

birthday_service/src/lib.rs:

```
impl IBirthdayService for BirthdayService {
    fn wishHappyBirthday(
        &self,
        name: &str,
        years: i32,
        text: &[String],
    ) -> binder::Result<String> {
        let mut msg = format!(
            "З днем народження {name}, вітаємо з {years} роками!",
        );

        for line in text {
            msg.push('\n');
```

```
            msg.push_str(line);
        }

        Ok(msg)
    }
}
```

birthday_service/src/client.rs:

```
let msg = service.wishHappyBirthday(
    &name,
    years,
    &[
        String::from("Habby birfday to yuuuuu"),
        String::from("А також: багато іншого"),
    ],
)?;
```

- TODO: Перемістити фрагменти коду у файли проекту, де вони будуть зібрані?

34.2 Робота з типами AIDL

Типи AIDL транслюються у відповідний ідіоматичний тип Rust:

- Примітивні типи здебільшого відображаються на ідіоматичні типи Rust.
- Підтримуються такі типи колекцій, як зрізи, `Vec` та рядкові типи.
- Посилання на об'єкти AIDL та дескриптори файлів можуть передаватися між клієнтами та сервісами.
- Повністю підтримуються дескриптори файлів та посилкові дані.

34.2.1 Примітивні типи

Примітивні типи відображаються (здебільшого) ідіоматично:

Тип AIDL	Тип Rust	Примітка
`boolean`	`bool`	
`byte`	`i8`	Зверніть увагу, що байти є знаковими.
`char`	`u16`	Зверніть увагу на використання `u16`, а не `u32`.
`int`	`i32`	
`long`	‘i64	
`float`	`f32`	
‘double	`f64`	
`String`	`String`	

34.2.2 Типи Масивів

Типи масивів (`T[]`, `byte[]` та `List<T>`) буде переведено до відповідного типу масиву Rust залежно від того, як вони використовуються у сигнатурі функції:

Позиція	Тип Rust
`in` аргумент	`&[T]`
`out`/`inout` аргумент	`&mut Vec<T>`
Повернення	`Vec<T>`

- В Android 13 і вище підтримуються масиви фіксованого розміру, тобто `T[N]` стає `[T; N]`. Масиви фіксованого розміру можуть мати декілька вимірів (наприклад, `int[3][4]`). У бекенді Java масиви фіксованого розміру представлені як типи масивів.
- Масиви у посилкових полях завжди перетворюються на `Vec<T>`.

34.2.3 Надсилання об'єктів

AIDL-об'єкти можна надсилати або як конкретний тип AIDL, або як інтерфейс `IBinder` зі стертим типом:

birthday_service/aidl/com/example/birthdayservice/IBirthdayInfoProvider.aidl:

```
package com.example.birthdayservice;

interface IBirthdayInfoProvider {
    String name();
    int years();
}
```

birthday_service/aidl/com/example/birthdayservice/IBirthdayService.aidl:

```
import com.example.birthdayservice.IBirthdayInfoProvider;

interface IBirthdayService {
    /** Те саме, але з використанням об'єкта-зв'язки. */
    String wishWithProvider(IBirthdayInfoProvider provider);

    /** Те саме, але з використанням `IBinder`. */
    String wishWithErasedProvider(IBinder provider);
}
```

birthday_service/src/client.rs:

```
/// Rust структурна структура, що реалізує інтерфейс `IBirthdayInfoProvider`.
struct InfoProvider {
    name: String,
    age: u8,
}

impl binder::Interface for InfoProvider {}

impl IBirthdayInfoProvider for InfoProvider {
    fn name(&self) -> binder::Result<String> {
        Ok(self.name.clone())
    }
```

```
    fn years(&self) -> binder::Result<i32> {
        Ok(self.age as i32)
    }
}

fn main() {
    binder::ProcessState::start_thread_pool();
    let service = connect().expect("Не вдалося підключитися до BirthdayService");

    // Створюємо об'єкт-зв'язувач для інтерфейсу `IBirthdayInfoProvider`.
    let provider = BnBirthdayInfoProvider::new_binder(
        InfoProvider { name: name.clone(), age: years as u8 },
        BinderFeatures::default(),
    );

    // Надсилаємо об'єкт-зв'язку до сервісу.
    service.wishWithProvider(&provider)?;

    // Виконуємо ту саму операцію, але передаємо провайдера як `SpIBinder`.
    service.wishWithErasedProvider(&provider.as_binder())?;
}
```

- Зверніть увагу на використання `BnBirthdayInfoProvider`. Він слугує тій самій меті, що й `BnBirthdayService`, який ми бачили раніше.

34.2.4 Посилкові данні

Binder для Rust підтримує пряме надсилання посилкових данних:

birthday_service/aidl/com/example/birthdayservice/BirthdayInfo.aidl:

```
package com.example.birthdayservice;

parcelable BirthdayInfo {
    String name;
    int years;
}
```

birthday_service/aidl/com/example/birthdayservice/IBirthdayService.aidl:

```
import com.example.birthdayservice.BirthdayInfo;

interface IBirthdayService {
    /** Те саме, але з посилковими даними. */
    String wishWithInfo(in BirthdayInfo info);
}
```

birthday_service/src/client.rs:

```
fn main() {
    binder::ProcessState::start_thread_pool();
    let service = connect().expect("Не вдалося підключитися до BirthdayService");

    let info = BirthdayInfo { name: "Alice".into(), years: 123 };
```

```
    service.wishWithInfo(&info)?;
}
```

34.2.5 Надсилання файлів

Файли можна надсилати між клієнтами/серверами Binder, використовуючи тип `ParcelFileDescriptor`:

birthday_service/aidl/com/example/birthdayservice/IBirthdayService.aidl:

```
interface IBirthdayService {
    /** Те саме, але завантажує інформацію з файлу. */
    String wishFromFile(in ParcelFileDescriptor infoFile);
}
```

birthday_service/src/client.rs:

```
fn main() {
    binder::ProcessState::start_thread_pool();
    let service = connect().expect("Не вдалося підключитися до BirthdayService");

    // Відкриваємо файл і записуємо до нього інформацію про день народження.
    let mut file = File::create("/data/local/tmp/birthday.info").unwrap();
    writeln!(file, "{name}")?;
    writeln!(file, "{years}")?;

    // Створюємо `ParcelFileDescriptor` з файлу та надсилаємо його.
    let file = ParcelFileDescriptor::new(file);
    service.wishFromFile(&file)?;
}
```

birthday_service/src/lib.rs:

```
impl IBirthdayService for BirthdayService {
    fn wishFromFile(
        &self,
        info_file: &ParcelFileDescriptor,
    ) -> binder::Result<String> {
        // Перетворюємо дескриптор файлу в `File`. `ParcelFileDescriptor` обертає
        // `OwnedFd`, який може бути клонований і потім використаний для створення об'єк
        // `File`.
        let mut info_file = info_file
            .as_ref()
            .try_clone()
            .map(File::from)
            .expect("Неправильний дескриптор файлу");

        let mut contents = String::new();
        info_file.read_to_string(&mut contents).unwrap();

        let mut lines = contents.lines();
        let name = lines.next().unwrap();
        let years: i32 = lines.next().unwrap().parse().unwrap();
```

```
        Ok(format!("З днем народження {name}, вітаємо з {years} роками!"))
    }
}
```

- Дескриптор `ParcelFileDescriptor` обгортає `OwnedFd`, тому може бути створений з `File` (або будь-якого іншого типу, який обгортає `OwnedFd`), і може бути використаний для створення нового дескриптора `File` на іншій стороні.
- Інші типи дескрипторів файлів можуть бути загорнуті та надіслані, наприклад, TCP, UDP та UNIX-сокети.

Розділ 35

Тестування в Android

Спираючись на Тестування, ми розглянемо, як працюють юніт-тести в AOSP. Використовуйте модуль `rust_test` для ваших модульних тестів:

testing/Android.bp:

```
rust_library {
    name: "libleftpad",
    crate_name: "leftpad",
    srcs: ["src/lib.rs"],
}

rust_test {
    name: "libleftpad_test",
    crate_name: "leftpad_test",
    srcs: ["src/lib.rs"],
    host_supported: true,
    test_suites: ["general-tests"],
}
```

testing/src/lib.rs:

```
//! Бібліотека лівих відступів.

/// Додати `s` зліва до `width`.
pub fn leftpad(s: &str, width: usize) -> String {
    format!("{s:>width$}")
}

mod tests {
    use super::*;

    fn short_string() {
        assert_eq!(leftpad("foo", 5), "  foo");
    }

    fn long_string() {
```

```
        assert_eq!(leftpad("foobar", 6), "foobar");
    }
}
```

Тепер ви можете запустити тест за допомогою

```
atest --host libleftpad_test
```

Результат має такий вигляд:

```
INFO: Elapsed time: 2.666s, Critical Path: 2.40s
INFO: 3 processes: 2 internal, 1 linux-sandbox.
INFO: Build completed successfully, 3 total actions
//comprehensive-rust-android/testing:libleftpad_test_host            PASSED in 2.3s
    PASSED  libleftpad_test.tests::long_string (0.0s)
    PASSED  libleftpad_test.tests::short_string (0.0s)
Test cases: finished with 2 passing and 0 failing out of 2 test cases
```

Зверніть увагу, що ви згадуєте лише корінь крейта бібліотеки. Тести знаходяться рекурсивно у вкладених модулях.

35.1 GoogleTest

Крейт GoogleTest дозволяє створювати гнучкі тестові твердження за допомогою *зрівнювачів*.

```
use googletest::prelude::*;

fn test_elements_are() {
    let value = vec!["foo", "bar", "baz"];
    expect_that!(value, elements_are!(eq(&"foo"), lt(&"xyz"), starts_with("b")));
}
```

Якщо ми змінимо останній елемент на `"!"`, тест завершиться невдачею зі структурованим повідомленням про помилку, яке точно вказує на помилку:

```
---- test_elements_are stdout ----
Value of: value
Expected: has elements:
  0. is equal to "foo"
  1. is less than "xyz"
  2. starts with prefix "!"
Actual: ["foo", "bar", "baz"],
  where element #2 is "baz", which does not start with "!"
  at src/testing/googletest.rs:6:5
Error: See failure output above
```

- GoogleTest не є частиною Rust Playground, тому вам потрібно запустити цей приклад у локальному середовищі. Скористайтеся `cargo add googletest`, щоб швидко додати його до існуючого проекту Cargo.
- У стрічці `use googletest::prelude::*;` імпортується низка загальновживаних макросів і типів.

- Це лише поверхневий огляд, є багато вбудованих зрівнювачів. Подумайте про те, щоб прочитати перший розділ "Поглиблене тестування для прикладних програм на Rust", самовчитель з Rust: він надає керований вступ до бібліотеки з вправами, які допоможуть вам освоїтися з макросами `googletest` , його зрівнювачами і його загальною філософією.
- Особливо приємною особливістю є те, що розбіжності в багаторядкових рядках відображаються у вигляді diff:

```
fn test_multiline_string_diff() {
    let haiku = "Memory safety found,\n\
                 Rust's strong typing guides the way,\n\
                 Secure code you'll write.";
    assert_that!(
        haiku,
        eq("Memory safety found,\n\
            Rust's silly humor guides the way,\n\
            Secure code you'll write.")
    );
}
```

показує кольорову різницю (кольори тут не показано):

```
    Value of: haiku
Expected: is equal to "Memory safety found,\nRust's silly humor guides the way,\nSecure
Actual: "Memory safety found,\nRust's strong typing guides the way,\nSecure code you'll
  which isn't equal to "Memory safety found,\nRust's silly humor guides the way,\nSecure
Difference(-actual / +expected):
 Memory safety found,
-Rust's strong typing guides the way,
+Rust's silly humor guides the way,
 Secure code you'll write.
  at src/testing/googletest.rs:17:5
```

- Цей крейт - Rust-порт GoogleTest for C++.

35.2 Mocking

Для імітації широко використовується бібліотека Mockall. Вам потрібно рефакторити свій код, щоб використовувати трейти, які потім можна швидко імітувати:

```
use std::time::Duration;

pub trait Pet {
    fn is_hungry(&self, since_last_meal: Duration) -> bool;
}

fn test_robot_dog() {
    let mut mock_dog = MockPet::new();
    mock_dog.expect_is_hungry().return_const(true);
    assert_eq!(mock_dog.is_hungry(Duration::from_secs(10)), true);
}
```

- Mockall - рекомендована бібліотека для створення імітацій в Android (AOSP). На crates.io доступні й інші бібліотеки для імітації, зокрема для імітації HTTP-сервісів. Інші бібліотеки імітацій працюють подібно до Mockall, тобто вони дозволяють легко отримати імітаційну реалізацію заданого трейту.
- Зауважте, що імітація дещо *суперечлива*: імітації дозволяють повністю ізолювати тест від його залежностей. Безпосереднім результатом є швидше і стабільніше виконання тесту. З іншого боку, імітатори можуть бути налаштовані неправильно і повертати результат, відмінний від того, який виводили б реальні залежності.

  Якщо це можливо, рекомендується використовувати реальні залежності. Наприклад, багато баз даних дозволяють налаштовувати бекенд в пам'яті. Це означає, що ви отримаєте правильну поведінку у ваших тестах, до того ж вони швидкі і автоматично прибиратимуть за собою.

  Аналогічно, багато веб-фреймворків дозволяють запускати сервер у процесі роботи, який прив'язується до випадкового порту на `localhost`. Завжди віддавайте перевагу цьому, а не імітаційному фреймворку, оскільки це допоможе вам протестувати ваш код у реальному середовищі.
- Mockall не є частиною Rust Playground, тому вам потрібно запустити цей приклад у локальному середовищі. Використовуйте `cargo add mockall` для швидкого додавання Mockall до існуючого проекту Cargo.
- Mockall має набагато більше функціональних можливостей. Зокрема, ви можете встановлювати очікування, які залежать від переданих аргументів. Тут ми використовуємо це, щоб імітувати кота, який зголоднів через 3 години після того, як його востаннє годували:

```
fn test_robot_cat() {
    let mut mock_cat = MockPet::new();
    mock_cat
        .expect_is_hungry()
        .with(mockall::predicate::gt(Duration::from_secs(3 * 3600)))
        .return_const(true);
    mock_cat.expect_is_hungry().return_const(false);
    assert_eq!(mock_cat.is_hungry(Duration::from_secs(1 * 3600)), false);
    assert_eq!(mock_cat.is_hungry(Duration::from_secs(5 * 3600)), true);
}
```

- Ви можете використати `.times(n)`, щоб обмежити кількість викликів імітаційного методу до `n` --- імітація автоматично панікує при звільненні, якщо ця умова не виконується.

Розділ 36

Журналювання

Ви повинні використовувати крейт `log` для автоматичної реєстрації в `logcat` (на пристрої) або `stdout` (на хості):

hello_rust_logs/Android.bp:

```
rust_binary {
    name: "hello_rust_logs",
    crate_name: "hello_rust_logs",
    srcs: ["src/main.rs"],
    rustlibs: [
        "liblog_rust",
        "liblogger",
    ],
    host_supported: true,
}
```

hello_rust_logs/src/main.rs:

```
//! Демонстрація журналу Rust.

use log::{debug, error, info};

/// Реєструє привітання.
fn main() {
    logger::init(
        logger::Config::default()
            .with_tag_on_device("rust")
            .with_max_level(log::LevelFilter::Trace),
    );
    debug!("Запуск програми.");
    info!("Справи йдуть добре.");
    error!("Щось пішло не так!");
}
```

Створіть, завантажте і запустіть бінарний файл на своєму пристрої:

```
m hello_rust_logs
```

```
adb push "$ANDROID_PRODUCT_OUT/system/bin/hello_rust_logs" /data/local/tmp
adb shell /data/local/tmp/hello_rust_logs
```

Журнали відображаються в `adb logcat`:

```
adb logcat -s rust
```

```
09-08 08:38:32.454  2420  2420 D rust: hello_rust_logs: Starting program.
09-08 08:38:32.454  2420  2420 I rust: hello_rust_logs: Things are going fine.
09-08 08:38:32.454  2420  2420 E rust: hello_rust_logs: Something went wrong!
```

- Реалізація логгера у `liblogger` потрібна лише у фінальній версії, якщо ви логіруєте з бібліотеки, вам знадобиться лише фасадний крейт`log`.

Розділ 37

Інтероперабельність

Rust чудово підтримує взаємодію з іншими мовами. Це означає, що ви можете:

- Викликати функції Rust з інших мов.
- Функції виклику, написані іншими мовами з Rust.

Коли ви викликаєте функції з іншої мови, ми говоримо, що ви використовуєте *foreign function interface*, також відомий як FFI.

37.1 Взаємодія з C

Rust має повну підтримку зв'язування об'єктних файлів за допомогою угоди про виклики C. Так само ви можете експортувати функції Rust і викликати їх із C.

Ви можете зробити це вручну, якщо хочете:

```
unsafe extern "C" {
    safe fn abs(x: i32) -> i32;
}

fn main() {
    let x = -42;
    let abs_x = abs(x);
    println!("{x}, {abs_x}");
}
```

Ми вже бачили це у вправі Safe FFI Wrapper.

> Це передбачає повне знання цільової платформи. Не рекомендується для використання.

Далі ми розглянемо кращі варіанти.

37.1.1 Використання Bindgen

Інструмент bindgen може автоматично генерувати прив'язки з файлу заголовка C.

Спочатку створіть невелику бібліотеку C:

interoperability/bindgen/libbirthday.h:

```
typedef struct card {
  const char* name;
  int years;
} card;

void print_card(const card* card);
```

interoperability/bindgen/libbirthday.c:

```
#include <stdio.h>
#include "libbirthday.h"

void print_card(const card* card) {
  printf("+--------------\n");
  printf("| З днем народженняу %s!\n", card->name);
  printf("| Вітаємо з %i роками!\n", card->years);
  printf("+--------------\n");
}
```

Додайте це до свого файлу `Android.bp`:

interoperability/bindgen/Android.bp:

```
cc_library {
    name: "libbirthday",
    srcs: ["libbirthday.c"],
}
```

Створіть файл заголовка оболонки для бібліотеки (у цьому прикладі це не обов'язково):

interoperability/bindgen/libbirthday_wrapper.h:

```
#include "libbirthday.h"
```

Тепер ви можете автоматично генерувати прив'язки:

interoperability/bindgen/Android.bp:

```
rust_bindgen {
    name: "libbirthday_bindgen",
    crate_name: "birthday_bindgen",
    wrapper_src: "libbirthday_wrapper.h",
    source_stem: "прив'язки",
    static_libs: ["libbirthday"],
}
```

Нарешті, ми можемо використовувати прив'язки в нашій програмі Rust:

interoperability/bindgen/Android.bp:

```
rust_binary {
    name: "print_birthday_card",
    srcs: ["main.rs"],
    rustlibs: ["libbirthday_bindgen"],
}
```

interoperability/bindgen/main.rs:

```rust
//! Демонстрація Bindgen.

use birthday_bindgen::{card, print_card};

fn main() {
    let name = std::ffi::CString::new("Peter").unwrap();
    let card = card { name: name.as_ptr(), years: 42 };
    // БЕЗПЕКА: Вказівник, який ми передаємо, є дійсним, оскільки він прийшов з
    // Rust посилання, а `name`, яке воно містить, посилається на `name`
    // вище, яке також залишається дійсним. `print_card` не зберігає жодного з вказівни
    // їх пізніше після повернення.
    unsafe {
        print_card(&card as *const card);
    }
}
```

Створіть, завантажте і запустіть бінарний файл на своєму пристрої:

```
m print_birthday_card
adb push "$ANDROID_PRODUCT_OUT/system/bin/print_birthday_card" /data/local/tmp
adb shell /data/local/tmp/print_birthday_card
```

Нарешті, ми можемо запустити автоматично згенеровані тести, щоб переконатися, що прив'язки працюють:

interoperability/bindgen/Android.bp:

```
rust_test {
    name: "libbirthday_bindgen_test",
    srcs: [":libbirthday_bindgen"],
    crate_name: "libbirthday_bindgen_test",
    test_suites: ["general-tests"],
    auto_gen_config: true,
    clippy_lints: "none", // Згенерований файл, пропустити лінтування
    lints: "none",
}
```

```
atest libbirthday_bindgen_test
```

37.1.2 Виклик Rust

Експортувати функції та типи Rust на C легко:

interoperability/rust/libanalyze/analyze.rs

```rust
//! Демонстрація Rust FFI.

use std::os::raw::c_int;

/// Аналізуємо цифри.
// БЕЗПЕКА: Іншої глобальної функції з таким ім'ям не існує.
pub extern "C" fn analyze_numbers(x: c_int, y: c_int) {
    if x < y {
        println!("x ({x}) є найменшим!");
    } else {
```

```
        println!("y ({y}) ймовірно більше, ніж x ({x})");
    }
}
```

interoperability/rust/libanalyze/analyze.h

```
#ifndef ANALYSE_H
#define ANALYSE_H

void analyze_numbers(int x, int y);

#endif
```

interoperability/rust/libanalyze/Android.bp

```
rust_ffi {
    name: "libanalyze_ffi",
    crate_name: "analyze_ffi",
    srcs: ["analyze.rs"],
    include_dirs: ["."],
}
```

Тепер ми можемо викликати це з бінарного файлу C:

interoperability/rust/analyze/main.c

```
#include "analyze.h"

int main() {
  analyze_numbers(10, 20);
  analyze_numbers(123, 123);
  return 0;
}
```

interoperability/rust/analyze/Android.bp

```
cc_binary {
    name: "analyze_numbers",
    srcs: ["main.c"],
    static_libs: ["libanalyze_ffi"],
}
```

Створіть, завантажте і запустіть бінарний файл на своєму пристрої:

```
m analyze_numbers
adb push "$ANDROID_PRODUCT_OUT/system/bin/analyze_numbers" /data/local/tmp
adb shell /data/local/tmp/analyze_numbers
```

`#[unsafe(no_mangle)]` вимикає звичайне перетворення назв Rust, тому експортований символ буде просто назвою функції. Ви також можете використовувати `#[unsafe(export_name = "some_name")]`, щоб вказати будь-яке ім'я.

37.2 З C++

Крейт CXX дає змогу безпечно взаємодіяти між Rust і C++.

Загальний підхід виглядає так:

37.2.1 Модуль Bridge

CXX покладається на опис сигнатур функцій, які будуть передаватися з однієї мови до іншої. Ви надаєте цей опис за допомогою блоків extern у модулі Rust, анотованому макросом з атрибутом `#[cxx::bridge]`.

```
mod ffi {
    // Спільні структури з полями, видимими для обох мов.
    struct BlobMetadata {
        size: usize,
        tags: Vec<String>,
    }

    // Типи та сигнатури Rust, що доступні у C++.
    extern "Rust" {
        type MultiBuf;

        fn next_chunk(buf: &mut MultiBuf) -> &[u8];
    }

    // Типи та сигнатури C++, доступні у Rust.
    unsafe extern "C++" {
        include!("include/blobstore.h");

        type BlobstoreClient;

        fn new_blobstore_client() -> UniquePtr<BlobstoreClient>;
        fn put(self: Pin<&mut BlobstoreClient>, parts: &mut MultiBuf) -> u64;
        fn tag(self: Pin<&mut BlobstoreClient>, blobid: u64, tag: &str);
        fn metadata(&self, blobid: u64) -> BlobMetadata;
    }
}
```

- Міст зазвичай оголошується у модулі `ffi` у вашому крейті.
- На основі оголошень, зроблених у модулі-містку, CXX згенерує відповідні визначення типів/функцій Rust та C++, щоб зробити ці елементи доступними для обох мов.
- Щоб переглянути згенерований код Rust, скористайтеся cargo-expand для перегляду розширеного макросу proc. У більшості прикладів ви можете використовувати `cargo expand ::ffi` для розгортання лише модуля `ffi` (хоча це не стосується проектів для Android).
- Щоб переглянути згенерований C++ код, подивіться у `target/cxxbridge`.

37.2.2 Декларації мосту на мові Rust

```
mod ffi {
    extern "Rust" {
        type MyType; // Непрозорий тип
        fn foo(&self); // Метод на `MyType`
```

```
        fn bar() -> Box<MyType>; // Вільна функція
    }
}

struct MyType(i32);

impl MyType {
    fn foo(&self) {
        println!("{}", self.0);
    }
}

fn bar() -> Box<MyType> {
    Box::new(MyType(123))
}
```

- Елементи, оголошені у посиланнях `extern "Rust"`, які знаходяться в області видимості батьківського модуля.
- Генератор коду CXX використовує вашу секцію (секції) `extern "Rust"` для створення заголовного файлу C++, що містить відповідні оголошення C++. Створений заголовок має той самий шлях, що і вихідний файл Rust, який містить міст, за винятком розширення файлу .rs.h.

37.2.3 Згенерований C++

```
mod ffi {
    // Типи та сигнатури Rust, що доступні у C++.
    extern "Rust" {
        type MultiBuf;

        fn next_chunk(buf: &mut MultiBuf) -> &[u8];
    }
}
```

В результаті маємо (приблизно) наступний код на C++:

```
struct MultiBuf final : public ::rust::Opaque {
  ~MultiBuf() = delete;

private:
  friend ::rust::layout;
  struct layout {
    static ::std::size_t size() noexcept;
    static ::std::size_t align() noexcept;
  };
};

::rust::Slice<::std::uint8_t const> next_chunk(::org::blobstore::MultiBuf &buf) noexcept
```

37.2.4 Декларації мосту на мові C++

```
mod ffi {
    // Типи та сигнатури C++, доступні у Rust.
    unsafe extern "C++" {
        include!("include/blobstore.h");

        type BlobstoreClient;

        fn new_blobstore_client() -> UniquePtr<BlobstoreClient>;
        fn put(self: Pin<&mut BlobstoreClient>, parts: &mut MultiBuf) -> u64;
        fn tag(self: Pin<&mut BlobstoreClient>, blobid: u64, tag: &str);
        fn metadata(&self, blobid: u64) -> BlobMetadata;
    }
}
```

В результаті отримуємо (приблизно) такий Rust:

```
pub struct BlobstoreClient {
    _private: ::cxx::private::Opaque,
}

pub fn new_blobstore_client() -> ::cxx::UniquePtr<BlobstoreClient> {
    extern "C" {
        fn __new_blobstore_client() -> *mut BlobstoreClient;
    }
    unsafe { ::cxx::UniquePtr::from_raw(__new_blobstore_client()) }
}

impl BlobstoreClient {
    pub fn put(&self, parts: &mut MultiBuf) -> u64 {
        extern "C" {
            fn __put(
                _: &BlobstoreClient,
                parts: *mut ::cxx::core::ffi::c_void,
            ) -> u64;
        }
        unsafe {
            __put(self, parts as *mut MultiBuf as *mut ::cxx::core::ffi::c_void)
        }
    }
}

// ...
```

- Програмісту не потрібно обіцяти, що введені ним сигнатури є точними. CXX виконує статичні перевірки того, що сигнатури точно відповідають тому, що оголошено у C++.
- Блоки `unsafe extern` дозволяють вам оголошувати функції C++, які безпечно викликати з Rust.

37.2.5 Спільні типи

```
mod ffi {
    struct PlayingCard {
        suit: Suit,
        value: u8,  // A=1, J=11, Q=12, K=13
    }

    enum Suit {
        Clubs,
        Diamonds,
        Hearts,
        Spades,
    }
}
```

- Підтримуються тільки C-подібні (одиничні) переліки.
- Для `#[derive()]` на спільних типах підтримується обмежена кількість трейтів. Відповідна функціональність також генерується для C++ коду, наприклад, якщо ви виводите `Hash`, також генерується реалізація `std::hash` для відповідного типу C++.

37.2.6 Спільні переліки

```
mod ffi {
    enum Suit {
        Clubs,
        Diamonds,
        Hearts,
        Spades,
    }
}
```

Згенерований Rust

```
pub struct Suit {
    pub repr: u8,
}

impl Suit {
    pub const Clubs: Self = Suit { repr: 0 };
    pub const Diamonds: Self = Suit { repr: 1 };
    pub const Hearts: Self = Suit { repr: 2 };
    pub const Spades: Self = Suit { repr: 3 };
}
```

Згенерований C++:

```
enum class Suit : uint8_t {
  Clubs = 0,
  Diamonds = 1,
  Hearts = 2,
  Spades = 3,
```

```
};
```

- З боку Rust, код, що генерується для спільних переліків, насправді є структурою, що обгортає числове значення. Це пов'язано з тим, що у C++ це не є UB для класу переліку зберігати значення, відмінне від усіх перелічених варіантів, і наше представлення у Rust повинно мати таку саму поведінку.

37.2.7 Обробка помилок в Rust

```
mod ffi {
    extern "Rust" {
        fn fallible(depth: usize) -> Result<String>;
    }
}

fn fallible(depth: usize) -> anyhow::Result<String> {
    if depth == 0 {
        return Err(anyhow::Error::msg("fallible1 вимагає глибини > 0"));
    }

    Ok("Успіх!".into())
}
```

- Функції Rust, які повертають `Result`, транслюються у виняткові ситуації на стороні C++.
- Виняткова ситуація, яку буде згенеровано, завжди матиме тип `rust::Error`, який, насамперед, надає можливість отримати рядок з повідомленням про помилку. Повідомлення про помилку буде отримано з імплементації `Display` для типу помилки.
- Паніка при переході з Rust на C++ завжди призведе до негайного завершення процесу.

37.2.8 Обробка помилок в C++

```
mod ffi {
    unsafe extern "C++" {
        include!("example/include/example.h");
        fn fallible(depth: usize) -> Result<String>;
    }
}

fn main() {
    if let Err(err) = ffi::fallible(99) {
        eprintln!("Помилка: {}", err);
        process::exit(1);
    }
}
```

- Функції C++, оголошені як такі, що повертають `Result`, перехоплять будь-яке згенероване виключення на стороні C++ і повернуть його у вигляді значення `Err` до викликаючої функції Rust.

- Якщо виключна ситуація виникає з функції extern "C++", яка не оголошена мостом CXX і повертає `Result`, програма викликає `std::terminate` у C++. Поведінка еквівалентна тій самій виключній ситуації, яка виникає через функцію C++ `noexcept`.

37.2.9 Додаткові типи

Тип Rust	Тип C++
`String`	`rust::String`
`&str`	`rust::Str`
`CxxString`	`std::string`
`&[T]/&mut [T]`	`rust::Slice`
`Box<T>`	`rust::Box<T>`
`UniquePtr<T>`	`std::unique_ptr<T>`
`Vec<T>`	`rust::Vec<T>`
`CxxVector<T>`	`std::vector<T>`

- Ці типи можна використовувати в полях спільних структур, а також в аргументах і поверненнях зовнішніх функцій.
- Зверніть увагу, що `String` у Rust не відображається безпосередньо у `std::string`. На це є декілька причин:
  - `std::string` не підтримує інваріант UTF-8, якого вимагає `String`.
  - Ці два типи мають різне розташування в пам'яті, тому їх не можна передавати безпосередньо між мовами.
  - `std::string` вимагає конструктора переміщення, який не відповідає семантиці переміщення Rust, тому `std::string` не може бути переданий за значенням до Rust

37.2.10 Збірка в Android

Створіть `cc_library_static`, щоб зібрати бібліотеку C++, включаючи згенерований CXX заголовок і вихідний файл.

```
cc_library_static {
    name: "libcxx_test_cpp",
    srcs: ["cxx_test.cpp"],
    generated_headers: [
        "cxx-bridge-header",
        "libcxx_test_bridge_header"
    ],
    generated_sources: ["libcxx_test_bridge_code"],
}
```

- Зверніть увагу, що `libcxx_test_bridge_header` і `libcxx_test_bridge_code` є залежностями для CXX-згенерованих зв'язок C++. Ми покажемо, як їх налаштовувати, на наступному слайді.
- Зауважте, що вам також потрібно залежати від бібліотеки `cxx-bridge-header`, щоб отримати загальні визначення CXX.

- Повну документацію щодо використання CXX в Android можна знайти в документації для Android. Ви можете поділитися цим посиланням з класом, щоб студенти знали, де вони можуть знайти ці інструкції у майбутньому.

37.2.11 Збірка в Android

Створіть два правила генерування: Одне для створення заголовка CXX, а інше для створення вихідного файлу CXX. Потім їх буде використано як вхідні дані для `cc_library_static`.

```
// Згенерує C++ заголовок, що містить C++ прив'язки до
// експортованих функцій Rust у lib.rs.
genrule {
    name: "libcxx_test_bridge_header",
    tools: ["cxxbridge"],
    cmd: "$(location cxxbridge) $(in) --header > $(out)",
    srcs: ["lib.rs"],
    out: ["lib.rs.h"],
}

// Згенерує C++ код, до якого звертається Rust.
genrule {
    name: "libcxx_test_bridge_code",
    tools: ["cxxbridge"],
    cmd: "$(location cxxbridge) $(in) > $(out)",
    srcs: ["lib.rs"],
    out: ["lib.rs.cc"],
}
```

- Інструмент `cxxbridge` - це окремий інструмент, який генерує C++ частину модуля моста. Він входить до складу Android і доступний як інструмент Soong.
- За домовленістю, якщо ваш вихідний файл Rust має ім'я `lib.rs`, ваш заголовний файл буде називатися `lib.rs.h`, а вихідний файл буде називатися `lib.rs.cc`. Втім, цей порядок іменування не є обов'язковим.

37.2.12 Збірка в Android

Створіть `rust_binary`, який залежить від `libcxx` і вашої `cc_library_static`.

```
rust_binary {
    name: "cxx_test",
    srcs: ["lib.rs"],
    rustlibs: ["libcxx"],
    static_libs: ["libcxx_test_cpp"],
}
```

37.3 Взаємодія з Java

Java може завантажувати спільні об’єкти через Java Native Interface (JNI). Крейт `jni` дозволяє створити сумісну бібліотеку.

Спочатку ми створюємо функцію Rust для експорту в Java:

interoperability/java/src/lib.rs:

```
//! Rust <-> Java FFI демонстрація.

use jni::objects::{JClass, JString};
use jni::sys::jstring;
use jni::JNIEnv;

/// Реалізація методу HelloWorld::hello.
// БЕЗПЕКА: Не існує іншої глобальної функції з таким іменем.
pub extern "system" fn Java_HelloWorld_hello(
    mut env: JNIEnv,
    _class: JClass,
    name: JString,
) -> jstring {
    let input: String = env.get_string(&name).unwrap().into();
    let greeting = format!("Привіт, {input}!");
    let output = env.new_string(greeting).unwrap();
    output.into_raw()
}
```

interoperability/java/Android.bp:

```
rust_ffi_shared {
    name: "libhello_jni",
    crate_name: "hello_jni",
    srcs: ["src/lib.rs"],
    rustlibs: ["libjni"],
}
```

Потім ми можемо викликати цю функцію з Java:

interoperability/java/HelloWorld.java:

```
class HelloWorld {
    private static native String hello(String name);

    static {
        System.loadLibrary("hello_jni");
    }

    public static void main(String[] args) {
        String output = HelloWorld.hello("Alice");
        System.out.println(output);
    }
}
```

interoperability/java/Android.bp:

```
java_binary {
    name: "helloworld_jni",
    srcs: ["HelloWorld.java"],
    main_class: "HelloWorld",
```

```
    required: ["libhello_jni"],
}
```

Нарешті, ви можете створити, синхронізувати та запустити бінарний файл:

```
m helloworld_jni
adb sync  # requires adb root && adb remount
adb shell /system/bin/helloworld_jni
```

- The `unsafe(no_mangle)` attribute instructs Rust to emit the `Java_HelloWorld_hello` symbol exactly as written. This is important so that Java can recognize the symbol as a `hello` method on the `HelloWorld` class.
  - By default, Rust will mangle (rename) symbols so that a binary can link in two versions of the same Rust crate.

Частина X

Chromium

Розділ 38

Ласкаво просимо до Rust в Chromium

Rust підтримується для бібліотек сторонніх розробників у Chromium, а також стороннім кодом для з'єднання між Rust та існуючим кодом Chromium C++.

> Сьогодні ми будемо викликати Rust, щоб зробити дещо безглузде з рядками. Якщо у вас є ділянка коду, де ви показуєте користувачеві рядок у кодуванні UTF8, сміливо використовуйте цей рецепт у вашій частині коду, замість тієї частини, про яку ми говоримо.

Розділ 39

Установка

Переконайтеся, що ви можете зібрати та запустити Chromium. Підійде будь-яка платформа і набір флажків збірки, якщо ваш код відносно свіжий (позиція коміту 1223636 і далі, що відповідає листопаду 2023 року):

```
gn gen out/Debug
autoninja -C out/Debug chrome
out/Debug/chrome # or on Mac, out/Debug/Chromium.app/Contents/MacOS/Chromium
```

(Рекомендується використовувати налагоджувальну збірку компонента для скорочення часу ітерацій. Це збірка за замовчуванням!)

Дивіться Як зібрати Chromium, якщо ви ще не зробили цього. Зауважте: підготовка до збірки Chromium потребує часу.

Також рекомендується, щоб у вас був встановлений код Visual Studio.

Про вправи

Ця частина курсу складається з серії вправ, які будуються одна на одній. Ми будемо виконувати їх протягом усього курсу, а не лише наприкінці. Якщо ви не встигнете виконати певну частину, не хвилюйтеся: ви зможете надолужити згаяне на наступному занятті.

Розділ 40

Порівняння екосистем Chromium і Cargo

Спільнота Rust зазвичай використовує `cargo` та бібліотеки з crates.io. Chromium збирається за допомогою `gn` і `ninja` та курованого набору залежностей.

Коли ви пишете код на Rust, у вас є вибір:

- Використовувати `gn` і `ninja` за допомогою шаблонів з `//build/rust/*.gni` (наприклад, `rust_static_library`, з яким ми познайомимося пізніше). Для цього використовується перевірений інструментарій та крейти Chromium.
- Використовувати `cargo`, але обмежитися перевіреним інструментарієм та крейтами Chromium
- Використовувати `cargo`, довіряючи інструментарію та/або крейтам, завантаженим з Інтернету.

Відтепер ми зосередимося на `gn` та `ninja`, тому що саме так код Rust можна вбудувати в браузер Chromium. У той же час, Cargo є важливою частиною екосистеми Rust, і ви повинні мати його у своєму арсеналі інструментів.

Міні вправа

Розділіться на невеликі групи та:

- Проведіть мозковий штурм сценаріїв, де `cargo` може дати перевагу, і оцініть профіль ризику цих сценаріїв.
- Обговоріть, яким інструментам, бібліотекам і групам людей варто довіряти при використанні `gn` і `ninja`, офлайнового `cargo` тощо.

Попросіть студентів не підглядати в нотатки доповідача до завершення вправи. Припускаючи, що учасники курсу фізично знаходяться разом, попросіть їх обговорити питання в малих групах по 3-4 особи.

Зауваження/підказки, пов'язані з першою частиною вправи (”сценарії, в яких Cargo може мати перевагу”):

- Це фантастично, що при написанні інструменту або прототипуванні частини Chromium, ви маєте доступ до багатої екосистеми бібліотек crates.io. Майже для будь-чого є своя бібліотека, і вони, як правило, досить приємні у використанні. (`clap` для розбору командного рядка, `serde` для серіалізації/десеріалізації у/з різні формати, `itertools` для роботи з ітераторами тощо).
  - `cargo` дозволяє легко спробувати бібліотеку (просто додайте один рядок до `Cargo.toml` і починайте писати код)
  - Можливо, варто порівняти, як CPAN допоміг зробити `perl` популярним вибором. Або порівняти з `python` + `pip`.
- Процес розробки полегшують не лише основні інструменти Rust (наприклад, використання `rustup` для перемикання на іншу версію `rustc` при тестуванні крейту, який має працювати на нічних, поточних стабільних та старих стабільних версіях), але й екосистема сторонніх інструментів (наприклад, Mozilla надає `cargo vet` для впорядкування та спільного використання аудитів безпеки; крейт `criterion` надає спрощений спосіб запуску бенчмарків).
  - `cargo` спрощує додавання інструмента за допомогою `cargo install --locked cargo-vet`.
  - Можливо, варто порівняти з розширеннями Chrome або VScode.
- Широкі, загальні приклади проектів, де `cargo` може бути правильним вибором:
  - Можливо, це дивно, але Rust стає все більш популярним в індустрії написання інструментів командного інтерфейсу. Широта та ергономічність бібліотек порівнянна з Python, при цьому вона більш надійна (завдяки багатій системі типів) і працює швидше (як скомпільована, а не інтерпретована мова).
  - Для участі в екосистемі Rust потрібно використовувати стандартні інструменти Rust, такі як Cargo. Бібліотекам, які хочуть отримати зовнішні надходження і використовувати їх поза межами Chromium (наприклад, у середовищах збірки Bazel або Android/Soong), ймовірно, варто використовувати Cargo.
- Приклади проектів, пов'язаних з Chromium, які базуються на `cargo`:
  - `serde_json_lenient` (з ним експериментували в інших частинах Google, що призвело до PRs з покращенням продуктивності)
  - Бібліотеки шрифтів на кшталт `font-types`
  - Інструмент `gnrt` (ми познайомимося з ним пізніше у курсі), який залежить від `clap` для розбору командного рядка і від `toml` для конфігураційних файлів.
    * Застереження: єдиною причиною використання `cargo` була недоступність `gn` при збиранні та завантаженні стандартної бібліотеки Rust під час побудови інструментарію Rust.
    * У `run_gnrt.py` використовується копія `cargo` та `rustc` у Chromium. `gnrt` залежить від сторонніх бібліотек, завантажених з інтернету, тому `run_gnrt.py` запитує `cargo` про те, що лише `--locked` контент дозволено через `Cargo.lock`.

Студенти можуть визначити наступні пункти як такі, що викликають у них явну чи неявну довіру:

- `rustc` (компілятор Rust), який, у свою чергу, залежить від бібліотек LLVM, компілятор Clang, вихідні коди `rustc` (отримані з GitHub, переглянуті командою компілятора Rust), бінарний компілятор Rust, завантажений для початкової

обробки
- `rustup` (варто зазначити, що `rustup` розробляється під егідою організації https://github.com/rust-lang/ - так само, як і `rustc`)
- `cargo`, `rustfmt` тощо.
- Різноманітна внутрішня інфраструктура (боти, що збирають `rustc`, система розповсюдження готового інструментарію серед інженерів Chromium тощо).
- Інструменти Cargo, такі як `cargo audit`, `cargo vet`тощо.
- Бібліотеки Rust, що постачаються у `//third_party/rust` (перевірено security@chromium.org)
- Інші бібліотеки Rust (деякі нішеві, деякі досить популярні та часто застосовуються)

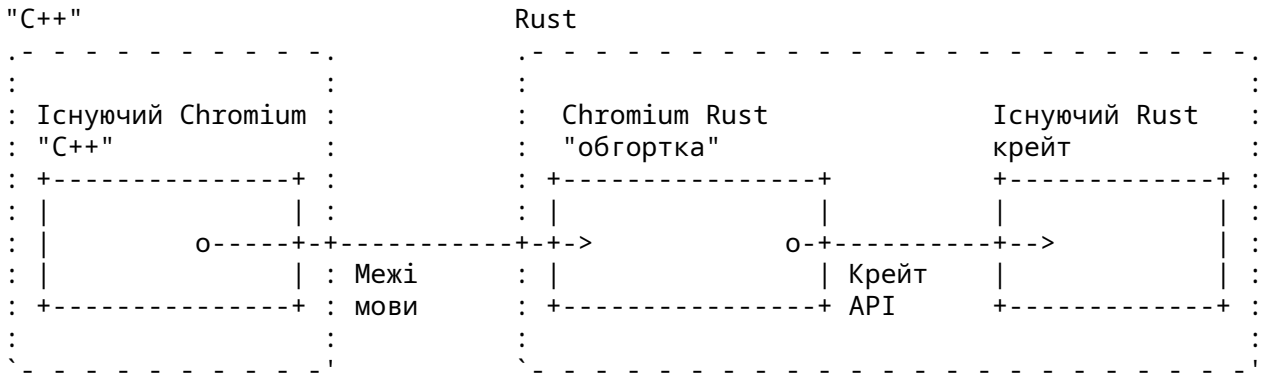
Розділ 41

Політика Chromium щодо Rust

Chromium поки що не дозволяє сторонній Rust, за винятком рідкісних випадків, схвалених Chromium Area Tech Leads.

Політика Chromium щодо сторонніх бібліотек описана тут - Rust дозволяється для сторонніх бібліотек за різних обставин, зокрема, якщо вони є найкращим варіантом для продуктивності або безпеки.

Дуже мало бібліотек Rust безпосередньо надають C/C++ API, а це означає, що майже всі такі бібліотеки потребують невеликої кількості стороннього коду для склеювання.



Код склейки Rust від сторонніх розробників для конкретного стороннього скрипта зазвичай слід зберігати у `third_party/rust/<crate>/<version>/wrapper`.

Через це сьогоднішній курс буде значною мірою сфокусований на:

- Залучення сторонніх бібліотек Rust (”крейтів”)
- Написання коду для використання цих крейтів з Chromium C++.

Якщо ця політика з часом зміниться, курс буде розвиватися, щоб не відставати від неї.

Розділ 42

Правила побудови

Код Rust зазвичай збирається за допомогою `cargo`. Chromium збирає за допомогою `gn` та `ninja` для ефективності --- його статичні правила дозволяють максимальний паралелізм. Rust не є винятком.

Додавання коду Rust до Chromium

У деякому існуючому файлі Chromium `BUILD.gn` оголосіть `rust_static_library`:

```
import("//build/rust/rust_static_library.gni")

rust_static_library("my_rust_lib") {
  crate_root = "lib.rs"
  sources = [ "lib.rs" ]
}
```

Ви також можете додати `deps` на інших цілях Rust. Пізніше ми будемо використовувати це для залежності від стороннього коду.

Ви маєте вказати *одночасно* і корінь крейту, *і* повний список вхідних кодів. `crate_root` - це файл, який передається компілятору Rust, що представляє собою кореневий файл блоку компіляції --- зазвичай це `lib.rs`. `sources` - це повний список усіх вхідних файлів, який потрібен `ninja` для того, щоб визначити, коли потрібна перезбірка.

(У Rust не існує такого поняття, як `source_set`, оскільки у Rust одиницею компіляції є цілий крейт. Найменшою одиницею є `static_library`).

Студентам може бути цікаво, навіщо нам потрібен шаблон gn, а не використання вбудованої підтримки статичних бібліотек Rust у gn. Відповідь полягає у тому, що цей шаблон надає підтримку взаємодії CXX, функцій Rust та модульних тестів, деякі з яких ми використаємо пізніше.

42.1 Включаючи `unsafe` код Rust

Небезпечний Rust-код заборонено у `rust_static_library` за замовчуванням --- він не буде скомпільований. Якщо вам потрібен небезпечний Rust-код, додайте `allow_unsafe`

= true до цілі gn. (Пізніше у курсі ми побачимо обставини, за яких це необхідно).

```
import("//build/rust/rust_static_library.gni")

rust_static_library("my_rust_lib") {
  crate_root = "lib.rs"
  sources = [
    "lib.rs",
    "hippopotamus.rs"
  ]
  allow_unsafe = true
}
```

42.2 Залежність Chromium C++ від коду Rust

Просто додайте наведену вище ціль до deps деякої цілі Chromium C++.

```
import("//build/rust/rust_static_library.gni")

rust_static_library("my_rust_lib") {
  crate_root = "lib.rs"
  sources = [ "lib.rs" ]
}

# or source_set, static_library etc.
component("preexisting_cpp") {
  deps = [ ":my_rust_lib" ]
}
```

42.3 Visual Studio Code

Типи в Rust коді усуваються, що робить хорошу IDE ще більш корисною, ніж для C++. Код Visual Studio добре працює для Rust у Chromium. Щоб скористатися ним,

- Переконайтеся, що ваш VSCode має розширення `rust-analyzer`, а не більш ранні форми підтримки Rust
- `gn gen out/Debug --export-rust-project` (або еквівалент для вашого вихідного каталогу)
- `ln -s out/Debug/rust-project.json rust-project.json`

Демонстрація деяких можливостей rust-analyzer з анотування та дослідження коду може бути корисною, якщо аудиторія скептично ставиться до IDE.

Наступні кроки можуть допомогти з демонстрацією (але не соромтеся використовувати частину Rust, пов'язану з Chromium, з якою ви найбільш знайомі):

- Відкрийте `components/qr_code_generator/qr_code_generator_ffi_glue.rs`
- Наведіть курсор на виклик `QrCode::new` (біля рядка 26) у ‘qr_code_generator_ffi_glue.rs
- Продемонструйте **show documentation** (типові прив'язки: vscode = ctrl k i; vim/CoC = K).
- Продемонструйте **go to definition** (типові прив'язки: vscode = F12; vim/CoC = g d). (Звідси ви потрапите на `//third_party/rust/.../qr_code-.../src/lib.rs`.)

- Продемонструйте **outline** і перейдіть до методу `QrCode::with_bits` (біля рядка 164; контур знаходиться на панелі провідника файлів у vscode; типові прив'язки vim/CoC = space o)
- Продемонструйте **type annotations** (у методі `QrCode::with_bits` наведено декілька гарних прикладів)

Варто зазначити, що команду `gn gen ... --export-rust-project` потрібно буде виконати повторно після редагування файлів `BUILD.gn` (що ми будемо робити кілька разів під час виконання вправ у цій сесії).

42.4 Вправа правил побудови

У вашій збірці Chromium додайте нову ціль Rust до файлу `//ui/base/BUILD.gn`, що містить:

```
// БЕЗПЕКА: Не існує іншої глобальної функції з таким іменем.
pub extern "C" fn hello_from_rust() {
    println!("Привіт від Rust!")
}
```

Важливо: зауважте, що `no_mangle` тут розглядається компілятором Rust як тип небезпеки, тому вам потрібно буде дозволити небезпечний код у вашій цілі gn.

Додайте цю нову ціль Rust як залежність від `//ui/base:base`. Оголосіть цю функцію у верхній частині файлу `ui/base/resource/resource_bundle.cc` (пізніше ми побачимо, як це можна автоматизувати за допомогою інструментів генерації прив'язок):

```
extern "C" void hello_from_rust();
```

Викличте цю функцію звідкись з `ui/base/resource/resource_bundle.cc` - радимо зверху `ResourceBundle::MaybeMangleLocalizedString`. Зберіть і запустіть Chromium, і переконайтеся, що ”Hello from Rust!” виводиться багато разів.

Якщо ви використовуєте VSCode, налаштуйте Rust для роботи у VSCode. Це стане у нагоді у наступних вправах. Якщо вам це вдалося, ви зможете скористатися командою ”Go to definition” правою кнопкою миші на `println!`.

Де знайти допомогу

- Опції, доступні для `rust_static_library`шаблону gn
- Інформація про `#[unsafe(no_mangle)]`
- Інформація про `extern "C"`
- Інформація про перемикач `--export-rust-project` gn
- Як встановити rust-analyzer у VSCode

Цей приклад є незвичайним, тому що він зводиться до мови взаємодії з найменшим спільним знаменником - C. І C++, і Rust можуть оголошувати та викликати функції C ABI на мові C. Пізніше у курсі ми підключимо C++ безпосередньо до Rust.

Тут потрібен `allow_unsafe = true`, оскільки `#[unsafe(no_mangle)]` може дозволити Rust згенерувати дві функції з однаковими іменами, і Rust більше не зможе гарантувати, що буде викликано правильну функцію.

Якщо вам потрібен чистий виконуваний файл Rust, ви також можете зробити це за допомогою шаблону gn `rust_executable`.

Розділ 43

Тестування

Учасники спільноти Rust зазвичай пишуть модульні тести у модулі, розміщеному у тому самому вхідному файлі, що й код, який тестується. Це було розглянуто раніше у курсі і має такий вигляд:

```
mod tests {
    fn my_test() {
        todo!()
    }
}
```

У Chromium ми розміщуємо модульні тести в окремому вхідному файлі і продовжуємо дотримуватися цієї практики для Rust --- це робить тести стабільно доступними для виявлення і допомагає уникнути повторної збірки `.rs`-файлів (у конфігурації `test`).

Це призводить до наступних варіантів тестування Rust-коду в Chromium:

- Нативні тести Rust (тобто `#[test]`). Не рекомендується використовувати поза `//third_party/rust`.
- Тести `gtest`, написані на C++, які виконують Rust за допомогою викликів FFI. Достатньо, коли код Rust є лише тонким прошарком FFI, а наявні модульні тести забезпечують достатнє покриття для функції.
- Тести `gtest`, написані на Rust, що використовують крейт який тестується через його публічний API (з використанням `pub mod for_testing { ... }`, якщо потрібно). Це тема наступних кількох слайдів.

Зауважте, що нативне Rust-тестування сторонніх крейтів має зрештою здійснюватися ботами Chromium. (Таке тестування потрібне рідко --- лише після додавання або оновлення сторонніх крейтів).

Деякі приклади можуть допомогти проілюструвати, коли слід використовувати C++ `gtest` проти Rust `gtest`:

- QR має дуже мало функціональності у сторонньому прошарку Rust (це просто тонкий FFI клей) і тому використовує існуючі модульні тести C++ для тестування як C++, так і Rust-реалізації (параметризуючи тести так, щоб вони вмикали або вимикали Rust за допомогою `ScopedFeatureList`).
- Гіпотетична/WIP інтеграція з PNG може потребувати безпечної реалізації

перетворень пікселів, які надаються `libpng`, але відсутні у крейті png - наприклад, RGBA => BGRA, або гамма-корекція. Така функціональність може отримати вигоду від окремих тестів, написаних у Rust.

43.1 Бібліотека `rust_gtest_interop`

Бібліотека `rust_gtest_interop` надає можливість для:

- Використовувати функцію Rust як тестовий приклад `gtest` (використовуючи атрибут `#[gtest(...)]`)
- Використовувати `expect_eq!` та подібні макроси (подібні до `assert_eq!`, але не панікувати і не завершувати тест, коли твердження не спрацьовує).

Приклад:

```
use rust_gtest_interop::prelude::*;

fn test_addition() {
    expect_eq!(2 + 2, 4);
}
```

43.2 Правила GN для тестів Rust

Найпростіший спосіб створити тести Rust `gtest` - це додати їх до існуючого тестового бінарного файлу, який вже містить тести, написані на C++. Наприклад:

```
test("ui_base_unittests") {
  ...
  sources += [ "my_rust_lib_unittest.rs" ]
  deps += [ ":my_rust_lib" ]
}
```

Створення тестів Rust в окремій `static_library` також працює, але вимагає ручного оголошення залежності від допоміжних бібліотек:

```
rust_static_library("my_rust_lib_unittests") {
  testonly = true
  is_gtest_unittests = true
  crate_root = "my_rust_lib_unittest.rs"
  sources = [ "my_rust_lib_unittest.rs" ]
  deps = [
    ":my_rust_lib",
    "//testing/rust_gtest_interop",
  ]
}

test("ui_base_unittests") {
  ...
  deps += [ ":my_rust_lib_unittests" ]
}
```

43.3 Макрос `chromium::import!`

Після додавання `:my_rust_lib` до GN `deps` нам все ще потрібно навчитися імпортувати та використовувати `my_rust_lib` з `my_rust_lib_unittest.rs`. Ми не надали явного `crate_name` для `my_rust_lib`, тому його ім'я буде обчислено на основі повного шляху та імені. На щастя, ми можемо уникнути роботи з такою громіздкою назвою за допомогою макросу `chromium::import!` з автоматично імпортованого крейту `chromium`:

```
chromium::import! {
    "//ui/base:my_rust_lib";
}

use my_rust_lib::my_function_under_test;
```

Під ковдрою макрос розширюється до чогось схожого на це:

```
extern crate ui_sbase_cmy_urust_ulib as my_rust_lib;

use my_rust_lib::my_function_under_test;
```

Додаткову інформацію можна знайти у коментарі документації макросу `chromium::import`.

Бібліотека `rust_static_library` підтримує вказівку явної назви через властивість `crate_name`, але робити це не рекомендується. Не рекомендується, тому що ім'я крейту має бути глобально унікальним. crates.io гарантує унікальність імен своїх крейтів, тому GN цілі `cargo_crate` (створені за допомогою інструменту `gnrt`, описаного в наступному розділі) використовують короткі імена крейтів.

43.4 Тестова вправа

Час для наступної вправи!

У вашій збірці Chromium:

- Додайте тестову функцію поруч з `hello_from_rust`. Деякі пропозиції: додавання двох цілих чисел, отриманих як аргументи, обчислення n-го числа Фібоначчі, підсумовування цілих чисел у зрізі тощо.
- Додайте окремий файл `..._unittest.rs` з тестом для нової функції.
- Додайте нові тести до `BUILD.gn`.
- Побудуйте тести, запустіть їх і перевірте, чи працює новий тест.

Розділ 44

Взаємодія з C++

Спільнота Rust пропонує кілька варіантів взаємодії C++/Rust, при цьому постійно розробляються нові інструменти. Наразі у Chromium використовується інструмент під назвою CXX.

Ви описуєте всю вашу мовну границю мовою визначення інтерфейсів (яка дуже схожа на Rust), а потім інструменти CXX генерують оголошення для функцій і типів як на Rust, так і на C++.

Перегляньте підручник з CXX, щоб отримати повний приклад використання цього.

Поговоріть про схему. Поясніть, що за лаштунками відбувається те саме, що ви робили раніше. Зазначте, що автоматизація процесу має такі переваги:

- Інструмент гарантує, що сторони C++ та Rust збігаються (наприклад, ви отримаєте помилки компіляції, якщо `#[cxx::bridge]` не збігається з фактичними визначеннями C++ або Rust, а з несинхронізованими ручними прив'язками ви отримаєте Undefined Behavior).
- Інструмент автоматизує генерацію заглушок FFI (невеликих, C-ABI-сумісних, вільних функцій) для не-C функціоналу (наприклад, увімкнення викликів FFI в методи Rust або C++; ручне прив'язування вимагало б написання таких вільних функцій верхнього рівня вручну).
- Інструмент і бібліотека можуть працювати з набором основних типів, наприклад:
  - `&[T]` можна передавати через межу FFI, навіть якщо це не гарантує певного ABI або розміщення пам'яті. При ручному зв'язуванні `std::span<T>` / `&[T]` потрібно вручну деструктурувати і відновити з вказівника і довжини - це може призвести до помилок, оскільки кожна мова представляє порожні зрізи дещо по-різному
  - Розумні вказівники типу `std::unique_ptr<T>`, `std::shared_ptr<T>` та/або `Box` підтримуються за замовчуванням. При ручному прив'язуванні потрібно було б передавати C-ABI-сумісні необроблені вказівники, що збільшило б ризики для тривалості життя та безпеки пам'яті.
  - Типи `rust::String` і `CxxString` розуміють і підтримують відмінності у представленні рядків у різних мовах (наприклад, `rust::String::lossy` може створити рядок Rust із вхідних даних не у форматі UTF8, а `rust::String::c_str` може завершити рядок NUL).

44.1 Приклади прив'язок

CXX вимагає, щоб вся межа C++/Rust була оголошена в модулях `cxx::bridge` у вхідному коді `.rs`.

```
mod ffi {
    extern "Rust" {
        type MultiBuf;

        fn next_chunk(buf: &mut MultiBuf) -> &[u8];
    }

    unsafe extern "C++" {
        include!("example/include/blobstore.h");

        type BlobstoreClient;

        fn new_blobstore_client() -> UniquePtr<BlobstoreClient>;
        fn put(self: &BlobstoreClient, buf: &mut MultiBuf) -> Result<u64>;
    }
}

// Визначення типів та функцій Rust можна знайти тут
```

Вкажіть:

- Хоча це виглядає як звичайний `mod` у Rust, процедурний макрос `#[cxx::bridge]` робить з ним складні речі. Згенерований код є дещо складнішим - хоча це все одно призведе до появи у вашому коді `mod` з назвою `ffi`.
- Вбудована підтримка `std::unique_ptr` з C++ у Rust
- Вбудована підтримка зрізів Rust у C++
- Виклики з C++ на Rust та типи Rust (у верхній частині)
- Виклики з Rust на C++ та типи C++ (у нижній частині)

Поширена помилка: *Виглядає* так, ніби заголовок C++ розбирається Rust'ом, але це оманлива думка. Цей заголовок ніколи не інтерпретується Rust'ом, а просто `#included` у згенерований C++ код на користь компіляторів C++.

Обмеження CXX

Безумовно, найбільш корисною сторінкою при використанні CXX є довідник типів.

CXX принципово підходить для випадків, коли:

- Ваш інтерфейс Rust-C++ достатньо простий, щоб ви могли оголосити все це.
- Ви використовуєте лише типи, які вже підтримуються CXX, наприклад, `std::unique_ptr`, `std::string`, `&[u8]` тощо.

Це має багато обмежень --- наприклад, відсутність підтримки типу `Option` у Rust.

Ці обмеження обмежують нас у використанні Rust у Chromium лише для добре ізольованих "листових вузлів", а не для довільної взаємодії Rust-C++. Розглядаючи варіанти використання Rust у Chromium, гарною відправною точкою є складання проекту прив'язки CXX для мовної межі, щоб побачити, чи виглядає він достатньо простим.

Ви також повинні обговорити деякі інші проблемні моменти з CXX, наприклад:

- Обробка помилок базується на винятках C++ (наведені на наступному слайді)
- Функціональні покажчики незручні у використанні.

44.2 Обробка помилок в CXX

У CXX підтримка `Result<T,E>` покладається на винятки C++, тому ми не можемо використовувати її у Chromium. Альтернативи:

- Частина `T` у `Result<T, E>` може бути:
  - Повернута через вихідні параметри (наприклад, через `&mut T`). Для цього потрібно, щоб `T` можна було передати через межу FFI - наприклад, `T` має бути:
    * Примітивний тип (наприклад, `u32` або `usize`)
    * Тип, що підтримується `cxx` (наприклад, `UniquePtr<T>`), який має відповідне значення за замовчуванням для використання у випадку невдачі (*на відміну* від `Box<T>`).
  - Збережена на стороні Rust та доступна за посиланням. Це може знадобитися, коли `T` є типом Rust, який не може бути переданий через межу FFI і не може бути збережений у `UniquePtr<T>`.
- Частина `E` у `Result<T, E>` може бути:
  - Повернута як булеве значення (наприклад, `true` означає успіх, а `false` - невдачу)
  - Збереження деталей помилок теоретично можливе, але поки що на практиці воно не було потрібне.

44.2.1 Обробка помилок CXX: Приклад з QR

Генератор QR-кодів - це приклад, де булеве значення використовується для передачі інформації про успіх чи невдачу, і де успішний результат може бути переданий через межу FFI:

```
mod ffi {
    extern "Rust" {
        fn generate_qr_code_using_rust(
            data: &[u8],
            min_version: i16,
            out_pixels: Pin<&mut CxxVector<u8>>,
            out_qr_size: &mut usize,
        ) -> bool;
    }
}
```

Студентам може бути цікаво дізнатися про семантику виведення `out_qr_size`. Це не розмір вектора, а розмір QR-коду (і слід визнати, що це трохи зайве - це квадратний корінь з розміру вектора).

Варто звернути увагу на важливість ініціалізації `out_qr_size` перед викликом функції Rust. Створення посилання у Rust, яке вказує на неініціалізовану пам'ять, призводить до Undefined Behavior (на відміну від C++, де лише акт розіменування такої пам'яті призводить до UB).

Якщо студенти запитають про `Pin`, поясніть, навіщо він потрібен CXX для змінних посилань на дані C++: відповідь полягає в тому, що дані C++ не можна переміщати, як дані Rust, оскільки вони можуть містити самопосилальні вказівники.

44.2.2 Обробка помилок CXX: Приклад PNG

Прототип декодера PNG ілюструє, що можна зробити, коли успішний результат не може бути переданий через межу FFI:

```
mod ffi {
    extern "Rust" {
        /// Повертає дружній до FFI еквівалент `Result<PngReader<'a>,
        /// ()>`.
        fn new_png_reader<'a>(input: &'a [u8]) -> Box<ResultOfPngReader<'a>>;

        /// Зв'язування C++ для типу `crate::png::ResultOfPngReader`.
        type ResultOfPngReader<'a>;
        fn is_err(self: &ResultOfPngReader) -> bool;
        fn unwrap_as_mut<'a, 'b>(
            self: &'b mut ResultOfPngReader<'a>,
        ) -> &'b mut PngReader<'a>;

        /// Зв'язування C++ для типу `crate::png::PngReader`.
        type PngReader<'a>;
        fn height(self: &PngReader) -> u32;
        fn width(self: &PngReader) -> u32;
        fn read_rgba8(self: &mut PngReader, output: &mut [u8]) -> bool;
    }
}
```

`PngReader` та `ResultOfPngReader` є типами Rust --- об'єкти цих типів не можуть перетинати межу FFI без опосередкування `Box<T>`. Ми не можемо мати `out_parameter: &mut PngReader`, оскільки CXX не дозволяє C++ зберігати об'єкти Rust за значенням.

Цей приклад ілюструє, що навіть якщо CXX не підтримує довільні узагальнення або шаблони, ми все одно можемо передати їх через межу FFI, вручну спеціалізувавши / мономорфізувавши їх до не узагальненого типу. У прикладі `ResultOfPngReader` є не узагальненим типом, який передається у відповідні методи `Result<T, E>` (наприклад, у `is_err`, `unwrap` та/або `as_mut`).

Використання cxx у Chromium

У Chromium ми визначаємо незалежний `#[cxx::bridge] mod` для кожного листового вузла, де ми хочемо використовувати Rust. Зазвичай у вас буде по одному модулю для кожної `rust_static_library`. Просто додайте

```
cxx_bindings = [ "my_rust_file.rs" ]
   # список файлів, що містять #[cxx::bridge], не всі вхідні файли
allow_unsafe = true
```

до вашої існуючої цілі `rust_static_ibrary` разом з `crate_root` та `sources`.

C++ заголовки будуть згенеровані в доцільному місці, тому ви можете просто

```
#include "ui/base/my_rust_file.rs.h"
```

У `//base` ви знайдете деякі утиліти для перетворення типів Chromium C++ у типи CXX Rust --- наприклад SpanToRustSlice.

Студенти можуть запитати --- навіщо нам все ще потрібно `allow_unsafe = true`?

Загальна відповідь полягає у тому, що жоден C/C++ код не є "безпечним" за звичайними стандартами Rust. Виклик C/C++ з Rust може призвести до довільних дій з пам'яттю та поставити під загрозу безпеку власних структур даних Rust. Наявність *занадто* `unsafe` ключових слів у взаємодії C/C++ може погіршити співвідношення сигнал/шум такого ключового слова, що є суперечливим, але строго кажучи, внесення будь-якого стороннього коду у бінарний файл Rust може спричинити неочікувану поведінку з точки зору Rust'у.

Вузька відповідь міститься на діаграмі у верхній частині цієї сторінки --- за завісою CXX генерує `unsafe` та `extern "C"` функції Rust так само, як ми робили це вручну у попередньому розділі.

44.3 Вправа: Інтероперабельність з C++

Частина перша

- У створений раніше файл Rust додайте `#[cxx::bridge]`, який визначає єдину функцію для виклику з C++ під назвою `hello_from_rust`, яка не отримує параметрів і не повертає жодного значення.
- Змініть вашу попередню функцію `hello_from_rust`, видаливши `extern "C"` і `#[unsafe(no_mangle)]`. Тепер це просто стандартна функція Rust.
- Змініть вашу ціль gn, щоб створити ці прив'язки.
- У вашому C++ коді видаліть форвардне оголошення `hello_from_rust`. Замість цього додайте згенерований заголовний файл.
- Будуємо і запускаємо!

Частина друга

Це гарна ідея - трохи погратися з CXX. Це допоможе вам зрозуміти, наскільки гнучким є Rust у Chromium.

Декілька речей, які варто спробувати:

- Зворотний виклик у C++ з Rust. Вам знадобиться:
  - Додатковий заголовний файл, який ви можете `include!` до вашого `cxx::bridge`. Вам потрібно буде оголосити вашу функцію C++ у цьому новому заголовному файлі.
  - `unsafe` блок для виклику такої функції, або вкажіть ключове слово `unsafe` у вашому `#[cxx::bridge]` як описано тут.
  - Вам також може знадобитися `#include "third_party/rust/cxx/v1/crate/include/cxx.h"`
- Передати рядок C++ з C++ у Rust.
- Передати в Rust посилання на об'єкт C++.
- Навмисно зробити так, щоб сигнатури функцій Rust не співпадали з `#[cxx::bridge]`, і звикати до помилок, які ви побачите.
- Навмисно зробити так, щоб сигнатури функцій C++ не співпадали з `#[cxx::bridge]`, і звикати до помилок, які ви побачите.

- Передати `std::unique_ptr` деякого типу з C++ у Rust, щоб Rust міг володіти деяким об'єктом C++.
- Створити об'єкт Rust і передати його в C++ так, щоб C++ володів ним. (Підказка: вам потрібен `Box`).
- Оголосити деякі методи на типі C++. Викликати їх з Rust.
- Оголосити декілька методів на типі Rust. Викликати їх з C++.

Частина третя

Тепер, коли ви розумієте сильні та слабкі сторони взаємодії CXX, подумайте про пару варіантів використання Rust у Chromium, де інтерфейс був би достатньо простим. Накидайте ескіз того, як ви могли б визначити цей інтерфейс.

Де знайти допомогу

- Довідник `cxx` зв'язувань
- `rust_static_library` шаблон gn

Ви можете зіткнутися з деякими питаннями:

- Я бачу проблему з ініціалізацією змінної типу X типом Y, де X і Y є типами функцій. Це пов'язано з тим, що ваша функція C++ не зовсім відповідає оголошенню у вашому `cxx::bridge`.
- Здається, я можу вільно конвертувати посилання на C++ у посилання на Rust. Чи не загрожує це UB? Для *непрозорих* типів CXX - ні, тому що вони мають нульовий розмір. Для тривіальних типів CXX так, це *можливо* спричинити UB, хоча дизайн CXX робить досить складним створення такого прикладу.

Розділ 45

Додавання крейтів третіх сторін

Бібліотеки Rust називаються ”крейтами” і знаходяться на crates.io. Для крейтів Rust дуже легко залежати один від одного. Так вони і роблять!

Власивість	Бібліотека C++	Крейт Rust
Система збірки	Багато	Послідовна: `Cargo.toml`
Типовий розмір бібліотеки	Великий	Маленький
Транзитивні залежності	Небагато	Багато

Для інженера Chromium це має плюси та мінуси:

- Всі крейти використовують спільну систему збірки, тому ми можемо автоматизувати їхнє включення до Chromium ...
- ... але, як правило, крейти мають транзитивні залежності, тому вам, ймовірно, доведеться залучити декілька бібліотек.

Ми обговоримо:

- Як розмістити крейт у дереві вхідного коду Chromium
- Як зробити так, щоб gn будував правила для нього
- Як провести аудит його вхідного коду на предмет достатньої безпеки.

45.1 Налаштування файлу `Cargo.toml` для додавання крейтів

Chromium має єдиний набір централізовано керованих прямих залежностей крейтів. Вони управляються через єдиний `Cargo.toml`:

```
[dependencies]
bitflags = "1"
cfg-if = "1"
cxx = "1"
# lots more...
```

Як і для будь-якого іншого `Cargo.toml`, ви можете вказати більш детальну інформацію про залежності --- найчастіше, вам потрібно вказати `features`, які ви хочете увімкнути в крейті.

При додаванні крейту до Chromium вам часто потрібно надати додаткову інформацію у додатковому файлі `gnrt_config.tml`, з яким ми познайомимося далі.

45.2 Налаштування `gnrt_config.toml`

Поряд з `Cargo.toml` знаходиться `gnrt_config.toml`. Він містить специфічні для Chromium розширення для роботи з крейтами.

Якщо ви додаєте новий крейт, ви повинні вказати принаймні `group`. Це одна з них:

```
#   'safe': The library satisfies the rule-of-2 and can be used in any process.
#   'sandbox': The library does not satisfy the rule-of-2 and must be used in
#              a sandboxed process such as the renderer or a utility process.
#   'test': The library is only used in tests.
```

Наприклад,

```
[crate.my-new-crate]
group = 'test' # only used in test code
```

Залежно від компонування вхідного коду крейту, вам також може знадобитися використовувати цей файл, щоб вказати, де можна знайти його файл(и) `LICENSE`.

Пізніше ми розглянемо деякі інші речі, які вам потрібно буде налаштувати в цьому файлі для вирішення проблем.

45.3 Завантаження крейтів

Інструмент під назвою `gnrt` знає, як завантажувати крейти і як генерувати правила `BUILD.gn`.

Для початку завантажте потрібний вам крейт ось так:

```
cd chromium/src
vpython3 tools/crates/run_gnrt.py -- vendor
```

> Хоча інструмент `gnrt` є частиною вхідного коду Chromium, виконуючи цю команду, ви завантажите і запустите його залежності з `crates.io`. Дивіться попередній розділ, де описано це рішення з безпеки.

Ця команда `vendor` може завантажити:

- Ваш крейт
- Прямі та транзитивні залежності
- Нові версії інших крейтів, які вимагаються `cargo` для встановлення повного набору крейтів, необхідних для Chromium.

Chromium підтримує патчі для деяких крейтів, які зберігаються у `//third_party/rust/chromium_crates`. Їх буде повторно застосовано автоматично, але якщо виправлення не вдасться, вам може знадобитися вжити заходів вручну.

45.4 Створення правил побудови gn

Після того, як ви завантажили крейт, згенеруйте файли `BUILD.gn`, як показано нижче:

```
vpython3 tools/crates/run_gnrt.py -- gen
```

Тепер запустіть `git status`. Ви повинні знайти:

- Щонайменше один новий вхідний код скриньки у `third_party/rust/chromium_crates_io/vendor`
- Щонайменше один новий `BUILD.gn` у `third_party/rust/<crate name>/v<major semver version>`
- Відповідний `README.chromium`

Тут "major semver version" - це номер версії "semver" Rust .

Уважно подивіться, особливо на те, що генерується в `third_party/rust`.

Поговоримо трохи про семантичну версифікацію (semver) --- і, зокрема, про те, як у Chromium вона дозволяє створювати кілька несумісних версій крейту, що не рекомендується, але іноді необхідно в екосистемі Cargo.

45.5 Вирішення проблем

Якщо ваша збірка не вдається, це може бути пов'язано з `build.rs`: програмами, які виконують довільні дії під час збирання. Це принципово суперечить принципам роботи gn та `ninja`, які передбачають статичні, детерміновані правила збирання для максимізації паралелізму та повторюваності збірок.

Деякі дії `build.rs` підтримуються автоматично, інші потребують втручання:

ефект скрипту збірки	Підтримується нашими шаблонами gn	Робота, яка потрібна від вас
Перевірка версії rustc для ввімкнення та вимкнення можливостей	Так	Нічого
Перевірка платформи або процесора для ввімкнення та вимкнення можливостей	Так	Нічого
Генерація коду	Так	Так - вкажіть у файлі `gnrt_config.toml`
Збірка C/C++	Немає	Залатати навколо
Довільні інші дії	Немає	Залатати навколо

На щастя, більшість крейтів не містять скриптів збірки, і, на щастя, більшість скриптів збірки виконують лише перші дві дії.

45.5.1 Скрипти збірки, які генерують код

Якщо `ninja` скаржиться на відсутність файлів, перевірте `build.rs`, чи пише він файли вхідного коду.

Якщо так, змініть `gnrt_config.toml`, щоб додати `build-script-outputs` до сховища. Якщо це транзитивна залежність, тобто така, від якої код Chromium не повинен безпосередньо залежати, також додайте `allow-first-party-usage=false`. У цьому файлі вже є кілька прикладів:

```
[crate.unicode-linebreak]
allow-first-party-usage = false
build-script-outputs = ["tables.rs"]
```

Тепер повторно запустіть `gnrt.py -- gen` для регенерації файлів `BUILD.gn`, щоб повідомити ninja , що саме цей вихідний файл буде використано на наступних кроках збірки.

45.5.2 Скрипти збірки, які будують C++ або виконують довільні дії

Деякі крейти використовують крейт `cc` для збірки та компонування бібліотек C/C++. Інші крейти розбирають C/C++ за допомогою `bindgen` у своїх скриптах збірки. Ці дії не підтримуються у контексті Chromium --- наша система збірки gn, ninja та LLVM дуже специфічна у вираженні взаємозв'язків між діями збірки.

Отже, у вас є наступні варіанти:

- Уникайте цих крейтів
- Накладіть патч на крейт.

Патчі слід зберігати у `third_party/rust/chromium_crates_io/patches/<crate>` - дивіться, наприклад, патчі на крейтисхх - і вони будуть автоматично застосовуватися `gnrt` під час кожного оновлення крейту.

45.6 Залежнісь від крейта

Після того, як ви додали сторонній крейт і згенерували правила збірки, залежність від крейту є простою. Знайдіть ціль `rust_static_library` і додайте dep до цілі `:lib` у вашій збірці.

А саме,

```
                     +------------+      +----------------------+
"//third_party/rust" | crate name | "/v" | major semver version | ":lib"
                     +------------+      +----------------------+
```

Наприклад,

```
rust_static_library("my_rust_lib") {
  crate_root = "lib.rs"
  sources = [ "lib.rs" ]
  deps = [ "//third_party/rust/example_rust_crate/v1:lib" ]
}
```

45.7 Аудит сторонніх крейтів

Додавання нових бібліотек підпорядковується стандартним політикам Chromium, але, звісно, також підлягає перевірці безпеки. Оскільки ви можете додати не лише один крейт, але й транзитивні залежності, то може бути багато коду для перевірки. З іншого боку, безпечний код Rust може мати обмежені негативні побічні ефекти. Як ви повинні його перевіряти?

З часом Chromium планує перейти на процес, заснований навколо cargo vet.

Тим часом, для кожного нового доданого крейту ми перевіряємо наступне:

- Зрозуміти, для чого використовується кожен крейт. Який взаємозв'язок між крейтами? Якщо система збірки для кожного крейту містить `build.rs` або процедурні макроси, з'ясуйвати, для чого вони призначені. Чи сумісні вони зі звичайним способом збирання Chromium?
- Перевірити, щоб кожен крейт був достатньо добре доглянутий
- За допомогою `cd third-party/rust/chromium_crates_io; cargo audit` перевірити наявність відомих уразливостей (спочатку потрібно `cargo install cargo-audit`, що за іронією долі передбачає завантаження великої кількості залежностей з інтернету2).
- Переконатися, що будь-який `unsafe` код достатньо підходить для Правила двох.
- Перевірити, чи не використовуються API `fs` або `net`
- Прочитати весь код на достатньому рівні, щоб знайти все, що могло бути вставлено зловмисниками. (Ви не можете реально прагнути до 100% досконалості тут: часто коду просто занадто багато).

Це лише рекомендації - попрацюйте з рецензентами з `security@chromium.org`, щоб виробити правильний спосіб отримати впевненість в крейті.

45.8 Включення крейтів у вхідний код Chromium

`git status` повинен показати:

- Код крейту в `//third_party/rust/chromium_crates_io`
- Метадані (`BUILD.gn` та `README.chromium`) у `//third_party/rust/<crate>/<version>`

Будь ласка, додайте також файл `OWNERS` в останнє місце.

Все це, разом зі змінами в файлах `Cargo.toml` і `gnrt_config.toml`, слід завантажити в репозиторій Chromium.

Важливо: ви повинні використовувати `git add -f`, оскільки інакше файли `.gitignore` можуть бути пропущені.

У процесі цього ви можете виявити, що перевірка перед відправкою не спрацьовує через неінклюзивну термінологію. Це пов'язано з тим, що дані крейту Rust, як правило, містять назви гілок git'а, а у багатьох проектах все ще використовується неінклюзивна термінологія. Тож, можливо, вам доведеться запустити:

```
infra/update_inclusive_language_presubmit_exempt_dirs.sh > infra/inclusive_language_pres
git add -p infra/inclusive_language_presubmit_exempt_dirs.txt # add whatever changes are
```

45.9 Підтримання крейтів в актуальному стані

Як ВЛАСНИК будь-якої сторонньої залежності від Chromium, ви маєте підтримувати її в актуальному стані з будь-якими виправленнями безпеки. Сподіваємося, що незабаром ми автоматизуємо цю процедуру для крейтів Rust, але наразі ви все ще несете відповідальність за це, як і за будь-яку іншу сторонню залежність.

45.10 Вправа

Додайте uwuify до Chromium, вимкнувши можливості за замовчуванням. Передбачається, що крейт буде використовуватися при постачанні Chromium, але не буде використовуватися для обробки ненадійних вхідних даних.

(У наступній вправі ми будемо використовувати uwuify з Chromium, але ви можете зробити це прямо зараз, якщо хочете. Або ви можете створити нову ціль `rust_executable`, яка використовує `uwuify`).

Студентам потрібно буде завантажити багато транзитивних залежностей.

Загальна кількість необхідних крейтів:

- `instant`,
- `lock_api`,
- `parking_lot`,
- `parking_lot_core`,
- `redox_syscall`,
- `scopeguard`,
- `smallvec`, та
- uwuify‘.

Якщо студенти завантажують ще більше, то, ймовірно, вони забули вимкнути можливості за замовчуванням.

Дякуємо Daniel Liu за цей крейт!

Розділ 46

Збираємо все докупи --- Вправа

У цій вправі ви спробуєте додати абсолютно нову функцію Chromium, об'єднавши все, що ви вже вивчили.

Коротка доповідь від продуктового менеджменту

У віддаленому тропічному лісі виявили спільноту ельфів, які живуть там. Важливо, щоб ми доставили їм Chromium for Pixies якнайшвидше.

Вимога полягає в тому, щоб перекласти всі рядки інтерфейсу користувача Chromium на мову ельфів.

Немає часу чекати на нормальний переклад, але, на щастя, мова ельфів дуже близька до англійської, і, виявляється, є крейт Rust, яка робить переклад.

Насправді, ви вже імпортували цей крейт у попередній вправі.

(Очевидно, що справжні переклади для Chrome вимагають неймовірної ретельності та старанності. Не публікуйте це!)

Кроки

Змініть `ResourceBundle::MaybeMangleLocalizedString` так, щоб він використовував uwuify для усіх рядків перед відображенням. У цій спеціальній збірці Chromium він має робити це завжди, незалежно від значення параметра `mangle_localized_strings_`.

Якщо ви зробили все правильно у всіх цих вправах, вітаємо, вам варто було створити Chrome для ельфів!

- UTF16 vs UTF8. Студенти повинні знати, що рядки Rust завжди мають кодування UTF8, і, ймовірно, вирішать, що краще зробити перетворення на стороні C++ за допомогою `base::UTF16ToUTF8` і навпаки.
- Якщо студенти вирішать виконати перетворення на стороні Rust, їм потрібно буде розглянути `String::from_utf16`, обміркувати обробку помилок і визначити, які CXX-підтримувані типи можуть передавати багато u16s.

- Студенти можуть створити межу між C++ і Rust кількома різними способами, наприклад, приймати і повертати рядки за значенням, або приймати мутабельне посилання на рядок. Якщо використовується мутабільне посилання, CXX, ймовірно, скаже студенту, що потрібно використовувати `Pin`. Можливо, вам доведеться пояснити, що робить `Pin`, а потім пояснити, навіщо він потрібен CXX для мутабельних посилань на дані C++: відповідь полягає у тому, що дані C++ не можна переміщувати, як дані Rust, оскільки вони можуть містити самопосилальні вказівники.
- Ціль C++, що містить `ResourceBundle::MaybeMangleLocalizedString`, повинна залежати від цілі `rust_static_biblioteka`. Студенти, ймовірно, вже зробили це.
- Ціль `rust_static_library` має залежати від `//third_party/rust/uwuify/v0_2:lib`.

Розділ 47

Рішення вправ

Рішення вправ з Chromium можна знайти в цій серії CLs.

Частина XI

Залізо: Ранок

Розділ 48

Ласкаво просимо до Rust на голому залізі

Це окремий одноденний курс про Rust на голому залізі, призначений для людей, які знайомі з основами Rust (можливо, після завершення комплексного курсу Rust), а в ідеалі також мають певний досвід програмування на голому залізі якоюсь іншою мовою, такою як C.

Сьогодні ми поговоримо про Rust на 'голому залізі': запуск коду Rust без операційної системи під нами. Цей розділ буде розділено на кілька частин:

- Що таке `no_std` Rust?
- Написання мікропрограм для мікроконтролерів.
- Написання коду завантажувача/ядра для прикладних процесорів.
- Кілька корисних крейтів для розробки Rust на голому залізі.

Для частини курсу, присвяченої мікроконтролеру, ми використаємо BBC micro:bit v2 як приклад. Це плата розробки на основі мікроконтролера Nordic nRF52833 із деякими світлодіодами та кнопками, акселерометром і компасом, підключеними до I2C, і вбудованим налагоджувачем SWD.

Для початку встановіть деякі інструменти, які нам знадобляться пізніше. У gLinux або Debian:

```
sudo apt install gdb-multiarch libudev-dev picocom pkg-config qemu-system-arm
rustup update
rustup target add aarch64-unknown-none thumbv7em-none-eabihf
rustup component add llvm-tools-preview
cargo install cargo-binutils
curl --proto '=https' --tlsv1.2 -LsSf https://github.com/probe-rs/probe-rs/releases/late
```

І надайте користувачам у групі `plugdev` доступ до програматора micro:bit:

```
echo 'SUBSYSTEM=="hidraw", ATTRS{idVendor}=="0d28", MODE="0660", GROUP="logindev", TAG+=
  sudo tee /etc/udev/rules.d/50-microbit.rules
sudo udevadm control --reload-rules
```

У MacOS:

```
xcode-select --install
brew install gdb picocom qemu
rustup update
rustup target add aarch64-unknown-none thumbv7em-none-eabihf
rustup component add llvm-tools-preview
cargo install cargo-binutils
curl --proto '=https' --tlsv1.2 -LsSf https://github.com/probe-rs/probe-rs/releases/late
```

Розділ 49

`no_std`

`core`

`alloc`

`std`

- Зрізи, `&str`, `CStr`
- `NonZeroU8`...
- `Option`, `Result`
- `Display`, `Debug`, `write!`...
- `Ітератор`
- `Error`
- `panic!`, `assert_eq!`...
- `NonNull` і всі звичайні функції, пов'язані з покажчиками
- `Future` та `async`/`await`
- `fence`, `AtomicBool`, `AtomicPtr`, `AtomicU32`...
- `Duration`

- `Box`, `Cow`, `Arc`, `Rc`
- `Vec`, `BinaryHeap`, `BtreeMap`, `LinkedList`, `VecDeque`
- `String`, `CString`, `format!`

- `HashMap`
- `Mutex`, `Condvar`, `Barrier`, `Once`, `RwLock`, `mpsc`
- `File` та решта `fs`
- `println!`, `Read`, `Write`, `Stdin`, `Stdout` та решта `io`
- `Path`, `OsString`
- `net`
- `Command`, `Child`, `ExitCode`
- `spawn`, `sleep` та решта `thread`
- `SystemTime`, `Instant`

- `HashMap` залежить від RNG.
- `std` повторно експортує вміст як `core`, так і `alloc`.

49.1 Мінімальна програма no_std

```
use core::panic::PanicInfo;

fn panic(_panic: &PanicInfo) -> ! {
    loop {}
}
```

- Це буде скомпільовано в порожній бінарний файл.
- `std` надає обробник паніки; без нього ми повинні створити свій власний.
- Це також може бути забезпечено іншим крейтом, таким як `panic-halt`.
- Залежно від цілі, вам може знадобитися скомпілювати за допомогою `panic = "abort"`, щоб уникнути помилки щодо `eh_personality`.
- Зверніть увагу, що не існує `main` або будь-якої іншої точки входу; ви самі визначаєте свою точку входу. Зазвичай це може бути скрипт компонувальника та деякий код збірки, щоб підготувати все до запуску коду Rust.

49.2 alloc

Щоб використовувати `alloc`, ви повинні реалізувати глобальний розподільник (кучі).

```
extern crate alloc;
extern crate panic_halt as _;

use alloc::string::ToString;
use alloc::vec::Vec;
use buddy_system_allocator::LockedHeap;

static HEAP_ALLOCATOR: LockedHeap<32> = LockedHeap::<32>::new();

static mut HEAP: [u8; 65536] = [0; 65536];

pub fn entry() {
    // БЕЗПЕКА: `HEAP` використовується тільки тут і `entry` викликається тільки один ра
    unsafe {
        // Дати розподільнику трохи пам'яті для виділення.
        HEAP_ALLOCATOR.lock().init(HEAP.as_mut_ptr() as usize, HEAP.len());
    }

    // Тепер ми можемо робити речі, які вимагають виділення кучі.
    let mut v = Vec::new();
    v.push("Рядок".to_string());
}
```

- `buddy_system_allocator` — це сторонній крейт, який реалізує базовий системний розподільник між друзями. Доступні інші крейти, або ви можете написати свій власний або підключити до наявного розподільника.
- Параметр const у `LockedHeap` - це максимальний порядок розподільника, тобто у цьому випадку він може виділяти області розміром до 2**32 байт.

- Якщо будь-який крейт у вашому дереві залежностей залежить від `alloc`, тоді ви повинні мати точно один глобальний розподільник, визначений у вашому бінарному файлі. Зазвичай це робиться у бінарному крейті верхнього рівня.
- `extern crate panic_halt as _` необхідний для того, щоб переконатися, що буде зв'язано крейт `panic_halt` і ми отримаємо його обробник паніки.
- Цей приклад збиратиметься, але не запускатиметься, оскільки він не має точки входу.

Розділ 50

Мікроконтролери

Крейт `cortex_m_rt` містить (серед іншого) обробник скидання для мікроконтролерів Cortex M.

```
extern crate panic_halt as _;

mod interrupts;

use cortex_m_rt::entry;

fn main() -> ! {
    loop {}
}
```

Далі ми розглянемо, як отримати доступ до периферійних пристроїв із підвищенням рівня абстракції.

- Макрос `cortex_m_rt::entry` вимагає, щоб функція мала тип `fn() -> !`, оскільки повернення до обробника скидання не має сенсу.
- Запустіть приклад із `cargo embed --bin minimal`

50.1 Сирий ввід вивід з відображеної пам'яті (MMIO)

Більшість мікроконтролерів отримують доступ до периферійних пристроїв через відображений в пам’ять IO. Давайте спробуємо включити світлодіод на нашому micro:bit:

```
extern crate panic_halt as _;

mod interrupts;

use core::mem::size_of;
use cortex_m_rt::entry;
```

```rust
/// Периферійна адреса порту GPIO 0
const GPIO_P0: usize = 0x5000_0000;

// Зміщення периферії GPIO
const PIN_CNF: usize = 0x700;
const OUTSET: usize = 0x508;
const OUTCLR: usize = 0x50c;

// Поля PIN_CNF
const DIR_OUTPUT: u32 = 0x1;
const INPUT_DISCONNECT: u32 = 0x1 << 1;
const PULL_DISABLED: u32 = 0x0 << 2;
const DRIVE_S0S1: u32 = 0x0 << 8;
const SENSE_DISABLED: u32 = 0x0 << 16;

fn main() -> ! {
    // Налаштуйте виводи GPIO 0 21 та 28 як push-pull виводи.
    let pin_cnf_21 = (GPIO_P0 + PIN_CNF + 21 * size_of::<u32>()) as *mut u32;
    let pin_cnf_28 = (GPIO_P0 + PIN_CNF + 28 * size_of::<u32>()) as *mut u32;
    // БЕЗПЕКА: вказівники вказують на дійсні периферійні регістри
    // керування, і ніяких псевдонімів не існує.
    unsafe {
        pin_cnf_21.write_volatile(
            DIR_OUTPUT
                | INPUT_DISCONNECT
                | PULL_DISABLED
                | DRIVE_S0S1
                | SENSE_DISABLED,
        );
        pin_cnf_28.write_volatile(
            DIR_OUTPUT
                | INPUT_DISCONNECT
                | PULL_DISABLED
                | DRIVE_S0S1
                | SENSE_DISABLED,
        );
    }

    // Встановіть низький рівень на виводі 28 і високий на виводі 21, щоб увімкнути сві
    let gpio0_outset = (GPIO_P0 + OUTSET) as *mut u32;
    let gpio0_outclr = (GPIO_P0 + OUTCLR) as *mut u32;
    // БЕЗПЕКА: вказівники вказують на дійсні периферійні регістри
    // керування, і ніяких псевдонімів не існує.
    unsafe {
        gpio0_outclr.write_volatile(1 << 28);
        gpio0_outset.write_volatile(1 << 21);
    }

    loop {}
}
```

- Вивід 21 GPIO 0 підключений до першого стовпчика світлодіодної матриці, а вивід 28 – до першого рядка.

Запустіть приклад за допомогою:

```
cargo embed --bin mmio
```

50.2 Крейти периферійного доступу

svd2rust створює здебільшого безпечні оболонки Rust для периферійних пристроїв із відображенням пам'яті з CMSIS-SVD файлів.

```
extern crate panic_halt as _;

use cortex_m_rt::entry;
use nrf52833_pac::Peripherals;

fn main() -> ! {
    let p = Peripherals::take().unwrap();
    let gpio0 = p.P0;

    // Налаштуйте виводи GPIO 0 21 та 28 як push-pull виводи.
    gpio0.pin_cnf[21].write(|w| {
        w.dir().output();
        w.input().disconnect();
        w.pull().disabled();
        w.drive().s0s1();
        w.sense().disabled();
        w
    });
    gpio0.pin_cnf[28].write(|w| {
        w.dir().output();
        w.input().disconnect();
        w.pull().disabled();
        w.drive().s0s1();
        w.sense().disabled();
        w
    });

    // Встановіть низький рівень на виводі 28 і високий на виводі 21, щоб увімкнути сві
    gpio0.outclr.write(|w| w.pin28().clear());
    gpio0.outset.write(|w| w.pin21().set());

    loop {}
}
```

- Файли SVD (System View Description) — це XML-файли, які зазвичай надають постачальники кремнію, які описують карту пам'яті пристрою.
  - Вони організовані за периферією, регістром, полем і значенням, з назвами, описами, адресами тощо.

  - Файли SVD часто є помилковими та неповними, тому існують різні проекти, які виправляють помилки, додають відсутні деталі та публікують згенеровані крейти.
- `cortex-m-rt` надає векторну таблицю, серед іншого.
- Якщо ви `cargo install cargo-binutils`, ви можете запустити `cargo objdump --bin pac -- -d --no-show-raw-insn`, щоб побачити результуючий бінарний файл.

Запустіть приклад за допомогою:

```
cargo embed --bin pac
```

50.3 Крейти HAL

Крейти HAL для багатьох мікроконтролерів забезпечують оболонки для різних периферійних пристроїв. Зазвичай вони реалізують трейти з `embedded-hal`.

```
extern crate panic_halt as _;

use cortex_m_rt::entry;
use embedded_hal::digital::OutputPin;
use nrf52833_hal::gpio::{p0, Level};
use nrf52833_hal::pac::Peripherals;

fn main() -> ! {
    let p = Peripherals::take().unwrap();

    // Створити HAL-обгортку для порту GPIO 0
    let gpio0 = p0::Parts::new(p.P0);

    // Налаштуйте виводи GPIO 0 21 та 28 як push-pull виводи.
    let mut col1 = gpio0.p0_28.into_push_pull_output(Level::High);
    let mut row1 = gpio0.p0_21.into_push_pull_output(Level::Low);

    // Встановіть низький рівень на виводі 28 і високий на виводі 21, щоб увімкнути сві
    col1.set_low().unwrap();
    row1.set_high().unwrap();

    loop {}
}
```

- `set_low` і `set_high` — це методи трейту `embedded_hal` `OutputPin`.
- Існують крейти HAL для багатьох пристроїв Cortex-M і RISC-V, включаючи різні мікроконтролери STM32, GD32, nRF, NXP, MSP430, AVR і PIC.

Запустіть приклад за допомогою:

```
cargo embed --bin hal
```

50.4 Крейти для підтримки плат

Крейти для підтримки плат забезпечують додатковий рівень обгортання для конкретної дошки для зручності.

```
extern crate panic_halt as _;

use cortex_m_rt::entry;
use embedded_hal::digital::OutputPin;
use microbit::Board;

fn main() -> ! {
    let mut board = Board::take().unwrap();

    board.display_pins.col1.set_low().unwrap();
    board.display_pins.row1.set_high().unwrap();

    loop {}
}
```

- У цьому випадку крейти для підтримки плати просто надає корисніші назви та трохи ініціалізації.
- Крейт також може містити драйвери для деяких вбудованих пристроїв за межами самого мікроконтролера.
  - `microbit-v2` містить простий драйвер для світлодіодної матриці.

Запустіть приклад за допомогою:

```
cargo embed --bin board_support
```

50.5 Шаблон стану типу

```
fn main() -> ! {
    let p = Peripherals::take().unwrap();
    let gpio0 = p0::Parts::new(p.P0);

    let pin: P0_01<Disconnected> = gpio0.p0_01;

    // let gpio0_01_again = gpio0.p0_01; // Помилка, переміщено.
    let mut pin_input: P0_01<Input<Floating>> = pin.into_floating_input();
    if pin_input.is_high().unwrap() {
        // ...
    }
    let mut pin_output: P0_01<Output<OpenDrain>> = pin_input
        .into_open_drain_output(OpenDrainConfig::Disconnect0Standard1, Level::Low);
    pin_output.set_high().unwrap();
    // pin_input.is_high(); // Помилка, переміщено.

    let _pin2: P0_02<Output<OpenDrain>> = gpio0
        .p0_02
        .into_open_drain_output(OpenDrainConfig::Disconnect0Standard1, Level::Low);
```

```
    let _pin3: P0_03<Output<PushPull>> =
        gpio0.p0_03.into_push_pull_output(Level::Low);

    loop {}
}
```

- Піни не реалізують `Copy` або `Clone`, тому може існувати лише один екземпляр кожного з них. Після того, як пін буде переміщено зі структури порту, ніхто інший не зможе його взяти.
- Зміна конфігурації піна поглинає старий екземпляр піна, тому ви не можете продовжувати використовувати старий екземпляр після цього.
- Тип значення вказує на стан, у якому воно перебуває: наприклад, у цьому випадку це стан конфігурації піна GPIO. Це кодує машину станів у систему типів і гарантує, що ви не спробуєте використати пін певним чином, не налаштувавши його належним чином. Незаконні переходи станів перехоплюються під час компіляції.
- Ви можете викликати `is_high` на вхідному піні та `set_high` на вихідному піні, але не навпаки.
- Багато крейтів HAL дотримуються цієї моделі.

50.6 `embedded-hal`

Крейт `embedded-hal` надає низку трейтів, що охоплюють поширені периферійні пристрої мікроконтролерів:

- GPIO
- PWM
- Таймери затримки
- Шини та пристрої I2C і SPI

Аналогічні трейти для байтових потоків (наприклад, UART), CAN-шини та ГВЧ (RNGs) і розбиті на `embedded-io`, `embedded-can` та `rand_core`, відповідно.

Інші крейти потім реалізують драйвери у термінах цих трейтів, наприклад драйверу акселерометра може знадобитися кземпляр пристрою I2C або SPI.

- Трейти охоплюють використання периферійних пристроїв, але не їх ініціалізацію чи конфігурацію, оскільки ініціалізація та конфігурація, як правило, сильно залежить від платформи.
- Існують реалізації для багатьох мікроконтролерів, а також інших платформ, таких як Linux на Raspberry Pi.
- Крейт `embedded-hal-async` надає асинхронні версії трейтів.
- Крейт `embedded-hal-nb` надає інший підхід до неблокуючого вводу/виводу, заснований на крейті `nb`.

50.7 `probe-rs` та `cargo-embed`

probe-rs — це зручний набір інструментів для вбудованого налагодження, як OpenOCD, але краще інтегрований.

- SWD (Serial Wire Debug) і JTAG через CMSIS-DAP, ST-Link і J-Link зонди
- GDB заглушка та сервер Microsoft DAP (Debug Adapter Protocol)
- Інтеграція Cargo

`cargo-embed` - це підкоманда cargo для збирання та прошивання двійкових файлів, ведення журналу RTT (Real Time Transfers) та підключення GDB. Вона налаштовується за допомогою файлу `Embed.toml` у каталозі вашого проекту.

- CMSIS-DAP - це стандартний протокол Arm через USB для внутрішньосхемного налагоджувача для доступу до порту CoreSight Debug Access Port різних процесорів Arm Cortex. Це те, що використовує вбудований відладчик на BBC micro:bit.
- ST-Link — це ряд внутрішньосхемних налагоджувачів від ST Microelectronics, J-Link — це ряд від SEGGER.
- Порт доступу для налагодження зазвичай являє собою або 5-контактний інтерфейс JTAG, або 2-контактний Serial Wire Debug.
- probe-rs — це бібліотека, яку ви можете інтегрувати у власні інструменти, якщо хочете.
- Протокол адаптера налагодження Microsoft дозволяє VSCode та іншим IDE налагоджувати код, запущений на будь-якому підтримуваному мікроконтролері.
- cargo-embed — бінарний файл, створений за допомогою бібліотеки probe-rs.
- RTT (Real Time Transfers) — це механізм передачі даних між хостом налагодження та налагоджуваною цільовою системою через кілька кільцевих буферів.

50.7.1 Налагодження

Embed.toml:

```
[default.general]
chip = "nrf52833_xxAA"

[debug.gdb]
enabled = true
```

В одному терміналі в `src/bare-metal/microcontrollers/examples/`:

```
cargo embed --bin board_support debug
```

В іншому терміналі в тому ж каталозі:

На gLinux або Debian:

```
gdb-multiarch target/thumbv7em-none-eabihf/debug/board_support --eval-command="target re
```

У MacOS:

```
arm-none-eabi-gdb target/thumbv7em-none-eabihf/debug/board_support --eval-command="targe
```

У GDB спробуйте запустити:

```
b src/bin/board_support.rs:29
b src/bin/board_support.rs:30
b src/bin/board_support.rs:32
c
c
c
```

50.8 Інші проекти

- RTIC

  - ”Одночасність виконання, керована перериванням у реальному часі”.
  - Управління спільними ресурсами, передача повідомлень, планування завдань, черга таймера.
- Embassy
  - `async` виконавці з пріоритетами, таймерами, мережею, USB.
- TockOS
  - Орієнтована на безпеку RTOS з випереджальним плануванням і підтримкою модуля захисту пам’яті.
- Hubris
  - Мікроядерна RTOS від Oxide Computer Company із захистом пам'яті, непривілейованими драйверами, IPC.
- Прив’язки для FreeRTOS.

Деякі платформи мають реалізацію `std`, наприклад esp-idf.

- RTIC можна вважати або RTOS, або фреймворком паралельного виконання.
  - Він не містить HAL.
  - Він використовує Cortex-M NVIC (вкладений віртуальний контролер переривань) для планування, а не належне ядро.
  - Тільки Cortex-M.
- Google використовує TockOS на мікроконтролері Haven для ключів безпеки Titan.
- FreeRTOS здебільшого написаний на C, але є прив’язки Rust для написання програм.

Розділ 51

Вправи

Ми прочитаємо напрямок із компаса I2C і запишемо показання до послідовного порту.

Переглянувши вправи, ви можете переглянути надані рішення.

51.1 Компас

Ми прочитаємо напрямок із компаса I2C і запишемо показання до послідовного порту. Якщо у вас є час, спробуйте ще якось відобразити його на світлодіодах або якось кнопками.

Підказки:

- Перегляньте документацію для `lsm303agr` і [`microbit-v2`](https://docs.rs/microbit-v2/latest/microbit /) крейтів, а також micro:bit hardware.
- Інерційний вимірювальний блок LSM303AGR підключено до внутрішньої шини I2C.
- TWI — це інша назва I2C, тому головний периферійний пристрій I2C називається TWIM.
- Драйверу LSM303AGR потрібно щось, що реалізує трейт `embedded_hal::i2c::I2c`. Структура `microbit::hal::Twim` реалізує це.
- У вас є структура `microbit::Board` з полями для різних контактів і периферійних пристроїв.
- Ви також можете переглянути технічну таблицю nRF52833, якщо хочете, але це не обов'язково для цієї вправи.

Завантажте шаблон вправи і знайдіть у каталозі `compass` наступні файли.

src/main.rs:

```
extern crate panic_halt as _;

use core::fmt::Write;
use cortex_m_rt::entry;
use microbit::{hal::{Delay, uarte::{Baudrate, Parity, Uarte}}, Board};
```

```rust
fn main() -> ! {
    let mut board = Board::take().unwrap();

    // Configure serial port.
    let mut serial = Uarte::new(
        board.UARTE0,
        board.uart.into(),
        Parity::EXCLUDED,
        Baudrate::BAUD115200,
    );

    // Use the system timer as a delay provider.
    let mut delay = Delay::new(board.SYST);

    // Set up the I2C controller and Inertial Measurement Unit.
    // TODO

    writeln!(serial, "Ready.").unwrap();

    loop {
        // Read compass data and log it to the serial port.
        // TODO
    }
}
```

Cargo.toml (це не потрібно змінювати):

```toml
[workspace]

[package]
name = "compass"
version = "0.1.0"
edition = "2021"
publish = false

[dependencies]
cortex-m-rt = "0.7.5"
embedded-hal = "1.0.0"
lsm303agr = "1.1.0"
microbit-v2 = "0.15.1"
panic-halt = "1.0.0"
```

Embed.toml (це не потрібно змінювати):

```toml
[default.general]
chip = "nrf52833_xxAA"

[debug.gdb]
enabled = true

[debug.reset]
halt_afterwards = true
```

.cargo/config.toml (це не потрібно змінювати):

```
[build]
target = "thumbv7em-none-eabihf" # Cortex-M4F

[target.'cfg(all(target_arch = "arm", target_os = "none"))']
rustflags = ["-C", "link-arg=-Tlink.x"]
```

Перегляньте послідовний вивід у Linux за допомогою:

```
picocom --baud 115200 --imap lfcrlf /dev/ttyACM0
```

Або в Mac OS щось на зразок (назва пристрою може трохи відрізнятися):

```
picocom --baud 115200 --imap lfcrlf /dev/tty.usbmodem14502
```

Використовуйте Ctrl+A Ctrl+Q, щоб вийти з picocom.

51.2 Ранкова зарядка Bare Metal Rust

Компас

(назад до вправи)

```
extern crate panic_halt as _;

use core::fmt::Write;
use cortex_m_rt::entry;
use core::cmp::{max, min};
use embedded_hal::digital::InputPin;
use lsm303agr::{
    AccelMode, AccelOutputDataRate, Lsm303agr, MagMode, MagOutputDataRate,
};
use microbit::display::blocking::Display;
use microbit::hal::twim::Twim;
use microbit::hal::uarte::{Baudrate, Parity, Uarte};
use microbit::hal::{Delay, Timer};
use microbit::pac::twim0::frequency::FREQUENCY_A;
use microbit::Board;

const COMPASS_SCALE: i32 = 30000;
const ACCELEROMETER_SCALE: i32 = 700;

fn main() -> ! {
    let mut board = Board::take().unwrap();

    // Налаштувати послідовний порт.
    let mut serial = Uarte::new(
        board.UARTE0,
        board.uart.into(),
        Parity::EXCLUDED,
        Baudrate::BAUD115200,
```

```
);

// Використовувати системний таймер як джерело затримки.
let mut delay = Delay::new(board.SYST);

// Налаштувати контролер I2C та блок інерційних вимірювань.
writeln!(serial, "Налаштування IMU...").unwrap();
let i2c = Twim::new(board.TWIM0, board.i2c_internal.into(), FREQUENCY_A::K100);
let mut imu = Lsm303agr::new_with_i2c(i2c);
imu.init().unwrap();
imu.set_mag_mode_and_odr(
    &mut delay,
    MagMode::HighResolution,
    MagOutputDataRate::Hz50,
)
.unwrap();
imu.set_accel_mode_and_odr(
    &mut delay,
    AccelMode::Normal,
    AccelOutputDataRate::Hz50,
)
.unwrap();
let mut imu = imu.into_mag_continuous().ok().unwrap();

// Налаштувати дисплей і таймер.
let mut timer = Timer::new(board.TIMER0);
let mut display = Display::new(board.display_pins);

let mut mode = Mode::Compass;
let mut button_pressed = false;

writeln!(serial, "Готовий.").unwrap();

loop {
    // Зчитати дані компаса і записати їх у послідовний порт.
    while !(imu.mag_status().unwrap().xyz_new_data()
        && imu.accel_status().unwrap().xyz_new_data())
    {}
    let compass_reading = imu.magnetic_field().unwrap();
    let accelerometer_reading = imu.acceleration().unwrap();
    writeln!(
        serial,
        "{},{},{}\t{},{},{}",
        compass_reading.x_nt(),
        compass_reading.y_nt(),
        compass_reading.z_nt(),
        accelerometer_reading.x_mg(),
        accelerometer_reading.y_mg(),
        accelerometer_reading.z_mg(),
    )
    .unwrap();
```

```
        let mut image = [[0; 5]; 5];
        let (x, y) = match mode {
            Mode::Compass => (
                scale(-compass_reading.x_nt(), -COMPASS_SCALE, COMPASS_SCALE, 0, 4)
                    as usize,
                scale(compass_reading.y_nt(), -COMPASS_SCALE, COMPASS_SCALE, 0, 4)
                    as usize,
            ),
            Mode::Accelerometer => (
                scale(
                    accelerometer_reading.x_mg(),
                    -ACCELEROMETER_SCALE,
                    ACCELEROMETER_SCALE,
                    0,
                    4,
                ) as usize,
                scale(
                    -accelerometer_reading.y_mg(),
                    -ACCELEROMETER_SCALE,
                    ACCELEROMETER_SCALE,
                    0,
                    4,
                ) as usize,
            ),
        };
        image[y][x] = 255;
        display.show(&mut timer, image, 100);

        // Якщо натиснута кнопка A, перейти в наступний режим і короткочасно
        // увімкнути всі світлодіоди.
        if board.buttons.button_a.is_low().unwrap() {
            if !button_pressed {
                mode = mode.next();
                display.show(&mut timer, [[255; 5]; 5], 200);
            }
            button_pressed = true;
        } else {
            button_pressed = false;
        }
    }
}

enum Mode {
    Compass,
    Accelerometer,
}

impl Mode {
    fn next(self) -> Self {
        match self {
```

```
            Self::Compass => Self::Accelerometer,
            Self::Accelerometer => Self::Compass,
        }
    }
}

fn scale(value: i32, min_in: i32, max_in: i32, min_out: i32, max_out: i32) -> i32 {
    let range_in = max_in - min_in;
    let range_out = max_out - min_out;
    cap(min_out + range_out * (value - min_in) / range_in, min_out, max_out)
}

fn cap(value: i32, min_value: i32, max_value: i32) -> i32 {
    max(min_value, min(value, max_value))
}
```

Частина XII

Залізо: Полудень

Розділ 52

Прикладні процесори

Досі ми говорили про мікроконтролери, такі як серія Arm Cortex-M. Тепер давайте спробуємо написати щось для Cortex-A. Для простоти ми просто працюватимемо з платою QEMU aarch64 'virt'.

- Загалом кажучи, мікроконтролери не мають MMU або кількох рівнів привілеїв (рівні виключень на центральних процесорах Arm, кільця на x86), тоді як процесори прикладних програм мають.
- QEMU підтримує емуляцію різних машин або моделей плат для кожної архітектури. Плата 'virt' не відповідає жодному конкретному реальному апаратному забезпеченню, а розроблена виключно для віртуальних машин.

52.1 Підготовка до Rust

Перш ніж ми зможемо запускати код Rust, нам потрібно виконати деяку ініціалізацію.

```
.section .init.entry, "ax"
.global entry
entry:
    /*
     * Завантаження та застосування конфігурації керування пам'яттю, готової
     * до ввімкнення MMU та
     * кешів.
     */
    adrp x30, idmap
    msr ttbr0_el1, x30

    mov_i x30, .Lmairval
    msr mair_el1, x30

    mov_i x30, .Ltcrval
    /* Скопіювати підтримуваний діапазон PA у TCR_EL1.IPS. */
    mrs x29, id_aa64mmfr0_el1
    bfi x30, x29, #32, #4
```

```
    msr tcr_el1, x30

    mov_i x30, .Lsctlrval

    /*
     * Перевірити все до завершення цього пункту, а потім зробити недійсними всі
     * потенційно застарілі локальні записи TLB до того, як вони почнуть використовувати
     */
    isb
    tlbi vmalle1
    ic iallu
    dsb nsh
    isb

    /*
     * Налаштувати sctlr_el1 на ввімкнення MMU та кешу і не продовжувати,  доки це
     * не буде зроблено.
     */
    msr sctlr_el1, x30
    isb

    /* Вимкнути перехоплення доступу з плаваючою комою в EL1. */
    mrs x30, cpacr_el1
    orr x30, x30, #(0x3 << 20)
    msr cpacr_el1, x30
    isb

    /* Обнуліть секцію bss. */
    adr_l x29, bss_begin
    adr_l x30, bss_end
0:  cmp x29, x30
    b.hs 1f
    stp xzr, xzr, [x29], #16
    b 0b

1:  /* Підготувати стек. */
    adr_l x30, boot_stack_end
    mov sp, x30

    /* Налаштування вектора виключень. */
    adr x30, vector_table_el1
    msr vbar_el1, x30

    /* Виклик коду Rust. */
    bl main

    /* Постійно циклічно чекаємо на переривання. */
2:  wfi
    b 2b
```

- Це те саме, що було б для C: ініціалізація стану процесора, обнулення BSS і

налаштування покажчика стека.
  - BSS (символ початку блоку, з історичних причин) — це частина об'єктного файлу, яка містить статично виділені змінні, які ініціалізуються нулем. Вони пропущені на зображенні, щоб не витрачати місце на зайві нулі. Компілятор припускає, що завантажувач подбає про їх обнулення.
- BSS може бути вже обнулено, залежно від того, як ініціалізовано пам'ять і завантажено зображення, але ми обнуляємо його, щоб бути впевненими.
- Нам потрібно ввімкнути MMU та кеш перед читанням або записом пам'яті. Якщо ми цього не зробимо:
  - Невирівняні доступи призведуть до помилки. Ми створюємо код Rust для цілі `aarch64-unknown-none`, яка встановлює `+strict-align`, щоб запобігти створенню компілятором невирівняних доступів, тому в цьому випадку це має бути гаразд, але це не обов'язково так загалом.
  - Якщо це було запущено у віртуальній машині, це може призвести до проблеми з узгодженістю кешу. Проблема полягає в тому, що віртуальна машина звертається до пам'яті безпосередньо з вимкненим кешем, в той час як хост має кешовані псевдоніми до тієї ж пам'яті. Навіть якщо хост не має явного доступу до пам'яті, спекулятивні доступи можуть призвести до заповнення кешу, а потім зміни з того чи іншого будуть втрачені, коли кеш буде очищено або віртуальна машина ввімкне кеш. (Кеш використовується за фізичною адресою, а не VA чи IPA.)
- Для спрощення ми просто використовуємо жорстко закодовану таблицю сторінок (дивиться `idmap.S`), яка ідентифікує перший 1 ГіБ адресного простору для пристроїв, наступний 1 ГіБ для DRAM і ще 1 ГіБ вище для інших пристроїв. Це відповідає розміщенню пам'яті, яке використовує QEMU.
- Ми також встановили вектор виключень (`vbar_el1`), про який ми розповімо більше пізніше.
- Усі приклади цього дня припускають, що ми будемо працювати на рівні виключення 1 (EL1). Якщо вам потрібно запустити на іншому рівні виключення, вам потрібно буде відповідно змінити `entry.S`.

52.2 Вбудований асемблер

Іноді нам потрібно використовувати асемблер для того, щоб робити речі, які неможливо зробити за допомогою коду на Rust. Наприклад, зробити HVC (виклик гіпервізора), щоб сказати прошивці вимкнути систему:

```
use core::arch::asm;
use core::panic::PanicInfo;

mod exceptions;

const PSCI_SYSTEM_OFF: u32 = 0x84000008;

// БЕЗПЕКА: Не існує іншої глобальної функції з таким іменем.
extern "C" fn main(_x0: u64, _x1: u64, _x2: u64, _x3: u64) {
    // БЕЗПЕКА: тут використовуються тільки оголошені регістри
    // і нічого не робиться з пам'яттю.
    unsafe {
```

```
        asm!("hvc #0",
            inout("w0") PSCI_SYSTEM_OFF => _,
            inout("w1") 0 => _,
            inout("w2") 0 => _,
            inout("w3") 0 => _,
            inout("w4") 0 => _,
            inout("w5") 0 => _,
            inout("w6") 0 => _,
            inout("w7") 0 => _,
            options(nomem, nostack)
        );
    }

    loop {}
}
```

(Якщо ви справді хочете це зробити, скористайтеся крейтом `smccc`, у якому є оболонки для всіх цих функцій.)

- PSCI — це Arm Power State Coordination Interface, стандартний набір функцій для керування станами живлення системи та CPU, серед іншого. Він реалізований прошивкою EL3 і гіпервізорами на багатьох системах.
- Синтаксис `0 => _` означає ініціалізацію реєстру до 0 перед виконанням вбудованого асемблеру та ігнорування його вмісту після цього. Нам потрібно використовувати `inout`, а не `in`, оскільки виклик потенційно може знищити вміст реєстрів.
- Ця `main` функція має бути `#[unsafe(no_mangle)]` і `extern "C"`, оскільки вона викликається з нашої точки входу в `entry.S`.
- `_x0`–`_x3` – це значення регістрів `x0`–`x3`, які традиційно використовуються завантажувачем для передачі таких речей, як покажчик на дерево пристроїв. Відповідно до стандартної угоди про виклики aarch64 (це те, що вказує `extern "C"`), регістри `x0`–`x7` використовуються для перших 8 аргументів, що передаються до функції, тому `entry.S` не потрібно робити нічого особливого, окрім як переконатися, що він не змінює ці регістри.
- Запустіть приклад у QEMU за допомогою `make qemu_psci` в `src/bare-metal/aps/examples`.

52.3 Здійснення непостійного доступу до пам'яті для MMIO

- Використовуйте `pointer::read_volatile` та `pointer::write_volatile`.
- Ніколи не тримайте посилання.
- Використовуйте `&raw` для отримання полів структур без створення проміжного посилання.

- Непостійний доступ: операції читання або запису можуть мати побічні ефекти, тому не дозволяйте компілятору чи апаратному забезпеченню їх перевпорядковувати, дублювати чи видаляти.
  - Зазвичай, якщо ви пишете, а потім читаєте, напр. через змінне посилання, компілятор може припустити, що прочитане значення є таким самим, як щойно записане значення, і не турбуватися про фактичне читання пам’яті.

- Деякі існуючі крейти для непостійного доступу до апаратного забезпечення містять посилання, але це нерозумно. Кожного разу, коли існує посилання, компілятор може вирішити розіменувати його.
- Використовуйте `&raw`, щоб отримати покажчики полів структури від покажчика на структуру.
- Для сумісності зі старими версіями Rust ви можете скористатися замість цього макросом `addr_of!`.

52.4 Давайте напишемо драйвер UART

Машина QEMU 'virt' має PL011 UART, тож давайте напишемо для нього драйвер.

```
const FLAG_REGISTER_OFFSET: usize = 0x18;
const FR_BUSY: u8 = 1 << 3;
const FR_TXFF: u8 = 1 << 5;

/// Мінімальний драйвер для PL011 UART.
pub struct Uart {
    base_address: *mut u8,
}

impl Uart {
    /// Створює новий екземпляр драйвера UART для пристрою PL011
    /// за заданою базовою адресою.
    ///
    /// # Безпека
    ///
    /// Задана базова адреса повинна вказувати на 8 керуючих регістрів MMIO пристрою
    /// PL011, які повинні бути відображені в адресному просторі процесу
    /// як пам'ять пристрою і не мати ніяких інших псевдонімів.
    pub unsafe fn new(base_address: *mut u8) -> Self {
        Self { base_address }
    }

    /// Записує один байт до UART.
    pub fn write_byte(&self, byte: u8) {
        // Чекаємо, поки не звільниться місце в буфері TX.
        while self.read_flag_register() & FR_TXFF != 0 {}

        // БЕЗПЕКА: ми знаємо, що базова адреса вказує на регістри
        // керування пристрою PL011, які відповідним чином відображені.
        unsafe {
            // Записуємо в буфер TX.
            self.base_address.write_volatile(byte);
        }

        // Чекаємо, поки UART більше не буде зайнято.
        while self.read_flag_register() & FR_BUSY != 0 {}
    }
```

```
    fn read_flag_register(&self) -> u8 {
        // БЕЗПЕКА: ми знаємо, що базова адреса вказує на регістри
        // керування пристрою PL011, які відповідним чином відображені.
        unsafe { self.base_address.add(FLAG_REGISTER_OFFSET).read_volatile() }
    }
}
```

- Зауважте, що `Uart::new` є небезпечним, тоді як інші методи є безпечними. Це пов'язано з тим, що доки викликач `Uart::new` гарантує, що його вимоги безпеки дотримано (тобто, що існує лише один екземпляр драйвера для даного UART, і ніщо інше не змінює його адресний простір), доти безпечно викликати `write_byte` пізніше, оскільки ми можемо припустити, що виконано необхідні передумови.
- Ми могли б зробити це навпаки (зробити `new` безпечним, але `write_byte` небезпечним), але це було б набагато менш зручно використовувати, оскільки кожне місце, яке викликає `write_byte`, мало б міркувати про безпеку
- Це загальний шаблон для написання безпечних оболонок небезпечного коду: перенесення тягаря доведення правильності з великої кількості місць на меншу кількість місць.

52.4.1 Більше трейтів

Ми вивели трейт Debug. Також було б корисно реалізувати ще кілька трейтів.

```
use core::fmt::{self, Write};

impl Write for Uart {
    fn write_str(&mut self, s: &str) -> fmt::Result {
        for c in s.as_bytes() {
            self.write_byte(*c);
        }
        Ok(())
    }
}

// БЕЗПЕКА: `Uart` містить лише покажчик на пам'ять пристрою, до якого
// можна отримати доступ з будь-якого контексту.
unsafe impl Send for Uart {}
```

- Реалізація `Write` дозволяє використовувати макроси `write!` і `writeln!` з нашим типом `Uart`.
- Запустіть приклад у QEMU за допомогою `make qemu_minimal` у `src/bare-metal/aps/examples`.

52.5 Кращий драйвер UART

PL011 насправді має набагато більше регістрів, і додавання зміщень до вказівників для конструювання доступу до них може призвести до помилок, і читається важко. Крім того, деякі з них є бітовими полями, до яких було б добре мати структурований доступ.

Зміщення	Ім'я регістру	Ширина
0x00	DR	12
0x04	RSR	4
0x18	FR	9
0x20	ILPR	8
0x24	IBRD	16
0x28	FBRD	6
0x2c	LCR_H	8
0x30	CR	16
0x34	IFLS	6
0x38	IMSC	11
0x3c	RIS	11
0x40	MIS	11
0x44	ICR	11
0x48	DMACR	3

- Є також деякі ID регістри, які були пропущені для стислості.

52.5.1 Бітові прапорці (крейт bitflags)

Крейт `bitflags` корисний для роботи з бітовими флагами.

```
use bitflags::bitflags;

bitflags! {
    /// Прапорці з регістру прапорів UART.
    struct Flags: u16 {
        /// Очистити для відправки.
        const CTS = 1 << 0;
        /// Набір даних готовий.
        const DSR = 1 << 1;
        /// Визначення носія даних.
        const DCD = 1 << 2;
        /// UART зайнятий передачею даних.
        const BUSY = 1 << 3;
        /// FIFO отримання порожній.
        const RXFE = 1 << 4;
        /// FIFO передачі заповнено.
        const TXFF = 1 << 5;
        /// FIFO отримання заповнено.
        const RXFF = 1 << 6;
        /// FIFO передачі порожній.
        const TXFE = 1 << 7;
        /// Індикатор кільця.
        const RI = 1 << 8;
    }
}
```

- Макрос `bitflags!` створює новий тип, щось на кшталт `Flags(u16)` разом із купою реалізацій методів для отримання та встановлення прапорів.

52.5.2 Кілька регістрів

Ми можемо використовувати структуру для представлення розташування пам'яті регістрів UART.

```
struct Registers {
    dr: u16,
    _reserved0: [u8; 2],
    rsr: ReceiveStatus,
    _reserved1: [u8; 19],
    fr: Flags,
    _reserved2: [u8; 6],
    ilpr: u8,
    _reserved3: [u8; 3],
    ibrd: u16,
    _reserved4: [u8; 2],
    fbrd: u8,
    _reserved5: [u8; 3],
    lcr_h: u8,
    _reserved6: [u8; 3],
    cr: u16,
    _reserved7: [u8; 3],
    ifls: u8,
    _reserved8: [u8; 3],
    imsc: u16,
    _reserved9: [u8; 2],
    ris: u16,
    _reserved10: [u8; 2],
    mis: u16,
    _reserved11: [u8; 2],
    icr: u16,
    _reserved12: [u8; 2],
    dmacr: u8,
    _reserved13: [u8; 3],
}
```

- `#[repr(C)]` каже компілятору розмістити поля структури в потрібному порядку, дотримуючись тих самих правил, що й C. Це необхідно для того, щоб наша структура мала передбачуваний порядок розміщення, оскільки представлення Rust за замовчуванням дозволяє компілятору (між іншим) змінювати порядок полів, як він вважає за потрібне.

52.5.3 Драйвер

Тепер давайте використаємо нову структуру `Registers` у нашому драйвері.

```
/// Драйвер для PL011 UART.
pub struct Uart {
    registers: *mut Registers,
}

impl Uart {
```

```
    /// Створює новий екземпляр драйвера UART для пристрою PL011
    /// за заданою базовою адресою.
    ///
    /// # Безпека
    ///
    /// Задана базова адреса повинна вказувати на 8 керуючих регістрів MMIO пристрою
    /// PL011, які повинні бути відображені в адресному просторі процесу
    /// як пам'ять пристрою і не мати ніяких інших псевдонімів.
    pub unsafe fn new(base_address: *mut u32) -> Self {
        Self { registers: base_address as *mut Registers }
    }

    /// Записує один байт до UART.
    pub fn write_byte(&self, byte: u8) {
        // Чекаємо, поки не звільниться місце в буфері TX.
        while self.read_flag_register().contains(Flags::TXFF) {}

        // БЕЗПЕКА: ми знаємо, що self.registers вказує на керуючі
        // регістри пристрою PL011, який відповідним чином відображено.
        unsafe {
            // Записуємо в буфер TX.
            (&raw mut (*self.registers).dr).write_volatile(byte.into());
        }

        // Чекаємо, поки UART більше не буде зайнято.
        while self.read_flag_register().contains(Flags::BUSY) {}
    }

    /// Читає і повертає байт очікування, або `None`, якщо нічого не було
    /// отримано.
    pub fn read_byte(&self) -> Option<u8> {
        if self.read_flag_register().contains(Flags::RXFE) {
            None
        } else {
            // БЕЗПЕКА: ми знаємо, що self.registers вказує на керуючі
        // регістри пристрою PL011, який відповідним чином відображено.
            let data = unsafe { (&raw const (*self.registers).dr).read_volatile() };
            // TODO: Перевірити на наявність помилок у бітах 8-11.
            Some(data as u8)
        }
    }

    fn read_flag_register(&self) -> Flags {
        // БЕЗПЕКА: ми знаємо, що self.registers вказує на керуючі
        // регістри пристрою PL011, який відповідним чином відображено.
        unsafe { (&raw const (*self.registers).fr).read_volatile() }
    }
}
```

- Зверніть увагу на використання `&raw const` / `&raw mut` для отримання вказівників на окремі поля без створення проміжного посилання, що було б

нерозумним.

52.5.4 Використання

Давайте напишемо невелику програму, використовуючи наш драйвер для запису в послідовну консоль і відлуння вхідних байтів.

```
mod exceptions;
mod pl011;

use crate::pl011::Uart;
use core::fmt::Write;
use core::panic::PanicInfo;
use log::error;
use smccc::psci::system_off;
use smccc::Hvc;

/// Базова адреса основного PL011 UART.
const PL011_BASE_ADDRESS: *mut u32 = 0x900_0000 as _;

// БЕЗПЕКА: Не існує іншої глобальної функції з таким іменем.
extern "C" fn main(x0: u64, x1: u64, x2: u64, x3: u64) {
    // БЕЗПЕКА: `PL011_BASE_ADDRESS` є базовою адресою пристрою PL011,
    // і ніщо інше не має доступу до цього діапазону адресації.
    let mut uart = unsafe { Uart::new(PL011_BASE_ADDRESS) };

    writeln!(uart, "main({x0:#x}, {x1:#x}, {x2:#x}, {x3:#x})").unwrap();

    loop {
        if let Some(byte) = uart.read_byte() {
            uart.write_byte(byte);
            match byte {
                b'\r' => {
                    uart.write_byte(b'\n');
                }
                b'q' => break,
                _ => continue,
            }
        }
    }

    writeln!(uart, "\n\nБувайте!").unwrap();
    system_off::<Hvc>().unwrap();
}
```

- Як і у прикладі вбудована збірка, ця функція `main` викликається з нашого коду точки входу в `entry.S`. Докладніше дивиться у примітках доповідача.
- Запустіть приклад у QEMU за допомогою `make qemu` у `src/bare-metal/aps/examples`.

52.6 Журналювання

Було б чудово мати можливість використовувати макроси журналювання з крейту log. Ми можемо зробити це, реалізувавши трейт Log.

```
use crate::pl011::Uart;
use core::fmt::Write;
use log::{LevelFilter, Log, Metadata, Record, SetLoggerError};
use spin::mutex::SpinMutex;

static LOGGER: Logger = Logger { uart: SpinMutex::new(None) };

struct Logger {
    uart: SpinMutex<Option<Uart>>,
}

impl Log for Logger {
    fn enabled(&self, _metadata: &Metadata) -> bool {
        true
    }

    fn log(&self, record: &Record) {
        writeln!(
            self.uart.lock().as_mut().unwrap(),
            "[{}] {}",
            record.level(),
            record.args()
        )
        .unwrap();
    }

    fn flush(&self) {}
}

/// Ініціалізує логгер UART.
pub fn init(uart: Uart, max_level: LevelFilter) -> Result<(), SetLoggerError> {
    LOGGER.uart.lock().replace(uart);

    log::set_logger(&LOGGER)?;
    log::set_max_level(max_level);
    Ok(())
}
```

- Розгортання в `log` є безпечним, оскільки ми ініціалізуємо `LOGGER` перед викликом `set_logger`.

52.6.1 Використання

Нам потрібно ініціалізувати логгер перед його використанням.

```
mod exceptions;
```

```
mod logger;
mod pl011;

use crate::pl011::Uart;
use core::panic::PanicInfo;
use log::{error, info, LevelFilter};
use smccc::psci::system_off;
use smccc::Hvc;

/// Базова адреса основного PL011 UART.
const PL011_BASE_ADDRESS: *mut u32 = 0x900_0000 as _;

// БЕЗПЕКА: Не існує іншої глобальної функції з таким іменем.
extern "C" fn main(x0: u64, x1: u64, x2: u64, x3: u64) {
    // БЕЗПЕКА: `PL011_BASE_ADDRESS` є базовою адресою пристрою PL011,
    // і ніщо інше не має доступу до цього діапазону адресації.
    let uart = unsafe { Uart::new(PL011_BASE_ADDRESS) };
    logger::init(uart, LevelFilter::Trace).unwrap();

    info!("main({x0:#x}, {x1:#x}, {x2:#x}, {x3:#x})");

    assert_eq!(x1, 42);

    system_off::<Hvc>().unwrap();
}

fn panic(info: &PanicInfo) -> ! {
    error!("{info}");
    system_off::<Hvc>().unwrap();
    loop {}
}
```

- Зверніть увагу, що наш обробник паніки тепер може реєструвати деталі паніки.
- Запустіть приклад у QEMU за допомогою `make qemu_logger` у `src/bare-metal/aps/examples`.

52.7 Виключення

AArch64 визначає векторну таблицю винятків із 16 записами для 4 типів винятків (синхронний, IRQ, FIQ, SError) із 4 станів (поточний EL із SP0, поточний EL із SPx, нижчий EL із використанням AArch64, нижчий EL із застосуванням AArch32). Ми реалізуємо це в асемблері, щоб зберегти непостійні регістри в стеку перед викликом коду Rust:

```
use log::error;
use smccc::psci::system_off;
use smccc::Hvc;

// SAFETY: There is no other global function of this name.
extern "C" fn sync_exception_current(_elr: u64, _spsr: u64) {
    error!("sync_exception_current");
```

```
    system_off::<Hvc>().unwrap();
}

// SAFETY: There is no other global function of this name.
extern "C" fn irq_current(_elr: u64, _spsr: u64) {
    error!("irq_current");
    system_off::<Hvc>().unwrap();
}

// SAFETY: There is no other global function of this name.
extern "C" fn fiq_current(_elr: u64, _spsr: u64) {
    error!("fiq_current");
    system_off::<Hvc>().unwrap();
}

// SAFETY: There is no other global function of this name.
extern "C" fn serr_current(_elr: u64, _spsr: u64) {
    error!("serr_current");
    system_off::<Hvc>().unwrap();
}

// SAFETY: There is no other global function of this name.
extern "C" fn sync_lower(_elr: u64, _spsr: u64) {
    error!("sync_lower");
    system_off::<Hvc>().unwrap();
}

// SAFETY: There is no other global function of this name.
extern "C" fn irq_lower(_elr: u64, _spsr: u64) {
    error!("irq_lower");
    system_off::<Hvc>().unwrap();
}

// SAFETY: There is no other global function of this name.
extern "C" fn fiq_lower(_elr: u64, _spsr: u64) {
    error!("fiq_lower");
    system_off::<Hvc>().unwrap();
}

// SAFETY: There is no other global function of this name.
extern "C" fn serr_lower(_elr: u64, _spsr: u64) {
    error!("serr_lower");
    system_off::<Hvc>().unwrap();
}
```

- EL - це рівень винятків; усі наші приклади сьогодні працюють на EL1.
- Для простоти ми не розрізняємо SP0 і SPx для поточних винятків EL або між AArch32 і AArch64 для нижчих винятків EL.
- У цьому прикладі ми просто реєструємо виняток і вимикаємо живлення, оскільки ми не очікуємо, що будь-що з цього станеться.
- Ми можемо розглядати обробники винятків і наш основний контекст виконання

більш-менш як різні потоки. Send і Sync керуватимуть тим, чим ми можемо обмінюватися між ними, як і з потоками. Наприклад, якщо ми хочемо поділитися деяким значенням між обробниками винятків та рештою програми, і це Send, але не Sync, тоді нам потрібно буде загорнути його в щось на зразок Mutex і помістити у статику.

52.8 Інші проекти

- oreboot
  - ”coreboot без C”.
  - Підтримує x86, aarch64 і RISC-V.
  - Покладається на LinuxBoot, замість того, щоб самому мати багато драйверів.
- Навчальний посібник Rust з ОС RaspberryPi
  - Ініціалізація, драйвер UART, простий завантажувач, JTAG, рівні винятків, обробка винятків, таблиці сторінок.
  - Деякі хитрощі щодо обслуговування кешу та ініціалізації в Rust, не обов’язково хороший приклад для копіювання для виробничого коду.
- `cargo-call-stack`
  - Статичний аналіз для визначення максимального використання стека.

- Підручник з ОС RaspberryPi запускає код Rust до ввімкнення MMU та кешу. Це дозволить читати та записувати пам’ять (наприклад, стек). Однак:
  - Без MMU та кешу невирівняні доступи призведуть до помилки. Код створюється за допомогою `aarch64-unknown-none`, який встановлює `+strict-align`, щоб запобігти генерації компілятором невирівняних доступів, тому це має бути гаразд, але це не обов’язково так загалом.
  - Якщо код працював у віртуальній машині, це може призвести до проблем узгодженості кешу. Проблема полягає в тому, що віртуальна машина звертається до пам’яті безпосередньо з вимкненою кеш-пам’яттю, тоді як хост має кешовані псевдоніми для тієї самої пам’яті. Навіть якщо хост явно не звертається до пам’яті, спекулятивні доступи можуть призвести до заповнення кешу, і тоді зміни з одного або іншого боку будуть втрачені. Знову ж таки, це нормально в цьому конкретному випадку (працює безпосередньо на апаратному забезпеченні без гіпервізора), але це не дуже гарний шаблон загалом.

Розділ 53

Корисні крейти

Ми розглянемо кілька крейтів, які вирішують деякі поширені проблеми програмування на голому залізі.

53.1 `zerocopy`

Крейт `zerocopy` (від Fuchsia) надає трейти та макроси для безпечного перетворення між послідовностями байтів та іншими типами.

```
use zerocopy::{Immutable, IntoBytes};

enum RequestType {
    In = 0,
    Out = 1,
    Flush = 4,
}

struct VirtioBlockRequest {
    request_type: RequestType,
    reserved: u32,
    sector: u64,
}

fn main() {
    let request = VirtioBlockRequest {
        request_type: RequestType::Flush,
        sector: 42,
        ..Default::default()
    };

    assert_eq!(
        request.as_bytes(),
        &[4, 0, 0, 0, 0, 0, 0, 0, 42, 0, 0, 0, 0, 0, 0, 0]
    );
}
```

Це не підходить для MMIO (оскільки він не використовує непостійні читання та записи), але може бути корисним для роботи зі структурами, спільними з обладнанням, наприклад, з прямим доступом до пам'яті (DMA), або переданими через зовнішній інтерфейс.

- `FromBytes` можна реалізувати для типів, для яких дійсний будь-який шаблон байтів, і тому його можна безпечно перетворити з ненадійної послідовності байтів.
- Спроба отримати `FromBytes` для цих типів не вдасться, оскільки `RequestType` не використовує всі можливі значення u32 як дискримінанти, тому не всі шаблони байтів є дійсними.
- `zerocopy::byteorder` має типи для числових примітивів з урахуванням порядку байтів.
- Запустіть приклад із `cargo run` у `src/bare-metal/useful-crates/zerocopy-example/`. (Він не працюватиме на Rust Playground через залежність від крейту.)

53.2 aarch64-paging

Крейт aarch64-paging дозволяє створювати таблиці сторінок відповідно до архітектури системи віртуальної пам’яті AArch64.

```
use aarch64_paging::{
    idmap::IdMap,
    paging::{Attributes, MemoryRegion},
};

const ASID: usize = 1;
const ROOT_LEVEL: usize = 1;

// Створити нову таблицю сторінок з відображенням ідентичності.
let mut idmap = IdMap::new(ASID, ROOT_LEVEL);
// Відобразити область пам'яті розміром 2 MiB як доступну тільки для читання.
idmap.map_range(
    &MemoryRegion::new(0x80200000, 0x80400000),
    Attributes::NORMAL | Attributes::NON_GLOBAL | Attributes::READ_ONLY,
).unwrap();
// Встановити `TTBR0_EL1` для активації таблиці сторінок.
idmap.activate();
```

- Наразі він підтримує лише EL1, але підтримка інших рівнів винятків має бути легко додана.
- Це використовується в Android для прошивки захищеної віртуальної машини.
- Немає простого способу запустити цей приклад, оскільки він повинен працювати на реальному обладнанні або під керуванням QEMU.

53.3 buddy_system_allocator

buddy_system_allocator — це сторонній крейт, який реалізує базовий системний розподільник між друзями. Його можна використовувати як для LockedHeap, так і для реалізації GlobalAlloc, щоб ви могли використовувати стандартний крейт `alloc`

(як ми бачили раніше), або для виділення іншого адресного простору. Наприклад, ми можемо захотіти виділити простір MMIO для шин PCI:

```
use buddy_system_allocator::FrameAllocator;
use core::alloc::Layout;

fn main() {
    let mut allocator = FrameAllocator::<32>::new();
    allocator.add_frame(0x200_0000, 0x400_0000);

    let layout = Layout::from_size_align(0x100, 0x100).unwrap();
    let bar = allocator
        .alloc_aligned(layout)
        .expect("Failed to allocate 0x100 byte MMIO region");
    println!("Allocated 0x100 byte MMIO region at {:#x}", bar);
}
```

- Шини PCI завжди мають вирівнювання відповідно до їх розміру.
- Запустіть приклад із `cargo run` у `src/bare-metal/useful-crates/allocator-example/`. (Він не працюватиме на Rust Playground через залежність від крейту.)

53.4 `tinyvec`

Іноді вам потрібне щось, розмір якого можна змінити, наприклад `Vec`, але без виділення купи. `tinyvec` надає це: вектор, підкріплений масивом або зрізом, який може бути статично розміщений або в стеку, який відстежує, скільки елементів використовується та впадає в паніку, якщо ви намагаєтеся використати більше, ніж виділено.

```
use tinyvec::{array_vec, ArrayVec};

fn main() {
    let mut numbers: ArrayVec<[u32; 5]> = array_vec!(42, 66);
    println!("{numbers:?}");
    numbers.push(7);
    println!("{numbers:?}");
    numbers.remove(1);
    println!("{numbers:?}");
}
```

- `tinyvec` вимагає, щоб тип елемента реалізував `Default` для ініціалізації.
- Rust Playground містить `tinyvec`, тож цей приклад добре працюватиме вбудовано.

53.5 `spin`

`std::sync::Mutex` та інші примітиви синхронізації з `std::sync` недоступні в `core` або `alloc`. Як ми можемо керувати синхронізацією або внутрішньою мутабельністю, наприклад, для обміну станом між різними CPU?

Крейт `spin` надає еквіваленти багатьох із цих примітивів на основі спін-блокування.

```rust
use spin::mutex::SpinMutex;

static counter: SpinMutex<u32> = SpinMutex::new(0);

fn main() {
    println!("count: {}", counter.lock());
    *counter.lock() += 2;
    println!("count: {}", counter.lock());
}
```

- Будьте обережні, щоб уникнути взаємоблокувань, якщо ви використовуєте блокування в обробниках переривань.
- `spin` також має реалізацію квиткового м'ютексу блокування; еквіваленти `RwLock`, `Barrier` і `Once` з `std::sync`; і `Lazy` для ледачої ініціалізації.
- Крейт `once_cell` також має кілька корисних типів для пізньої ініціалізації з дещо іншим підходом до `spin::once::Once`.
- Rust Playground містить `spin`, тож цей приклад добре працюватиме вбудовано.

Розділ 54

Залізо на Android

Щоб зібрати бінарник Rust в AOSP для голого заліза, вам потрібно використати правило `rust_ffi_static` Soong для створення коду Rust, потім `cc_binary` зі сценарієм компонування, щоб створити сам бінарний файл, а потім `raw_binary` для перетворення ELF у необроблений бінарний файл, готовий до запуску.

```
rust_ffi_static {
    name: "libvmbase_example",
    defaults: ["vmbase_ffi_defaults"],
    crate_name: "vmbase_example",
    srcs: ["src/main.rs"],
    rustlibs: [
        "libvmbase",
    ],
}

cc_binary {
    name: "vmbase_example",
    defaults: ["vmbase_elf_defaults"],
    srcs: [
        "idmap.S",
    ],
    static_libs: [
        "libvmbase_example",
    ],
    linker_scripts: [
        "image.ld",
        ":vmbase_sections",
    ],
}

raw_binary {
    name: "vmbase_example_bin",
    stem: "vmbase_example.bin",
    src: ":vmbase_example",
    enabled: false,
```

```
    target: {
        android_arm64: {
            enabled: true,
        },
    },
}
```

54.1 vmbase

Для віртуальних машин, що працюють під керуванням crosvm на aarch64, бібліотека vmbase надає сценарій компонування та корисні параметри за замовчуванням для правил збірки разом із точкою входу, журналювання консолі UART тощо.

```
use vmbase::{main, println};

main!(main);

pub fn main(arg0: u64, arg1: u64, arg2: u64, arg3: u64) {
    println!("Hello world");
}
```

- Макрос `main!` позначає вашу основну функцію, яку потрібно викликати з точки входу `vmbase`.
- Точка входу `vmbase` обробляє ініціалізацію консолі та видає PSCI_SYSTEM_OFF для завершення роботи віртуальної машини, якщо основна функція повертається.

Розділ 55

Вправи

Напишемо драйвер для пристрою годин реального часу PL031.

Переглянувши вправи, ви можете переглянути надані рішення.

55.1 RTC драйвер

Віртуальна машина QEMU aarch64 має PL031 годинник реального часу за адресою 0x9010000. Для цієї вправи ви повинні написати для неї драйвер.

1. Використовуйте його для друку поточного часу на послідовній консолі. Ви можете використовувати крейт `chrono` для форматування дати/часу.
2. Використовуйте регістр збігу та необроблений стан переривання для очікування зайнятості до заданого часу, наприклад 3 секунди в майбутньому. (Викличте `core::hint::spin_loop` усередині циклу.)
3. *Розширення, якщо у вас є час:* Увімкніть і обробіть переривання, створене збігом RTC. Ви можете використовувати драйвер, наданий у крейті `arm-gic`, щоб налаштувати загальний контролер переривань Arm.
    - Використовуйте переривання RTC, яке підключено до GIC як `IntId::spi(2)`.
    - Коли переривання ввімкнено, ви можете перевести ядро в режим сну за допомогою `arm_gic::wfi()`, що призведе до того, що ядро буде спати, доки воно не отримає переривання.

Завантажте шаблон вправи і знайдіть у каталозі `rtc` наступні файли.

src/main.rs:

```
mod exceptions;
mod logger;
mod pl011;

use crate::pl011::Uart;
use arm_gic::gicv3::GicV3;
use core::panic::PanicInfo;
use log::{error, info, trace, LevelFilter};
use smccc::psci::system_off;
```

```rust
use smccc::Hvc;

/// Base addresses of the GICv3.
const GICD_BASE_ADDRESS: *mut u64 = 0x800_0000 as _;
const GICR_BASE_ADDRESS: *mut u64 = 0x80A_0000 as _;

/// Base address of the primary PL011 UART.
const PL011_BASE_ADDRESS: *mut u32 = 0x900_0000 as _;

// SAFETY: There is no other global function of this name.
extern "C" fn main(x0: u64, x1: u64, x2: u64, x3: u64) {
    // SAFETY: `PL011_BASE_ADDRESS` is the base address of a PL011 device, and
    // nothing else accesses that address range.
    let uart = unsafe { Uart::new(PL011_BASE_ADDRESS) };
    logger::init(uart, LevelFilter::Trace).unwrap();

    info!("main({:#x}, {:#x}, {:#x}, {:#x})", x0, x1, x2, x3);

    // SAFETY: `GICD_BASE_ADDRESS` and `GICR_BASE_ADDRESS` are the base
    // addresses of a GICv3 distributor and redistributor respectively, and
    // nothing else accesses those address ranges.
    let mut gic = unsafe { GicV3::new(GICD_BASE_ADDRESS, GICR_BASE_ADDRESS) };
    gic.setup();

    // TODO: Create instance of RTC driver and print current time.

    // TODO: Wait for 3 seconds.

    system_off::<Hvc>().unwrap();
}

fn panic(info: &PanicInfo) -> ! {
    error!("{info}");
    system_off::<Hvc>().unwrap();
    loop {}
}
```

`src/exceptions.rs` (вам потрібно буде змінити його лише для 3-ї частини вправи):

```rust
// Copyright 2023 Google LLC
//
// Licensed under the Apache License, Version 2.0 (the "License");
// you may not use this file except in compliance with the License.
// You may obtain a copy of the License at
//
//      http://www.apache.org/licenses/LICENSE-2.0
//
// Unless required by applicable law or agreed to in writing, software
// distributed under the License is distributed on an "AS IS" BASIS,
// WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
// See the License for the specific language governing permissions and
// limitations under the License.
```

```rust
use arm_gic::gicv3::GicV3;
use log::{error, info, trace};
use smccc::psci::system_off;
use smccc::Hvc;

// SAFETY: There is no other global function of this name.
extern "C" fn sync_exception_current(_elr: u64, _spsr: u64) {
    error!("sync_exception_current");
    system_off::<Hvc>().unwrap();
}

// SAFETY: There is no other global function of this name.
extern "C" fn irq_current(_elr: u64, _spsr: u64) {
    trace!("irq_current");
    let intid =
        GicV3::get_and_acknowledge_interrupt().expect("No pending interrupt");
    info!("IRQ {intid:?}");
}

// SAFETY: There is no other global function of this name.
extern "C" fn fiq_current(_elr: u64, _spsr: u64) {
    error!("fiq_current");
    system_off::<Hvc>().unwrap();
}

// SAFETY: There is no other global function of this name.
extern "C" fn serr_current(_elr: u64, _spsr: u64) {
    error!("serr_current");
    system_off::<Hvc>().unwrap();
}

// SAFETY: There is no other global function of this name.
extern "C" fn sync_lower(_elr: u64, _spsr: u64) {
    error!("sync_lower");
    system_off::<Hvc>().unwrap();
}

// SAFETY: There is no other global function of this name.
extern "C" fn irq_lower(_elr: u64, _spsr: u64) {
    error!("irq_lower");
    system_off::<Hvc>().unwrap();
}

// SAFETY: There is no other global function of this name.
extern "C" fn fiq_lower(_elr: u64, _spsr: u64) {
    error!("fiq_lower");
    system_off::<Hvc>().unwrap();
}

// SAFETY: There is no other global function of this name.
```

```
extern "C" fn serr_lower(_elr: u64, _spsr: u64) {
    error!("serr_lower");
    system_off::<Hvc>().unwrap();
}
```

src/logger.rs (вам не потрібно це змінювати):

```
// Copyright 2023 Google LLC
//
// Licensed under the Apache License, Version 2.0 (the "License");
// you may not use this file except in compliance with the License.
// You may obtain a copy of the License at
//
//      http://www.apache.org/licenses/LICENSE-2.0
//
// Unless required by applicable law or agreed to in writing, software
// distributed under the License is distributed on an "AS IS" BASIS,
// WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
// See the License for the specific language governing permissions and
// limitations under the License.

// ANCHOR: main
use crate::pl011::Uart;
use core::fmt::Write;
use log::{LevelFilter, Log, Metadata, Record, SetLoggerError};
use spin::mutex::SpinMutex;

static LOGGER: Logger = Logger { uart: SpinMutex::new(None) };

struct Logger {
    uart: SpinMutex<Option<Uart>>,
}

impl Log for Logger {
    fn enabled(&self, _metadata: &Metadata) -> bool {
        true
    }

    fn log(&self, record: &Record) {
        writeln!(
            self.uart.lock().as_mut().unwrap(),
            "[{}] {}",
            record.level(),
            record.args()
        )
        .unwrap();
    }

    fn flush(&self) {}
}

/// Initialises UART logger.
```

```
pub fn init(uart: Uart, max_level: LevelFilter) -> Result<(), SetLoggerError> {
    LOGGER.uart.lock().replace(uart);

    log::set_logger(&LOGGER)?;
    log::set_max_level(max_level);
    Ok(())
}
```

src/pl011.rs (вам не потрібно це змінювати):

```
// Copyright 2023 Google LLC
//
// Licensed under the Apache License, Version 2.0 (the "License");
// you may not use this file except in compliance with the License.
// You may obtain a copy of the License at
//
//      http://www.apache.org/licenses/LICENSE-2.0
//
// Unless required by applicable law or agreed to in writing, software
// distributed under the License is distributed on an "AS IS" BASIS,
// WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
// See the License for the specific language governing permissions and
// limitations under the License.


use core::fmt::{self, Write};

// ANCHOR: Flags
use bitflags::bitflags;

bitflags! {
    /// Flags from the UART flag register.
    struct Flags: u16 {
        /// Clear to send.
        const CTS = 1 << 0;
        /// Data set ready.
        const DSR = 1 << 1;
        /// Data carrier detect.
        const DCD = 1 << 2;
        /// UART busy transmitting data.
        const BUSY = 1 << 3;
        /// Receive FIFO is empty.
        const RXFE = 1 << 4;
        /// Transmit FIFO is full.
        const TXFF = 1 << 5;
        /// Receive FIFO is full.
        const RXFF = 1 << 6;
        /// Transmit FIFO is empty.
        const TXFE = 1 << 7;
        /// Ring indicator.
        const RI = 1 << 8;
    }
```

```
}
// ANCHOR_END: Flags

bitflags! {
    /// Flags from the UART Receive Status Register / Error Clear Register.
    struct ReceiveStatus: u16 {
        /// Framing error.
        const FE = 1 << 0;
        /// Parity error.
        const PE = 1 << 1;
        /// Break error.
        const BE = 1 << 2;
        /// Overrun error.
        const OE = 1 << 3;
    }
}

// ANCHOR: Registers
struct Registers {
    dr: u16,
    _reserved0: [u8; 2],
    rsr: ReceiveStatus,
    _reserved1: [u8; 19],
    fr: Flags,
    _reserved2: [u8; 6],
    ilpr: u8,
    _reserved3: [u8; 3],
    ibrd: u16,
    _reserved4: [u8; 2],
    fbrd: u8,
    _reserved5: [u8; 3],
    lcr_h: u8,
    _reserved6: [u8; 3],
    cr: u16,
    _reserved7: [u8; 3],
    ifls: u8,
    _reserved8: [u8; 3],
    imsc: u16,
    _reserved9: [u8; 2],
    ris: u16,
    _reserved10: [u8; 2],
    mis: u16,
    _reserved11: [u8; 2],
    icr: u16,
    _reserved12: [u8; 2],
    dmacr: u8,
    _reserved13: [u8; 3],
}
// ANCHOR_END: Registers

// ANCHOR: Uart
```

```rust
/// Driver for a PL011 UART.
pub struct Uart {
    registers: *mut Registers,
}

impl Uart {
    /// Constructs a new instance of the UART driver for a PL011 device at the
    /// given base address.
    ///
    /// # Safety
    ///
    /// The given base address must point to the MMIO control registers of a
    /// PL011 device, which must be mapped into the address space of the process
    /// as device memory and not have any other aliases.
    pub unsafe fn new(base_address: *mut u32) -> Self {
        Self { registers: base_address as *mut Registers }
    }

    /// Writes a single byte to the UART.
    pub fn write_byte(&self, byte: u8) {
        // Wait until there is room in the TX buffer.
        while self.read_flag_register().contains(Flags::TXFF) {}

        // SAFETY: We know that self.registers points to the control registers
        // of a PL011 device which is appropriately mapped.
        unsafe {
            // Write to the TX buffer.
            (&raw mut (*self.registers).dr).write_volatile(byte.into());
        }

        // Wait until the UART is no longer busy.
        while self.read_flag_register().contains(Flags::BUSY) {}
    }

    /// Reads and returns a pending byte, or `None` if nothing has been
    /// received.
    pub fn read_byte(&self) -> Option<u8> {
        if self.read_flag_register().contains(Flags::RXFE) {
            None
        } else {
            // SAFETY: We know that self.registers points to the control
            // registers of a PL011 device which is appropriately mapped.
            let data = unsafe { (&raw const (*self.registers).dr).read_volatile() };
            // TODO: Check for error conditions in bits 8-11.
            Some(data as u8)
        }
    }

    fn read_flag_register(&self) -> Flags {
        // SAFETY: We know that self.registers points to the control registers
        // of a PL011 device which is appropriately mapped.
```

```
        unsafe { (&raw const (*self.registers).fr).read_volatile() }
    }
}
// ANCHOR_END: Uart

impl Write for Uart {
    fn write_str(&mut self, s: &str) -> fmt::Result {
        for c in s.as_bytes() {
            self.write_byte(*c);
        }
        Ok(())
    }
}

// Safe because it just contains a pointer to device memory, which can be
// accessed from any context.
unsafe impl Send for Uart {}
```

Cargo.toml (це не потрібно змінювати):

```
[workspace]

[package]
name = "rtc"
version = "0.1.0"
edition = "2021"
publish = false

[dependencies]
arm-gic = "0.1.2"
bitflags = "2.6.0"
chrono = { version = "0.4.38", default-features = false }
log = "0.4.22"
smccc = "0.1.1"
spin = "0.9.8"

[build-dependencies]
cc = "1.2.2"
```

build.rs (вам не потрібно це змінювати):

```
// Copyright 2023 Google LLC
//
// Licensed under the Apache License, Version 2.0 (the "License");
// you may not use this file except in compliance with the License.
// You may obtain a copy of the License at
//
//      http://www.apache.org/licenses/LICENSE-2.0
//
// Unless required by applicable law or agreed to in writing, software
// distributed under the License is distributed on an "AS IS" BASIS,
// WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
// See the License for the specific language governing permissions and
```

```
// limitations under the License.

use cc::Build;
use std::env;

fn main() {
    env::set_var("CROSS_COMPILE", "aarch64-none-elf");
    env::set_var("CC", "clang");

    Build::new()
        .file("entry.S")
        .file("exceptions.S")
        .file("idmap.S")
        .compile("empty")
}
```

entry.S (вам не потрібно це змінювати):

```
/*
 * Copyright 2023 Google LLC
 *
 * Licensed under the Apache License, Version 2.0 (the "License");
 * you may not use this file except in compliance with the License.
 * You may obtain a copy of the License at
 *
 *     https://www.apache.org/licenses/LICENSE-2.0
 *
 * Unless required by applicable law or agreed to in writing, software
 * distributed under the License is distributed on an "AS IS" BASIS,
 * WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
 * See the License for the specific language governing permissions and
 * limitations under the License.
 */

.macro adr_l, reg:req, sym:req
    adrp \reg, \sym
    add \reg, \reg, :lo12:\sym
.endm

.macro mov_i, reg:req, imm:req
    movz \reg, :abs_g3:\imm
    movk \reg, :abs_g2_nc:\imm
    movk \reg, :abs_g1_nc:\imm
    movk \reg, :abs_g0_nc:\imm
.endm

.set .L_MAIR_DEV_nGnRE, 0x04
.set .L_MAIR_MEM_WBWA,  0xff
.set .Lmairval, .L_MAIR_DEV_nGnRE | (.L_MAIR_MEM_WBWA << 8)

/* 4 KiB granule size for TTBR0_EL1. */
.set .L_TCR_TG0_4KB, 0x0 << 14
```

```
/* 4 KiB granule size for TTBR1_EL1. */
.set .L_TCR_TG1_4KB, 0x2 << 30
/* Disable translation table walk for TTBR1_EL1, generating a translation fault instead
.set .L_TCR_EPD1, 0x1 << 23
/* Translation table walks for TTBR0_EL1 are inner sharable. */
.set .L_TCR_SH_INNER, 0x3 << 12
/*
 * Translation table walks for TTBR0_EL1 are outer write-back read-allocate write-alloca
 * cacheable.
 */
.set .L_TCR_RGN_OWB, 0x1 << 10
/*
 * Translation table walks for TTBR0_EL1 are inner write-back read-allocate write-alloca
 * cacheable.
 */
.set .L_TCR_RGN_IWB, 0x1 << 8
/* Size offset for TTBR0_EL1 is 2**39 bytes (512 GiB). */
.set .L_TCR_T0SZ_512, 64 - 39
.set .Ltcrval, .L_TCR_TG0_4KB | .L_TCR_TG1_4KB | .L_TCR_EPD1 | .L_TCR_RGN_OWB
.set .Ltcrval, .Ltcrval | .L_TCR_RGN_IWB | .L_TCR_SH_INNER | .L_TCR_T0SZ_512

/* Stage 1 instruction access cacheability is unaffected. */
.set .L_SCTLR_ELx_I, 0x1 << 12
/* SP alignment fault if SP is not aligned to a 16 byte boundary. */
.set .L_SCTLR_ELx_SA, 0x1 << 3
/* Stage 1 data access cacheability is unaffected. */
.set .L_SCTLR_ELx_C, 0x1 << 2
/* EL0 and EL1 stage 1 MMU enabled. */
.set .L_SCTLR_ELx_M, 0x1 << 0
/* Privileged Access Never is unchanged on taking an exception to EL1. */
.set .L_SCTLR_EL1_SPAN, 0x1 << 23
/* SETEND instruction disabled at EL0 in aarch32 mode. */
.set .L_SCTLR_EL1_SED, 0x1 << 8
/* Various IT instructions are disabled at EL0 in aarch32 mode. */
.set .L_SCTLR_EL1_ITD, 0x1 << 7
.set .L_SCTLR_EL1_RES1, (0x1 << 11) | (0x1 << 20) | (0x1 << 22) | (0x1 << 28) | (0x1 <<
.set .Lsctlrval, .L_SCTLR_ELx_M | .L_SCTLR_ELx_C | .L_SCTLR_ELx_SA | .L_SCTLR_EL1_ITD |
.set .Lsctlrval, .Lsctlrval | .L_SCTLR_ELx_I | .L_SCTLR_EL1_SPAN | .L_SCTLR_EL1_RES1

/**
 * This is a generic entry point for an image. It carries out the operations required to
 * loaded image to be run. Specifically, it zeroes the bss section using registers x25 a
 * prepares the stack, enables floating point, and sets up the exception vector. It pres
 * for the Rust entry point, as these may contain boot parameters.
 */
.section .init.entry, "ax"
.global entry
entry:
    /* Load and apply the memory management configuration, ready to enable MMU and cache
    adrp x30, idmap
    msr ttbr0_el1, x30
```

```
    mov_i x30, .Lmairval
    msr mair_el1, x30

    mov_i x30, .Ltcrval
    /* Copy the supported PA range into TCR_EL1.IPS. */
    mrs x29, id_aa64mmfr0_el1
    bfi x30, x29, #32, #4

    msr tcr_el1, x30

    mov_i x30, .Lsctlrval

    /*
     * Ensure everything before this point has completed, then invalidate any potentiall
     * local TLB entries before they start being used.
     */
    isb
    tlbi vmalle1
    ic iallu
    dsb nsh
    isb

    /*
     * Configure sctlr_el1 to enable MMU and cache and don't proceed until this has comp
     */
    msr sctlr_el1, x30
    isb

    /* Disable trapping floating point access in EL1. */
    mrs x30, cpacr_el1
    orr x30, x30, #(0x3 << 20)
    msr cpacr_el1, x30
    isb

    /* Zero out the bss section. */
    adr_l x29, bss_begin
    adr_l x30, bss_end
0:  cmp x29, x30
    b.hs 1f
    stp xzr, xzr, [x29], #16
    b 0b

1:  /* Prepare the stack. */
    adr_l x30, boot_stack_end
    mov sp, x30

    /* Set up exception vector. */
    adr x30, vector_table_el1
    msr vbar_el1, x30
```

```
    /* Call into Rust code. */
    bl main

    /* Loop forever waiting for interrupts. */
2:  wfi
    b 2b
```

exceptions.S (вам не потрібно це змінювати):

```
/*
 * Copyright 2023 Google LLC
 *
 * Licensed under the Apache License, Version 2.0 (the "License");
 * you may not use this file except in compliance with the License.
 * You may obtain a copy of the License at
 *
 *     https://www.apache.org/licenses/LICENSE-2.0
 *
 * Unless required by applicable law or agreed to in writing, software
 * distributed under the License is distributed on an "AS IS" BASIS,
 * WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
 * See the License for the specific language governing permissions and
 * limitations under the License.
 */

/**
 * Saves the volatile registers onto the stack. This currently takes 14
 * instructions, so it can be used in exception handlers with 18 instructions
 * left.
 *
 * On return, x0 and x1 are initialised to elr_el2 and spsr_el2 respectively,
 * which can be used as the first and second arguments of a subsequent call.
 */
.macro save_volatile_to_stack
    /* Reserve stack space and save registers x0-x18, x29 & x30. */
    stp x0, x1, [sp, #-(8 * 24)]!
    stp x2, x3, [sp, #8 * 2]
    stp x4, x5, [sp, #8 * 4]
    stp x6, x7, [sp, #8 * 6]
    stp x8, x9, [sp, #8 * 8]
    stp x10, x11, [sp, #8 * 10]
    stp x12, x13, [sp, #8 * 12]
    stp x14, x15, [sp, #8 * 14]
    stp x16, x17, [sp, #8 * 16]
    str x18, [sp, #8 * 18]
    stp x29, x30, [sp, #8 * 20]

    /*
     * Save elr_el1 & spsr_el1. This such that we can take nested exception
     * and still be able to unwind.
     */
    mrs x0, elr_el1
```

```
    mrs x1, spsr_el1
    stp x0, x1, [sp, #8 * 22]
.endm

/**
 * Restores the volatile registers from the stack. This currently takes 14
 * instructions, so it can be used in exception handlers while still leaving 18
 * instructions left; if paired with save_volatile_to_stack, there are 4
 * instructions to spare.
 */
.macro restore_volatile_from_stack
    /* Restore registers x2-x18, x29 & x30. */
    ldp x2, x3, [sp, #8 * 2]
    ldp x4, x5, [sp, #8 * 4]
    ldp x6, x7, [sp, #8 * 6]
    ldp x8, x9, [sp, #8 * 8]
    ldp x10, x11, [sp, #8 * 10]
    ldp x12, x13, [sp, #8 * 12]
    ldp x14, x15, [sp, #8 * 14]
    ldp x16, x17, [sp, #8 * 16]
    ldr x18, [sp, #8 * 18]
    ldp x29, x30, [sp, #8 * 20]

    /* Restore registers elr_el1 & spsr_el1, using x0 & x1 as scratch. */
    ldp x0, x1, [sp, #8 * 22]
    msr elr_el1, x0
    msr spsr_el1, x1

    /* Restore x0 & x1, and release stack space. */
    ldp x0, x1, [sp], #8 * 24
.endm

/**
 * This is a generic handler for exceptions taken at the current EL while using
 * SP0. It behaves similarly to the SPx case by first switching to SPx, doing
 * the work, then switching back to SP0 before returning.
 *
 * Switching to SPx and calling the Rust handler takes 16 instructions. To
 * restore and return we need an additional 16 instructions, so we can implement
 * the whole handler within the allotted 32 instructions.
 */
.macro current_exception_sp0 handler:req
    msr spsel, #1
    save_volatile_to_stack
    bl \handler
    restore_volatile_from_stack
    msr spsel, #0
    eret
.endm

/**
```

```
 * This is a generic handler for exceptions taken at the current EL while using
 * SPx. It saves volatile registers, calls the Rust handler, restores volatile
 * registers, then returns.
 *
 * This also works for exceptions taken from EL0, if we don't care about
 * non-volatile registers.
 *
 * Saving state and jumping to the Rust handler takes 15 instructions, and
 * restoring and returning also takes 15 instructions, so we can fit the whole
 * handler in 30 instructions, under the limit of 32.
 */
.macro current_exception_spx handler:req
    save_volatile_to_stack
    bl \handler
    restore_volatile_from_stack
    eret
.endm

.section .text.vector_table_el1, "ax"
.global vector_table_el1
.balign 0x800
vector_table_el1:
sync_cur_sp0:
    current_exception_sp0 sync_exception_current

.balign 0x80
irq_cur_sp0:
    current_exception_sp0 irq_current

.balign 0x80
fiq_cur_sp0:
    current_exception_sp0 fiq_current

.balign 0x80
serr_cur_sp0:
    current_exception_sp0 serr_current

.balign 0x80
sync_cur_spx:
    current_exception_spx sync_exception_current

.balign 0x80
irq_cur_spx:
    current_exception_spx irq_current

.balign 0x80
fiq_cur_spx:
    current_exception_spx fiq_current

.balign 0x80
serr_cur_spx:
```

```
    current_exception_spx serr_current

.balign 0x80
sync_lower_64:
    current_exception_spx sync_lower

.balign 0x80
irq_lower_64:
    current_exception_spx irq_lower

.balign 0x80
fiq_lower_64:
    current_exception_spx fiq_lower

.balign 0x80
serr_lower_64:
    current_exception_spx serr_lower

.balign 0x80
sync_lower_32:
    current_exception_spx sync_lower

.balign 0x80
irq_lower_32:
    current_exception_spx irq_lower

.balign 0x80
fiq_lower_32:
    current_exception_spx fiq_lower

.balign 0x80
serr_lower_32:
    current_exception_spx serr_lower
```

idmap.S (вам не потрібно це змінювати):

```
/*
 * Copyright 2023 Google LLC
 *
 * Licensed under the Apache License, Version 2.0 (the "License");
 * you may not use this file except in compliance with the License.
 * You may obtain a copy of the License at
 *
 *     https://www.apache.org/licenses/LICENSE-2.0
 *
 * Unless required by applicable law or agreed to in writing, software
 * distributed under the License is distributed on an "AS IS" BASIS,
 * WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
 * See the License for the specific language governing permissions and
 * limitations under the License.
 */
```

```
.set .L_TT_TYPE_BLOCK, 0x1
.set .L_TT_TYPE_PAGE,  0x3
.set .L_TT_TYPE_TABLE, 0x3

/* Access flag. */
.set .L_TT_AF, 0x1 << 10
/* Not global. */
.set .L_TT_NG, 0x1 << 11
.set .L_TT_XN, 0x3 << 53

.set .L_TT_MT_DEV, 0x0 << 2         // MAIR #0 (DEV_nGnRE)
.set .L_TT_MT_MEM, (0x1 << 2) | (0x3 << 8)  // MAIR #1 (MEM_WBWA), inner shareable

.set .L_BLOCK_DEV, .L_TT_TYPE_BLOCK | .L_TT_MT_DEV | .L_TT_AF | .L_TT_XN
.set .L_BLOCK_MEM, .L_TT_TYPE_BLOCK | .L_TT_MT_MEM | .L_TT_AF | .L_TT_NG

.section ".rodata.idmap", "a", %progbits
.global idmap
.align 12
idmap:
    /* level 1 */
    .quad       .L_BLOCK_DEV | 0x0          // 1 GiB of device mappings
    .quad       .L_BLOCK_MEM | 0x40000000   // 1 GiB of DRAM
    .fill       254, 8, 0x0         // 254 GiB of unmapped VA space
    .quad       .L_BLOCK_DEV | 0x4000000000 // 1 GiB of device mappings
    .fill       255, 8, 0x0         // 255 GiB of remaining VA space
```

image.ld (вам не потрібно це змінювати):

```
/*
 * Copyright 2023 Google LLC
 *
 * Licensed under the Apache License, Version 2.0 (the "License");
 * you may not use this file except in compliance with the License.
 * You may obtain a copy of the License at
 *
 *     https://www.apache.org/licenses/LICENSE-2.0
 *
 * Unless required by applicable law or agreed to in writing, software
 * distributed under the License is distributed on an "AS IS" BASIS,
 * WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
 * See the License for the specific language governing permissions and
 * limitations under the License.
 */

/*
 * Code will start running at this symbol which is placed at the start of the
 * image.
 */
ENTRY(entry)

MEMORY
```

```
{
    image : ORIGIN = 0x40080000, LENGTH = 2M
}

SECTIONS
{
    /*
     * Collect together the code.
     */
    .init : ALIGN(4096) {
        text_begin = .;
        *(.init.entry)
        *(.init.*)
    } >image
    .text : {
        *(.text.*)
    } >image
    text_end = .;

    /*
     * Collect together read-only data.
     */
    .rodata : ALIGN(4096) {
        rodata_begin = .;
        *(.rodata.*)
    } >image
    .got : {
        *(.got)
    } >image
    rodata_end = .;

    /*
     * Collect together the read-write data including .bss at the end which
     * will be zero'd by the entry code.
     */
    .data : ALIGN(4096) {
        data_begin = .;
        *(.data.*)
        /*
         * The entry point code assumes that .data is a multiple of 32
         * bytes long.
         */
        . = ALIGN(32);
        data_end = .;
    } >image

    /* Everything beyond this point will not be included in the binary. */
    bin_end = .;

    /* The entry point code assumes that .bss is 16-byte aligned. */
    .bss : ALIGN(16)  {
```

```
        bss_begin = .;
        *(.bss.*)
        *(COMMON)
        . = ALIGN(16);
        bss_end = .;
    } >image

    .stack (NOLOAD) : ALIGN(4096) {
        boot_stack_begin = .;
        . += 40 * 4096;
        . = ALIGN(4096);
        boot_stack_end = .;
    } >image

    . = ALIGN(4K);
    PROVIDE(dma_region = .);

    /*
     * Remove unused sections from the image.
     */
    /DISCARD/ : {
        /* The image loads itself so doesn't need these sections. */
        *(.gnu.hash)
        *(.hash)
        *(.interp)
        *(.eh_frame_hdr)
        *(.eh_frame)
        *(.note.gnu.build-id)
    }
}
```

Makefile (вам не потрібно це змінювати):

```
# Copyright 2023 Google LLC
#
# Licensed under the Apache License, Version 2.0 (the "License");
# you may not use this file except in compliance with the License.
# You may obtain a copy of the License at
#
#      http://www.apache.org/licenses/LICENSE-2.0
#
# Unless required by applicable law or agreed to in writing, software
# distributed under the License is distributed on an "AS IS" BASIS,
# WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
# See the License for the specific language governing permissions and
# limitations under the License.

.PHONY: build qemu_minimal qemu qemu_logger

all: rtc.bin

build:
```

```
    cargo build

rtc.bin: build
    cargo objcopy -- -O binary $@

qemu: rtc.bin
    qemu-system-aarch64 -machine virt,gic-version=3 -cpu max -serial mon:stdio -display

clean:
    cargo clean
    rm -f *.bin
```

.cargo/config.toml (це не потрібно змінювати):

```
[build]
target = "aarch64-unknown-none"
rustflags = ["-C", "link-arg=-Timage.ld"]
```

Запустіть код у QEMU за допомогою `make qemu`.

55.2 Rust на голому залізі. Полудень.

RTC драйвер

(назад до вправи)

main.rs:

```
mod exceptions;
mod logger;
mod pl011;
mod pl031;

use crate::pl031::Rtc;
use arm_gic::gicv3::{IntId, Trigger};
use arm_gic::{irq_enable, wfi};
use chrono::{TimeZone, Utc};
use core::hint::spin_loop;
use crate::pl011::Uart;
use arm_gic::gicv3::GicV3;
use core::panic::PanicInfo;
use log::{error, info, trace, LevelFilter};
use smccc::psci::system_off;
use smccc::Hvc;

/// Базові адреси GICv3.
const GICD_BASE_ADDRESS: *mut u64 = 0x800_0000 as _;
const GICR_BASE_ADDRESS: *mut u64 = 0x80A_0000 as _;

/// Базова адреса основного PL011 UART.
const PL011_BASE_ADDRESS: *mut u32 = 0x900_0000 as _;
```

```rust
/// Базова адреса PL031 RTC.
const PL031_BASE_ADDRESS: *mut u32 = 0x901_0000 as _;
/// IRQ, що використовується PL031 RTC.
const PL031_IRQ: IntId = IntId::spi(2);

// БЕЗПЕКА: Не існує іншої глобальної функції з таким іменем.
extern "C" fn main(x0: u64, x1: u64, x2: u64, x3: u64) {
    // БЕЗПЕКА: `PL011_BASE_ADDRESS` є базовою адресою пристрою PL011,
    // і ніщо інше не має доступу до цього діапазону адресації.
    let uart = unsafe { Uart::new(PL011_BASE_ADDRESS) };
    logger::init(uart, LevelFilter::Trace).unwrap();

    info!("main({:#x}, {:#x}, {:#x}, {:#x})", x0, x1, x2, x3);

    // БЕЗПЕКА: `GICD_BASE_ADDRESS` і `GICR_BASE_ADDRESS` є базовими
    // адресами дистриб'ютора і редистриб'ютора GICv3 відповідно, і ніщо
    // інше не має доступу до цих адресних діапазонів.
    let mut gic = unsafe { GicV3::new(GICD_BASE_ADDRESS, GICR_BASE_ADDRESS) };
    gic.setup();

    // БЕЗПЕКА: `PL031_BASE_ADDRESS` є базовою адресою пристрою PL031,
    // і ніщо інше не має доступу до цього діапазону адресації.
    let mut rtc = unsafe { Rtc::new(PL031_BASE_ADDRESS) };
    let timestamp = rtc.read();
    let time = Utc.timestamp_opt(timestamp.into(), 0).unwrap();
    info!("RTC: {time}");

    GicV3::set_priority_mask(0xff);
    gic.set_interrupt_priority(PL031_IRQ, 0x80);
    gic.set_trigger(PL031_IRQ, Trigger::Level);
    irq_enable();
    gic.enable_interrupt(PL031_IRQ, true);

    // Чекаємо 3 секунди, без переривань.
    let target = timestamp + 3;
    rtc.set_match(target);
    info!("Чекаємо на {}", Utc.timestamp_opt(target.into(), 0).unwrap());
    trace!(
        "matched={}, interrupt_pending={}",
        rtc.matched(),
        rtc.interrupt_pending()
    );
    while !rtc.matched() {
        spin_loop();
    }
    trace!(
        "matched={}, interrupt_pending={}",
        rtc.matched(),
        rtc.interrupt_pending()
    );
```

```
    info!("Дочекалися");

    // Чекаємо ще 3 секунди на переривання.
    let target = timestamp + 6;
    info!("Чекаємо на {}", Utc.timestamp_opt(target.into(), 0).unwrap());
    rtc.set_match(target);
    rtc.clear_interrupt();
    rtc.enable_interrupt(true);
    trace!(
        "matched={}, interrupt_pending={}",
        rtc.matched(),
        rtc.interrupt_pending()
    );
    while !rtc.interrupt_pending() {
        wfi();
    }
    trace!(
        "matched={}, interrupt_pending={}",
        rtc.matched(),
        rtc.interrupt_pending()
    );
    info!("Дочекалися");

    system_off::<Hvc>().unwrap();
}

fn panic(info: &PanicInfo) -> ! {
    error!("{info}");
    system_off::<Hvc>().unwrap();
    loop {}
}
```

pl031.rs:

```
struct Registers {
    /// Регістр даних
    dr: u32,
    /// Регістр збігів
    mr: u32,
    /// Регістр завантаження
    lr: u32,
    /// Регістр управління
    cr: u8,
    _reserved0: [u8; 3],
    /// Регістр установки або очищення маски переривання
    imsc: u8,
    _reserved1: [u8; 3],
    /// Необроблений стан переривання
    ris: u8,
    _reserved2: [u8; 3],
    /// Маскований статус переривання
    mis: u8,
```

```
    _reserved3: [u8; 3],
    /// Регістр очищення переривання
    icr: u8,
    _reserved4: [u8; 3],
}

/// Драйвер для годинника реального часу PL031.
pub struct Rtc {
    registers: *mut Registers,
}

impl Rtc {
    /// Створює новий екземпляр драйвера RTC для пристрою PL031 за заданою
    /// базовою адресою.
    ///
    /// # Безпека
    ///
    /// Вказана базова адреса має вказувати на регістри керування MMIO
    /// пристрою PL031, які мають бути відображені у адресному просторі процесу
    /// як пам'ять пристрою і не мати інших псевдонімів.
    pub unsafe fn new(base_address: *mut u32) -> Self {
        Self { registers: base_address as *mut Registers }
    }

    /// Зчитує поточне значення RTC.
    pub fn read(&self) -> u32 {
        // БЕЗПЕКА: ми знаємо, що self.registers вказує на керуючі
        // регістри пристрою PL031, який відповідним чином відображено.
        unsafe { (&raw const (*self.registers).dr).read_volatile() }
    }

    /// Записує значення збігу. Коли значення RTC збігається з цим, буде
    /// згенеровано переривання (якщо його увімкнено).
    pub fn set_match(&mut self, value: u32) {
        // БЕЗПЕКА: ми знаємо, що self.registers вказує на керуючі
        // регістри пристрою PL031, який відповідним чином відображено.
        unsafe { (&raw mut (*self.registers).mr).write_volatile(value) }
    }

    /// Повертає, чи відповідає регістр збігу значенню RTC, незалежно від того,
    /// увімкнено переривання чи ні.
    pub fn matched(&self) -> bool {
        // БЕЗПЕКА: ми знаємо, що self.registers вказує на керуючі
        // регістри пристрою PL031, який відповідним чином відображено.
        let ris = unsafe { (&raw const (*self.registers).ris).read_volatile() };
        (ris & 0x01) != 0
    }

    /// Повертає, чи є переривання в очікуванні.
    ///
    /// Це значення має бути істинним тоді і тільки тоді, коли `matched`
```

```
    /// повертає істину і переривання замасковане.
    pub fn interrupt_pending(&self) -> bool {
        // БЕЗПЕКА: ми знаємо, що self.registers вказує на керуючі
        // регістри пристрою PL031, який відповідним чином відображено.
        let ris = unsafe { (&raw const (*self.registers).mis).read_volatile() };
        (ris & 0x01) != 0
    }

    /// Встановлює або очищує маску переривання.
    ///
    /// Якщо маска дорівнює істині, переривання увімкнено; якщо ні -
    /// переривання вимкнено.
    pub fn enable_interrupt(&mut self, mask: bool) {
        let imsc = if mask { 0x01 } else { 0x00 };
        // БЕЗПЕКА: ми знаємо, що self.registers вказує на керуючі
        // регістри пристрою PL031, який відповідним чином відображено.
        unsafe { (&raw mut (*self.registers).imsc).write_volatile(imsc) }
    }

    /// Очищає очікуване переривання, якщо таке є.
    pub fn clear_interrupt(&mut self) {
        // БЕЗПЕКА: ми знаємо, що self.registers вказує на керуючі
        // регістри пристрою PL031, який відповідним чином відображено.
        unsafe { (&raw mut (*self.registers).icr).write_volatile(0x01) }
    }
}

// БЕЗПЕКА: `Rtc` просто містить вказівник на пам'ять пристрою, до якого
// можна отримати доступ з будь-якого контексту.
unsafe impl Send for Rtc {}
```

Частина XIII

Одночасність виконання: Ранок

Розділ 56

Ласкаво просимо до одночасних обчислень у Rust

Rust має повну підтримку паралелізму та одночасних обчислень за допомогою потоків ОС із м'ютексами та каналами.

Система типів у Rust відіграє важливу роль у тому, що дозволяє зробити багато помилок з одночасним виконанням помилками часу компіляції. Це часто називають *безстрашним одночасним виконанням*, оскільки ви можете покластися на компілятор для забезпечення коректності під час виконання.

Розклад

Враховуючи 10-хвилинні перерви, ця сесія має тривати близько 3 годин та 20 хвилин. Вона містить:

Сегмент	Тривалість
Потоки	30 хвилин
Канали	20 хвилин
Send та Sync	15 хвилин
Спільний стан	30 хвилин
Вправи	1 година та 10 хвилин

- Rust дозволяє нам отримати доступ до інструментарію одночасності виконнання ОС: потоків, примітивів синхронізації тощо.
- Система типів дає нам безпеку для одночасного виконання без будь-яких спеціальних функцій.
- Ті самі інструменти, які допомагають з "одночасним" доступом в одному потоці (наприклад, викликана функція, яка може змінювати аргумент або зберігати посилання на нього, щоб прочитати пізніше), позбавляють нас від проблем багатопотоковості.

Розділ 57

Потоки

Цей сегмент повинен зайняти близько 30 хвилин. Він містить:

Слайд	Тривалість
Звичайні потоки	15 хвилин
Потоки з областю видимості	15 хвилин

57.1 Звичайні потоки

Потоки Rust працюють так само, як і в інших мовах:

```
use std::thread;
use std::time::Duration;

fn main() {
    thread::spawn(|| {
        for i in 0..10 {
            println!("Підрахунок в потоці: {i}!");
            thread::sleep(Duration::from_millis(5));
        }
    });

    for i in 0..5 {
        println!("Головний потік: {i}");
        thread::sleep(Duration::from_millis(5));
    }
}
```

- Породження нових потоків не призводить до автоматичної затримки завершення програми в кінці `main`.
- Паніка потоків не залежить одна від одної.
  - Паніки можуть нести корисне навантаження, яке можна розпакувати за допомогою `Any::downcast_ref`.
- Запустіть приклад.

  - Таймінг 5 мс є достатньо вільним, щоб головний і породжений потоки залишалися переважно в одному ритмі.
  - Зверніть увагу, що програма завершується до того, як породжений потік досягне 10!
  - Це тому, що `main` завершує програму, а породжені потоки не змушують її продовжувати.
    * За бажанням можна порівняти з `pthreads`/C++ `std::thread`/`boost::thread`.
- Як нам дочекатися завершення породженого потоку?
- `thread::spawn` повертає `JoinHandle`. Перегляньте документацію.
  - У `JoinHandle` є метод `.join()`, який блокує.
- Використовуйте `let handle = thread::spawn(...)`, а потім `handle.join()`, щоб дочекатися завершення потоку і змусити програму дорахувати до 10..
- А що, якщо ми хочемо повернути значення?
- Перегляньте документацію ще раз:
  - Закриття `thread::spawn` повертає `T`.
  - `JoinHandle` `.join()` повертає `thread::Result<T>`
- Використовуйте значення `Result`, що повертається з `handle.join()`, щоб отримати доступ до значення, що повертається.
- Гаразд, а як щодо іншого випадку?
  - Викликає паніку в потоці. Зауважте, що це не впливає на `main`.
  - Дає доступ до корисного навантаження паніки. Це гарний час, щоб поговорити про `Any`.
- Тепер ми можемо повертати значення з потоків! А як щодо отримання вхідних даних?
  - Захоплюємо щось за посиланням у закритті потоку.
  - Повідомлення про помилку вказує на те, що ми повинні його перемістити.
  - Переміщуємо його, бачимо, що можемо обчислити, а потім повертаємо похідне значення.
- Якщо ми хочемо позичити?
  - Main вбиває дочірні потоки, коли повертається, але інша функція просто повернеться і залишить їх працювати.
  - Це буде використання стеку після повернення, що порушує безпеку пам'яті!
  - Як цього уникнути? Дивіться наступний слайд.

57.2 Потоки з областю видимості

Звичайні потоки не можуть запозичувати зі свого середовища:

```
use std::thread;

fn foo() {
    let s = String::from("Привіт");
    thread::spawn(|| {
```

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

fn main() {
    foo();
}
```

Однак для цього можна використовувати потік із обмеженою областю:

```
use std::thread;

fn foo() {
    let s = String::from("Привіт");
    thread::scope(|scope| {
        scope.spawn(|| {
            println!("Довжина: {}", s.len());
        });
    });
}

fn main() {
    foo();
}
```

- Причина цього полягає в тому, що коли функція `thread::scope` завершується, усі потоки гарантовано об'єднуються, тому вони можуть повертати запозичені дані.
- Застосовуються звичайні правила запозичення Rust: ви можете запозичувати або мутабельно одним потоком, або іммутабельно будь-якою кількістю потоків.

Розділ 58

Канали

Цей сегмент повинен зайняти близько 20 хвилин. Він містить:

Слайд	Тривалість
Відправники та отримувачі	10 хвилин
Незав'язані канали	2 хвилини
Зав'язані канали	10 хвилин

58.1 Відправники та отримувачі

Канали Rust мають дві частини: `Sender<T>` і `Receiver<T>`. Дві частини з'єднані через канал, але ви бачите лише кінцеві точки.

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

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

    tx.send(10).unwrap();
    tx.send(20).unwrap();

    println!("Прийнято: {:?}", rx.recv());
    println!("Прийнято: {:?}", rx.recv());

    let tx2 = tx.clone();
    tx2.send(30).unwrap();
    println!("Прийнято: {:?}", rx.recv());
}
```

- `mpsc` означає багато виробників, один споживач (Multi-Producer, Single-Consumer). `Sender` і `SyncSender` реалізують `Clone` (тобто ви можете створити кілька виробників), а `Receiver` — ні.
- `send()` і `recv()` повертають `Result`. Якщо вони повертають `Err`, це означає, що відповідний `Sender` або `Receiver` видалено, а канал закрито.

58.2 Незав'язані канали

Ви отримуєте необмежений і асинхронний канал за допомогою `mpsc::channel()`:

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

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

    thread::spawn(move || {
        let thread_id = thread::current().id();
        for i in 0..10 {
            tx.send(format!("Повідомлення {i}")).unwrap();
            println!("{thread_id:?}: надіслано Повідомлення {i}");
        }
        println!("{thread_id:?}: виконано");
    });
    thread::sleep(Duration::from_millis(100));

    for msg in rx.iter() {
        println!("Головний: отримав  {msg}");
    }
}
```

58.3 Зав'язані канали

З обмеженими (синхронними) каналами `send` може блокувати поточний потік:

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

fn main() {
    let (tx, rx) = mpsc::sync_channel(3);

    thread::spawn(move || {
        let thread_id = thread::current().id();
        for i in 0..10 {
            tx.send(format!("Повідомлення {i}")).unwrap();
            println!("{thread_id:?}: надіслано Повідомлення {i}");
        }
        println!("{thread_id:?}: виконано");
    });
    thread::sleep(Duration::from_millis(100));

    for msg in rx.iter() {
        println!("Головний: отримав  {msg}");
    }
}
```

- Виклик `send()` заблокує поточний потік, доки в каналі не залишиться місця для нового повідомлення. Потік може бути заблокований на невизначений термін, якщо ніхто не читає з каналу.
- Виклик `send()` буде перервано з помилкою (ось чому він повертає `Result`), якщо канал закрито. Канал закривається, коли отримувача видалено.
- Обмежений канал з нульовим розміром називається ”каналом зустрічі”. Кожне надсилання блокуватиме поточний потік, доки інший потік не викличе `recv`.

Розділ 59

Send та Sync

Цей сегмент повинен зайняти близько 15 хвилин. Він містить:

Слайд	Тривалість
Маркерні трейти	2 хвилини
Send	2 хвилини
Sync	2 хвилини
Приклади	10 хвилин

59.1 Маркерні трейти

Як Rust знає, що потрібно заборонити спільний доступ до потоків? Відповідь полягає у двох трейтах:

- Send: тип T є Send, якщо безпечно переміщувати T через межу потоку.
- Sync: тип T є Sync, якщо безпечно переміщувати &T через межу потоку.

Send та Sync є небезпечними трейтами. Компілятор автоматично виведе їх для ваших типів, якщо вони містять лише типи Send і Sync. Ви також можете реалізувати їх вручну, якщо знаєте, що це допустимо.

- Ці трейти можна розглядати як маркери того, що тип має певні властивості безпеки потоків.
- Їх можна використовувати в загальних обмеженнях як звичайні трейти.

59.2 Send

> Тип T є Send, якщо безпечно переміщати значення T в інший потік.

Наслідком перенесення права власності на інший потік є те, що *деструктори* будуть виконани в цьому потоці. Отже, питання полягає в тому, коли ви можете виділити значення в одному потоці та звільнити його в іншому.

Як приклад, підключення до бібліотеки SQLite має бути доступне лише з одного потоку.

59.3 Sync

> Тип `T` є Sync, якщо безпечно отримувати доступ до значення `T` з кількох потоків водночас.

Точніше, визначення таке:

> `T` є Sync тоді і тільки тоді коли `&T` є Send

Це твердження, по суті, є скороченим способом сказати, що якщо тип є потокобезпечним для спільного використання, також потоково безпечно передавати посилання на нього між потоками.

Це пояснюється тим, що якщо тип є Sync, це означає, що він може використовуватися кількома потоками без ризику перегонів даних або інших проблем із синхронізацією, тому його безпечно перемістити в інший потік. Посилання на тип також безпечно перемістити в інший потік, оскільки дані, на які воно посилається, можуть бути безпечно доступні з будь-якого потоку.

59.4 Приклади

`Send + Sync`

Більшість типів, які ви зустрічаєте, це `Send + Sync`:

- `i8`, `f32`, `bool`, `char`, `&str`, ...
- `(T1, T2)`, `[T; N]`, `&[T]`, `struct { x: T }`, ...
- `String`, `Option<T>`, `Vec<T>`, `Box<T>`, ...
- `Arc<T>`: явно потокобезпечний через кількість атомарних посилань.
- `Mutex<T>`: явно потокобезпечний через внутрішнє блокування.
- `mpsc::Sender<T>`: Починаючи з 1.72.0.
- `AtomicBool`, `AtomicU8`, ...: використовує спеціальні атомарні інструкції.

Загальні типи, як правило, є `Send + Sync, коли параметри типу є також Send + Sync`.

`Send + !Sync`

Ці типи можна переміщувати в інші потоки, але вони не є потокобезпечними. Зазвичай через внутрішню мутабільність:

- `mpsc::Receiver<T>`
- `Cell<T>`
- `RefCell<T>`

`!Send + Sync`

До цих типів можна безпечно отримати доступ (через спільні посилання) з декількох потоків, але їх не можна перемістити в інший потік:

- `MutexGuard<T: Sync>`: Використовує примітиви рівня ОС, які мають бути звільнені у потоці, що їх створив. Проте, вже заблокований м'ютекс може мати захищену змінну, яку може читати будь-який потік, з яким розділяється захист.

`!Send + !Sync`

Ці типи є потоконебезпечними і не можуть бути переміщені в інші потоки:

- `Rc<T>`: кожен `Rc<T>` має посилання на `RcBox<T>`, який містить неатомарний лічильник посилань.
- `*const T`, `*mut T`: Rust припускає, що необроблені покажчики можуть мати особливі міркування щодо одночасного використання.

Розділ 60

Спільний стан

Цей сегмент повинен зайняти близько 30 хвилин. Він містить:

Слайд	Тривалість
Arc	5 хвилин
Mutex	15 хвилин
Приклад	10 хвилин

60.1 `Arc`

`Arc<T>` дозволяє спільний доступ лише для читання через `Arc::clone`:

```
use std::sync::Arc;
use std::thread;

fn main() {
    let v = Arc::new(vec![10, 20, 30]);
    let mut handles = Vec::new();
    for _ in 0..5 {
        let v = Arc::clone(&v);
        handles.push(thread::spawn(move || {
            let thread_id = thread::current().id();
            println!("{thread_id:?}: {v:?}");
        }));
    }

    handles.into_iter().for_each(|h| h.join().unwrap());
    println!("v: {v:?}");
}
```

- `Arc` означає ”Atomic Reference Counted”, потокобезпечну версію `Rc`, яка використовує атомарні операції.
- `Arc<T>` реалізує `Clone` незалежно від того, чи `T` реалізує це. Він реалізує `Send` і `Sync` тоді і тільки тоді коли `T` реалізує їх обидва.

- `Arc::clone()` має вартість атомарних операцій, які виконуються, але після цього використання `T` є безкоштовним.
- Остерігайтеся циклів посилань, `Arc` не використовує збирач сміття для їх виявлення.
  - `std::sync::Weak` може допомогти.

60.2 `Mutex`

`Mutex<T>` забезпечує взаємовиключення *та* дозволяє мутабельний доступ до `T` за інтерфейсом лише для читання (інша форма внутрішньої мутабельності):

```
use std::sync::Mutex;

fn main() {
    let v = Mutex::new(vec![10, 20, 30]);
    println!("v: {:?}", v.lock().unwrap());

    {
        let mut guard = v.lock().unwrap();
        guard.push(40);
    }

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

Зверніть увагу, що ми маємо `impl<T: Send> Sync for Mutex<T>` загальну реалізацію.

- `Mutex` у Rust виглядає як колекція лише з одним елементом --- захищеними даними.
  - Неможливо забути отримати м'ютекс перед доступом до захищених даних.
- Ви можете отримати `&mut T` від `&Mutex<T>`, взявши блокування. `MutexGuard` гарантує, що `&mut T` не переживе утримуване блокування.
- `Mutex<T>` реалізує як `Send`, так і `Sync` тоді і тільки тоді коли `T` реалізує `Send`.
- Аналог блокування читання-запису: `RwLock`.
- Чому `lock()` повертає `Result`?
  - Якщо потік, який утримував `Mutex`, запанікував, `Mutex` стає "отруєним", сигналізуючи про те, що дані, які він захищає, можуть перебувати в неузгодженому стані. Виклик `lock()` для отруєнного м'ютексу зазнає невдачі з `PoisonError`. Ви можете викликати `into_inner()` для помилки, щоб відновити дані незалежно від цього.

60.3 Приклад

Давайте подивимося на `Arc` і `Mutex` в дії:

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

fn main() {
    let v = vec![10, 20, 30];
    let handle = thread::spawn(|| {
```

```
        v.push(10);
    });
    v.push(1000);

    handle.join().unwrap();
    println!("v: {v:?}");
}
```

Можливе рішення:

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

fn main() {
    let v = Arc::new(Mutex::new(vec![10, 20, 30]));

    let v2 = Arc::clone(&v);
    let handle = thread::spawn(move || {
        let mut v2 = v2.lock().unwrap();
        v2.push(10);
    });

    {
        let mut v = v.lock().unwrap();
        v.push(1000);
    }

    handle.join().unwrap();

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

Визначні частини:

- `v` обертається як в `Arc`, так і в `Mutex`, тому що їхні інтереси ортогональні.
  - Обгортання `Mutex` в `Arc` є загальним шаблоном для обміну змінним станом між потоками.
- `v: Arc<_>` потрібно клонувати як `v2`, перш ніж це можна буде перемістити в інший потік. Зверніть увагу, що до сигнатури лямбда було додано `move`.
- Блоки вводяться для того, щоб максимально звузити область використання `LockGuard`.

Розділ 61

Вправи

Цей сегмент повинен зайняти близько 1 години 10 хвилин. Він містить:

Слайд	Тривалість
Вечеря філософів	20 хвилин
Перевірка багатопоточних посилань	20 хвилин
Рішення	30 хвилин

61.1 Вечеря філософів

Проблема вечері філософів - це класична проблема одночасного виконання:

> П'ятеро філософів вечеряють разом за одним столом. У кожного філософа своє місце за столом. Між кожною тарілкою є виделка. Страва, що подається, являє собою різновид спагетті, яке потрібно їсти двома виделками. Кожен філософ може лише поперемінно мислити і їсти. Крім того, філософ може їсти свої спагетті лише тоді, коли у нього є і ліва, і права виделка. Таким чином, дві виделки будуть доступні лише тоді, коли його найближчі сусіди думають, а не їдять. Після того, як окремий філософ закінчує їсти, він кладе обидві виделки.

Для цієї вправи вам знадобиться локальний встановленний Cargo. Скопіюйте наведений нижче код у файл під назвою `src/main.rs`, заповніть порожні поля та перевірте, чи `cargo run` не блокує:

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

struct Fork;

struct Philosopher {
    name: String,
    // left_fork: ...
```

```
    // right_fork: ...
    // thoughts: ...
}

impl Philosopher {
    fn think(&self) {
        self.thoughts
            .send(format!("Еврика! {} має нову ідею!", &self.name))
            .unwrap();
    }

    fn eat(&self) {
        // Беремо виделки...
        println!("{} їсть...", &self.name);
        thread::sleep(Duration::from_millis(10));
    }
}

static PHILOSOPHERS: &[&str] =
    &["Сократ", "Гіпатія", "Платоне", "Аристотель", "Піфагор"];

fn main() {
    // Створюємо виделки

    // Створюємо філософів

    // Змусимо кожного з них подумати і з'їсти 100 разів

    // Вивести свої думки
}
```

Ви можете використовувати наступний `Cargo.toml`:

```
[package]
name = "dining-philosophers"
version = "0.1.0"
edition = "2021"
```

61.2 Перевірка багатопоточних посилань

Давайте використаємо наші нові знання, щоб створити багатопотоковий засіб перевірки лінків. Він має початися з веб-сторінки та перевірити, чи лінкі на сторінці дійсні. Він повинен рекурсивно перевіряти інші сторінки в тому самому домені та продовжувати робити це, доки всі сторінки не будуть перевірені.

Для цього вам знадобиться HTTP-клієнт, наприклад, `reqwest`. Вам також знадобиться спосіб пошуку лінків, ми можемо використати `scraper`. Нарешті, нам знадобиться спосіб обробки помилок, ми скористаємося `thiserror`.

Створіть новий проект Cargo та додайте `reqwest` як залежность з:

```
cargo new link-checker
```

```
cd link-checker
cargo add --features blocking,rustls-tls reqwest
cargo add scraper
cargo add thiserror
```

> Якщо `cargo add` завершується помилкою `error: no such subcommand`, будь ласка, відредагуйте файл `Cargo.toml` вручну. Додайте перелічені нижче залежності.

Виклики `cargo add` оновлять файл `Cargo.toml` таким чином:

```
[package]
name = "link-checker"
version = "0.1.0"
edition = "2021"
publish = false

[dependencies]
reqwest = { version = "0.11.12", features = ["blocking", "rustls-tls"] }
scraper = "0.13.0"
thiserror = "1.0.37"
```

Тепер ви можете завантажити стартову сторінку. Спробуйте з невеликим сайтом, наприклад `https://www.google.org/`.

Ваш файл `src/main.rs` має виглядати приблизно так:

```
use reqwest::blocking::Client;
use reqwest::Url;
use scraper::{Html, Selector};
use thiserror::Error;

enum Error {
    ReqwestError(#[from] reqwest::Error),
    BadResponse(String),
}

struct CrawlCommand {
    url: Url,
    extract_links: bool,
}

fn visit_page(client: &Client, command: &CrawlCommand) -> Result<Vec<Url>, Error> {
    println!("Перевіряємо {:#}", command.url);
    let response = client.get(command.url.clone()).send()?;
    if !response.status().is_success() {
        return Err(Error::BadResponse(response.status().to_string()));
    }

    let mut link_urls = Vec::new();
    if !command.extract_links {
        return Ok(link_urls);
    }
```

```
    let base_url = response.url().to_owned();
    let body_text = response.text()?;
    let document = Html::parse_document(&body_text);

    let selector = Selector::parse("a").unwrap();
    let href_values = document
        .select(&selector)
        .filter_map(|element| element.value().attr("href"));
    for href in href_values {
        match base_url.join(href) {
            Ok(link_url) => {
                link_urls.push(link_url);
            }
            Err(err) => {
                println!("На  {base_url:#}: проігноровано нерозбірливий {href:?}: {err}"
            }
        }
    }
    Ok(link_urls)
}

fn main() {
    let client = Client::new();
    let start_url = Url::parse("https://www.google.org").unwrap();
    let crawl_command = CrawlCommand{ url: start_url, extract_links: true };
    match visit_page(&client, &crawl_command) {
        Ok(links) => println!("Лінкі: {links:#?}"),
        Err(err) => println!("Не вдалося витягти лінки: {err:#}"),
    }
}
```

Запустіть код у `src/main.rs` за допомогою

```
cargo run
```

Завдання

- Використовуйте потоки для паралельної перевірки лінків: надішліть URL-адреси для перевірки на канал і дозвольте кільком потокам перевіряти URL-адреси паралельно.
- Розширте це, щоб рекурсивно отримувати лінкі з усіх сторінок домену `www.google.org`. Встановіть верхню межу приблизно в 100 сторінок, щоб вас не заблокував сайт.

61.3 Рішення

Вечеря філософів

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

```rust
struct Fork;

struct Philosopher {
    name: String,
    left_fork: Arc<Mutex<Fork>>,
    right_fork: Arc<Mutex<Fork>>,
    thoughts: mpsc::SyncSender<String>,
}

impl Philosopher {
    fn think(&self) {
        self.thoughts
            .send(format!("Еврика! {} має нову ідею!", &self.name))
            .unwrap();
    }

    fn eat(&self) {
        println!("{} намагається їсти", &self.name);
        let _left = self.left_fork.lock().unwrap();
        let _right = self.right_fork.lock().unwrap();

        println!("{} їсть...", &self.name);
        thread::sleep(Duration::from_millis(10));
    }
}

static PHILOSOPHERS: &[&str] =
    &["Сократ", "Гіпатія", "Платоне", "Аристотель", "Піфагор"];

fn main() {
    let (tx, rx) = mpsc::sync_channel(10);

    let forks = (0..PHILOSOPHERS.len())
        .map(|_| Arc::new(Mutex::new(Fork)))
        .collect::<Vec<_>>();

    for i in 0..forks.len() {
        let tx = tx.clone();
        let mut left_fork = Arc::clone(&forks[i]);
        let mut right_fork = Arc::clone(&forks[(i + 1) % forks.len()]);

        // Щоб уникнути глухого кута, ми повинні порушити симетрію
        // десь. Це дозволить поміняти місцями виделки без деініціалізації
        // жодної з них.
        if i == forks.len() - 1 {
            std::mem::swap(&mut left_fork, &mut right_fork);
        }

        let philosopher = Philosopher {
            name: PHILOSOPHERS[i].to_string(),
```

```
            thoughts: tx,
            left_fork,
            right_fork,
        };

        thread::spawn(move || {
            for _ in 0..100 {
                philosopher.eat();
                philosopher.think();
            }
        });
    }

    drop(tx);
    for thought in rx {
        println!("{thought}");
    }
}
```

Перевірка лінків

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

use reqwest::blocking::Client;
use reqwest::Url;
use scraper::{Html, Selector};
use thiserror::Error;

enum Error {
    ReqwestError(#[from] reqwest::Error),
    BadResponse(String),
}

struct CrawlCommand {
    url: Url,
    extract_links: bool,
}

fn visit_page(client: &Client, command: &CrawlCommand) -> Result<Vec<Url>, Error> {
    println!("Перевіряємо {:#}", command.url);
    let response = client.get(command.url.clone()).send()?;
    if !response.status().is_success() {
        return Err(Error::BadResponse(response.status().to_string()));
    }

    let mut link_urls = Vec::new();
    if !command.extract_links {
        return Ok(link_urls);
    }
```

```
    let base_url = response.url().to_owned();
    let body_text = response.text()?;
    let document = Html::parse_document(&body_text);

    let selector = Selector::parse("a").unwrap();
    let href_values = document
        .select(&selector)
        .filter_map(|element| element.value().attr("href"));
    for href in href_values {
        match base_url.join(href) {
            Ok(link_url) => {
                link_urls.push(link_url);
            }
            Err(err) => {
                println!("На  {base_url:#}: проігноровано нерозбірливий {href:?}: {err}"
            }
        }
    }
    Ok(link_urls)
}

struct CrawlState {
    domain: String,
    visited_pages: std::collections::HashSet<String>,
}

impl CrawlState {
    fn new(start_url: &Url) -> CrawlState {
        let mut visited_pages = std::collections::HashSet::new();
        visited_pages.insert(start_url.as_str().to_string());
        CrawlState { domain: start_url.domain().unwrap().to_string(), visited_pages }
    }

    /// Визначаємо, чи потрібно витягувати лінки на даній сторінці.
    fn should_extract_links(&self, url: &Url) -> bool {
        let Some(url_domain) = url.domain() else {
            return false;
        };
        url_domain == self.domain
    }

    /// Відмітимо дану сторінку як відвідану, повернувши false, якщо вона вже
    /// була відвідана.
    fn mark_visited(&mut self, url: &Url) -> bool {
        self.visited_pages.insert(url.as_str().to_string())
    }
}

type CrawlResult = Result<Vec<Url>, (Url, Error)>;
fn spawn_crawler_threads(
    command_receiver: mpsc::Receiver<CrawlCommand>,
```

```
    result_sender: mpsc::Sender<CrawlResult>,
    thread_count: u32,
) {
    let command_receiver = Arc::new(Mutex::new(command_receiver));

    for _ in 0..thread_count {
        let result_sender = result_sender.clone();
        let command_receiver = command_receiver.clone();
        thread::spawn(move || {
            let client = Client::new();
            loop {
                let command_result = {
                    let receiver_guard = command_receiver.lock().unwrap();
                    receiver_guard.recv()
                };
                let Ok(crawl_command) = command_result else {
                    // Відправника було видалено. Більше команд не надходить.
                    break;
                };
                let crawl_result = match visit_page(&client, &crawl_command) {
                    Ok(link_urls) => Ok(link_urls),
                    Err(error) => Err((crawl_command.url, error)),
                };
                result_sender.send(crawl_result).unwrap();
            }
        });
    }
}

fn control_crawl(
    start_url: Url,
    command_sender: mpsc::Sender<CrawlCommand>,
    result_receiver: mpsc::Receiver<CrawlResult>,
) -> Vec<Url> {
    let mut crawl_state = CrawlState::new(&start_url);
    let start_command = CrawlCommand { url: start_url, extract_links: true };
    command_sender.send(start_command).unwrap();
    let mut pending_urls = 1;

    let mut bad_urls = Vec::new();
    while pending_urls > 0 {
        let crawl_result = result_receiver.recv().unwrap();
        pending_urls -= 1;

        match crawl_result {
            Ok(link_urls) => {
                for url in link_urls {
                    if crawl_state.mark_visited(&url) {
                        let extract_links = crawl_state.should_extract_links(&url);
                        let crawl_command = CrawlCommand { url, extract_links };
                        command_sender.send(crawl_command).unwrap();
```

```
                        pending_urls += 1;
                    }
                }
            }
            Err((url, error)) => {
                bad_urls.push(url);
                println!("Виникла помилка при скануванні: {:#}", error);
                continue;
            }
        }
    }
    bad_urls
}

fn check_links(start_url: Url) -> Vec<Url> {
    let (result_sender, result_receiver) = mpsc::channel::<CrawlResult>();
    let (command_sender, command_receiver) = mpsc::channel::<CrawlCommand>();
    spawn_crawler_threads(command_receiver, result_sender, 16);
    control_crawl(start_url, command_sender, result_receiver)
}

fn main() {
    let start_url = reqwest::Url::parse("https://www.google.org").unwrap();
    let bad_urls = check_links(start_url);
    println!("Неправильні URL-адреси: {:#?}", bad_urls);
}
```

Частина XIV

Одночасність виконання: Полудень

Розділ 62

Ласкаво просимо

”Async” — це модель одночасного виконання декількох завдань, при якій кожне завдання виконується одночасно доти, доки воно не заблокується, а потім перемикається на інше завдання, яке готове до виконання. Модель дозволяє виконувати більшу кількість завдань на обмеженій кількості потоків. Це пов'язано з тим, що накладні витрати на кожну задачу зазвичай дуже низькі, а операційні системи надають примітиви для ефективного визначення вводу/виводу, який може продовжувати роботу.

Асинхронна робота Rust базується на ”ф'ючерсах”, які представляють роботу, яка може бути завершена в майбутньому. Ф'ючерси ”опитуються”, доки вони не сигналізують, що вони завершені.

Ф’ючерси опитуються асинхронним середовищем виконання, і доступно кілька різних середовищ виконання.

Порівняння

- Python має подібну модель у своєму `asyncio`. Однак його тип `Future` базується на зворотному виклику, а не опитується. Програми на асинхронному Python вимагають ”циклу”, подібного до середовища виконання в Rust.
- Тип `Promise` JavaScript подібний, але знову ж таки на основі зворотного виклику. Середовище виконання мови реалізує цикл подій, тому багато деталей вирішення `Promise` приховані.

Розклад

Враховуючи 10-хвилинні перерви, ця сесія має тривати близько 3 годин та 20 хвилин. Вона містить:

Сегмент	Тривалість
Основи асинхронізації	30 хвилин
Канали та потік управління	20 хвилин

Сегмент	Тривалість
Підводні камені	55 хвилин
Вправи	1 година та 10 хвилин

Розділ 63

Основи асинхронізації

Цей сегмент повинен зайняти близько 30 хвилин. Він містить:

Слайд	Тривалість
async/await	10 хвилин
Futures	4 хвилини
Середовища виконання	10 хвилин
Завдання	10 хвилин

63.1 `async`/`await`

На високому рівні асинхронний код Rust дуже схожий на ”звичайний” послідовний код:

```
use futures::executor::block_on;

async fn count_to(count: i32) {
    for i in 0..count {
        println!("Підрахунок: {i}!");
    }
}

async fn async_main(count: i32) {
    count_to(count).await;
}

fn main() {
    block_on(async_main(10));
}
```

Ключові моменти:

- Зауважте, що це спрощений приклад для демонстрації синтаксису. У ньому немає тривалої операції чи реального одночасного виконання!

- Ключове слово ”async” - це синтаксичний цукор. Компілятор замінює тип повернення на ф'ючерс.
- Ви не можете зробити `main` асинхронним без додаткових інструкцій для компілятора щодо використання повернутого ф'ючерса.
- Вам потрібен виконавець для запуску асинхронного коду. `block_on` блокує поточний потік, доки наданий ф'ючерс не завершиться.
- `.await` асинхронно очікує на завершення іншої операції. На відміну від `block_on`, `.await` не блокує поточний потік.
- `.await` можна використовувати тільки всередині функції `async` (або блоку; вони будуть представлені пізніше).

63.2 Futures

`Future` — це трейт, реалізований об’єктами, які представляють операцію, яка може бути ще не завершеною. Ф'ючерс можна опитувати, і `poll` повертає `Poll`.

```
use std::pin::Pin;
use std::task::Context;

pub trait Future {
    type Output;
    fn poll(self: Pin<&mut Self>, cx: &mut Context<'_>) -> Poll<Self::Output>;
}

pub enum Poll<T> {
    Ready(T),
    Pending,
}
```

Асинхронна функція повертає `impl Future`. Також можливо (але рідко) реалізувати `Future` для ваших власних типів. Наприклад, `JoinHandle`, отриманий від `tokio::spawn`, реалізує `Future`, щоб дозволити приєднання до нього.

Ключове слово `.await`, застосоване до `Future`, змушує поточну асинхронну функцію призупинятися, доки це `Future` не буде готове, а потім обчислює її вихідні дані.

- Типи `Future` та `Poll` реалізовано саме так, як показано на малюнку; натисніть на лінки, щоб переглянути реалізацію в документації.
- Ми не будемо переходити до `Pin` і `Context`, оскільки ми зосередимося на написанні асинхронного коду, а не на створенні нових асинхронних примітивів. Коротко:
  - `Context` дозволяє Future запланувати повторне опитування, при настанні певної події.
  - `Pin` гарантує, що Future не буде переміщено в пам'яті, тому покажчики на цей ф'ючерс залишатимуться дійсними. Це потрібно, щоб дозволити посиланням залишатися дійсними після `.await`.

63.3 Середовища виконання

Середовище виконанняе забезпечує підтримку асинхронного виконання операцій (*реактор*) і відповідає за виконання ф'ючерсів (*виконавець*). Rust не має "вбудованого" середовища виконання, але доступні кілька варіантів:

- Tokio: ефективний, із добре розвиненою екосистемою функціональності, наприклад Hyper для HTTP або Tonic для gRPC.
- async-std: прагне бути "std for async" та включає базове середовище виконання в `async::task`.
- smol: простий і легкий

Кілька великих програм мають власний час виконання. Наприклад, Fuchsia вже має один.

- Зверніть увагу, що з перелічених середовищ виконання лише Tokio підтримується на ігровому майданчику Rust. Ігровий майданчик також не дозволяє будь-який ввід-вивід, тому більшість цікавих асинхронних речей не можуть працювати на ігровому майданчику.
- Ф'ючерси "інертні" в тому, що вони нічого не роблять (навіть не починають операцію вводу-виводу), якщо немає виконавця, який їх опитує. Це відрізняється від, наприклад, JS Promises, які виконуватимуться до кінця, навіть якщо їх ніколи не використовувати.

63.3.1 Tokio

Tokio надає:

- Багатопотокове середовище виконання для виконання асинхронного коду.
- Асинхронну версію стандартної бібліотеки.
- Велику екосистему бібліотек.

```
use tokio::time;

async fn count_to(count: i32) {
    for i in 0..count {
        println!("Підрахунок у завданні: {i}!");
        time::sleep(time::Duration::from_millis(5)).await;
    }
}

async fn main() {
    tokio::spawn(count_to(10));

    for i in 0..5 {
        println!("Основне завдання: {i}");
        time::sleep(time::Duration::from_millis(5)).await;
    }
}
```

- За допомогою макросу `tokio::main` ми тепер можемо зробити `main` асинхронною.
- Функція `spawn` створює нове, одночасне "завдання".

- Примітка: `spawn` приймає `Future`, ви не викликаєте `.await` на `count_to`.

Подальше дослідження:

- Чому `count_to` (зазвичай) не досягає 10? Це приклад асинхронного скасування. `tokio::spawn` повертає дескриптор, який можна чекати, поки він не завершиться.
- Спробуйте `count_to(10).await` замість породження.
- Спробуйте дочекатися завдання, повернутого з `tokio::spawn`.

63.4 Завдання

У Rust є система завдань, яка є формою полегшеного потокового програмування.

Завдання має єдиний ф'ючерс верхнього рівня, яке виконавець опитує для прогресу. Цей ф'ючерс може мати один або декілька вкладених ф'ючерсів, які опитує його метод `poll`, що приблизно відповідає стеку викликів. Одночасність виконання у межах завдання можлива за допомогою опитування кількох дочірніх ф'ючерсів, таких як перегони таймера та операції введення/виведення.

```
use tokio::io::{self, AsyncReadExt, AsyncWriteExt};
use tokio::net::TcpListener;

async fn main() -> io::Result<()> {
    let listener = TcpListener::bind("127.0.0.1:0").await?;
    println!("слухаємо на порту {}", listener.local_addr()?.port());

    loop {
        let (mut socket, addr) = listener.accept().await?;

        println!("з'єднання з {addr:?}");

        tokio::spawn(async move {
            socket.write_all(b"Хто ви?\n").await.expect("помилка сокета");

            let mut buf = vec![0; 1024];
            let name_size = socket.read(&mut buf).await.expect("помилка сокета");
            let name = std::str::from_utf8(&buf[..name_size]).unwrap().trim();
            let reply = format!("Дякуємо за дзвінок, {name}!\n");
            socket.write_all(reply.as_bytes()).await.expect("помилка сокета");
        });
    }
}
```

Скопіюйте цей приклад у ваш підготовлений `src/main.rs` і запустіть його звідти.

Спробуйте підключитися до нього за допомогою TCP-з'єднання, наприклад, nc або telnet.

- Попросіть студентів візуалізувати стан сервера прикладу з кількома підключеними клієнтами. Які існують завдання? Які їхні Futures?

- Це перший раз, коли ми бачимо блок `async`. Це схоже на закриття, але не приймає жодних аргументів. Він повертає значення Future, подібно до `async fn`.
- Перетворіть асинхронний блок у функцію та покращіть обробку помилок за допомогою `?`.

Розділ 64

Канали та потік управління

Цей сегмент повинен зайняти близько 20 хвилин. Він містить:

Слайд	Тривалість
Асинхронні канали	10 хвилин
Join	4 хвилини
Select	5 хвилин

64.1 Асинхронні канали

Кілька крейтів підтримують асинхронні канали. Наприклад `tokio`:

```
use tokio::sync::mpsc;

async fn ping_handler(mut input: mpsc::Receiver<()>) {
    let mut count: usize = 0;

    while let Some(_) = input.recv().await {
        count += 1;
        println!("Отримано {count} пінгів до цього часу.");
    }

    println!("ping_handler завершено");
}

async fn main() {
    let (sender, receiver) = mpsc::channel(32);
    let ping_handler_task = tokio::spawn(ping_handler(receiver));
    for i in 0..10 {
        sender.send(()).await.expect("Не вдалося надіслати пінг.");
        println!("Поки що надіслано {} пінгів.", i + 1);
    }

    drop(sender);
```

```
    ping_handler_task.await.expect("Щось пішло не так у завданні обробника пінгу.");
}
```

- Змініть розмір каналу на 3 і подивіться, як це вплине на виконання.
- Загалом, інтерфейс подібний до каналів `sync`, які ми бачили в ранковому класі.
- Спробуйте видалити виклик `std::mem::drop`. Що сталося? Чому?
- Крейт Flume має канали, які реалізують як `sync`, так і `async` `send` і `recv`. Це може бути зручно для складних програм, що використовують як ввід-вивід, так і важкі процесорні завдання.
- Що робить роботу з `async` каналами більш кращою, так це можливість комбінувати їх з іншими `future`, щоб об'єднувати їх і створювати складні потоки управління.

64.2 Join

Операція об’єднання очікує, поки весь набір ф’ючерсів буде готовий, і повертає колекцію їхніх результатів. Це схоже на `Promise.all` у JavaScript або `asyncio.gather` у Python.

```
use anyhow::Result;
use futures::future;
use reqwest;
use std::collections::HashMap;

async fn size_of_page(url: &str) -> Result<usize> {
    let resp = reqwest::get(url).await?;
    Ok(resp.text().await?.len())
}

async fn main() {
    let urls: [&str; 4] = [
        "https://google.com",
        "https://httpbin.org/ip",
        "https://play.rust-lang.org/",
        "BAD_URL",
    ];
    let futures_iter = urls.into_iter().map(size_of_page);
    let results = future::join_all(futures_iter).await;
    let page_sizes_dict: HashMap<&str, Result<usize>> =
        urls.into_iter().zip(results.into_iter()).collect();
    println!("{page_sizes_dict:?}");
}
```

Скопіюйте цей приклад у ваш підготовлений `src/main.rs` і запустіть його звідти.

- Для кількох ф’ючерсів непересічних типів ви можете використовувати `std::future::join!`, але ви повинні знати, скільки ф’ючерсів у вас буде під час компіляції. Наразі це в коейті `futures`, незабаром буде стабілізовано в `std::future`.

- Ризик `join` полягає в тому, що один із ф'ючерсів може ніколи не вирішитися, це призведе до зависання вашої програми.
- Ви також можете поєднати `join_all` з `join!`, наприклад, щоб об'єднати всі запити до служби http, а також запит до бази даних. Спробуйте додати `tokio::time::sleep` до ф'ючерсу, використовуючи `futures::join!`. Це не тайм-аут (який вимагає `select!`, пояснюється в наступному розділі), але демонструє `join!`.

64.3 Select

Операція select очікує, поки будь-який із набору ф'ючерсів буде готовий, і відповідає на результат цього ф'ючерсу. У JavaScript це схоже на `Promise.race`. У Python це порівнюється з `asyncio.wait(task_set, return_when=asyncio.FIRST_COMPLETED)`.

Подібно до оператора порівняння, тіло `select!` має кілька гілок, кожен з яких має вигляд `pattern = future => statement`. Коли `future` готовий, його значення, що повертається, деструктурується за допомогою `pattern`. Потім виконується `statement` з отриманими змінними. Результат `statement` стає результатом макросу `select!`.

```
use tokio::sync::mpsc;
use tokio::time::{sleep, Duration};

async fn main() {
    let (tx, mut rx) = mpsc::channel(32);
    let listener = tokio::spawn(async move {
        tokio::select! {
            Some(msg) = rx.recv() => println!("отримав: {msg}"),
            _ = sleep(Duration::from_millis(50)) => println!("тайм-аут"),
        };
    });
    sleep(Duration::from_millis(10)).await;
    tx.send(String::from("Привіт!")).await.expect("Не вдалося надіслати привітання.");

    listener.await.expect("Listener зазнав невдачі");
}
```

- Блок асинхронізації `listener` тут є звичайною формою: очікування деякої асинхронної події або таймауту. Змініть `sleep` на довший час, щоб побачити, що він не спрацює. Чому в цій ситуації також не спрацьовує send?
- Команда `select!` також часто використовується у циклі в "actor" архітектурах, де завдання реагує на події у циклі. Це має деякі підводні камені, які буде обговорено у наступному розділі.

Розділ 65

Підводні камені

Async / await забезпечує зручну та ефективну абстракцію для асинхронного програмування з одночасним виконанням. Однак, модель async/await у Rust також має свої підводні камені та пастки. Ми проілюструємо деякі з них у цьому розділі.

Цей сегмент повинен зайняти близько 55 хвилин. Він містить:

Слайд	Тривалість
Блокування Виконавця	10 хвилин
Pin	20 хвилин
Асинхронні трейти	5 хвилин
Скасування	20 хвилин

65.1 Блокування виконавця

Більшість асинхронних середовищ виконання дозволяють лише одночасний запуск завдань вводу/виводу. Це означає, що завдання, що блокують процесор, блокуватимуть виконавця та запобігатимуть виконанню інших завдань. Простим обхідним шляхом є використання еквівалентних асинхронних методів, де це можливо.

```
use futures::future::join_all;
use std::time::Instant;

async fn sleep_ms(start: &Instant, id: u64, duration_ms: u64) {
    std::thread::sleep(std::time::Duration::from_millis(duration_ms));
    println!(
        "ф'ючерс {id} спав протягом {duration_ms}ms, закінчив після {}ms",
        start.elapsed().as_millis()
    );
}

async fn main() {
    let start = Instant::now();
    let sleep_futures = (1..=10).map(|t| sleep_ms(&start, t, t * 10));
```

```
    join_all(sleep_futures).await;
}
```

- Запустіть код і подивіться, що засинання відбуваються послідовно, а не одночасно.
- Варіант `"current_thread"` поміщає всі завдання в один потік. Це робить ефект більш очевидним, але помилка все ще присутня в багатопоточному варіанті.
- Переключіть `std::thread::sleep` на `tokio::time::sleep` і дочекайтеся результату.
- Іншим виправленням було б `tokio::task::spawn_blocking`, який породжує фактичний потік і перетворює його дескриптор у ф'ючерс, не блокуючи виконавця.
- Ви не повинні думати про завдання як про потоки ОС. Вони не відображаються 1 до 1, і більшість виконавців дозволять виконувати багато завдань в одному потоці ОС. Це особливо проблематично під час взаємодії з іншими бібліотеками через FFI, де ця бібліотека може залежати від локального сховища потоку або зіставлятися з певними потоками ОС (наприклад, CUDA). У таких ситуаціях віддайте перевагу `tokio::task::spawn_blocking`.
- Обережно використовуйте м'ютекси синхронізації. Утримування м'ютексу над `.await` може призвести до блокування іншого завдання, яке може виконуватися в тому самому потоці.

65.2 `Pin`

Асинхронні блоки та функції повертають типи, що реалізують трейт `Future`. Тип, що повертається, є результатом трансформації компілятора, який перетворює локальні змінні на дані, що зберігаються у ф'ючерсі.

Деякі з цих змінних можуть містити вказівники на інші локальні змінні. Через це ф'ючерс ніколи не слід переміщувати в іншу комірку пам'яті, оскільки це зробить ці вказівники недійсними.

Щоб запобігти переміщенню ф'ючерсного типу у пам'яті, його можна опитувати лише через закріплений вказівник. Закріплення - це обгортка навколо посилання, яка забороняє всі операції, що можуть перемістити екземпляр, на який воно вказує, в іншу ділянку пам'яті.

```
use tokio::sync::{mpsc, oneshot};
use tokio::task::spawn;
use tokio::time::{sleep, Duration};

// Робочий елемент. У цьому випадку просто заснути на заданий час і
// відповісти повідомленням на каналі `respond_on`.
struct Work {
    input: u32,
    respond_on: oneshot::Sender<u32>,
}

// Робочий, який чекає на роботу у черзі та виконує її.
```

```
async fn worker(mut work_queue: mpsc::Receiver<Work>) {
    let mut iterations = 0;
    loop {
        tokio::select! {
            Some(work) = work_queue.recv() => {
                sleep(Duration::from_millis(10)).await; // Вдається, що працює.
                work.respond_on
                    .send(work.input * 1000)
                    .expect("не вдалося надіслати відповідь");
                iterations += 1;
            }
            // TODO: виводити кількість ітерацій кожні 100 мс
        }
    }
}

// Запитувач, який надсилає запит на виконання роботи і чекає на її завершення.
async fn do_work(work_queue: &mpsc::Sender<Work>, input: u32) -> u32 {
    let (tx, rx) = oneshot::channel();
    work_queue
        .send(Work { input, respond_on: tx })
        .await
        .expect("не вдалося відправити в робочу чергу");
    rx.await.expect("не вдалося дочекатися відповіді")
}

async fn main() {
    let (tx, rx) = mpsc::channel(10);
    spawn(worker(rx));
    for i in 0..100 {
        let resp = do_work(&tx, i).await;
        println!("результат роботи для ітерації {i}: {resp}");
    }
}
```

- Ви можете розпізнати це як приклад шаблону актора. Актори зазвичай викликають `select!` у циклі.
- Це є підсумком кількох попередніх уроків, тож не поспішайте з цим.
  - Наївно додайте `_ = sleep(Duration::from_millis(100)) => { println!(..) }` до `select!`. Це ніколи не буде виконано. Чому?
  - Замість цього додайте `timeout_fut`, що містить цей ф'юсчерс за межами `loop`:

    ```
    let timeout_fut = sleep(Duration::from_millis(100));
    loop {
        select! {
            ..,
            _ = timeout_fut => { println!(..); },
        }
    }
    ```

- Це все ще не працює. Слідкуйте за помилками компілятора, додавши `&mut` до `timeout_fut` у `select!`, щоб обійти переміщення, а потім використовуючи `Box::pin`:

  ```
  let mut timeout_fut = Box::pin(sleep(Duration::from_millis(100)));
  loop {
      select! {
          ..,
          _ = &mut timeout_fut => { println!(..); },
      }
  }
  ```

- Це компілюється, але після закінчення часу очікування на кожній ітерації відображається `Poll::Ready` (злитий ф'ючерс міг би допомогти в цьому). Оновіть, щоб скидати `timeout_fut` кожного разу, коли він спливає:

  ```
  let mut timeout_fut = Box::pin(sleep(Duration::from_millis(100)));
  loop {
      select! {
          _ = &mut timeout_fut => {
              println!(..);
              timeout_fut = Box::pin(sleep(Duration::from_millis(100)));
          },
      }
  }
  ```

- Box виділяє у купі. У деяких випадках, `std::pin::pin!` (лише нещодавно стабілізовано, у старому коді часто використовується `tokio::pin!`) також є варіантом, але його важко використовувати для фьючерсів, які перепризначено.
- Інша альтернатива — взагалі не використовувати pin, а створювати інше завдання, яке буде надсилати на канал oneshot кожні 100 мс.
- Дані, які містять вказівники на себе, називаються самопосилальними. Зазвичай, перевірка запозичень у Rust запобігає переміщенню самопосилань, оскільки посилання не можуть пережити дані, на які вони вказують. Однак, перетворення коду для асинхронних блоків і функцій не перевіряється перевіркою запозичень.
- `Pin` - це обгортка навколо вказівника. Об'єкт не можна перемістити з його місця за допомогою закріпленого вказівника. Однак, його можна переміщати за допомогою незакріпленого вказівника.
- Метод `poll` трейту `Future` використовує `Pin<&mut Self>` замість `&mut Self` для посилання на екземпляр. Тому його можна викликати лише на закріпленому покажчику.

65.3 Асинхронні трейти

Асинхронні методи у трейтах було стабілізовано у випуску 1.75. Це вимагало підтримки використання `impl Trait` з позицією повернення у трейтах, оскільки десигнування для `async fn` включає `-> impl Future<Output = ...>`.

Однак, навіть з нативною підтримкою, існують деякі підводні камені навколо `async fn`:

- Позиція повернення `impl Trait` фіксує всі терміни життя в межах області застосування (тому деякі моделі запозичення не можуть бути виражені).
- Асинхронні трейти не можна використовувати з об'єктами трейтів (підтримка `dyn Trait`).

Крейт async_trait надає обхідний шлях за допомогою макросу, з деякими застереженнями:

```
use async_trait::async_trait;
use std::time::Instant;
use tokio::time::{sleep, Duration};

trait Sleeper {
    async fn sleep(&self);
}

struct FixedSleeper {
    sleep_ms: u64,
}

impl Sleeper for FixedSleeper {
    async fn sleep(&self) {
        sleep(Duration::from_millis(self.sleep_ms)).await;
    }
}

async fn run_all_sleepers_multiple_times(
    sleepers: Vec<Box<dyn Sleeper>>,
    n_times: usize,
) {
    for _ in 0..n_times {
        println!("Запуск всіх сплячих..");
        for sleeper in &sleepers {
            let start = Instant::now();
            sleeper.sleep().await;
            println!("Проспав {}мс", start.elapsed().as_millis());
        }
    }
}

async fn main() {
    let sleepers: Vec<Box<dyn Sleeper>> = vec![
        Box::new(FixedSleeper { sleep_ms: 50 }),
        Box::new(FixedSleeper { sleep_ms: 100 }),
    ];
    run_all_sleepers_multiple_times(sleepers, 5).await;
}
```

- `async_trait` простий у використанні, але зауважте, що для цього він використовує виділення в купі. Цей розподіл купи має накладні витрати на продуктивність.
- Проблеми мовної підтримки `async trait` є надто глибокими, щоб детально

описати їх у цьому уроці. Якщо ви зацікавлені у глибшому вивченні, перегляньте цей запис у блозі Ніко Мацакіса. Дивіться також ці ключові слова:

  - RPIT: скорочення від return-position `impl Trait`.
  - RPITIT: скорочення від return-position `impl Trait` у трейті (RPIT у трейті).

- Спробуйте створити нову сплячу структуру, яка буде спати протягом випадкового періоду часу, і додайте її до `Vec`.

65.4 Скасування

Видалення ф'ючерсу означає, що він більше ніколи не може бути опитаний. Це називається *скасуванням*, і воно може відбутися в будь-який `await` момент. Потрібно бути обережним, щоб система працювала правильно, навіть якщо ф'ючерс скасовано. Наприклад, вона не повинна зайти в глухий кут або втратити дані.

```
use std::io;
use std::time::Duration;
use tokio::io::{AsyncReadExt, AsyncWriteExt, DuplexStream};

struct LinesReader {
    stream: DuplexStream,
}

impl LinesReader {
    fn new(stream: DuplexStream) -> Self {
        Self { stream }
    }

    async fn next(&mut self) -> io::Result<Option<String>> {
        let mut bytes = Vec::new();
        let mut buf = [0];
        while self.stream.read(&mut buf[..]).await? != 0 {
            bytes.push(buf[0]);
            if buf[0] == b'\n' {
                break;
            }
        }
        if bytes.is_empty() {
            return Ok(None);
        }
        let s = String::from_utf8(bytes)
            .map_err(|_| io::Error::new(io::ErrorKind::InvalidData, "не UTF-8"))?;
        Ok(Some(s))
    }
}

async fn slow_copy(source: String, mut dest: DuplexStream) -> io::Result<()> {
    for b in source.bytes() {
        dest.write_u8(b).await?;
        tokio::time::sleep(Duration::from_millis(10)).await
```

```
    }
    Ok(())
}

async fn main() -> io::Result<()> {
    let (client, server) = tokio::io::duplex(5);
    let handle = tokio::spawn(slow_copy("всім\nпривіт\n".to_owned(), client));

    let mut lines = LinesReader::new(server);
    let mut interval = tokio::time::interval(Duration::from_millis(60));
    loop {
        tokio::select! {
            _ = interval.tick() => println!("тік!"),
            line = lines.next() => if let Some(l) = line? {
                print!("{}", l)
            } else {
                break
            },
        }
    }
    handle.await.unwrap()?;
    Ok(())
}
```

- Компілятор не допомагає з безпекою скасування. Вам потрібно прочитати документацію до API і звернути увагу на те, який стан містить ваша `async fn`.
- На відміну від `panic` і `?`, скасування є частиною звичайного потоку керування (на відміну від обробки помилок).
- Приклад втрачає частини рядка.
  - Щоразу, коли гілка `tick()` завершується першою, `next()` і його `buf` відкидаються.
  - `LinesReader` можна зробити безпечним для скасування, зробивши `buf` частиною структури:

    ```
    struct LinesReader {
        stream: DuplexStream,
        bytes: Vec<u8>,
        buf: [u8; 1],
    }

    impl LinesReader {
        fn new(stream: DuplexStream) -> Self {
            Self { stream, bytes: Vec::new(), buf: [0] }
        }
        async fn next(&mut self) -> io::Result<Option<String>> {
            // префікс buf та байти з self.
            // ...
            let raw = std::mem::take(&mut self.bytes);
            let s = String::from_utf8(raw)
                .map_err(|_| io::Error::new(io::ErrorKind::InvalidData, "не UTF-8"))
    ```

```
            // ...
        }
    }
```

- `Interval::tick` є безпечним для скасування, оскільки він відстежує, чи був тік 'доставлений'.
- `AsyncReadExt::read` є безпечним для скасування, оскільки він повертає або не читає дані.
- `AsyncBufReadExt::read_line` схожий на приклад і *не є* безпечним для скасування. Подробиці та альтернативи дивится у його документації.

Розділ 66

Вправи

Цей сегмент повинен зайняти близько 1 години 10 хвилин. Він містить:

Слайд	Тривалість
Вечеря філософів	20 хвилин
Програма широкомовного чату	30 хвилин
Рішення	20 хвилин

66.1 Вечеря філософів --- Async

Перегляньте вечерю філософів для опису проблеми.

Як і раніше, для виконання цієї вправи вам знадобиться локальний встановленний Cargo. Скопіюйте наведений нижче код у файл під назвою `src/main.rs`, заповніть порожні поля та перевірте, чи `cargo run` не блокує:

```
use std::sync::Arc;
use tokio::sync::{mpsc, Mutex};
use tokio::time;

struct Fork;

struct Philosopher {
    name: String,
    // left_fork: ...
    // right_fork: ...
    // thoughts: ...
}

impl Philosopher {
    async fn think(&self) {
        self.thoughts
            .send(format!("Еврика! {} має нову ідею!", &self.name))
            .await
```

```
            .unwrap();
    }

    async fn eat(&self) {
        // Продовжуємо пробувати, поки не знайдемо обидві виделки
        println!("{} їсть...", &self.name);
        time::sleep(time::Duration::from_millis(5)).await;
    }
}

static PHILOSOPHERS: &[&str] =
    &["Сократ", "Гіпатія", "Платоне", "Аристотель", "Піфагор"];

async fn main() {
    // Створюємо виделки

    // Створюємо філософів

    // Змусимо їх думати і їсти

    // Вивести свої думки
}
```

Оскільки цього разу ви використовуєте Async Rust, вам знадобиться залежність `tokio`. Ви можете використовувати наступний `Cargo.toml`:

```
[package]
name = "dining-philosophers-async-dine"
version = "0.1.0"
edition = "2021"

[dependencies]
tokio = { version = "1.26.0", features = ["sync", "time", "macros", "rt-multi-thread"] }
```

Також зауважте, що цього разу вам доведеться використовувати `Mutex` і модуль `mpsc` з крейту `tokio`.

- Чи можете ви зробити вашу реалізацію однопотоковою?

66.2 Програма широкомовного чату

У цій вправі ми хочемо використати наші нові знання для реалізації програми чату. У нас є чат-сервер, до якого підключаються клієнти і публікують свої повідомлення. Клієнт читає повідомлення користувача зі стандартного вводу і надсилає їх на сервер. Сервер чату транслює кожне повідомлення, яке він отримує, усім клієнтам.

Для цього ми використовуємо трансляційний канал на сервері та `tokio_websockets` для зв'язку між клієнтом і сервером.

Створіть новий проект Cargo та додайте такі залежності:

Cargo.toml:

```toml
[package]
name = "chat-async"
version = "0.1.0"
edition = "2021"

[dependencies]
futures-util = { version = "0.3.31", features = ["sink"] }
http = "1.1.0"
tokio = { version = "1.41.1", features = ["full"] }
tokio-websockets = { version = "0.10.1", features = ["client", "fastrand", "server", "sh
```

Необхідні API

Вам знадобляться такі функції з `tokio` і `tokio_websockets`. Витратьте кілька хвилин на ознайомлення з API.

- StreamExt::next(), реалізований `WebsocketStream`: для асинхронного читання повідомлень з потоку Websocket.
- SinkExt::send(), реалізований `WebsocketStream`: для асинхронного надсилання повідомлень у потоці Websocket.
- Lines::next_line(): для асинхронного читання повідомлень користувача зі стандартного вводу.
- Sender::subscribe(): для підписки на канал трансляції.

Два бінарні файли

Зазвичай у проекті Cargo можна мати лише один бінарний файл і один файл `rc/main.rs`. У цьому проекті нам потрібні два бінарних файли. Один для клієнта і один для сервера. Потенційно ви могли б зробити їх двома окремими проектами Cargo, але ми збираємося помістити їх в один проект Cargo з двома бінарними файлами. Для того, щоб це працювало, клієнтський і серверний код має знаходитися у каталозі `src/bin` (дивиться документацію).

Скопіюйте наступний серверний та клієнтський код у файли `src/bin/server.rs` та `src/bin/client.rs` відповідно. Ваше завдання - доповнити ці файли, як описано нижче.

src/bin/server.rs:

```rust
use futures_util::sink::SinkExt;
use futures_util::stream::StreamExt;
use std::error::Error;
use std::net::SocketAddr;
use tokio::net::{TcpListener, TcpStream};
use tokio::sync::broadcast::{channel, Sender};
use tokio_websockets::{Message, ServerBuilder, WebSocketStream};

async fn handle_connection(
    addr: SocketAddr,
    mut ws_stream: WebSocketStream<TcpStream>,
    bcast_tx: Sender<String>,
) -> Result<(), Box<dyn Error + Send + Sync>> {
```

```
    // TODO: Підказку дивіться в описі завдання нижче.

}

async fn main() -> Result<(), Box<dyn Error + Send + Sync>> {
    let (bcast_tx, _) = channel(16);

    let listener = TcpListener::bind("127.0.0.1:2000").await?;
    println!("слухаємо на порту 2000");

    loop {
        let (socket, addr) = listener.accept().await?;
        println!("Нове з'єднання з {addr:?}");
        let bcast_tx = bcast_tx.clone();
        tokio::spawn(async move {
            // Обернути необроблений TCP потік у веб-сокет.
            let ws_stream = ServerBuilder::new().accept(socket).await?;

            handle_connection(addr, ws_stream, bcast_tx).await
        });
    }
}
```

src/bin/client.rs:

```
use futures_util::stream::StreamExt;
use futures_util::SinkExt;
use http::Uri;
use tokio::io::{AsyncBufReadExt, BufReader};
use tokio_websockets::{ClientBuilder, Message};

async fn main() -> Result<(), tokio_websockets::Error> {
    let (mut ws_stream, _) =
        ClientBuilder::from_uri(Uri::from_static("ws://127.0.0.1:2000"))
            .connect()
            .await?;

    let stdin = tokio::io::stdin();
    let mut stdin = BufReader::new(stdin).lines();


    // TODO: Підказку дивіться в описі завдання нижче.

}
```

Запуск бінарних файлів

Запустіть сервер за допомогою:

```
cargo run --bin server
```

і клієнт за допомогою:

```
cargo run --bin client
```

Завдання

- Реалізуйте функцію `handle_connection` у `src/bin/server.rs`.
  - Підказка: використовуйте `tokio::select!` для одночасного виконання двох завдань у безперервному циклі. Одне завдання отримує повідомлення від клієнта і транслює їх. Інше надсилає повідомлення, отримані сервером, клієнту.
- Завершіть основну функцію в `src/bin/client.rs`.
  - Підказка: як і раніше, використовуйте `tokio::select!` у безперервному циклі для одночасного виконання двох завдань: (1) читання повідомлень користувача зі стандартного вводу та надсилання їх на сервер, і (2) отримання повідомлень від сервера, і відображення їх для користувача.
- Необов'язково: коли ви закінчите, змініть код, щоб транслювати повідомлення всім клієнтам, крім відправника повідомлення.

66.3 Рішення

Вечеря філософів --- Async

```rust
use std::sync::Arc;
use tokio::sync::{mpsc, Mutex};
use tokio::time;

struct Fork;

struct Philosopher {
    name: String,
    left_fork: Arc<Mutex<Fork>>,
    right_fork: Arc<Mutex<Fork>>,
    thoughts: mpsc::Sender<String>,
}

impl Philosopher {
    async fn think(&self) {
        self.thoughts
            .send(format!("Еврика! {} має нову ідею!", &self.name))
            .await
            .unwrap();
    }

    async fn eat(&self) {
        // Продовжуємо пробувати, поки не знайдемо обидві виделки
        // Беремо виделки...
        let _left_fork = self.left_fork.lock().await;
        let _right_fork = self.right_fork.lock().await;

        println!("{} їсть...", &self.name);
        time::sleep(time::Duration::from_millis(5)).await;
```

```
        // Тут скидаються блокування
    }
}

static PHILOSOPHERS: &[&str] =
    &["Сократ", "Гіпатія", "Платоне", "Аристотель", "Піфагор"];

async fn main() {
    // Створюємо виделки
    let mut forks = vec![];
    (0..PHILOSOPHERS.len()).for_each(|_| forks.push(Arc::new(Mutex::new(Fork))));

    // Створюємо філософів
    let (philosophers, mut rx) = {
        let mut philosophers = vec![];
        let (tx, rx) = mpsc::channel(10);
        for (i, name) in PHILOSOPHERS.iter().enumerate() {
            let mut left_fork = Arc::clone(&forks[i]);
            let mut right_fork = Arc::clone(&forks[(i + 1) % PHILOSOPHERS.len()]);
            if i == PHILOSOPHERS.len() - 1 {
                std::mem::swap(&mut left_fork, &mut right_fork);
            }
            philosophers.push(Philosopher {
                name: name.to_string(),
                left_fork,
                right_fork,
                thoughts: tx.clone(),
            });
        }
        (philosophers, rx)
        // tx відкидається тут, тому нам не потрібно явно відкидати його пізніше
    };

    // Змусимо їх думати і їсти
    for phil in philosophers {
        tokio::spawn(async move {
            for _ in 0..100 {
                phil.think().await;
                phil.eat().await;
            }
        });
    }

    // Вивести свої думки
    while let Some(thought) = rx.recv().await {
        println!("Є така думка: {thought}");
    }
}
```

Програма широкомовного чату

src/bin/server.rs:

```
use futures_util::sink::SinkExt;
use futures_util::stream::StreamExt;
use std::error::Error;
use std::net::SocketAddr;
use tokio::net::{TcpListener, TcpStream};
use tokio::sync::broadcast::{channel, Sender};
use tokio_websockets::{Message, ServerBuilder, WebSocketStream};

async fn handle_connection(
    addr: SocketAddr,
    mut ws_stream: WebSocketStream<TcpStream>,
    bcast_tx: Sender<String>,
) -> Result<(), Box<dyn Error + Send + Sync>> {

    ws_stream
        .send(Message::text("Ласкаво просимо до чату! Введіть повідомлення".to_string()))
        .await?;
    let mut bcast_rx = bcast_tx.subscribe();

    // Безперервний цикл для одночасного виконання двох задач: (1) отримання
    // повідомлень з `ws_stream` та їх трансляції, та (2) отримання
    // повідомлень на `bcast_rx` і відправлення їх клієнту.
    loop {
        tokio::select! {
            incoming = ws_stream.next() => {
                match incoming {
                    Some(Ok(msg)) => {
                        if let Some(text) = msg.as_text() {
                            println!("Від клієнта {addr:?} {text:?}");
                            bcast_tx.send(text.into())?;
                        }
                    }
                    Some(Err(err)) => return Err(err.into()),
                    None => return Ok(()),
                }
            }
            msg = bcast_rx.recv() => {
                ws_stream.send(Message::text(msg?)).await?;
            }
        }
    }
}

async fn main() -> Result<(), Box<dyn Error + Send + Sync>> {
    let (bcast_tx, _) = channel(16);

    let listener = TcpListener::bind("127.0.0.1:2000").await?;
    println!("слухаємо на порту 2000");
```

```
    loop {
        let (socket, addr) = listener.accept().await?;
        println!("Нове з'єднання з {addr:?}");
        let bcast_tx = bcast_tx.clone();
        tokio::spawn(async move {
            // Обернути необроблений TCP потік у веб-сокет.
            let ws_stream = ServerBuilder::new().accept(socket).await?;

            handle_connection(addr, ws_stream, bcast_tx).await
        });
    }
}
```

src/bin/client.rs:

```
use futures_util::stream::StreamExt;
use futures_util::SinkExt;
use http::Uri;
use tokio::io::{AsyncBufReadExt, BufReader};
use tokio_websockets::{ClientBuilder, Message};

async fn main() -> Result<(), tokio_websockets::Error> {
    let (mut ws_stream, _) =
        ClientBuilder::from_uri(Uri::from_static("ws://127.0.0.1:2000"))
            .connect()
            .await?;

    let stdin = tokio::io::stdin();
    let mut stdin = BufReader::new(stdin).lines();

    // Безперервний цикл для одночасної відправки та отримання повідомлень.
    loop {
        tokio::select! {
            incoming = ws_stream.next() => {
                match incoming {
                    Some(Ok(msg)) => {
                        if let Some(text) = msg.as_text() {
                            println!("З сервера: {}", text);
                        }
                    },
                    Some(Err(err)) => return Err(err.into()),
                    None => return Ok(()),
                }
            }
            res = stdin.next_line() => {
                match res {
                    Ok(None) => return Ok(()),
                    Ok(Some(line)) => ws_stream.send(Message::text(line.to_string())).aw
                    Err(err) => return Err(err.into()),
                }
            }
```

```
            }
        }
    }
```

Частина XV

Заключні слова

Розділ 67

Дякую!

Дякуємо, що прослухали курс Comprehensive Rust 🦀! Сподіваємося, вам сподобалось і було корисно.

Нам було дуже весело створювати курс. Курс не ідеальний, тож якщо ви помітили будь-які помилки або маєте ідеї щодо покращення, зв'яжіться з нами на GitHub. Ми будемо раді почути від вас.

- Thank you for reading the speaker notes! We hope they have been useful. If you find pages without notes, please send us a PR and link it to issue #1083. We are also very grateful for fixes and improvements to the existing notes.

Розділ 68

Глосарій

Нижче наведено глосарій, який має на меті дати коротке визначення багатьох термінів Rust. У перекладах він також допомагає пов'язати термін з англійським оригіналом.

- виділяти:
  Динамічний розподіл пам'яті на купі.
- аргумент:
  Інформація, яка передається у функцію або метод.
- асоційований тип:
  Тип, пов'язаний з певним трейтом. Корисний для визначення взаємозв'язку між типами.
- Rust на голому залізі:
  Низькорівнева розробка Rust, часто розгорнута на системі без операційної системи. See Bare-metal Rust. Дивіться Rust на голому залізі.
- блок:
  Дивіться Блоки та *область видимості*.
- позичати:
  Дивіться Запозичення.
- перевірка запозичень:
  Частина компілятора Rust, яка перевіряє допустимість усіх запозичень.
- дужка:
  { та }. Також називаються *фігурними дужками*, вони розмежовують *блоки*.
- збірка:
  Процес перетворення вихідного коду у виконуваний код або придатну для використання програму.
- виклик:
  Виклик або виконання функції або методу.
- канал:
  Використовується для безпечної передачі повідомлень між потоками.
- Comprehensive Rust 🦀:
  Ці курси мають спільну назву Comprehensive Rust 🦀.
- одночасність виконання:
  Виконання декількох завдань або процесів одночасно.
- Одночасність виконання у Rust:
  Дивіться Одночасність виконання у Rust.
- константа:

Значення, яке не змінюється під час виконання програми.
- потік керування:
  Порядок, у якому виконуються окремі оператори або інструкції у програмі.
- крах:
  Неочікуваний і некерований збій або завершення роботи програми.
- перелік:
  Тип даних, який містить одну з декількох іменованих констант, можливо, з асоційованим кортежем або структурою.
- помилка:
  Неочікувана умова або результат, що відхиляється від очікуваної поведінки.
- обробка помилок:
  Процес управління та реагування на помилки, що виникають під час виконання програми.
- вправа:
  Завдання або проблема, призначена для практики та перевірки навичок програмування.
- функція:
  Багаторазово використовуваний блок коду, який виконує певне завдання.
- збирач сміття:
  Механізм, який автоматично звільняє пам'ять, зайняту об'єктами, які більше не використовуються.
- узагальнення:
  Це можливість написання коду із заповнювачами для типів, що дозволяє повторно використовувати код з різними типами даних.
- незмінний:
  Неможливо змінити після створення.
- інтеграційний тест:
  Тип тесту, який перевіряє взаємодію між різними частинами або компонентами системи.
- ключове слово:
  Зарезервоване слово в мові програмування, яке має певне значення і не може використовуватися як ідентифікатор.
- бібліотека:
  Колекція попередньо скомпільованих процедур або коду, які можуть бути використані програмами.
- макрос:
  Макроси використовуються, коли звичайних функцій недостатньо. Типовим прикладом є `format!`, який приймає змінну кількість аргументів, що не підтримується функціями Rust.
- `main` функція:
  Rust-програми починають виконуватися з функції `main`.
- match:
  Конструкція потоку керування у Rust, яка дозволяє виконувати шаблонний пошук за значенням виразу.
- витік пам'яті:
  Ситуація, коли програма не звільнює пам'ять, яка більше не потрібна, що призводить до поступового збільшення використання пам'яті.
- метод:
  Функція, пов'язана з об'єктом або типом у Rust.
- модуль:
  Простір імен, який містить визначення, такі як функції, типи або трейти, для

організації коду в Rust.
- move:
  Передача права власності на значення від однієї змінної до іншої у Rust.
- мутабельний:
  Це властивість у Rust, яка дозволяє змінювати змінні після того, як їх було оголошено.
- володіння:
  Концепція в Rust, яка визначає, яка частина коду відповідає за управління пам'яттю, пов'язаною зі значенням.
- паніка:
  Невиправна помилка у Rust, яка призводить до завершення роботи програми.
- параметр:
  Значення, яке передається у функцію або метод при її виклику.
- шаблон:
  Комбінація значень, літералів або структур, які можна зіставити з виразом у Rust.
- корисне навантаження:
  Дані або інформація, яку несе повідомлення, подія або структура даних.
- програма:
  Набір інструкцій, які комп'ютер може виконати, щоб виконати певне завдання або вирішити певну проблему.
- мова програмування:
  Формальна система, що використовується для передачі інструкцій комп'ютеру, наприклад, Rust.
- приймач:
  Перший параметр у методі Rust, який представляє екземпляр, на якому викликається метод.
- підрахунок посилань:
  Метод керування пам'яттю, в якому відстежується кількість посилань на об'єкт, і об'єкт звільняється, коли цей показник досягає нуля.
- return:
  Ключове слово у Rust, яке використовується для позначення значення, що повертається з функції.
- Rust:
  Мова системного програмування, яка фокусується на безпеці, продуктивності та одночасності виконання.
- Основи Rust:
  Дні з 1 по 4 цього курсу.
- Rust в Android:
  Дивіться Rust в Android.
- Rust в Chromium:
  Дивіться Rust в Chromium.
- безпечний:
  Відноситься до коду, який дотримується правил власності та запозичень Rust, запобігаючи помилкам, пов'язаним з пам'яттю.
- область видимості:
  Область програми, де змінна є дійсною і може бути використана.
- стандартна бібліотека:
  Колекція модулів, що забезпечують необхідну функціональність у Rust.
- static:
  Ключове слово у Rust, що використовується для визначення статичних змінних або елементів зі `'static` часом життя.

- string:
  Тип даних, що зберігає текстові дані. Дивіться Strings для отримання додаткової інформації.
- struct:
  Комбінований тип даних у Rust, який об'єднує змінні різних типів під одним іменем.
- test:
  Модуль Rust, що містить функції, які перевіряють коректність інших функцій.
- потік:
  Окрема послідовність виконання в програмі, що дозволяє одночасне виконання.
- безпека потоків:
  Властивість програми, яка забезпечує коректну поведінку в багатопотоковому середовищ.
- трейт:
  Набір методів, визначених для невідомого типу, що забезпечує можливість досягнення поліморфізму у Rust.
- обмеження трейту:
  Абстракція, в якій ви можете вимагати, щоб типи реалізовували певні трейти, які вас цікавлять.
- кортеж:
  Комбінований тип даних, який містить змінні різних типів. Поля кортежу не мають імен, доступ до них здійснюється за їхніми порядковими номерами.
- тип:
  Класифікація, яка визначає, які операції можна виконувати над значеннями певного типу в Rust.
- виведення типу:
  Здатність компілятора Rust виводити тип змінної або виразу.
- невизначена поведінка:
  Дії або умови в Rust, які не мають визначеного результату, що часто призводить до непередбачуваної поведінки програми.
- об'єднання:
  Тип даних, який може містити значення різних типів, але лише по одному за раз.
- модульній тест:
  Rust має вбудовану підтримку для запуску невеликих модульних тестів і великих інтеграційних тестів. Дивіться Модульні тести.
- тип одиниці:
  Тип, що не містить даних, записаний як кортеж без членів.
- unsafe:
  Підмножина Rust, яка дозволяє викликати *невизначену поведінку*. Дивіться Небезпечний Rust.
- змінна:
  Ділянка пам'яті, в якій зберігаються дані. Змінні дійсні в межах *області видимості*.

Розділ 69

Інші ресурси Rust

Спільнота Rust створила безліч високоякісних і безкоштовних ресурсів онлайн.

Офіційна документація

Проект Rust містить багато ресурсів. Вони охоплюють Rust загалом:

- Мова програмування Rust: канонічна безкоштовна книга про Rust. Детально охоплює мову та містить кілька проектів для створення.
- Rust за прикладом: описує синтаксис Rust за допомогою серії прикладів, які демонструють різні конструкції. Іноді включає невеликі вправи, де вас просять розширити код у прикладах.
- Стандартна бібліотека Rust: повна документація стандартної бібліотеки для Rust.
- Довідник Rust: неповна книга, яка описує граматику та модель пам'яті Rust.

Більш спеціалізовані посібники розміщені на офіційному сайті Rust:

- The Rustonomicon: охоплює небезпечний Rust, зокрема роботу з необробленими покажчиками та взаємодію з іншими мовами (FFI).
- Асинхронне програмування в Rust: охоплює нову модель асинхронного програмування, яка була представлена після написання книги Rust.
- The Embedded Rust Book: ознайомлення з використанням Rust на вбудованих пристроях без операційної системи.

Неофіційний навчальний матеріал

Невелика добірка інших посібників і підручників для Rust:

- Вивчіть Rust небезпечним способом: розповідається про Rust з точки зору програмістів на C низького рівня.
- Rust для Embedded C програмістів: розповідається про Rust з точки зору розробників, які пишуть вбудоване програмне забезпечення на C.
- Rust для професіоналів: висвітлює синтаксис Rust, використовуючи порівняння з іншими мовами, такими як C, C++, Java, JavaScript та Python.
- Rust on Exercism: понад 100 вправ, які допоможуть вам вивчити Rust.

- Навчальний матеріал Ferrous: серія невеликих презентацій, що охоплюють базову та розширену частину мови Rust. Також розглядаються інші теми, такі як WebAssembly та async/await.
- Розширене тестування для прикладних програм Rust: воркшоп для самостійної роботи, який виходить за рамки вбудованого тестового фреймворку Rust. Він охоплює `googletest`, тестування знімками, імітацію, а також те, як написати свій власний тестовий інструментарій.
- Серія для початківців до Rust і Зробіть перші кроки з Rust: два посібники з Rust, призначені для нових розробників. Перший — це набір із 35 відео, а другий — набір із 11 модулів, які охоплюють синтаксис і базові конструкції Rust.
- Вивчіть Rust із надто великою кількістю пов'язаних списків: поглиблене вивчення правил керування пам'яттю Rust за допомогою реалізації кількох різних типів структур списків .

Будь ласка, перегляньте Маленьку книгу Rust книжок, щоб отримати ще більше книг Rust.

Розділ 70

Кредити

Цей матеріал базується на багатьох чудових джерелах документації Rust. Перегляньте сторінку інші ресурси, щоб отримати повний список корисних ресурсів.

Матеріали Comprehensive Rust надаються згідно з умовами ліцензії Apache 2.0, будь ласка, дивіться `LICENSE` для отримання детальної інформації.

Rust на прикладі

Деякі приклади та вправи скопійовано та адаптовано з Rust на прикладі. Будь ласка, перегляньте каталог `third_party/rust-by-example/` для отримання детальної інформації, включно з умовами ліцензії.

Rust on Exercism

Деякі вправи скопійовано та адаптовано з Rust on Exercism. Будь ласка, перегляньте каталог `third_party/rust-on-exercism/`, щоб отримати докладніші відомості, включно з умовами ліцензії.

CXX

У розділі Взаємодія з C++ використовується зображення з CXX. Будь ласка, дивіться каталог `third_party/cxx/` для отримання детальної інформації, включно з умовами ліцензії.