

Мова Програмування Rust

автори Steve Klabnik та Carol Nichols, за допомогою спільноти Rust

У цій версії тексту припускається, що ви використовуєте Rust 1.65 (випущено 03 листопада 2022 року) або пізніший. Див. розділ «Встановлення» [Розділу 1](#), щоб встановити або оновити Rust.

Книжка англійською у форматі HTML доступна онлайн за адресою <https://doc.rust-lang.org/stable/book/> і офлайн з встановленим Rust за допомогою `rustup`; запустіть `rustup docs --book`, щоб відкрити.

Також є декілька створених спільнотою [перекладів](#).

Цей текст англійською доступний в паперовому форматі та електронних книжках No Starch Press.



Хочете більш інтерактивного навчального досвіду? Спробуйте іншу версію книжки Rust, що включає: завдання, підсвічування, візуалізації і більше:
<https://rust-book.cs.brown.edu>

Передмова

Це не завжди можна ясно побачити, та мова програмування Rust принципово задумувалася для *розширення можливостей*: неважливо, код якого типу ви зараз пишете, та Rust надає вам більше можливостей сягати далі і програмувати впевнено у більшій кількості сфер, ніж раніше.

Візьмемо, наприклад, роботу "системного рівня", що має справу з низькорівневими деталями керування пам'яттю, представленням даних та конкурентністю. Традиційна ця галузь програмування розглядається як утаємничена, доступна лише небагатьом обраним, хто присвятив необхідну кількість років навчанню, щоб уникнути її сумнозвісних підводних каменів. І навіть ті, хто працює в ній, робить це з обережністю, інакше їхній код буде відкрито для експлойтів, збоїв і ушкоджень.

Rust пробиває ці бар'єри, усуваючи старі підводні камені і надаючи дружній, відшліфований набір інструментів, щоб допомогти вам у цьому шляху. Програмісти, яким треба "зануритися" у нижній рівень контролю, можуть зробити це за допомогою Rust, без прийняття традиційного ризику збоїв і дірок безпеки і без необхідності вивчати тонкощі мінливого набору інструментів. Навіть краще, ця мова розроблена, щоб природним чином привести вас до надійного коду, ефективного з точки зору швидкості та використання пам'яті.

Програмісти, які вже працюють з кодом низького рівня, можуть скористатися Rust для підвищення своїх амбіцій. Скажімо, додавання паралелізму в Rust є відносно малоризикованою операцією: компілятор виявить класичні помилки за вас. І ви можете повернути більш агресивні оптимізації на своєму коді з упевненістю, що ви не додасте випадково збої та вразливості.

Та Rust не обмежується низькорівневим програмуванням. Rust достатньо виразний та ергономічний, щоб зробити розробку застосунків командного рядка, вебсерверів і багатьох інших видів коду дуже приємною - ви знайдете прості приклади і того, й іншого далі в книжці. Робота з Rust дозволяє вам розбудовувати навички, що переходять з однієї сфери в іншу; ви можете вивчити Rust, написавши вебзастосунок, а потім застосувати ті ж навички вже до вашого Raspberry Pi.

Ця книжка повністю охоплює потенціал Rust для розширення можливостей його користувачів. Це дружній та доступний текст, покликаний допомогти вам підвищити рівень не лише вашого знання Rust, але також вашої рішучості та впевненості як програміста в цілому. Тож занурюйтеся, готуйтеся вчитися - і ласкаво просимо до спільноти Rust!

— Nicholas Matsakis and Aaron Turon

Вступ

Примітка: Ця редакція книжки така ж сама, як [Мова програмування Rust](#), доступна у форматі друку та електронної книжки у [No Starch Press](#).

Вітаємо вас у *Мові програмування Rust*, вступній книзі про Rust. Мова програмування Rust допоможе вам писати швидше та надійніше програмне забезпечення. Ергономіка високого рівня та контроль низького рівня часто є несумісними у дизайні мови програмування; Rust кидає виклик цій суперечності. Завдяки балансу потужних технічних можливостей та великого досвіду розробників Rust надає вам можливість контролювати деталі низького рівня (наприклад, використання пам'яті) без будь-яких клопотів, традиційно пов'язаних із таким контролем.

Для Кого Призначено Rust

Rust ідеально підходить багатьом людям з різних причин. Розгляньмо кілька найважливіших груп.

Команди Розробників

Rust показав себе ефективним інструментом співпраці серед великих команд розробників з різним рівнем знання системного програмування. Низькорівневий код схильний до різних специфічних помилок, які в більшості інших мов програмування можна знайти лише завдяки глибокому тестуванню та ретельному перегляду коду досвідченими розробниками. У Rust компілятор виконує роль воротаря, відмовляючись компілювати код із цими неловними помилками, включно з помилками конкурентності. Працюючи разом із компілятором, команда може ефективніше витратити свій час, зосереджуючись на програмній логіці, а не на виловлюванні помилок.

Rust також привносить сучасні засоби розробники у світ системного програмування:

- Cargo, менеджер залежностей та інструмент побудови, включений у комплект, робить додавання, компіляцію та управління залежностями безболісним і послідовним в усій екосистемі Rust.
- Інструмент форматування Rustfmt забезпечує єдиний стиль кодування у різних розробників.

- Rust Language Server надає інтегрованому середовищу розробки (IDE) інтеграцію для завершення коду та вбудованих повідомлень про помилки.

Використовуючи ці та інші інструменти в екосистемі Rust, розробники можуть бути такими продуктивними під час написання коду системного рівня.

Студенти

Rust призначена для студентів та зацікавлених у вивченні системних концепцій. Використовуючи Rust, багато людей вивчили такі теми, як розробка операційних систем. Доброзичлива спільнота із задоволенням відповідає на питання студентів. Різними заходами, такими, як створення цієї книжки, команди Rust хочуть зробити системні концепції більш доступними для більшої кількості людей, особливо для програмістів-новачків.

Компанії

Сотні компаній, великих і малих, використовують Rust у виробництві для багатьох різних завдань, таких як інструменти командного рядка, вебслужби, інструменти DevOps, вбудовані пристрої, аудіо- та відеоаналіз та перекодування, криптовалюти, біоінформатика, пошукові системи, застосунки "Інтернету речей", машинне навчання та навіть суттєві частини веббраузера Firefox.

Розробники Відкритого Коду

Rust - для людей, які хочуть розбудовувати мову програмування Rust, спільноту, інструменти розробника та бібліотеки. Ми дуже хотіли б, щоб ви внесли свій внесок у мову Rust.

Люди, які Прагнуть Швидкості та Стабільності

Rust призначений для людей, які вимагають від мови швидкості та стабільності. Під швидкістю ми маємо на увазі як швидкість, з якою код Rust може виконуватися, так і швидкість, з якою Rust дозволяє вам писати програми. Перевірки компілятора Rust забезпечують стабільність під час розширення функціонала та рефакторинга. Це відрізняється від крихкого застарілого коду мовами без цих перевірок, який розробники часто бояться змінювати. Прагнучи до абстракцій з нульовою вартістю, конструкцій

вищого рівня, що компілюються в код нижчого рівня, так само швидкий, як і написаний вручну, Rust намагається зробити так, щоб безпечний код був швидким кодом.

Мова Rust сподівається підтримати й багатьох інших користувачів; тут згадані лише деякі з найбільш зацікавлених учасників. Загалом, найвищою метою Rust є усунення компромісів, на які програмісти йшли десятиліттями, забезпечуючи безпеку *та* продуктивність, швидкість *та* ергономічність. Спробуйте Rust і побачите, що його вибір вам підходить.

Для Кого Призначена Ця Книга

Ця книга передбачає, що ви вже писали код іншою мовою програмування, але не робить жодних припущень щодо того, якою саме. Ми спробували зробити матеріал широко доступним для тих, хто має найрізноманітніший досвід програмування. Ми не витрачаємо багато часу на розмови про те, *що* таке програмування або як думати про нього. Якщо ви зовсім ще новачок у програмуванні, вам краще буде прочитати книгу, яка спеціально написана для вступу до програмування.

Як Користуватися Цією Книгою

Загалом ця книжка передбачає, що ви читатимете її послідовно з початку до кінця. Пізніші розділи спираються на концепції, роз'яснені у попередніх розділах, а початкові розділи можуть не надто заглиблюватися в деталі конкретної теми; ми зазвичай повертаємося до таких тем у наступних розділах.

У цій книзі ви знайдете два типи розділів: розділи про концепції та розділи про проекти. У розділах про концепції ви дізнаєтесь про різні боки Rust. У розділах проєктів ми разом писатимемо невеликі програми, застосовуючи те, чому ви навчилися. Розділи 2, 12 та 20 - це розділи про проєкти; решта - це розділи про концепції.

Розділ 1 пояснює, як встановити Rust, як написати програму "Hello, world!" і як користуватися Cargo, менеджером пакунків і інструментом побудови Rust. Розділ 2 є практичним вступом до написання програми на Rust, де ви створите гру з відгадуванням чисел. Тут ми висвітлюємо концепції на високому рівні, а пізніші розділи нададуть додаткові подробиці. Якщо ви хочете одразу зануритися в мову, Розділ 2 дає добрий початок. Розділ 3 охоплює функціонал Rust, схожих на аналогічний в інших мовах програмування, а в Розділі 4 ви дізнаєтесь про систему володіння Rust. Якщо ви особливо скрупульозний учень і вважаєте за краще вивчити кожну деталь, перш ніж переходити до

наступної, можливо, ви захочете пропустити Розділ 2 і перейти прямо до Розділу 3, повернувшись до Розділу 2, коли захочете застосувати вивчене у проєкті.

У Розділі 5 обговорюються структури та методи, а в Розділі 6 - енуми, вирази `match` та керівні конструкції `if let`. Структури та енуми використовуються для створення власних типів у Rust.

У Розділі 7 ви дізнаєтесь про систему модулів Rust та про правила доступу для організації вашого коду та його публічного програмного інтерфейсу (API). У Розділі 8 розглядаються деякі узагальнені структури даних - колекції, які надає стандартна бібліотека, такі як вектори, рядки та геш-таблиці. Розділ 9 досліджує філософію та техніки обробки помилок Rust.

Розділ 10 заглиблюється в узагальнене програмування, трейти та часи існування, що надає вам змогу визначати код, застосований до різних типів. У Розділі 11 йдеться про тестування, яке є необхідним для забезпечення правильної логіки вашої програми навіть за гарантій безпеки Rust. У Розділі 12 ми створимо власну реалізацію підмножини функціонала інструмента командного рядка `grep`, який шукає заданий текст у файлах. Для цього ми використаємо багато концепцій, обговорених у попередніх розділах.

Розділ 13 досліджує замикання та ітератори - особливості Rust, які походять з функціональних мов програмування. У Розділі 14 ми детальніше розглянемо Cargo і поговоримо про кращі практики спільного використання ваших бібліотек. Розділ 15 обговорює розумні вказівники, надані стандартною бібліотекою, та трейти, що забезпечують їхній функціонал.

У Розділі 16 ми розглянемо різні моделі конкурентного програмування та поговоримо про те, як Rust допомагає вам програмувати декілька потоків без остраху. У Розділі 17 розглядається порівняння ідіом Rust із об'єктноорієнтованими принципами програмування, які вам можуть бути знайомі.

Розділ 18 - це посібник зі шаблонів та зіставлення з шаблонами, які є потужними способами вираження ідей у програмах Rust. Розділ 19 містить вінегрет із просунутих цікавих тем, включно з небезпечним Rust, макросами та різним про час існування, трейти, типи, функції та замикання.

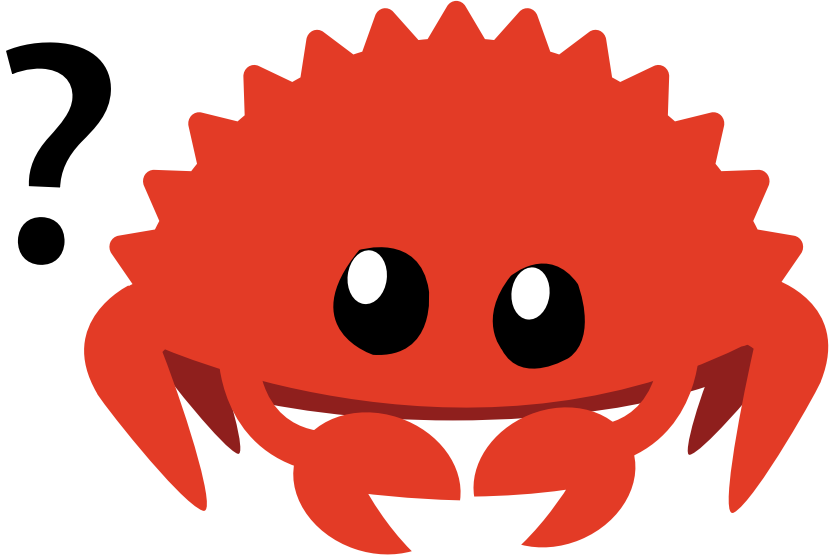
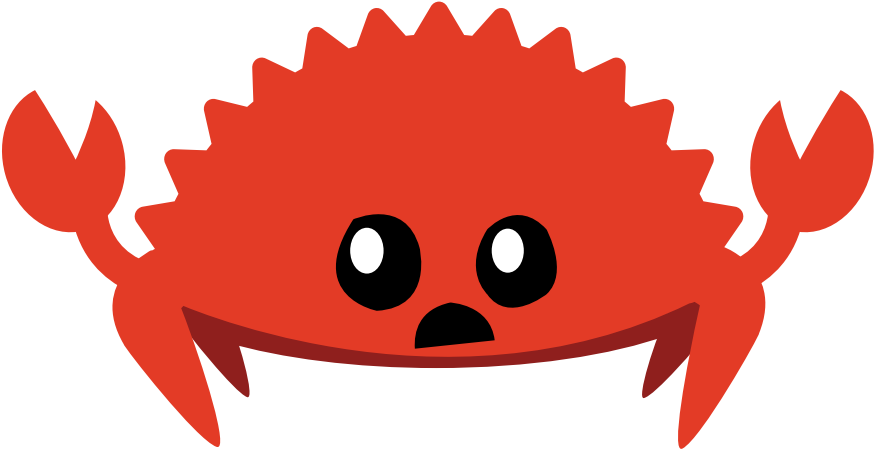
In Chapter 20, we'll complete a project in which we'll implement a low-level multithreaded web server!

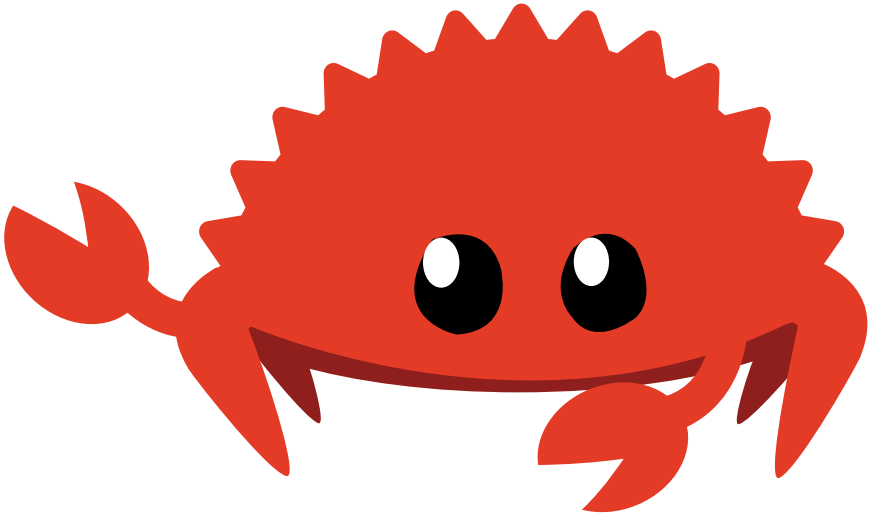
Нарешті, додатки містять корисну інформацію про мову у більш довідковому форматі. Додаток А охоплює ключові слова Rust, Додаток В охоплює оператори та символи Rust, Додаток С охоплює трейти, що їх можна успадковувати, надані стандартною бібліотекою, Додаток D охоплює деякі корисні інструменти розробки, а Додаток E пояснює видання

Rust. У Додатку F ви можете знайти переклади книги, а в Додатку G ми висвітлюємо, як робиться Rust, і що таке нічний Rust.

Немає неправильного способу прочитати цю книгу: якщо ви хочете пропустити щось - вперед! Можливо, вам доведеться повернутися до попередніх розділів, якщо ви відчуєте, що заплуталися. Але робіть все, що вам підходить.

Важливою складовою у навчанні Rust є навчання читати повідомлення про помилки, які відображає компілятор: вони спрямовують вас до робочого коду. Відтак, ми наведемо багато прикладів, які не компілюються, разом із повідомленням про помилку, яке компілятор покаже вам у кожній ситуації. Знайте, що якщо ви введете та запустите випадковий приклад, він може не скомпілюватись! Не забудьте прочитати навколишній текст, щоб побачити, чи приклад, який ви намагаєтесь запустити, мав на меті помилку. Ferris також допоможе вам розрізнити код, що не має працювати:

Ферріс	Значення
	Цей код не компілюється!
	Цей код призводить до паніки!

Ферріс	Значення
	Цей код не робить того, що від нього очікували.

У більшій частині ситуацій, ми направимо вас до правильної версії коду, що не компілюється.

Початковий Код

Файли з початковим кодом, з яких створено цю книжку, можна знайти на [GitHub](https://github.com).

Починаємо

Почнемо вашу подорож по Rust! Вивчати треба багато, але кожна подорож десь починається. В цьому розділі ми розглянемо:

- Встановлення Rust на Linux, macOS та Windows
- Написання програми, яка виводить `Hello, world!`
- Використання `cargo`, менеджера пакунків і системи збірки Rust

Встановлення

Наш перший крок - встановити Rust. Ми завантажимо Rust за допомогою `rustup`, інструмента командного рядка для керування виданнями Rust і пов'язаних інструментів. Для завантаження вам знадобиться з'єднання з Інтернетом.

Примітка: Якщо ви з якихось причин не бажаєте використовувати `rustup`, прочитайте [сторінку інших методів встановлення Rust](#), щоб дізнатися про інші варіанти.

Наступні кроки встановлять найостаннішу стабільну версію компілятора Rust. Принципи стабільності Rust гарантують, що всі приклади в цій книжці, які можна скомпілювати, будуть компілюватися в новіших версіях Rust. Повідомлення можуть незначно змінюватися від версії до версії, бо Rust часто покращує повідомлення і попередження про помилки. Іншими словами, будь-яка новіша стабільна версія Rust, яку ви встановите за цією інструкцією, має працювати відповідно до змісту цієї книжки.

Нотація Командного Рядка

У цьому розділі та надалі в книжці ми використовуватимемо команди терміналу. Рядки, що треба вводити в термінал, починаються з `$`. Не треба вводити сам символ `$`; це запрошення командного рядка, що лише позначає початок команди. Рядки, що не починаються з `$` зазвичай показують те, що виводить попередня команда. Приклади, специфічні для PowerShell, будуть починатися на `>` замість `$`.

Встановлення `rustup` на Linux або macOS

Якщо ви користувач Linux або macOS, відкрийте термінал і введіть цю команду:

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

Ця команда завантажить сценарій і почне встановлення інструменту `rustup`, що встановить останню стабільну версію Rust. Можливо, у вас запитатимуть ваш пароль. Якщо встановлення буде успішним, з'явиться цей рядок:

```
Rust is installed now. Great!
```

Крім того, вам знадобиться якийсь *компонувальник (linker)*, тобто програма, яку Rust використовує, щоб об'єднати результати компіляції в один файл. Швидше за все, він уже встановлений. Якщо ви отримаєте повідомлення про помилки компоувальника, вам слід встановити компілятор C, який зазвичай включає компоувальник. Компілятор C також корисний, бо деякі поширені пакунки Rust залежать від коду на C і потребуватимуть компілятора C.

На macOS, ви можете отримати C компілятор, виконавши команду:

```
$ xcode-select --install
```

Користувачі Linux зазвичай мають встановлювати GCC або Clang, відповідно до документації свого дистрибутиву. Скажімо, якщо ви використовуєте Ubuntu, ви можете встановити пакунок `build-essential`.

Встановлення `rustup` на Windows

У Windows перейдіть за адресою <https://www.rust-lang.org/tools/install> і дотримуйтесь вказаних там інструкцій для встановлення Rust. У певний момент встановлення ви отримаєте повідомлення, що вам також знадобляться інструменти збірки MSVC для Visual Studio 2013 чи пізнішої.

Щоб отримати інструменти збірки, вам потрібно встановити [Visual Studio 2022](#). На питання, які робочі завантаження потрібно встановити, вкажіть:

- “Desktop Development with C++”
- SDK для Windows 10 чи 11
- Англійський мовний компонент, а також будь-який інший мовний пакет на ваш вибір

Надалі книжка використовує команди, які працюють як у `cmd.exe`, так і в PowerShell. Якщо будуть відмінності, ми пояснимо, що робити.

Виправлення Несправностей

Щоб перевірити, чи правильно встановлено Rust, відкрийте оболонку і введіть рядок:

```
$ rustc --version
```

Ви маєте побачити номер версії, хеш коміту і дату коміту останньої стабільної версії, яку було випущено, в наступному форматі:

```
rustc x.y.z (abccabccc yyyy-mm-dd)
```

Якщо ви це бачите, Rust було успішно встановлено! Якщо ви не бачите цю інформацію, перевірте, чи є Rust у системній змінній `%PATH%`.

У Windows CMD наберіть:

```
> echo %PATH%
```

У PowerShell наберіть:

```
> echo $env:Path
```

У Linux і macOS наберіть:

```
$ echo $PATH
```

Якщо все правильно і Rust все ще не працює, є декілька місць, де ви можете звернутися по допомогу. Дізнайтеся, як зв'язатися з іншими рустцеанцями (так ми себе називаємо, від англ. crustacean - "ракоподібний") на [сторінці спільноти](#).

Оновлення та Видалення

Після встановлення Rust за допомогою `rustup` легко можна оновитися до нової версії після її виходу. З командної оболонки запустіть такий сценарій оновлення:

```
$ rustup update
```

Щоб видалити Rust і `rustup`, виконайте такий сценарій видалення з командної оболонки:

```
$ rustup self uninstall
```

Локальна Документація

Установлений Rust також включає локальну копію документації, тож ви можете читати її в офлайн. Запустіть `rustup doc`, щоб відкрити локальну документацію у веббраузері.

Кожного разу, коли ви зустрінете тип чи функцію у стандартній бібліотеці й не будете певні, що воно робить чи як ним користатися, скористайтеся документацією прикладного програмного інтерфейсу (API), щоб з'ясувати це!

Hello, World!

Після встановлення Rust настав час написати першу програму цією мовою. Давно стало традицією при вивченні нової мови програмування писати маленьку програму, що виводить на екран текст `Hello, world!`, і ми не будемо відступати від цієї традиції.

Примітка: Ця книжка передбачає базове знайомство із командним рядком. Rust як така не висуває особливих вимог до редакторів, інструментів і розміщення коду, тому якщо вам зручніше використовувати інтегроване середовище розробки (IDE) замість командного рядка, можете користуватися вашим улюбленим IDE. Багато сучасних IDE певною мірою підтримують Rust; зверніться до документації IDE, щоб дізнатися більше. Останнім часом команда Rust зосередилася на підтримці IDE за допомогою `rust-analyzer`. Перегляньте [Додаток D](#), для більш докладної інформації.

Створення Теки Проєкту

Для початку, створіть теку для розміщення вашого Rust коду. Для Rust немає значення, де розміщено ваш код, але для вправ і проєктів у цій книжці ми рекомендуємо зробити теку *projects* у вашій домашній теці і тримати всі проєкти там.

Запустіть термінал і введіть такі команди, щоб створити теку *projects* та теку для проєкту "Hello, world!" усередині теки *projects*.

У Linux, macOS та PowerShell на Windows, введіть це:

```
$ mkdir ~/projects
$ cd ~/projects
$ mkdir hello_world
$ cd hello_world
```

Для Windows CMD введіть це:

```
> mkdir "%USERPROFILE%\projects"
> cd /d "%USERPROFILE%\projects"
> mkdir hello_world
> cd hello_world
```

Написання і Запуск Rust-Програми

Тепер створіть новий вихідний файл і назвіть його *main.rs*. Файли Rust завжди закінчуються розширенням *.rs*. Якщо у назві файлу використовується більш ніж одне слово, домовлено для розділення використовувати підкреслення. Наприклад, можна назвати файл *hello_world.rs*, але не *helloworld.rs*.

Тепер відкрийте щойно створений файл *main.rs* і наберіть код з Блоку коду 1-1.

Файл: main.rs

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

Блок коду 1-1: програма, що виводить **Hello, world!**

Збережіть цей файл і поверніться до вікна терміналу у теці *~/projects/hello_world*. На Linux або macOS наберіть такі команди, щоб скомпілювати та запустити файл:

```
$ rustc main.rs  
$ ./main  
Hello, world!
```

У Windows запустіть команду **.\main.exe** замість **./main**:

```
> rustc main.rs  
> .\main.exe  
Hello, world!
```

Незалежно від вашої операційної системи, у терміналі буде виведено рядок **Hello, world!**. Якщо він не вивівся, зверніться до підрозділу [Розв'язання проблем](#) розділу Встановлення, щоб дізнатися, як отримати допомогу.

Якщо вивелося **Hello, world!** - вітаємо! Ви щойно офіційно написали програму мовою Rust. Тобто ви стали Rust програмістом! Ласкаво просимо!

Анатомія Програми Rust

Розгляньмо програму "Hello, world!" докладніше. Ось перший шматок пазла:

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

Ці рядки визначають функцію, що зветься `main`. Функція `main` особлива: вона завжди є першим кодом, що запускається у кожній виконуваний програмі Rust. Перший рядок проголошує функцію з назвою `main` без параметрів і яка нічого не повертає. Якби були параметри, їхні імена треба було розмістити між дужками `()`.

Тіло функції обгорнуто у `{}`. Rust вимагає фігурних дужок навколо тіл усіх функцій. Вважається хорошим стилем розміщувати початкову дужку на тому ж рядку, що й проголошення функції, з відступом в один пробіл.

Примітка: Якщо ви бажаєте дотримуватися стандартного стилю у всіх проєктах Rust, ви можете скористатися інструментом автоматичного форматування `rustfmt` для форматування вашого коду в певному стилі (докладніше про `rustfmt` у [Додатку D](#)). Команда Rust додала цей інструмент до стандартного набору програм Rust, на кшталт `rustc`, тобто він уже має бути встановленим на вашому комп'ютері!

Тіло функції `main` містить такий код:

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

Цей рядок виконує всю роботу в цій маленькій програмі: виводить текст на екран. Тут треба зазначити чотири важливі деталі.

По-перше, у Rust заведено робити відступи в чотири пробіли, а не табуляцію.

По-друге, `println!` викликає макрос Rust. Якби він викликав функцію, то це виглядало б як `println` (без `!`). Ми поговоримо про макроси в Rust детальніше в Розділі 19. Поки що вам достатньо знати, що коли ви бачите `!`, це означає, що ви викликаєте макрос, а не звичайну функцію, і що макроси не завжди дотримуються тих самих правил, що функції.

По-третє, ви бачите стрічку `Hello, world!`. Ми передаємо цю стрічку як аргумент до `println!`, і вона виводиться на екран.

По-четверте, рядок завершується крапкою із комою `;`, що позначає, що цей вираз завершено, і можна починати наступний. Більшість рядків в коді Rust завершується крапкою із комою.

Компіляція та Запуск є Окремими Крокami

Ви щойно запустили новостворену програму, тож дослідимо кожен крок у цьому процесі.

Перед запуском програми Rust необхідно її скомпілювати за допомогою компілятора Rust, набравши команду `rustc` і передавши їй ім'я вихідного файлу, отак:

```
$ rustc main.rs
```

Якщо ви маєте досвід роботи з C чи C++, ви можете помітити, що це схоже на `gcc` чи `clang`. Після вдалої компіляції Rust створює двійковий виконуваний файл.

На Linux, macOS чи PowerShell на Windows можна побачити цей файл, ввівши команду `ls` у командній оболонці:

```
$ ls  
main  main.rs
```

На Linux і macOS ви побачите два файли. У PowerShell на Windows ви побачите ті ж три файли, що й за допомогою CMD. У CMD на Windows ви побачите таке:

```
> dir r /B %= опція /B означає показувати лише імена файлів %=  
main.exe  
main.pdb  
main.rs
```

Тут показано вихідний файл з розширенням `.rs`, виконуваний файл (`main.exe` на Windows, але просто `main` на інших платформах) і, на Windows, файл з інформацією для зневадження з розширенням `.pdb`. Тепер можна запустити `main` чи `main.exe`, ось так:

```
$ ./main # чи .\main.exe у Windows
```

Якщо `main.rs` - це ваша програма "Hello, world!", вона виведе `Hello, world!` у ваш термінал.

Якщо ви більше знайомі з динамічними мовами на кшталт Ruby, Python чи JavaScript, вам може бути незвичним, що компіляція і виконання програми - окремі кроки. Rust є *завчасно компільованою* мовою, тобто ви можете скомпілювати програму, передати виконуваний файл комусь іншому, і він зможе запустити її навіть якщо у нього не встановлено Rust. Якщо ви дасте комусь файл `.rb`, `.py` чи `.js`, то у нього має бути встановлена реалізація Ruby, Python чи Javascript (відповідно), але в тих мовах потрібна лише одна команда, щоб скомпілювати та запустити вашу програму. Всі переваги мови програмування мають свою ціну.

Проста компіляція за допомогою `rustc` годиться для простеньких програм, але зі зростанням вашого проєкту вам захочеться мати можливість керувати всіма параметрами і легко ділитися кодом. Наступний крок - інструмент Cargo, що допоможе вам писати програми Rust для реального світу.

Привіт, Cargo!

Cargo - це система збірки та пакетний менеджер Rust. Більшість рустаєанців використовують цей інструмент для керування проєктами Rust, бо Cargo виконує багато задач, таких, як збірка коду, завантаження бібліотек, від яких залежить ваш код, та збірка цих бібліотек. (Бібліотеки, що потрібні вашому коду, звуться *залежностями*.)

Найпростіші програми Rust, як та, яку ми щойно написали, не мають жодних залежностей. Якби ми зібрали проєкт "Hello world" за допомогою Cargo, то скористалися б тільки тією частиною Cargo, що відповідає за збірку коду. Коли ж ви писатимете складніші програми Rust, то додаватимете залежності, і якщо почнете проєкт за допомогою Cargo, додавати залежності буде значно легше.

Оскільки переважна більшість проєктів Rust використовують Cargo, надалі в книзі вважатиметься, що ви теж використовуєте Cargo. Cargo встановлюється з Rust, якщо ви скористалися офіційним встановлювачем, як сказано в підрозділі [Встановлення](#). Якщо ви встановили Rust у інший спосіб, перевірте, чи встановлений Cargo, ввівши це у свій термінал:

```
$ cargo --version
```

Якщо ви побачите номер версії, то Cargo встановлений! Але якщо ви бачите помилку на кшталт **не знайдено команду**, зверніться до документації по вашому методу встановлення, щоб визначити, як окремо встановити Cargo.

Створення Проєкту з Cargo

Створімо новий проєкт за допомогою Cargo і подивімося, як він відрізняється від нашого початкового проєкту Hello World. Поверніться до вашої теки *projects* (чи іншої, де ви зберегли ваш код) і введіть команди (незалежно від системи):

```
$ cargo new hello_cargo  
$ cd hello_cargo
```

Перша команда створює нову теку і проєкт, що зветься *hello_cargo*. Ми назвали наш проєкт *hello_cargo*, і Cargo створює свої файли у теці з такою назвою.

Перейдіть до теки *hello_cargo* і перегляньте файли. Ви побачите, що Cargo створив два файли і одну теку: *Cargo.toml* і теку *src* із файлом *main.rs*.

Також він розпочав новий репозиторій Git, додавши файл `.gitignore`. Файли Git не будуть створені, якщо ви запустите `cargo new` в уже створеному репозиторії Git; ви можете змінити цю поведінку за допомогою `cargo new --vcs=git`.

Примітка: Git - це поширена система контролю версій. Ви можете сказати `cargo new` використовувати іншу систему контролю версій чи не використовувати жодної за допомогою прапорця `--vcs`. Запустіть `cargo new --help`, щоб побачити можливі варіанти.

Відкрийте файл `Cargo.toml` у будь-якому текстовому редакторі. Він має виглядати десь так, як показано у Роздруку 1-2.

Файл: `Cargo.toml`

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

# See more keys and their definitions at https://doc.rust-
lang.org/cargo/reference/manifest.html

[dependencies]
```

Блок коду 1-2: Вміст файлу `Cargo.toml`, створеного командою `cargo new`

Це файл у форматі [TOML](#) (*Tom's Obvious, Minimal Language* - "Томова очевидна мінімальна мова"), який Cargo використовує як формат для конфігурації.

Перший рядок, `[package]` (пакет) - це заголовок розділу, що показує, що наступні інструкції стосуються конфігурації пакета. Коли ми додамо більше інформації до цього файлу, ми додамо й інші розділи.

Наступні три рядки встановлюють конфігураційну інформацію, потрібну Cargo для компілювання вашої програми: ім'я, версію і яке видання Rust використовувати. Про ключ `edition` (видання) детальніше розповідається в [Додатку E](#).

Останній рядок, `[dependencies]`, розпочинає розділ, де можна вказувати залежності вашого проекту. Пакети з кодом в Rust зветься *крейтами* (`crate`). Нам не потрібні інші крейти для цього проекту, але вони знадобляться для першого проекту у Розділі 2, і тоді ми скористаємося цим розділом.

Тепер відкрийте файл `src/main.rs` і подивіться на його вміст:

Файл: src/main.rs

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

Cargo створив для вас “Hello World!”, точно такий, який ми написали в Роздруку 1-1! Поки що відмінності між нашим попереднім проєктом та згенерованим Cargo полягає в тому, що Cargo розмістив код у теці *src* і додав конфігураційний файл *Cargo.toml* в основній теці.

Cargo очікує, що вихідні файли будуть розташовані в теці *src*, а основна тека міститиме лише README, ліцензійну інформацію, конфігураційні файли і все таке, що не стосується вашого коду. Використання Cargo допомагає організувати ваші проєкт. Все має своє місце, і все лежить на своїх місцях.

Якщо ви почали проєкт, що не використовує Cargo, як було із нашим проєктом “Hello, world!”, його можна перетворити на проєкт із підтримкою Cargo, перемістивши код до теки *src* і створивши відповідний файл *Cargo.toml*.

Збірка та Запуск Проєкту Cargo

Погляньмо, як відрізняється збірка і запуск програми “Hello, world!” за допомогою Cargo. Зберіть проєкт такими командами з теки *hello_cargo*:

```
$ cargo build  
Compiling hello_cargo v0.1.0 (file:///projects/hello_cargo)  
Finished dev [unoptimized + debuginfo] target(s) in 2.85 secs
```

Ця команда створить виконаний файл *target/debug/hello_cargo* (чи *target\debug\hello_cargo.exe* на Windows), а не в поточній теці. Оскільки усталена збірка - дебажна, Cargo розміщує двійковий файл у теці, що зветься *debug*. Виконуваний файл можна запустити такою командою:

```
$ ./target/debug/hello_cargo # чи .\target\debug\hello_cargo.exe на Windows  
Hello, world!
```

Якщо все пройшло добре, в термінал виведеться **Hello, world!**. Запуск **cargo build** уперше також призводить до створення Cargo нового файлу в теці верхнього рівня: *Cargo.lock*. Цей файл відстежує конкретні версії залежностей вашого проєкту. Цей проєкт не має залежностей, тому файл дещо порожній. Вам не треба нічого самостійно змінювати у цьому файлі, його вмістом займається Cargo.

Ми щойно зібрали проєкт за допомогою `cargo build` і запустили його за допомогою `target/debug/hello_cargo`, але ми також можемо використати `cargo run`, щоб скомпілювати код, а потім запустити отриманий виконуваний файл однією командою:

```
$ cargo run
  Finished dev [unoptimized + debuginfo] target(s) in 0.0 secs
  Running `target/debug/hello_cargo`
Hello, world!
```

Використання `cargo run` зручніше, ніж запам'ятовувати виконати `cargo build`, а потім писати весь шлях до двійкового файлу, тому більшість розробників використовують `cargo run`.

Зверніть увагу, що цього разу ми не побачили повідомлення про те, що Cargo компілює `hello_cargo`. Cargo зрозумів, що файли не змінилися, тому не перезібрав, а просто запустив двійковий файл. Якби ви змінили вихідний код, Cargo б довелося перебудувати проєкт перед виконанням, і ви б побачили такий вивід:

```
$ cargo run
  Compiling hello_cargo v0.1.0 (file:///projects/hello_cargo)
  Finished dev [unoptimized + debuginfo] target(s) in 0.33 secs
  Running `target/debug/hello_cargo`
Hello, world!
```

Крім того, Cargo має команду `cargo check`. Ця команда швидко перевіряє ваш код, щоб переконатися, що він компілюється, але не створює виконанного файлу:

```
$ cargo check
  Compiling hello_cargo v0.1.0 (file:///projects/hello_cargo)
  Finished dev [unoptimized + debuginfo] target(s) in 0.32 secs
```

Чому виконуваний файл може бути непотрібним? `cargo check` зазвичай працює значно швидше за `cargo build`, бо пропускає створення виконанного файлу. Якщо ви постійно перевіряєте свій код під час його написання, використання `cargo check` дозволить вам швидше дізнатися, чи ваш проєкт все ще компілюється! Ось чому багато ратацеанців запускають `cargo check` час від часу поки пишуть програму, щоб переконатися, що вона компілюється, а потім запускають `cargo build`, коли готові працювати з виконуваним файлом.

Підіб'ємо підсумок, що ж ми дізналися про Cargo:

- Ми можемо створити проєкт за допомогою `cargo new`.
- Ми можемо зібрати проєкт за допомогою `cargo build`.
- Ми можемо зібрати і запустити проєкт в одну дію за допомогою `cargo run`.

- Ми можемо зібрати проєкт без створення двійкового файлу для пошуку помилок за допомогою `cargo check`.
- Cargo зберігає результат збірки не в одній теці з кодом, а в теці `target/debug`.

Додаткова перевага використання Cargo полягає в тому, що його команди однакові незалежно від операційної системи, в якій ви працюєте. Тому з цього моменту ми більше не надаватимемо окремих команд для Linux, macOS чи Windows.

Збірка для Релізу

Коли ваш проєкт нарешті готовий для релізу, ви можете запустити `cargo build --release`, щоб скомпілювати його з оптимізаціями. Ця команда створить виконуваний файл у теці `target/release` замість `target/debug`. Ці оптимізації дозволяють коду Rust працювати швидше, але подовжують час, потрібний для компіляції програми. Ось чому є два різні профілі: один для розробки, щоб можна було перебудовувати часто і швидко, і другий для збірки фінальної програми, яку можна дати користувачеві, яку не треба часто перебудовувати і яка буде виконуватися якомога швидше. Якщо ви робите бенчмарк вашого коду, запускайте `cargo build --release` і робіть бенчмарк виконуваного файлу у `target/release`.

Cargo як Стандарт

У простих проєктах Cargo надає ненабагато більше можливостей за `rustc`, але в подальшому він виявить свою цінність, коли ваші програми стануть складнішими. Щойно програми доростають до декількох файлів чи потребують залежності, набагато простіше дозволити Cargo координувати збірку.

Хоча проєкт `hello_cargo` і нескладний, тепер він використовує багато інструментів, якими ви будете користуватися решту вашої кар'єри з Rust. Фактично, щоб працювати із будь-яким існуючим проєктом ви можете скористатися цими командами, щоб завантажити код за допомогою Git, перейти до теки проєкту і зібрати його:

```
$ git clone someurl.com/someproject
$ cd someproject
$ cargo build
```

Для отримання додаткової інформації про Cargo перегляньте [документацію](#).

Підсумок

Це був непоганий початок вашої подорожі по Rust! У цьому розділі ви навчилися:

- Встановлювати останню стабільну версію Rust за допомогою `rustup`
- Оновлюватися до нової версії Rust
- Відкривати локально встановлену документацію
- Писати і запускати програму “Hello, world!” безпосередньо з `rustc`
- Створювати і запускати новий проєкт за допомогою домовленостей Cargo

Настав час побудувати більш змістовну програму, щоб призвичаїтися до читання та написання коду Rust. У Розділі 2 ми створимо програму для відгадування числа. Якщо ви натомість бажаєте почати з вивчення загальних концепцій програмування в Rust, переходьте до Розділу 3, а потім поверніться до Розділу 2.

Програмування Гри Відгадайки

Розпочнемо вивчення Rust зі спільної розробки проєкту! Цей розділ ознайомить вас із кількома поширеними концепціями Rust, демонструючи як вони використовуються у реальній програмі. Ви дізнаєтеся про `let`, `match`, методи, асоційовані функції, використання зовнішніх крейтів і навіть більше! Наступні розділи розкриють ці концепції детальніше. У цьому розділі ви лише займатиметеся основами.

Ми розв'язуватимемо класичну задачу для програмістів-початківців: гру "відгадай число". Умови такі: програма генерує випадкове ціле число між 1 та 100. Потім пропонує гравцю ввести спробу відгадати. Після введення спроби вона скаже, чи число більше або менше за загадане. Якщо відгадано правильно, гра виведе привітання і припинить роботу.

Початок Нового Проєкту

Щоб розпочати новий проєкт, перейдіть до теки *projects*, яку ви створили у Розділі 1, і створіть новий проєкт за допомогою Cargo, ось так:

```
$ cargo new guessing_game
$ cd guessing_game
```

Перша команда, `cargo new`, приймає першим параметром ім'я проєкту (`guessing_game`). Друга команда переходить до теки нового проєкту.

Перегляньмо щойно створений файл *Cargo.toml*:

Файл: Cargo.toml

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

# See more keys and their definitions at https://doc.rust-
lang.org/cargo/reference/manifest.html

[dependencies]
```

Як ви вже бачили у Розділі 1, `cargo new` створює програму "Hello, world!". Подивімося, що міститься у файлі *src/main.rs*:

Файл: `src/main.rs`

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

Скомпілюймо цю програму "Hello, world!" і запустимо її за один крок за допомогою команди `cargo run`:

```
$ cargo run  
Compiling guessing_game v0.1.0 (file:///projects/guessing_game)  
Finished dev [unoptimized + debuginfo] target(s) in 1.50s  
Running `target/debug/guessing_game`  
Hello, world!
```

Команда `run` стає в пригоді, коли треба швидко розвивати проєкт, і ця гра є якраз таким проєктом: ми хочемо швидко тестувати кожен ітерацію перед тим, як переходити до наступної.

Відкрийте файл `src/main.rs`. Увесь код ми писатимемо у цьому файлі.

Обробляємо Здогадку

Перша частина програми буде просити у користувача ввести здогадку, обробляти те, що він увів, і перевіряти, чи ввів він дані у потрібній формі. Для початку, дозволимо користувачеві ввести здогадку. Введіть код з Блоку коду 2-1 до `src/main.rs`.

Файл: `src/main.rs`

```
use std::io;  
  
fn main() {  
    println!("Guess the number!");  
  
    println!("Please input your guess.");  
  
    let mut guess = String::new();  
  
    io::stdin()  
        .read_line(&mut guess)  
        .expect("Failed to read line");  
  
    println!("You guessed: {guess}");  
}
```

Блок коду 2-1: Код, що отримує здогадку у користувача і виводить її

Цей код містить багато інформації, тому розбиратимемо його рядок за рядком. Щоб отримати, що ввів користувач, і вивести результат, нам треба ввести бібліотеку введення/виведення `io` в область видимості. Бібліотека `io` входить до стандартної бібліотеки, що зветься `std`:

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

За замовчуванням Rust має набір елементів, визначених у стандартній бібліотеці, що їх він вводить до області видимості будь-якої програми. Цей набір зветься *прелюдією*, і ви можете побачити все, що в ній міститься, [в документації стандартної бібліотеки](#).

Якщо типу, який ви хочете використати, нема у прелюдії, вам доведеться явно вносити цей тип у область видимості за допомогою інструкції `use`. Використання бібліотеки `std::io` надає вам ряд корисних особливостей, включно з можливістю користувацького вводу.

Як ви вже бачили у Розділі 1, функція `main` є точкою входу у програму:

```
fn main() {
```

Синтаксична конструкція `fn` проголошує нову функцію, `()` показує, що вона не має параметрів, і фігурна дужка `{` починає тіло функції.

Як ви вже дізналися з того ж Розділу 1, `println!` - це макрос, що виводить стрічку на екран:

```
println!("Guess the number!");  
println!("Please input your guess.");
```

Цей код виводить повідомлення, що це за гра і запитує введення у користувача.

Зберігання Значень в Змінних

Тепер створімо *змінну* для зберігання того, що користувач увів, ось так:

```
let mut guess = String::new();
```

Тепер програма стає цікавішою! В цьому коротенькому рядку відбувається багато всього. Ми використовуємо інструкцію `let`, щоб створити змінну. Ось інший приклад:

```
let apples = 5;
```

Цей рядок створює нову змінну з назвою `apples` і зв'язує її зі значенням 5. У Rust змінні є немутабельними за замовчанням, тобто щойно ми надамо змінній значення, воно не зміниться. Детально ця концепція обговорюється в підрозділі "Змінні та мутабельність" Розділу 3. Щоб зробити змінну мутабельною, слід додати `mut` перед її іменем:

```
let apples = 5; // немутабельна  
let mut bananas = 5; // мутабельна
```

Примітка: синтаксична конструкція `//` починає коментар, що продовжується до кінця рядка. Rust ігнорує весь вміст коментаря. Про коментарі детальніше йдеться в Розділі 3.

Повернімося до нашої ігрової програми - відгадайки. Тепер ви знаєте, що `let mut guess` створить мутабельну змінну на ім'я `guess`. Знак рівності (`=`) каже Rust, що тепер ми хочемо зв'язати щось зі змінною. З правого боку знаку рівності знаходиться значення, з яким зв'язується `guess`, а саме результат виклику `String::new`, функції, що повертає новий екземпляр стрічки `String`. `String String` - це тип стрічки, що надається стандартною бібліотекою; це кодовані в UTF-8 шматки тексту, які можна нарощувати.

Синтаксична конструкція `::` в рядку `::new` позначає, що `new` - це асоційована функція типу `String`. Асоційована функція є реалізованою для типу, в цьому випадку `String`. Ця функція `new` створює нову, порожню `String`. Функція `new` зустрінеться вам у багатьох типах, оскільки це звичайна назва функції, що створює нове значення певного виду.

В цілому: рядок `let mut guess = String::new();` створив мутабельну змінну, що зараз зв'язана з новим, порожнім екземпляром `String`. Уф!

Отримання Вхідних Даних від Користувача

Згадаймо, що ми додали функціональність введення/виведення зі стандартної бібліотеки за допомогою `use std::io;` у першому рядку програми. Тепер викличмо функцію `stdin` з модуля `io`, що дозволить обробляти те, що вводить користувач:

```
io::stdin()  
    .read_line(&mut guess)
```


Якби ми не імпортували бібліотеку `io` за допомогою `use std::io` на початку програми, ми могли б використати цю функцію, написавши цей виклик як `std::io::stdin`. Функція `stdin` повертає екземпляр `std::io::Stdin`; цей тип являє собою дескриптор (handle) стандартного потоку введення термінала.

Далі рядок `.read_line(&mut guess)` викликає метод `read_line` дескриптора стандартного введення, щоб отримати, що ввів користувач. Ми також передаємо

`&mut guess` аргументом до `read_line`, щоб повідомити йому, до якої стрічки зберегти введення користувача. Повне завдання `read_line` - взяти те, що користувач набрав у стандартний потік введення і додати до стрічки (не перезаписавши її вміст), тому ми передаємо стрічку як аргумент. Стрічка-аргумент має бути мутабельною, щоб метод міг змінити її вміст.

`&` позначає, що цей аргумент - *посилання*, що дає вам можливість надати кільком частинам вашого коду доступ до одного фрагменту даних без кількаразового копіювання цих даних у пам'яті. Посилання - складна тема, але одна з основних переваг Rust полягає в безпеці та легкості використання посилань. Для завершення цієї програми вам не знадобляться особливо детальні знання про посилання. Поки що все, що вам треба знати - що посилання, як і зміні, типово є немутабельними. Тому необхідно писати `&mut guess`, а не просто `&guess`, щоб зробити його мутабельним. (Розділ 4 пояснить посилання ретельніше.)

Обробка Потенційної Невдачі з `Result`

Ми ще не закінчили розбиратися із цим рядком коду. Хоча ми обговорюємо третій рядок тексту, це все ще частина єдиного логічного рядка коду. Наступна частина - це ось цей метод:

```
.expect("Failed to read line");
```

Ми могли б написати цей код ось так:

```
io::stdin().read_line(&mut guess).expect("Failed to read line");
```

Однак один довгий рядок важко читати, тому краще його розділити. Коли ви викликаєте метод за допомогою синтаксичної конструкції `.method_name()` часто має сенс розпочати новий рядок і додати відступи, щоб розділити довгі рядки. Тепер розглянемо, що цей рядок робить.

Як уже було сказано, `read_line` додає те, що ввів користувач, у стрічку, яку ми передали як аргумент, але також повертає значення `Result`. Тип `Result` - це

перелік (enumeration), який часто звуть просто *енум*, і цей тип може перебувати в одному з кількох можливих станів. Кожен такий стан зветься *варіантом*.

Розділ 6 розповість про енуми детальніше. Призначення типів `Result` - представлення інформації для обробки помилок.

`Result` має варіанти `Ok` та `Err`. Варіант `Ok` показує, що операція була вдалою, і всередині варіанту `Ok` знаходиться успішно згенероване значення. Варіант `Err` позначає невдачу, і містить інформацію, як і чому операція була невдалою.

Значення типу `Result`, як і значення будь-якого іншого типу, мають визначені для них методи. Екземпляр `Result` має доступний для виклику метод `expect`. Якщо цей екземпляр `Result` має значення `Err`, то `expect` викличе аварійне завершення програми та виведе повідомлення, яке ви передали до `expect` параметром. Якщо метод `read_line` поверне `Err`, це, швидше за все, станеться внаслідок помилки, яка станеться в операційній системі. Якщо цей екземпляр `Result` має значення `Ok`, `expect` візьме повернуте значення, яке знаходиться в `Ok`, і поверне тільки це значення, щоб ним можна було скористатися. В цьому випадку це значення - кількість байтів, введених користувачем до стандартного потоку.

Якщо ви не викличете `expect`, програма скомпілюється, проте ви отримаєте попередження:

```
$ cargo build
  Compiling guessing_game v0.1.0 (file:///projects/guessing_game)
warning: unused `Result` that must be used
--> src/main.rs:10:5
10 |         io::stdin().read_line(&mut guess);
    |         ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
    = note: `[warn(unused_must_use)]` on by default
    = note: this `Result` may be an `Err` variant, which should be handled

warning: `guessing_game` (bin "guessing_game") generated 1 warning
Finished dev [unoptimized + debuginfo] target(s) in 0.59s
```

Rust попереджає, що ви не використали значення `Result`, повернуте з `read_line`, що означає, що програма не обробляє можливу помилку.

Правильний спосіб пригнітити попередження - власне, обробити помилку, але оскільки ми в цьому випадку просто хочемо, щоб програма аварійно завершилася, якщо виникне проблема, то можемо скористатися `expect`. Ви дізнаєтеся про те, як відновити роботу програми при помилці, у **Розділі 9**.

Виведення Значень в Консолі із Заповнювачами `println!`

Якщо не враховувати завершувальної фігурної дужки, лишився лише один рядок, який ми ще не обговорили:

```
println!("You guessed: {guess}");
```

Цей рядок виводить стрічку, в якій ми зберегли те, що ввів користувач. Фігурні дужки `{}` - це заповнювач: можна уявити, що `{}` - клешні маленького краба, що тримає значення на місці. При виведенні значення змінної, назву змінної можна розмістити у фігурних дужках. При виведенні результату обчислення виразу розмістіть порожні фігурні дужки у форматній стрічці, а потім додайте за форматною стрічкою список, розділений комами, виразів, які треба вивести у кожному порожньому заповнювачі з фігурних дужок у тому самому порядку. Виведення змінної і результату обчислення виразу в одному виклику `println!` виглядатиме так:

```
let x = 5;  
let y = 10;  
  
println!("x = {x} and y + 2 = {}", y + 2);
```

Цей код виведе `x = 5 and y + 2 = 12`.

Тестування Першої Частини

Протестуємо першу частину гри "відгадай число". Запустіть її за допомогою `cargo run`:

```
$ cargo run  
Compiling guessing_game v0.1.0 (file:///projects/guessing_game)  
Finished dev [unoptimized + debuginfo] target(s) in 6.44s  
Running `target/debug/guessing_game`  
Guess the number!  
Please input your guess.  
6  
You guessed: 6
```

На цей момент перша частина гри завершена: ми отримуємо дані з клавіатури та виводимо їх.

Генерація Таємного Числа

Тепер нам треба згенерувати таємне число, яке користувач пробуватиме відгадати. Таємне число має бути різним кожного разу, щоб у гру було цікаво грати більше одного разу. Використаймо випадкове число від 1 до 100, щоб гра була не надто складною. Rust поки що не має функціональності для генерації випадкових чисел у стандартній бібліотеці; натомість команда Rust надає [крейт](#) `rand` з таким функціоналом.

Використання Крейта для Отримання Додаткової Функціональності

Пам'ятайте, що крейт є набором файлів вихідного коду Rust. Проєкт, що ми збираємо - це *двійковий крейт*, який є виконуваним. Крейт `rand` - це *бібліотечний крейт*, і він містить код, призначений для використання в інших програмах, та не може бути запущеним самостійно.

Використання зовнішніх крейтів - найсильніший бік Cargo. Перед тим, як писати код, що використовує `rand`, ми маємо змінити файл *Cargo.toml*, додавши туди крейт `rand` як залежність. Відкрийте цей файл і додайте такий рядок унизу, під заголовком секції `[dependencies]`, яку для вас створив Cargo. Переконайтеся, що зазначили `rand` точно так, як тут, із цим номером версії, інакше приклади коду з цього розділу можуть не запрацювати:

Файл: Cargo.toml

```
[dependencies]
rand = "0.8.5"
```

У файлі *Cargo.toml* все, що йде після заголовку секції, належить до цієї секції - до початку нової секції. У секції `[dependencies]` ви повідомляєте Cargo, від яких зовнішніх крейтів залежить ваш проєкт і які версії цих крейтів вам потрібні. У цьому випадку, ми зазначаємо крейт `rand` із семантичним версіюванням `0.8.5`. Cargo розуміє [Семантичне версіювання](#) (яке іноді звуть *SemVer*), що є стандартом для запису номерів версій. Запис `0.8.5` насправді є скороченням для `^0.8.5`, що означає будь-яку версію, не меншу за 0.8.5, але меншу за 0.9.0.

Cargo вважає ці версії сумісними з публічним API з версією 0.8.5, і ця специфікація гарантує, що ви отримаєте останній патч реліз, який все ще буде скомпільований з кодом у цьому розділі. У версій 0.9.0 і більших не гарантується збереження API, яке використовується подальшими прикладами.

Тепер, не змінюючи коду, побудуємо проєкт, як показано в Блоці коду 2-2.

```
$ cargo build
  Updating crates.io index
  Downloaded rand v0.8.5
  Downloaded libc v0.2.127
  Downloaded getrandom v0.2.7
  Downloaded cfg-if v1.0.0
  Downloaded ppv-lite86 v0.2.16
  Downloaded rand_chacha v0.3.1
  Downloaded rand_core v0.6.3
  Compiling libc v0.2.127
  Compiling getrandom v0.2.7
  Compiling cfg-if v1.0.0
  Compiling ppv-lite86 v0.2.16
  Compiling rand_core v0.6.3
  Compiling rand_chacha v0.3.1
  Compiling rand v0.8.5
  Compiling guessing_game v0.1.0 (file:///projects/guessing_game)
  Finished dev [unoptimized + debuginfo] target(s) in 2.53s
```

Блок коду 2-2: Вивід команди `cargo build` після додавання крейту `rand` як залежності

Ви можете бачити інші номери версій (але всі вони будуть сумісні з кодом, завдяки SemVer!) і різні рядки (залежно від операційної системи), і рядки можуть бути в іншому порядку.

Тепер, коли ми маємо зовнішню залежність, Cargo витягає останні версії всього, що нам треба, з *реєстру*, тобто копії даних з [Crates.io](https://crates.io). На crates.io в екосистемі Rust люди викладають свої проєкти Rust з відкритим кодом, щоб ними могли скористатися інші.

Після оновлення реєстру Cargo перевіряє секцію `[dependencies]` і завантажує крейти, вказані там, але яких у вас бракує. В цьому випадку, хоча ми вказали тільки залежність від `rand`, Cargo також завантажив інші крейти, від яких залежить робота `rand`. Після завантаження крейтів Rust їх компілює, а потім компілює проєкт із доступними залежностями.

Якщо ви знову запустите `cargo build`, не зробивши жодних змін, ви не отримаєте жодної відповіді окрім рядка `Finished`. Cargo знає, що він вже завантажив і скомпілював залежності, а ви не змінили нічого, що б їх стосувалося, у файлі *Cargo.toml*. Cargo також знає, що ви не змінили нічого у коді, тому він не буде його перекомпілювати. Оскільки роботи у Cargo немає, він просто завершується.

Якщо ви відкриєте файл *src/main.rs*, зробите тривіальну зміну, збережете і знову зберете, то побачите тільки два рядки виводу:

```
$ cargo build
  Compiling guessing_game v0.1.0 (file:///projects/guessing_game)
  Finished dev [unoptimized + debuginfo] target(s) in 2.53 secs
```

Ці рядки показують, що Cargo оновив збірку тільки вашою дрібною правкою до файлу `src/main.rs`. Залежності не змінилися, і Cargo знає, що може заново використати те, що він вже завантажив і скомпілював.

Забезпечення Відтворюваності Збірок із Файлом *Cargo.lock*

Cargo має механізм, що гарантує однаковість збірки того самого артефакту кожного разу, коли ви чи хтось інший збирає ваш код: Cargo використає тільки ті версії залежностей, які ви зазначили, доки ви не вкажете інші. Наприклад, якщо наступного тижня вийде `rand` версії `0.8.6`, що міститиме важливе виправлення помилки, але також міститиме регресію, що зламає ваш код. Щоб упоратися з цим, при першому запуску `cargo build` Rust створює файл *Cargo.lock*, що відтепер розміщується у теці *guessing_game*.

Коли ви збираєте проєкт вперше, Cargo визначає всі версії залежностей, що відповідають критерію, і записує їх до файлу *Cargo.lock*. Коли ви пізніше збиратимете проєкт, Cargo побачить, що файл *Cargo.lock* існує, і використає версії, зазначені там, а не буде наново робити всю роботу з визначення версій. Це дозволяє автоматично робити відтворювану збірку. Іншими словами, ваш проєкт залишиться на версії 0.8.5, доки ви самі не захочете оновити її, завдяки файлу *Cargo.lock*. Оскільки файл *Cargo.lock* важливий для відтворюваної збірки, він часто додається до контролю початкового коду разом із рештою коду в проєкті.

Оновлення Крейта для Отримання Нової Версії

Коли ж ви *хочете* оновити крейт, Cargo надає іншу команду, `update`, яка ігнорує файл *Cargo.lock* і визначає всі останні версії, що відповідають специфікаціям у *Cargo.toml*. Cargo запише ці версії до файлу *Cargo.lock*. Але за замовчанням Cargo шукатиме тільки версії, більші за 0.8.5 і менші 0.9.0. Якщо крейт `rand` вийшов у двох нових версіях, 0.8.6 та 0.9.0, то запустивши `cargo update` ви побачите таке:

```
$ cargo update
Updating crates.io index
Updating rand v0.8.5 -> v0.8.6
```

Cargo проігнорує реліз 0.9.0. Тут також можна звернути увагу на зміну у файлі *Cargo.lock* - версія крейта `rand`, яку ви використовуєте, тепер 0.8.6. Якщо вам потрібен `rand` версії 0.9.0 чи будь-якої версії у гілці 0.9.x, вам доведеться оновити файл *Cargo.toml*, щоб він мав такий вигляд:

```
[dependencies]
rand = "0.9.0"
```

Наступного разу, коли ви запустите `cargo build`, Cargo оновить реєстр доступних крейтів і заново перечитає вимоги до `rand` відповідно до вказаної вами нової версії.

Можна ще багато розповісти про [Cargo і його екосистему](#), яка обговорюється у Розділі 14, але поки що цього знати достатньо. Cargo робить використання бібліотек дуже простим, що дозволяє рачацеанцям писати менші проєкти, зібрані з кількох пакетів.

Генерація Випадкового Числа

Використаймо `rand` для генерації числа, що треба відгадати. Наступний крок - оновити `src/main.rs`, як показано в Блоці коду 2-3.

Файл: `src/main.rs`

```
use std::io;
use rand::Rng;

fn main() {
    println!("Guess the number!");

    let secret_number = rand::thread_rng().gen_range(1..=100);

    println!("The secret number is: {secret_number}");

    println!("Please input your guess.");

    let mut guess = String::new();

    io::stdin()
        .read_line(&mut guess)
        .expect("Failed to read line");

    println!("You guessed: {guess}");
}
```

Блок коду 2-3: Додаємо код, необхідний для генерації випадкового числа

Спершу ми додаємо рядок `use rand::Rng`. Трейт `Rng` визначає методи, які реалізує генератор випадкових чисел, і цей трейт має бути в області видимості, щоб ми могли скористатися цими методами. Розділ 10 розповість про трейти детальніше.

Далі ми додаємо всередині ще два рядки. У першому рядку ми викликаємо функцію `rand::thread_rng`, що дає нам генератор випадкових чисел, яким ми користуватимемося: він прив'язаний до потоку виконання, а його початкове значення задане операційною системою. Потім ми викликаємо метод генератора випадкових чисел `gen_range`. Цей метод визначається трейтом `Rng`, який ми внесли до області видимості інструкцією `use`

`range::Rng`. Метод `gen_range` приймає параметрами два числа і генерує випадкове число в діапазоні між ними. Вираз для діапазону, що ми його тут застосували, має форму `початок..=кінець` і включає нижню і верхню межі, тому треба вказувати `1..=100`, щоб отримати число між 1 та 100.

Примітка: Ви, звісно, не можете одразу знати, які трейти використати і які методи та функції викликати з крейта, тому кожен крейт має документацію з інструкцією до використання. Ще одна корисна можливість Cargo полягає в тому, що команда `cargo doc --open` збере на вашому комп'ютері документацію, надану всіма залежностями, і відкриє її у вашому переглядачі. Якщо вам цікавий інший функціонал, скажімо, крейту `rand`, запустіть `cargo doc --open` і клацніть `rand` на боковій панелі ліворуч.

Другий рядок, який ми додали до коду, виводить таємне число. Це корисно, поки ми розробляємо програму, щоб можна було перевірити її роботу, але ми видалимо його у фінальній версії. Буде не дуже цікаво, якщо програма виводитиме відповідь одразу по запуску!

Спробуємо запустити програму кілька разів:

```
$ cargo run
  Compiling guessing_game v0.1.0 (file:///projects/guessing_game)
  Finished dev [unoptimized + debuginfo] target(s) in 2.53s
  Running `target/debug/guessing_game`
Guess the number!
The secret number is: 7
Please input your guess.
4
You guessed: 4

$ cargo run
  Finished dev [unoptimized + debuginfo] target(s) in 0.02s
  Running `target/debug/guessing_game`
Guess the number!
The secret number is: 83
Please input your guess.
5
You guessed: 5
```

Ви маєте побачити різні випадкові числа, і вони мають бути між 1 та 100. Чудова робота!

Порівняння Здогадки з Таємним Числом

Тепер, коли ми маємо введення користувачем і випадкове число, ми можемо їх порівняти. Цей крок показано в Блоці коду 2-4. Зверніть увагу, що цей код ще не компілюється, як ми зараз пояснимо.

Файл: src/main.rs

```
use rand::Rng;
use std::cmp::Ordering;
use std::io;

fn main() {
    // --snip--

    println!("You guessed: {guess}");

    match guess.cmp(&secret_number) {
        Ordering::Less => println!("Too small!"),
        Ordering::Greater => println!("Too big!"),
        Ordering::Equal => println!("You win!"),
    }
}
```

Блок коду 2-4: Різні дії в залежності від порівняння двох чисел

Спершу ми додали ще одну інструкцію `use`, яка вводить тип `std::cmp::Ordering` зі стандартної бібліотеки в область видимості. Тип `Ordering` ("впорядкування") - це ще один еnum, що має варіанти: `Less` ("менше"), `Greater` ("більше"), and `Equal` ("дорівнює"). Це три можливі результати порівняння двох значень.

Потім ми додали в кінець коду п'ять нових рядків, в яких використали тип `Ordering`. Метод `cmp` порівнює два значення і може бути викликаний для всього, що можна порівнювати. Він приймає параметром посилання на те, що ви хочете порівнювати; тут він порівнює `guess` із `secret_number`. Потім він повертає варіант еnumу `Ordering`, який ми внесли у область видимості за допомогою інструкції `use`. Ми скористалися виразом `match`, щоб визначити, що робити далі залежно від варіанту `Ordering`, що його повернув виклик `cmp` зі значеннями `guess` та `secret_number`.

Вираз `match` складається з *рукавів*. Рукав складається зі *шаблону* (`<0>pattern</0>`) для порівняння та коду, який буде виконано, якщо значення, передане виразу `match`, відповідає шаблону цього рукава. Rust бере значення, передане `match`, і по черзі перевіряє шаблони рукавів. Шаблони та конструкція `match` - потужні засоби Rust, які дозволяють вам виражати різноманітні ситуації, які можуть трапитися вам при

програмуванні, і допомагають переконатися, що ви обробили їх усіх. Детально ці можливості будуть розглянуті в Розділах 6 і 18, відповідно.

Розберімо крок за кроком цей приклад з виразом `match`. Нехай користувач увів 50, а випадково згенероване цього разу таємне число - 38.

Коли код порівнює 50 і 38, метод `cmp` поверне `Ordering::Greater`, бо 50 більше за 38. Вираз `match` отримує значення `Ordering::Greater` і починає перевіряти шаблони кожного рукава. Він перевіряє шаблон першого рукава, `Ordering::Less`, і бачить, що значення `Ordering::Greater` не відповідає `Ordering::Less`, тому пропускає рукав і переходить до наступного рукава. Шаблон наступного рукава, `Ordering::Greater`, відповідає `Ordering::Greater`! Код цього рукава буде виконано і виведе на екран `Too big!`. Вираз `match` завершується після першого вдалого порівняння, тому останній рукав в цьому випадку не буде перевірено.

Але Блок коду 2-4 все ще не компілюється. Спробуймо його скомпілювати:

```
$ cargo build
  Compiling libc v0.2.86
  Compiling getrandom v0.2.2
  Compiling cfg-if v1.0.0
  Compiling ppv-lite86 v0.2.10
  Compiling rand_core v0.6.2
  Compiling rand_chacha v0.3.0
  Compiling rand v0.8.5
  Compiling guessing_game v0.1.0 (file:///projects/guessing_game)
error[E0308]: mismatched types
--> src/main.rs:22:21
   |
22 |         match guess.cmp(&secret_number) {
   |                ^^^^^^^^^^^^^^^^^^^^^^^ expected struct `String`, found integer
   |
   = note: arguments to this function are incorrect
   = note: expected reference `&String`
             found reference `&{integer}`
note: associated function defined here

For more information about this error, try `rustc --explain E0308`.
error: could not compile `guessing_game` due to previous error
```

Суть цієї помилки в тому, що тут є *невідповідні типи*. Rust має сильну, статичну систему типів. Разом із тим, він має систему виведення типів. Коли ми писали `let mut guess = String::new()`, Rust зміг вивести, що `guess` має бути типу `String` і не просив нас написати тип. `secret_number`, з іншого боку, числового типу. Кілька числових типів Rust можуть мати значення між 1 та 100: `i32`, знакове 32-бітне число; `u32`, беззнакове 32-бітне число; `i64`, знакове 64-бітне число і кілька інших. Як не вказати іншого, Rust за

замовчанням обере `i32`, і це й буде типом `secret_number`, якщо ви не додасте інформацію про тип деінде, щоб змусити Rust вивести інший числовий тип. Причина ж цієї помилки полягає в тому, що Rust не може порівнювати стрічку і числовий тип.

Зрештою, ми хочемо перетворити стрічку `String`, яку програма прочитала з клавіатури, в числовий тип, щоб можна було порівняти його як число зі таємним числом. Це можна зробити, додавши ще один рядок до функції `main`:

Файл: `src/main.rs`

```
// --snip--

let mut guess = String::new();

io::stdin()
    .read_line(&mut guess)
    .expect("Failed to read line");

let guess: u32 = guess.trim().parse().expect("Please type a number!");

println!("You guessed: {guess}");

match guess.cmp(&secret_number) {
    Ordering::Less => println!("Too small!"),
    Ordering::Greater => println!("Too big!"),
    Ordering::Equal => println!("You win!"),
}
```

Ось цей рядок:

```
let guess: u32 = guess.trim().parse().expect("Please type a number!");
```

Ми створили змінну з назвою `guess`. Але чекайте, в програмі вже ніби існує змінна з назвою `guess`? Так, але Rust дозволяє затінити попереднє значення `guess` новим. *Затінення* дозволяє нам наново використати ім'я змінної `guess`, щоб не довелося створювати дві окремі змінні на кшталт `guess_str` і `guess`. Про це детальніше піде у [Розділі 3](#) Розділ 3 детальніше розповідає про затінення, а поки що знайте, що ця особливість часто використовується, коли нам треба перетворити значення з одного типу в інший.

Ми зв'язали нову змінну з виразом `guess.trim().parse()`. `guess` у цьому виразі стосується першої змінної `guess`, у якій міститься стрічка, введена користувачем. Метод `trim`, застосований до екземпляра `String`, видаляє всі пробільні символи на початку і в кінці, що треба зробити, аби порівняти стрічку з `u32`, який містить виключно числові дані. Користувач має натиснути на `enter`, щоб спрацював метод `read_line` і данні були введені, але це додає символ нового рядка до стрічки. Наприклад, якщо користувач набере 5 і

натисне enter, `guess` буде виглядати як `5\n`. `\n` позначає символ нового рядка. (У Windows натискання enter створює символи повернення каретки та нового рядка, `\r\n`.) Метод `trim` видалить `\n` чи `\r\n`, і залишиться просто `5`.

Метод `parse` для стрічок перетворює стрічку на інший тип. Тут ми застосовуємо його для перетворення стрічки в число. Ми маємо повідомити Rust, який саме числовий тип нам потрібен, за допомогою `let guess: u32`. Двокрапка (`:`) після `guess` каже Rust, що ми анотуємо тип змінної. У Rust є кілька вбудованих числових типів; `u32`, що ви бачите тут є беззнаковим 32-бітним цілим. Це непоганий вибір для невеликих додатних чисел. Ви дізнаєтесь про інші типи у [Розділі 3](#).

На додачу, саме анотація `u32` в цьому прикладі та порівняння із `secret_number` означає, що Rust виведе, що `secret_number` теж має бути `u32`. І тепер порівнюватимуться два значення одного типу!

Метод `parse` буде працювати тільки з символами, які можна логічно перетворити на числа, і тому легко може викликати помилки. Якщо, наприклад, стрічка містить `A👍%`, її неможливо буде перетворити на число. Оскільки метод `parse` може завершитися невдачею, він повертає `Result`, майже так само, як і метод `read_line` (про який ми вже говорили раніше в підрозділі "[Керування потенційною невдачею за допомогою `Result`](#)"). Ми обробимо цей `Result` так само - за допомогою методу `expect`. Якщо `parse` поверне варіант `Result Err`, бо він не зміг створити число зі стрічки, виклик `expect` аварійно припинить гру і виведе повідомлення, яке ми йому надали. Якщо `parse` вдало створив число зі стрічки, він поверне варіант `Result Ok`, а `expect` поверне потрібне нам число зі значення `Ok`.

А тепер запустімо програму:

```
$ cargo run
  Compiling guessing_game v0.1.0 (file:///projects/guessing_game)
  Finished dev [unoptimized + debuginfo] target(s) in 0.43s
  Running `target/debug/guessing_game`
Guess the number!
The secret number is: 58
Please input your guess.
76
You guessed: 76
Too big!
```

Чудово! Хоча ми й додали пробіли перед здогадкою, програма все одно зрозуміла, що користувач увів 76. Запустіть програму кілька разів, щоб перевірити різну поведінку на різних введених даних: введіть таємне число, більше за нього і менше.

Гра тепер майже працює, але користувачеві надається тільки одна можливість вгадати. Змінімо це, додавши цикл!

Введення Декількох Здогадок із Циклами

Ключове слово `loop` створює нескінченний цикл. Ми додамо цикл, щоб дати користувачам більше можливостей відгадати число:

Файл: `src/main.rs`

```
// --snip--

println!("The secret number is: {secret_number}");

loop {
    println!("Please input your guess.");

    // --snip--

    match guess.cmp(&secret_number) {
        Ordering::Less => println!("Too small!"),
        Ordering::Greater => println!("Too big!"),
        Ordering::Equal => println!("You win!"),
    }
}
```

Як ви можете бачити, ми перенесли в цикл усе від запрошення ввести здогадку і до кінця. Обов'язково додайте в ці рядки відступи у чотири пробілами та знову запустіть програму. Програма запрошує ввести нову здогадку до нескінченості, що, власне, є новою проблемою. Не схоже, що користувач може вийти!

Користувач завжди може перервати програму, натиснувши клавіатурне скорочення `ctrl-c`. Але є інший спосіб втекти від цього ненажерного чудовиська - згаданий при обговоренні `parse` у підрозділі "[Порівняння здогадки з таємним числом](#)": якщо користувач введе щось, крім числа, програма аварійно завершиться. Ми можемо скористатися з цього, щоб користувач зумів вийти з програми, як показано тут:

```

$ cargo run
  Compiling guessing_game v0.1.0 (file:///projects/guessing_game)
  Finished dev [unoptimized + debuginfo] target(s) in 1.50s
  Running `target/debug/guessing_game`
Guess the number!
The secret number is: 59
Please input your guess.
45
You guessed: 45
Too small!
Please input your guess.
60
You guessed: 60
Too big!
Please input your guess.
59
You guessed: 59
You win!
Please input your guess.
quit
thread 'main' panicked at 'Please type a number!: ParseIntError { kind:
InvalidDigit }', src/main.rs:28:47
note: run with `RUST_BACKTRACE=1` environment variable to display a backtrace

```

Введення `quit` ("вийти") дійсно призводить до виходу з гри, але так само спрацює будь-що, що не є числом. А все ж таки, це щонайменше не найкращий спосіб. Ми хочемо, щоб гра сама зупинялася, коли ми відгадали число.

Вихід Після Вдалої Здогадки

Запрограмуймо гру виходити, якщо користувач виграв, додавши інструкцію `break`:

Файл: `src/main.rs`

```

// --snip--

match guess.cmp(&secret_number) {
    Ordering::Less => println!("Too small!"),
    Ordering::Greater => println!("Too big!"),
    Ordering::Equal => {
        println!("You win!");
        break;
    }
}
}
}

```

Додавання рядку `break` після `You win!` примусить програму вийти з циклу, якщо користувач відгадав таємне число. Вихід із циклу призведе до виходу з програми, бо цикл - це остання частина функції `main`.

Обробка Неправильних Вхідних Даних

Для покращення роботи гри, замість аварійного виходу, коли користувач вводить не число, зробимо так, що гра ігнорувала те, що ввели, щоб користувач міг продовжувати відгадувати. Ми можемо зробити це, змінивши рядок, де `guess` перетворюється зі `String` на `u32`, як показано в Блоці коду 2-5.

Файл: `src/main.rs`

```
// --snip--

io::stdin()
    .read_line(&mut guess)
    .expect("Failed to read line");

let guess: u32 = match guess.trim().parse() {
    Ok(num) => num,
    Err(_) => continue,
};

println!("You guessed: {guess}");

// --snip--
```

Блок коду 2-5: ігнорування введеного не-числа і запит ввести іншу відгадку замість аварійного завершення програми

Ми замінили виклик `expect` на вираз `match`, щоб перейти від аварійного завершення програми до обробки помилки. Згадаймо, що метод `parse` повертає тип `Result`, а `Result` - це еnum, що має варіанти `Ok` та `Err`. Ми використовуємо тут вираз `match`, так само як робили з `Ordering`, що його повертає метод `cmp`.

Якщо `parse` зможе вдало перетворити стрічку на число, він поверне значення `Ok`, що міститиме результат - число. Це значення `Ok` буде відповідати зразку першого рукава, і весь вираз `match` поверне значення `num`, яке `parse` обчислив і поклав всередину значення `Ok`. Це число потрапить саме туди, куди нам треба - в нову змінну `guess`, яку ми створюємо.

Якщо `parse` не зможе перетворити стрічку на число, він поверне значення `Err`, що міститиме більше інформації про помилку. Значення `Err` не відповідає шаблону `Ok(num)`

у першому рукаві `match`, але відповідає шаблону `Err(_)` у другому. Підкреслення `_` перехопить будь-яке значення; в цьому випадку, ми кажемо, що вираз має відповідати будь-якому `Err`, незалежно від інформації, що міститься у ньому. Тож програма виконає код другого рукава, `continue`, який каже програмі перейти на наступну ітерацію циклу `loop` і знову запитати наступну спробу. Таким чином, програма ігнорує всі помилки, які можуть зустрітись `parse`!

Нарешті все у нашій програмі має працювати як треба. Спробуймо запустити її:

```
$ cargo run
  Compiling guessing_game v0.1.0 (file:///projects/guessing_game)
  Finished dev [unoptimized + debuginfo] target(s) in 4.45s
  Running `target/debug/guessing_game`
Guess the number!
The secret number is: 61
Please input your guess.
10
You guessed: 10
Too small!
Please input your guess.
99
You guessed: 99
Too big!
Please input your guess.
foo
Please input your guess.
61
You guessed: 61
You win!
```

Блискуче! Лишилася тільки одна дрібна правка, і гра-відгадайка буде завершена. Згадаймо, що програма все ще виводить таємне число. Це було потрібно для тестування, але псує гру. Видалімо `println!`, який виводить таємне число. Блок коду 2-6 показує остаточний код.

Файл: `src/main.rs`


```
use rand::Rng;
use std::cmp::Ordering;
use std::io;

fn main() {
    println!("Guess the number!");

    let secret_number = rand::thread_rng().gen_range(1..=100);

    loop {
        println!("Please input your guess.");

        let mut guess = String::new();

        io::stdin()
            .read_line(&mut guess)
            .expect("Failed to read line");

        let guess: u32 = match guess.trim().parse() {
            Ok(num) => num,
            Err(_) => continue,
        };

        println!("You guessed: {guess}");

        match guess.cmp(&secret_number) {
            Ordering::Less => println!("Too small!"),
            Ordering::Greater => println!("Too big!"),
            Ordering::Equal => {
                println!("You win!");
                break;
            }
        }
    }
}
```

Блок коду 2-6: Повний код гри відгадайки

Отже, ви зуміли вдало зібрати гру відгадайку. Вітаємо!

Підсумок

Цей проєкт був вступом до багатьох концепцій мови Rust через практику: `let`, `match`, функції, використання зовнішніх крейтів та інших. У кількох наступних розділах ми детальніше розберемо ці концепції. Розділ 3 розповідає про концепції, які є у більшості мов програмування, такі як змінні, типи даних, функції і показує, як ними користуватися в Rust. Розділ 4 досліджує володіння, концепцію мови Rust, що є найбільш відмінною від

інших мов. Розділ 5 обговорює синтаксис структур і методів, а Розділ 6 детально розкриває, як працюють енуми.

Загальні Концепції Програмування

Цей розділ описує концепції, які можна зустріти у майже кожній мові програмування, і як вони працюють у Rust. Чимало мов програмування мають багато спільного у своїй основі. Жодна з концепцій, представлених у цьому розділі, не унікальна для Rust, але ми говоритимемо про них в контексті Rust і роз'яснимо пов'язані із ними умовності.

Зокрема, ви дізнаєтеся про змінні, базові типи, функції, коментарі і потік керування. Ці базові поняття зустрічаються у кожній програмі Rust, і вивчення їх на початку надасть вам міцну основу для подальшого руху.

Ключові Слова

У мові Rust є набір *ключових слів*, зарезервованих для використання виключно самою мовою, так само як і в інших мовах. Пам'ятайте, що не можна використовувати ці слова як назви змінних та функцій. Більшість ключових слів мають особливе значення, і ви використовуватимете їх для виконання різноманітних задач у ваших програмах на Rust; декілька наразі не мають пов'язаної функціональності, проте лишаються зарезервованими для можливостей, які можуть бути додані до Rust в майбутньому. Список ключових слів можна знайти у [Додатку А](#).

Змінні і Мутабельність

Як уже згадувалося у підрозділі “Зберігання значень у змінних”, за замовчанням змінні є *немутабельними*. Це - один з численних штурханців, якими Rust заохочує вас писати код, що користується перевагами у безпеці та простоті написання конкретного коду, які надає Rust. З усім тим, ви все ж маєте можливість зробити змінні мутабельними. Дослідимо, як і чому Rust заохочує вас надавати перевагу немутабельності, та чому ви можете захотіти відмовитися від цього.

Якщо змінна є немутабельною, це означає, що відколи значення стає прив'язаним до імені, ви не можете змінити це значення. Щоб проілюструвати це, згенеруємо новий проєкт з назвою *variables* у вашій теці *projects* за допомогою `cargo new variables`.

Потім, у новоствореній теці *variables*, відкрийте *src/main.rs* і замініть його код цим, який поки що не компілюється:

Файл: *src/main.rs*

```
fn main() {  
    let x = 5;  
    println!("The value of x is: {x}");  
    x = 6;  
    println!("The value of x is: {x}");  
}
```

Збережіть і запустіть програму за допомогою `cargo run`. Ви маєте отримати повідомлення про помилку щодо помилки немутабельності, як показано на цьому виведенні:

```
$ cargo run
  Compiling variables v0.1.0 (file:///projects/variables)
error[E0384]: cannot assign twice to immutable variable `x`
  --> src/main.rs:4:5
   |
2  |     let x = 5;
   |     -
   |     |
   |     first assignment to `x`
   |     help: consider making this binding mutable: `mut x`
3  |     println!("The value of x is: {x}");
4  |     x = 6;
   |     ^^^^^ cannot assign twice to immutable variable

For more information about this error, try `rustc --explain E0384`.
error: could not compile `variables` due to previous error
```

Цей приклад показує, як компілятор допомагає вам знаходити помилки у програмах. Хоча повідомлення компілятора про помилки й можуть засмучувати, та вони лише означають, що ваша програма ще не робить те, що ви хотіли, у безпечний спосіб; вони *не* означають, що ви поганий програміст! Досвідчені растацеанці також отримують повідомлення про помилки від компілятора.

Ви отримали повідомлення про помилку `cannot assign twice to immutable variable `x``, бо ви намагалися присвоїти друге значення неmutable змінній `x`.

Важливо, що ми отримали помилку часу компіляції, коли намагалися змінити значення, яке раніше визначили як неmutable, тому що ця ситуація може призвести до вад у програмі. Якщо одна частина нашого коду працює з припущенням, що значення не буде змінене, а інша частина нашого коду змінює це значення, можливо, що перша частина коду буде робити не те, для чого вона була розроблена. Цю причину вад важко відслідкувати після виявлення, особливо коли другий фрагмент коду змінює значення лише *час від часу*. У Rust компілятор гарантує, що, якщо ми заявили, що змінна не зміниться, вона і дійсно не зміниться, тому не треба відстежувати її самостійно. Таким чином, ваш код буде легше зрозуміти.

Але mutable може бути дуже корисною і може бути зручнішим писати код з mutable. Хоча змінні за замовчуванням є неmutable, ви можете зробити їх mutable, додавши `mut` перед назвою змінної, як ви робили у [Розділі 2](#). Додавання `mut` також передає ваші наміри майбутнім читачам коду, вказавши, що інші частини коду буде змінювати значення цієї змінної.

Наприклад, змінімо `src/main.rs` на такий код:

Файл: `src/main.rs`

```
fn main() {  
    let mut x = 5;  
    println!("The value of x is: {x}");  
    x = 6;  
    println!("The value of x is: {x}");  
}
```

Запустивши програму ми отримаємо:

```
$ cargo run  
Compiling variables v0.1.0 (file:///projects/variables)  
Finished dev [unoptimized + debuginfo] target(s) in 0.30s  
Running `target/debug/variables`  
The value of x is: 5  
The value of x is: 6
```

Застосувавши `mut`, ми дозволили змінити значення, прив'язане до `x`, з `5` на `6`.
Остаточне рішення, використовувати мутабельність чи ні, належить вам і залежить від того, що ви вважаєте найочевиднішим у конкретній ситуації.

Константи

Подібно до немутабельних змінних, *константи* так само є значенням, прив'язаним до імені, які не можна змінювати, але є кілька відмінностей між константами і змінними.

По-перше, не можна використовувати `mut` з константами. Константи не просто немутабельні за замовчанням, вони завжди немутабельні. Константи проголошуються ключовим словом `const` замість `let`, і тип значення має явно позначатися. Ми розкажемо про типи і анотації типів у наступному підрозділі, "Типи даних", тому не хвилюйтеся зараз про деталі. Просто пам'ятайте, що тип констант треба зазначати завжди.

Константи можуть проголошуватися у будь-якій області видимості, у тому числі глобальній, що робить їх корисними для зберігання значень, що використовуються у багатьох частинах вашого коду.

Остання відмінність полягає в тому, що константи можуть набувати тільки значення константних виразів, а не значень, які можуть бути обчислені лише у час виконання.

Ось приклад визначення константи:

```
const THREE_HOURS_IN_SECONDS: u32 = 60 * 60 * 3;
```

Константа зветься `THREE_HOURS_IN_SECONDS` і її значення встановлене в результат множення 60 (числа секунд у хвилині) на 60 (числа хвилин у годині) на 3 (числа годин, що ми хочемо порахувати у цій програмі). Угода про назви констант в Rust вимагає використання верхнього регістру із підкресленнями між словами. Компілятор здатний обчислити невеликий набір операцій під час компіляції, що дозволяє нам вписати це значення так, щоб його було легше зрозуміти та перевірити, замість того щоб встановлювати константі значення 10800. Зверніться до [Підрозділу Довідника Rust про обчислення констант](#) за додатковою інформацією про те, які операції можна використовувати при проголошенні констант.

Константи є коректними протягом усього часу життя програми у тій області видимості, де вони були проголошені. Це робить константи корисними для зберігання значень з предметної області вашого застосунку, про які необхідно знати багатьом частинам програми, наприклад, максимальна кількість балів, яку може отримати гравець чи швидкість світла.

Корисно давати назви жорстко заданим значенням, що використовуються у вашій програмі, позначаючи їх константами, щоб передати сенс цього значення тим, хто супроводжуватиме код. Це також корисно тим, що в коді буде тільки одне місце, яке буде необхідно змінити у разі потреби оновити жорстко задане значення.

Затінення

Як ви бачили під час програмування гри - відгадайки у [Розділі 2](#), можна проголошувати нову змінну із таким самим іменем, як і в раніше проголошеної змінної. Растацеанці кажуть, що перша змінна *затінена* другою, що означає, що при використанні змінної компілятор бачить лише другу змінну. По суті, друга змінна перекриває першу, перехоплюючи будь-яку згадку імені змінної на себе до тих пір, поки вона сама не буде затінена або область видимості не закінчиться. Ми можемо затінити змінну за допомогою ключового слова `let` та імені цієї змінної, ось так:

Файл: src/main.rs

```
fn main() {
    let x = 5;

    let x = x + 1;

    {
        let x = x * 2;
        println!("The value of x in the inner scope is: {x}");
    }

    println!("The value of x is: {x}");
}
```

Ця програма спершу прив'язує `x` до значення `5`. Потім створює нову змінну `x`, повторюючи `let x =` і початкове значення та додає до нього `1`, так що значення `x` тепер `6`. Потім, у внутрішній області видимості, створеній фігурними дужками, третя інструкція `let` знову затінює `x` і створює нову змінну, домножуючи попереднє значення на `2`, щоб надати `x` значення `12`. Коли область видимості завершується, внутрішнє затінення теж завершується і `x` повертається до значення `6`. Якщо ми запустимо цю програму, вона виведе:

```
$ cargo run
Compiling variables v0.1.0 (file:///projects/variables)
Finished dev [unoptimized + debuginfo] target(s) in 0.31s
Running `target/debug/variables`
The value of x in the inner scope is: 12
The value of x is: 6
```

Затінення відрізняється від позначення змінної `mut`, адже, якщо ми випадково спробуємо переприсвоїти значення цій змінній, не додавши ключове слово `let`, то отримаємо помилку часу компіляції. Використовуючи `let`, ми можемо виконати кілька перетворень значення, але лишити змінну немутабельною виконання цих перетворень.

Інша різниця між `mut` та затіненням полягає в тому, що, оскільки коли ми пишемо знову ключове слово `let`, насправді ми створюємо нову змінну, тож можемо змінити тип значення, але залишити ім'я. Наприклад, хай наша програма просить користувача вказати, скільки пробілів має бути всередині якогось тексту, ввівши символи пробілу, але насправді ми хочемо зберігати це значення як число:

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

Перша змінна `spaces` має стрічковий тип, а друга змінна `spaces` має числовий тип. Затінення, таким чином, позбавляє нас необхідності придумувати різні імена, на кшталт `spaces_str` та `spaces_num`; натомість, ми можемо заново використати простіше ім'я

`spaces`. Але якщо ми спробуємо для цього скористатися `mut`, як показано далі, то дістанемо помилку часу компіляції:

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

Помилка каже, що не можна змінювати тип змінної:

```
$ cargo run
   Compiling variables v0.1.0 (file:///projects/variables)
error[E0308]: mismatched types
--> src/main.rs:3:14
   |
 2 |     let mut spaces = "  ";
   |                      ----- expected due to this value
 3 |     spaces = spaces.len();
   |               ^^^^^^^^^^^^^ expected `&str`, found `usize`

For more information about this error, try `rustc --explain E0308`.
error: could not compile `variables` due to previous error
```

Тепер, дослідивши, як працюють змінні, погляньмо, які типи даних вони можуть зберігати. [ch02-00-guessing-game-tutorial.html#comparing-the-guess-to-the-secret-number](https://pavloslav.github.io/RustBookUkr/print.html#comparing-the-guess-to-the-secret-number)

Типи Даних

Кожне значення в Rust має певний *тип даних*, який каже Rust, якого виду це дані, щоб мова знала, як працювати з цими даними. Ми поглянемо на дві категорії типів даних: скалярні і складені.

Пам'ятайте, що Rust - *статично типізована* мова, тобто типи всіх змінних має бути відомим під час компіляції. Компілятор зазвичай може вивести, який тип ми хочемо використати, виходячи зі значення і того, як ми його використовуємо. У випадках, коли може підійти кілька типів, наприклад коли якщо ми перетворювали `String` на числовий тип за допомогою `parse` у підрозділі “Порівняння здогадки з таємним числом” Розділу 2, ми маємо додавати анотацію типу, ось так:

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

Якщо ми не додамо анотацію типу `: u32`, показану у попередньому коді, Rust видасть наступну помилку, яка означає, що компілятору потрібно більше інформації від нас, щоб дізнатися, який тип ми хочемо використати:

```
$ cargo build
  Compiling no_type_annotations v0.1.0 (file:///projects/no_type_annotations)
error[E0282]: type annotations needed
  --> src/main.rs:2:9
   |
2  |     let guess = "42".parse().expect("Not a number!");
   |     ^^^^^
help: consider giving `guess` an explicit type
   |
2  |     let guess: _ = "42".parse().expect("Not a number!");
   |             +++

For more information about this error, try `rustc --explain E0282`.
error: could not compile `no_type_annotations` due to previous error
```

Ви побачите різні анотації типів для інших типів даних.

Скалярні Типи

Скалярний тип представляє єдине значення. У Rust є чотири первинні скалярні типи: цілі, числа з рухомою комою, булівські та символи. Ви можете впізнати їх з інших мов

програмування. Гайда подивимося, як вони працюють у Rust.

Цілочисельні Типи

Ціле число - це число без дробової частини. Ви використали один цілий тип у Розділі 2, а саме `u32`. Проголошення цього типу означає, що асоційоване з ним значення має бути беззнаковим цілим (знакові цілі типи починаються на `i`, на відміну від беззнакових `u`), що займає 32 біти пам'яті. Таблиця 3-1 показує вбудовані цілі типи в Rust. Ми можемо скористатися будь-яким з них для оголошення типу цілого числа.

Таблиця 3-1: Цілі типи в Rust

Довжина	Знаковий	Беззнаковий
8 бітів	<code>i8</code>	<code>u8</code>
16 бітів	<code>i16</code>	<code>u16</code>
32 біти	<code>i32</code>	<code>u32</code>
64 біти	<code>i64</code>	<code>u64</code>
128 бітів	<code>i128</code>	<code>u128</code>
Залежно від архітектури	<code>isize</code>	<code>usize</code>

Кожен цілий тип є знаковим чи беззнаковим і має явно зазначений розмір. *Знаковий* і *беззнаковий* стосується того, чи може число бути від'ємним — іншими словами, чи має число знак (знакове) чи воно буде лише додатним і, відтак, може бути представлене без знаку (беззнакове). Це як запис чисел на папері: якщо знак має значення, число записується зі знаком плюс чи знаком мінус; але, якщо можна вважати, що число буде додатним, воно записується без знаку. Знакові числа зберігаються у [доповняльному коді](#).

Кожен знаковий цілий тип може зберігати числа від $-(2^{n-1})$ до $2^{n-1} - 1$ включно, де n - кількість біт, які він використовує. Так, `i8` може зберігати числа від $-(2^7)$ до $2^7 - 1$, тобто від -128 до 127. Беззнакові цілі типи зберігають числа від 0 до $2^n - 1$, так, `u8` може зберігати числа від 0 до $2^8 - 1$, тобто від 0 до 255.

На додачу, типи `isize` та `usize` залежать від архітектури комп'ютера, на якому працює ваша програма: 64 біти, якщо це 64-бітна архітектура, чи 32 біти, якщо 32-бітна.

Ви можете писати цілі літерали в будь-якій формі, вказаній у Таблиці 3-2. Зверніть увагу, що числові літерали, які можуть бути різних типів, дозволяють використовувати суфікс типу на кшталт `57u8`, для визначення типу. Числові літерали також можуть використовувати `_` як роздільник для поліпшення читання, як-от `1_000`, що позначає те саме значення, що й запис `1000`.

Таблиця 3-2: Цілі літерали в Rust

Числові літерали	Приклад
Десятковий	<code>98_222</code>
Шістнадцятковий	<code>0xff</code>
Вісімковий	<code>0o77</code>
Двійковий	<code>0b1111_0000</code>
Байт (лише <code>u8</code>)	<code>b'A'</code>

Як же зрозуміти, який тип цілого використати? Якщо ви непевні, вибір Rust за замовчуванням зазвичай непоганий, а цілий тип за замовчуванням в Rust - `i32`. Основна ситуація, в якій варто використовувати `isize` та `usize` - індексація якого виду колекції.

Переповнення Цілого Числа

Скажімо, що у вас є змінна типу `u8`, що може мати значення між 0 та 255. Якщо ви спробуєте змінити її значення на те, що виходить за межі цього діапазону, скажімо 256, стається *переповнення*, що призводить однієї з двох поведінок. Коли ви компілюєте програму в режимі дебагу, Rust додає перевірки на переповнення, які призведуть до *паніки* під час роботи програми, якщо воно станеться. Rust використовує термін *паніка*, коли програма завершується із помилкою; ми обговоримо паніку детальніше у підрозділі "[Невідновлювані помилки за допомогою `panic!`](#)" Розділу 9.

Коли ж ви компілюєте в режимі релізу за допомогою прапорця `--release`, Rust *не* додає перевірок на переповнення, що спричинили б паніку. Натомість якщо виникає переповнення, Rust *загортає з доповненням до двох* це число. Якщо коротко, значення, більші за максимальне значення, що вміщується в тип, "загортаються" до мінімального значення, що вміщується в тип. У випадку з `u8`, значення 256 стає 0, 257 стає 1 і так далі. Програма не панікуватиме, але змінна матиме значення, що, мабуть, не відповідає вашим очікуванням. Не варто розраховувати на загортання при переповненні як на коректну поведінку, це помилка.

Щоб явно обробити можливість переповнення, ви можете використати такі групи методів, наданих стандартною бібліотекою для примітивних числових типів:

- Якщо вам потрібне саме загортання, використовуйте методи `wrapping_*`, наприклад `wrapping_add`.
- Якщо вам потрібне значення `None` при переповненні, використовуйте методи `checked_*`.

- Для виявлення переповнення методи `overflowing_*` повертають значення і булеве значення, що показує, чи сталося переповнення.
 - Якщо вам потрібне насичення до мінімального чи максимального значення, використовуйте методи `saturating_*`.
-

Типи Чисел з Рухомою Комою

Також Rust має два примітивні типи для *чисел з рухомою комою*, тобто чисел з десятковою комою. Числа з рухомою комою в Rust - це `f32` та `f64`, які мають розмір у 32 біти та 64 біти відповідно. Тип за замовчанням - `f64`, оскільки на сучасних процесорах його швидкість приблизно така ж сама, як і в `f32`, але він має вищу точність. Усі числа з рухомою комою знакові.

Ось приклад, що демонструє числа з рухомою комою у дії:

Файл: `src/main.rs`

```
fn main() {  
    let x = 2.0; // f64  
  
    let y: f32 = 3.0; // f32  
}
```

Числа з рухомою комою представлені відповідно до стандарту IEEE-754. Тип `f32` є числом одинарної точності, а `f64` має подвійну точність.

Числові Операції

Rust підтримує звичайні математичні операції, які ви очікуєте для будь-яких типів чисел: додавання, віднімання, множення, ділення й остача. Цілочисельне ділення округлює результат униз до найближчого цілого. Наступний код демонструє, як використовувати числові операції в інструкції `let`:

Файл: `src/main.rs`

```
fn main() {  
    // addition  
    let sum = 5 + 10;  
  
    // subtraction  
    let difference = 95.5 - 4.3;  
  
    // multiplication  
    let product = 4 * 30;  
  
    // division  
    let quotient = 56.7 / 32.2;  
    let truncated = -5 / 3; // Results in -1  
  
    // remainder  
    let remainder = 43 % 5;  
}
```

Кожен вираз використовує математичний оператор і обчислює значення, яке прив'язується до змінної. [Додаток В](#) містить список усіх операторів, які використовуються в мові Rust.

Булевий Тип

Як і в більшості інших мов програмування, булівський тип у Rust має два можливі значення: `true` ("істина") та `false` ("неправда"). Булівський тип займає 1 байт. Булівський тип у Rust позначається `bool`. Наприклад:

Файл: src/main.rs

```
fn main() {  
    let t = true;  
  
    let f: bool = false; // with explicit type annotation  
}
```

Основний спосіб використання булівських значень - умовні вирази, такі, як вираз `if`. Ми розкажемо, як працюють вирази `if`, у підрозділі [Потік виконання](#).

Символьний Тип

Тип `char` в Rust є найпростішим алфавітним типом. Ось кілька прикладів проголошення значень `char`:

Файл: src/main.rs

```
fn main() {  
    let c = 'z';  
    let z: char = 'Z'; // with explicit type annotation  
    let heart_eyed_cat = '😺';  
}
```

Зверніть увагу, що літерали `char` позначаються одинарними лапками, на відміну від стрічкових літералів, які послуговуються подвійними. Тип `char` в Rust має чотири байти і представляє скалярне значення в Юнікодi, тобто може представляти значно більше, ніж просто ASCII. Літери з наголосами, китайські, японські і корейські символи, смайлики і пробіли нульової ширини є коректними значеннями для `char` у Rust. Скалярні значення Юнікода можуть бути в діапазоні від `U+0000` до `U+D7FF` і `U+E000` до `U+10FFFF` включно. Однак "символ" насправді не є концепцією Юнікода, тому ваше інтуїтивне уявлення про те, що таке "символ" може не зовсім відповідати тому, чим є `char` у Rust. Цю тему ми детальніше обговоримо в підрозділі ["Зберігання тексту, кодованого в UTF-8, у стрічках"](#) Розділу 8.

Складені Типи

Складені типи дозволяють об'єднувати багато значень в один тип. Rust має два базових складених типи: кортежі та масиви.

Кортеж

Кортеж (tuple) - основний спосіб збирати до купи ряд значень різних типів у один складений тип. Кортежі мають фіксовану довжину: один раз проголошені, вони не можуть зростати чи скорочуватися.

Кортеж утворюється списком значень, розділених комами, в дужках. Кожна позиція в кортежі має тип, і типи різних значень у кортежі не обов'язково мають збігатися. Ми додали необов'язкову анотацію типу у цьому прикладі:

Файл: src/main.rs

```
fn main() {  
    let tup: (i32, f64, u8) = (500, 6.4, 1);  
}
```

Змінна `tup` зв'язується з усім кортежем, оскільки кортеж розглядається як єдиний складений елемент. Щоб отримати окремі значення з кортежу, можна скористатися зіставленням з шаблоном, щоб деструктуризувати значення кортежу, на кшталт цього:

Файл: src/main.rs

```
fn main() {  
    let tup = (500, 6.4, 1);  
  
    let (x, y, z) = tup;  
  
    println!("The value of y is: {y}");  
}
```

Ця програма спершу створює кортеж і зв'язує його зі змінною `tup`. Далі вона використовує шаблон з `let`, щоб перетворити `tup` на три окремі змінні: `x`, `y` і `z`. Це зветься *деструктуризацією*, бо розбирає єдиний кортеж на три частини. І врешті програма виводить значення `y`, тобто `6.4`.

Ми також можемо отримати доступ до елементу кортежу напряму за допомогою точки (`.`), за якою іде індекс значення, яке нам треба отримати. Наприклад:

Файл: src/main.rs

```
fn main() {  
    let x: (i32, f64, u8) = (500, 6.4, 1);  
  
    let five_hundred = x.0;  
  
    let six_point_four = x.1;  
  
    let one = x.2;  
}
```

Ця програма створює кортеж `x`, а потім створює нові змінні для кожного елементу за допомогою їхніх індексів. Як і в більшості мов програмування, перший індекс в кортежі - 0.

Кортеж без значень має особливу назву - *одиничний тип*. Це значення і відповідний тип обидва записуються як `()` і представляють порожнє значення чи порожній тип результату. Вирази неявно повертають одиничний тип, якщо вони не повертають жодного іншого значення.

Масив

Інший спосіб організувати колекцію з багатьох значень - це *масив*. На відміну від кортежу, всі елементи масиву мусять мати один тип. На відміну від масивів у деяких інших мовах, масиви в Rust мають фіксовану довжину.

Значення в масиві записуються як список, розділений комами, в квадратних дужках:

Файл: src/main.rs

```
fn main() {  
    let a = [1, 2, 3, 4, 5];  
}
```

Масиви корисні, коли дані мають бути розмішені в стеку, а не в купі (детальніше про це йдеться у [Розділі 4](#)), чи коли ви хочете бути певним, що завжди маєте фіксовану кількість елементів. Втім, масиви не такі гнучкі, як вектори. *Вектор* - це схожий тип-колекція, наданий стандартною бібліотекою, який *може* зростати і скорочуватися. Якщо ви не певні, використовувати вам масив чи вектор, швидше за все варто використати вектор. [Розділ 8](#) розповідає про вектори детальніше.

Разом із тим, масиви корисніші, коли ви знаєте, що кількість елементів не треба буде змінювати. Наприклад, коли ви використовуєте імена місяців у програмі, швидше за все ви використаєте масив, а не вектор, бо ви знаєте, що він завжди складатиметься з 12 елементів:

```
let months = ["January", "February", "March", "April", "May", "June", "July",  
              "August", "September", "October", "November", "December"];
```

Тип масиву записується за допомогою квадратних дужок з типом кожного елементу, крапки з комою і числом елементів масиву, ось так:

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

Тут `i32` є типом кожного елементу. Після крапки з комою, число `5` позначає, що масив містить п'ять елементів.

Ви також можете ініціалізувати масив однаковими значеннями для кожного елементу, вказавши початкове значення, потім крапку з комою і довжину масиву у квадратних дужках, як показано тут:

```
let a = [3; 5];
```

Масив, що зветься `a`, міститиме `5` елементів, що початково матимуть значення `3`. Це - те саме, що написати `let a = [3, 3, 3, 3, 3];` але стисліше.

Доступ до Елементів Масиву

Масив - це єдиний блок пам'яті відомого, фіксованого розміру, що може бути виділений у стеку. Ви можете працювати з елементами масиву за допомогою індексів, ось так:

Файл: src/main.rs

```
fn main() {  
    let a = [1, 2, 3, 4, 5];  
  
    let first = a[0];  
    let second = a[1];  
}
```

У цьому прикладі, змінна, що зветься `first`, отримає значення `1`, бо це значення в масиві за індексом `[0]`. Змінна, що зветься `second`, отримає значення `2` за індексом `[1]` у масиві.

Некоректний Доступ до Елементів Масиву

Подивімося, що станеться, якщо ви спробуєте дістатися до елемента масиву, що знаходиться за його кінцем. Скажімо, ви запустите цей код, схожий на гру-здогадайку з Розділу 2, щоб отримати індекс масиву від користувача:

Файл: src/main.rs

```
use std::io;  
  
fn main() {  
    let a = [1, 2, 3, 4, 5];  
  
    println!("Please enter an array index.");  
  
    let mut index = String::new();  
  
    io::stdin()  
        .read_line(&mut index)  
        .expect("Failed to read line");  
  
    let index: usize = index  
        .trim()  
        .parse()  
        .expect("Index entered was not a number");  
  
    let element = a[index];  
  
    println!("The value of the element at index {index} is: {element}");  
}
```

Цей код успішно компілюється. Якщо ви запустите цей код за допомогою `cargo run` і введете `0`, `1`, `2`, ```, або `4`, програма виведе на екран відповідне значення з цього індексу в масиві. Якщо ж ви натомість введете число за кінцем масиву, таке як `10`, програма виведе таке:

```
thread 'main' panicked at 'index out of bounds: the len is 5 but the index is 10',  
src/main.rs:19:19  
note: run with `RUST_BACKTRACE=1` environment variable to display a backtrace
```

Програма завершилася помилкою *часу виконання* у той момент, коли використала некоректне значення при застосуванні індексу. Програма вийшла з повідомленням про помилку і не виконала останню інструкцію `println!`. Коли ви намагаєтеся отримати доступ до елементу масиву, Rust перевіряє, чи зазначений індекс менший за довжину масиву. Якщо індекс більший чи дорівнює довжині, Rust панікує. Ця перевірка здійснюється під час виконання, особливо в цьому випадку, бо компілятор не має жодної можливості знати, яке значення введе користувач, коли код буде запущено.

Це приклад безпеки роботи з пам'яттю Rust у дії. У багатьох мовах низького рівня такої перевірки не відбувається, і коли ви задаєте некоректний індекс, може відбутися доступ до некоректної пам'яті. Rust захищає вас від такої помилки, одразу перериваючи роботу програми замість того, щоб дозволити некоректний доступ і продовжити роботу. Розділ 9 розповідає більше про обробку помилок у Rust і як ви можете писати читаний, безпечний код що не панікує і не дозволяє некоректний доступ до пам'яті. [ch02-00-guessing-game-tutorial.html#comparing-the-guess-to-the-secret-number](https://pavloslav.github.io/RustBookUkr/print.html#comparing-the-guess-to-the-secret-number)

Функції

Функції використовуються скрізь у коді на Rust. Ви вже бачили одну з найважливіших функцій у мові - функцію `main`, яка є точкою входу багатьох програм. Ви також бачили ключове слово `fn`, яке дозволяє вам оголошувати нові функції.

У мові Rust для назв функцій і змінних домовлено використовувати *зміїний регістр* - тобто всі літери маленькі, а слова відокремлюються підкресленнями. Ось приклад програми, що містить визначення функції:

Файл: `src/main.rs`

```
fn main() {  
    println!("Hello, world!");  
  
    another_function();  
}  
  
fn another_function() {  
    println!("Another function.");  
}
```

Визначення функцій у Rust починаються з `fn`, далі іде назва функції і пара дужок. Фігурні дужки кажуть компілятору, де починається і закінчується тіло функції.

Ми можемо викликати будь-яку визначену нами функцію, написавши її назву і пару дужок. Оскільки в програмі є визначеної `another_function`, її можна викликати зсередини функції `main`. Зверніть увагу, що ми визначили `another_function` у початковому коді після функції `main`; так само її можна було визначити до функції `main`. Для Rust не має значення, де ви визначаєте функції, важливо, щоб вони були визначені хоч де у області видимості, доступної з місця виклику.

Почнімо новий двійковий проєкт з назвою *functions*, щоб глибше дослідити функції. Помістіть приклад `another_function` до файлу `src/main.rs` і запустіть його. Ви маєте побачити таке:

```
$ cargo run  
Compiling functions v0.1.0 (file:///projects/functions)  
Finished dev [unoptimized + debuginfo] target(s) in 0.28s  
Running `target/debug/functions`  
Hello, world!  
Another function.
```

Рядки виконуються в порядку, в якому вони знаходяться в функції `main`. Спершу виводиться повідомлення "Hello, world!", а потім викликається `another_function` і виводить своє повідомлення.

Параметри

При визначенні функції ми можемо задати *параметри*, тобто спеціальні змінні, що є частиною сигнатури функції. Коли функція має параметри, ми можемо надати функції конкретні значення для цих параметрів. Формально, конкретні значення зветься *аргументами* або *фактичними параметрами*, а параметри у визначенні функції - *формальними параметрами*, але зазвичай слова `<0>параметр</0>` та `<0>аргумент</0>` використовуються як для частини визначення функції, так і для конкретних значень, які були передані при виклику функції.

Додамо параметр у нову версію `another_function`:

Файл: src/main.rs

```
fn main() {
    another_function(5);
}

fn another_function(x: i32) {
    println!("The value of x is: {x}");
}
```

Запустіть цю програму; вона має вивести таке:

```
$ cargo run
Compiling functions v0.1.0 (file:///projects/functions)
Finished dev [unoptimized + debuginfo] target(s) in 1.21s
Running `target/debug/functions`
The value of x is: 5
```

Проголошення `another_function` містить один параметр під назвою `x`. Тип `x` зазначено як `i32`. Коли в `another_function` передається `5`, макрос `println!` виведе `5` на місце фігурних дужок з `x` у форматній стрічці.

У сигнатурі функції ви *обов'язково* маєте проголошувати тип кожного параметру. Це свідоме рішення у дизайні мови Rust: обов'язкові анотації типів у визначенні функцій означають, що компілятору дуже рідко знадобиться просити вас використовувати їх деінде ще в коді, щоб зрозуміти, який тип ви мали на увазі. Також компілятор може надавати більш помічні повідомлення про помилки, якщо знатиме, які типи параметрів очікує функція.

При оголошенні кількох параметрів відокремлюйте окремі оголошення комами, ось так:

Файл: `src/main.rs`

```
fn main() {
    print_labeled_measurement(5, 'h');
}

fn print_labeled_measurement(value: i32, unit_label: char) {
    println!("The measurement is: {value}{unit_label}");
}
```

Цей приклад створює функцію `print_labeled_measurement` з двома параметрами. Перший параметр зветься `value` і має тип `i32`. Другий зветься `unit_label` і має тип `char`. Функція виводить текст, що містить і `value`, і `unit_label`.

Спробуймо запустити цей код. Замініть програму у файлі `src/main.rs` вашого проєкту `functions` попереднім прикладом, і запустіть його командою `cargo run`:

```
$ cargo run
   Compiling functions v0.1.0 (file:///projects/functions)
   Finished dev [unoptimized + debuginfo] target(s) in 0.31s
   Running `target/debug/functions`
The measurement is: 5h
```

Оскільки ми викликали функцію зі значенням `5` для параметра `value` і значенням `'h'` для `unit_label`, вивід програми містить обидва цих значення.

Інструкції та Вирази

Тіла функцій складаються з послідовності інструкцій, яка може закінчуватися виразом. Поки що функції, про які ми згадували, не мали виразу наприкінці, але вирази були частиною інструкцій. Оскільки Rust є мовою, що ґрунтується на виразах, важливо розуміти цю відмінність. Інші мови не мають таких відмінностей, тому роздивімося, що таке інструкції та вирази і як різниця між ними впливає на тіла функцій.

- **Інструкції** (`<0>statement</0>`) - це команди, що виконують певну дію і не повертають значення.
- **Вирази** (`<0>expression</0>`) обчислюються, в результаті даючи певне значення. Розгляньмо приклади.

Власне, ми вже використовували інструкції та вирази. Створення змінної та приписування їй значення за допомогою ключового слова `let` є інструкцією. У Блоці коду 3-1 `let y =`

`6;` є інструкцією.

Файл: `src/main.rs`

```
fn main() {  
    let y = 6;  
}
```

Блок коду 3-1. Визначення функції `main`, що містить одну інструкцію

Визначення функцій також є інструкціями; весь попередній приклад як такий є інструкцією.

Інструкції не повертають значень. Таким чином, не можна присвоїти інструкцію `let` іншій змінній, як ми намагаємося в наступному коді; ви отримаєте помилку:

Файл: `src/main.rs`

```
fn main() {  
    let x = (let y = 6);  
}
```

При спробі запустити цю програму, ви отримаєте повідомлення про помилку:

```

$ cargo run
  Compiling functions v0.1.0 (file:///projects/functions)
error: expected expression, found `let` statement
--> src/main.rs:2:14
   |
2  |     let x = (let y = 6);
   |               ^^^

error: expected expression, found statement (`let`)
--> src/main.rs:2:14
   |
2  |     let x = (let y = 6);
   |               ^^^^^^^^^^^

= note: variable declaration using `let` is a statement

error[E0658]: `let` expressions in this position are unstable
--> src/main.rs:2:14
   |
2  |     let x = (let y = 6);
   |               ^^^^^^^^^^^

= note: see issue #53667 <https://github.com/rust-lang/rust/issues/53667> for
more information

warning: unnecessary parentheses around assigned value
--> src/main.rs:2:13
   |
2  |     let x = (let y = 6);
   |               ^         ^

= note: `#[warn(unused_parens)]` on by default
help: remove these parentheses
   |
2 -     let x = (let y = 6);
2 +     let x = let y = 6;
   |

For more information about this error, try `rustc --explain E0658`.
warning: `functions` (bin "functions") generated 1 warning
error: could not compile `functions` due to 3 previous errors; 1 warning emitted

```

Інструкція `let y = 6` не повертає значення, тому немає нічого, з чим можна було б зв'язати `x`. Це відрізняється від інших мов, таких як C чи Ruby, де присвоєння повертає значення, яке воно присвоїло. У тих мовах можна написати `x = y = 6` і обидві змінні `x` та `y` набудуть значення `6`; у Rust так робити не можна.

Вирази обчислюються у певне значення і складають більшу частину коду, який ви писатимете на Rust. Розгляньмо просту математичну операцію, таку, як `5 + 6`, яка є виразом, що обчислюється у значення `11`. Вирази можуть бути частинами інструкцій: у

Блоці коду 3-1 в інструкції `let y = 6;`, `6` - це вираз, що обчислюється у значення `6`. Виразами також є виклик функції чи макросу; блок, що створює нову область видимості за допомогою фігурних дужок - це також вираз, наприклад:

Файл: src/main.rs

```
fn main() {  
    let y = {  
        let x = 3;  
        x + 1  
    };  
  
    println!("The value of y is: {y}");  
}
```

Цей вираз:

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

є блоком, який, в цьому випадку, обчислюється у `4`. Це значення прив'язується до `y`, як частина інструкції `let`. Зверніть увагу, що `x + 1` не має крапки з комою наприкінці, на відміну від більшості рядків, які нам поки що траплялися. Вирази не мають завершувальної крапки з комою. Якщо ви додасте крапку з комою в кінець виразу, ви зробите його інструкцією, яка не повертає значення. Пам'ятайте це, коли вивчатимете далі значення, які повертають функції та вирази.

Функції, що Повертають Значення

Функції можуть повертати значення в код, що їх викликав. Цим значенням ми не даємо власних імен, але маємо проголосити їхній тип після стрілочки (`->`). У Rust значення, що його повертає функція - це те саме, що значення останнього виразу в блоці - тілі функції. Ви можете також вийти з функції раніше за допомогою ключового слова `return` і вказання значення, але більшість функцій неявно повертають значення останнього виразу. Ось приклад функції, що повертає значення:

Файл: src/main.rs

```
fn five() -> i32 {
    5
}

fn main() {
    let x = five();

    println!("The value of x is: {x}");
}
```

У функції `five` немає викликів інших функцій, макросів чи навіть інструкцій `let` - саме тільки число `5`. І це абсолютно коректна функція в Rust. Зверніть увагу, що тут зазначено тип значення, яке функція повертає `-> i32`. Спробуймо запустити цей код; вивід має виглядати так:

```
$ cargo run
Compiling functions v0.1.0 (file:///projects/functions)
Finished dev [unoptimized + debuginfo] target(s) in 0.30s
Running `target/debug/functions`
The value of x is: 5
```

`5` у `five` є значенням, яке повертає функція, і тому тип, який повертає функція - `i32`. Розгляньмо це детальніше. Є два важливі моменти: по-перше, рядок `let x = five();` показує, що ми використовуємо значення, яке повернула функція, для ініціалізації змінної. Оскільки функція `five` повертає `5`, цей рядок робить те саме, що й такий:

```
let x = 5;
```

По-друге, функція `п'ять` не має параметрів і визначає тип значення, що повертає, але тіло функції -- самотнє `5` без крапки з комою -- це вираз, значення якого ми хочемо повернути.

Подивімося інший приклад:

Файл: `src/main.rs`

```
fn main() {
    let x = plus_one(5);

    println!("The value of x is: {x}");
}

fn plus_one(x: i32) -> i32 {
    x + 1
}
```

Якщо виконати цей код, він виведе `The value of x is: 6`. Але якщо ми поставимо крапку з комою в кінець рядка `x + 1`, щоб він став не виразом, а інструкцією, ми дістанемо помилку:

Файл: src/main.rs

```
fn main() {  
    let x = plus_one(5);  
  
    println!("The value of x is: {x}");  
}  
  
fn plus_one(x: i32) -> i32 {  
    x + 1;  
}
```

Компіляція цього коду призводить до такої помилки:

```
$ cargo run  
Compiling functions v0.1.0 (file:///projects/functions)  
error[E0308]: mismatched types  
--> src/main.rs:7:24  
7 | fn plus_one(x: i32) -> i32 {  
  | -----  
  |             ^^^ expected `i32`, found `()`  
  |  
  | implicitly returns `()` as its body has no tail or `return` expression  
8 |     x + 1;  
  |     - help: remove this semicolon  
  
For more information about this error, try `rustc --explain E0308`.  
error: could not compile `functions` due to previous error
```

Основне повідомлення про помилку `mismatched types` ("невідповідні типи") розкриває основну проблему цього коду. Визначення функції `plus_one` каже, що вона має повернути `i32`, але інструкції не обчислюються в значення, що позначається як `()`, одиничний тип. Таким чином, нічого не повертається, що суперечить визначенню функції й призводить до помилки. У цьому виведенні Rust повідомляє про можливість виправити цю проблему: він радить прибрати крапку з комою, що дійсно виправить помилку.

Коментарі

Всі програмісти прагнуть зробити свій код зрозумілішим, та деколи не завадить додаткове пояснення. В таких випадках програмісти лишають в початковому коді *коментарі*, які ігнорує компілятор, але які можуть бути корисними людям, що читатимуть цей код.

Ось простий коментар:

```
// hello, world
```

У Rust найтипівіший стиль коментарів починається з двох знаків дробу і продовжується до кінця рядка. Для коментарів, що займають більше одного рядка, вам доведеться ставити `///` у кожному рядку, ось так:

```
// Тут ми робимо щось складне, досить довге, щоб нам знадобилося кілька рядків  
// коментаря! Хух! Сподіваюся, цей коментар достатньо детально пояснює, що  
// тут відбувається.
```

Коментарі також можна розміщувати в кінці рядків, що містять код:

Файл: src/main.rs

```
fn main() {  
    let lucky_number = 7; // I'm feeling lucky today  
}
```

Та частіше ви бачитимете їх у форматі, де коментар знаходиться в окремому рядку перед кодом, якого він стосується:

Файл: src/main.rs

```
fn main() {  
    // I'm feeling lucky today  
    let lucky_number = 7;  
}
```

Rust також має інший вид коментарів, документаційні коментарі, які ми обговоримо в підрозділі "[Публікація крейта на Crates.io](#)" Розділу 14.

Потік Виконання

Здатність виконувати чи ні певний код залежно від того, чи умова істинна (`true`), чи повторити певний код кілька разів, доки умова `true` - базові будівельні елементи коду у більшості мов програмування. Найпоширеніші конструкції, що дозволяють вам управляти потоком виконання коду на Rust є вирази `if` та цикли.

Вирази `if`

Вираз `if` дозволяє розгалужувати код у залежності від умов. Ви задаєте умову, а потім вказуєте: "Якщо цю умову дотримано, запустити цей блок коду. Якщо ж умову не дотримано, не запускати цей блок коду".

Створіть новий проект з назвою *branches* у вашій теці *projects* для вправ із виразом `if`. У файл *src/main.rs* введіть таке:

Файл: *src/main.rs*

```
fn main() {  
    let number = 3;  
  
    if number < 5 {  
        println!("condition was true");  
    } else {  
        println!("condition was false");  
    }  
}
```

Всі вирази `if` починаються з ключового слова `if`, за яким іде умова. В цьому випадку умова перевіряє, має чи ні змінна `number` значення, менше за 5. Ми розміщуємо блок коду, який треба виконати, якщо умова `true`, одразу після умови в фігурних дужках. Блоки коду, прив'язані до умов у виразах `if`, іноді зовуть *рукавами*, так само як рукави у виразах `match`, що ми обговорювали у підрозділі [Порівняння здогадки з таємним числом](#) Розділу 2.

Також можна додати необов'язковий вираз `else`, як ми зробили тут, щоб надати програмі альтернативний блок коду для виконання, якщо умова виявиться `false`. Якщо ви не надасте виразу `else`, а умова буде `false`, програма просто пропустить блок `if` і перейде до наступного фрагмента коду.

Спробуйте запустити цей код; ви маєте побачити, що він виведе таке:

```
$ cargo run
Compiling branches v0.1.0 (file:///projects/branches)
Finished dev [unoptimized + debuginfo] target(s) in 0.31s
Running `target/debug/branches`
condition was true
```

Спробуймо змінити значення `number` на таке, що зробить умову `хибною`, і подивитися, що станеться:

```
let number = 7;
```

Запустіть програму знову і подивіться на результат:

```
$ cargo run
Compiling branches v0.1.0 (file:///projects/branches)
Finished dev [unoptimized + debuginfo] target(s) in 0.31s
Running `target/debug/branches`
condition was false
```

Також варто зазначити, що умова в цьому коді *має* бути типу `bool`. Якщо умова не `bool`, ми отримаємо помилку. Наприклад, спробуйте запустити такий код:

Файл: `src/main.rs`

```
fn main() {
    let number = 3;

    if number {
        println!("number was three");
    }
}
```

Умова у виразі `if` обчислюється тепер у значення `3`, і Rust повідомляє про помилку:

```
$ cargo run
Compiling branches v0.1.0 (file:///projects/branches)
error[E0308]: mismatched types
--> src/main.rs:4:8
   |
4 |     if number {
   |         ^^^^^ expected `bool`, found integer

For more information about this error, try `rustc --explain E0308`.
error: could not compile `branches` due to previous error
```

Помилка показує, що Rust очікував `bool`, але виявив ціле число. Rust не буде автоматично намагатися перетворити небулівські типи в булівський, на відміну від таких мов, як Ruby чи JavaScript. Ви маєте завжди явно надавати виразу `if` умову булівського типу. Якщо ми хочемо, щоб блок із кодом `if` виконувався тільки, скажімо, якщо число не дорівнює `0`, ми можемо змінити вираз `if` на такий:

Файл: `src/main.rs`

```
fn main() {  
    let number = 3;  
  
    if number != 0 {  
        println!("number was something other than zero");  
    }  
}
```

Виконання цього коду виведе `number was something other than zero`.

Обробка Декількох Умов з `else if`

Можливо обирати з багатьох умов, комбінуючи `if` та `else` у ланцюжок виразів `else if`. Наприклад:

Файл: `src/main.rs`

```
fn main() {  
    let number = 6;  
  
    if number % 4 == 0 {  
        println!("number is divisible by 4");  
    } else if number % 3 == 0 {  
        println!("number is divisible by 3");  
    } else if number % 2 == 0 {  
        println!("number is divisible by 2");  
    } else {  
        println!("number is not divisible by 4, 3, or 2");  
    }  
}
```

Ця програма має чотири можливі шляхи. Після запуску, ви маєте побачити таке:

```
$ cargo run  
Compiling branches v0.1.0 (file:///projects/branches)  
Finished dev [unoptimized + debuginfo] target(s) in 0.31s  
Running `target/debug/branches`  
number is divisible by 3
```

Коли ця програма виконується, вона перевіряє по черзі кожен вираз `if` і виконує перший блок, для якого умова справджується. Зверніть увагу, що, хоча 6 і ділиться на 2, ми не бачимо повідомлення `number is divisible by 2`, так само як не бачимо `number is not divisible by 4, 3 чи 2` з блоку `else` - бо Rust виконає тільки той блок, в якого першого умова буде `true`, а знайшовши його, не перевіряє всю решту умов.

Забгато виразів `else if` можуть захарастити ваш код, тому, якщо вам треба більш ніж одна така конструкція, цілком можливо, що знадобиться рефакторизувати ваш код. У Розділі 6 описана потужна конструкція мови Rust для розгалуження, що зветься `match`, для таких випадків.

Використання `if` в Інструкції `let`

Оскільки `if` є виразом, ми можемо використати його як праву частину інструкції `let`, щоб присвоїти результат змінній, як у Блоці коду 3-2.

Файл: `src/main.rs`

```
fn main() {  
    let condition = true;  
    let number = if condition { 5 } else { 6 };  
  
    println!("The value of number is: {number}");  
}
```

Блок коду 3-2: присвоєння результату виразу `if` змінній

Змінна `number` буде прив'язана до значення, залежно від результату обчислення виразу `if`. Запустіть цей код і подивіться, що відбудеться:

```
$ cargo run  
Compiling branches v0.1.0 (file:///projects/branches)  
Finished dev [unoptimized + debuginfo] target(s) in 0.30s  
Running `target/debug/branches`  
The value of number is: 5
```

Нагадаємо, що значенням блоку коду є значення останнього виразу в них, а числа як такі самі є виразами. В цьому випадку, значення всього виразу `if` залежить від того, який блок коду буде виконано. Це означає, що значення, які можуть бути результатами у кожному рукаві `if` мають бути одного типу; у Блоці коду 3-2 результати рукавів `if` та `else` є цілими числами типу `i32`. Якщо ж типи не будуть збігатися, як у наступному прикладі, ми отримаємо помилку:

Файл: `src/main.rs`


```
fn main() {
    let condition = true;

    let number = if condition { 5 } else { "six" };

    println!("The value of number is: {number}");
}
```

Якщо ми спробуємо запустити цей код, то отримаємо помилку. Рукави `if` та `else` мають несумісні типи значень, і Rust точно вказує, де шукати проблему в програмі:

```
$ cargo run
   Compiling branches v0.1.0 (file:///projects/branches)
error[E0308]: `if` and `else` have incompatible types
  --> src/main.rs:4:44
   |
4  |         let number = if condition { 5 } else { "six" };
   |                                -          ^^^^^^ expected integer, found
   |                                `&str`
   |                                |
   |                                expected because of this

For more information about this error, try `rustc --explain E0308`.
error: could not compile `branches` due to previous error
```

Вираз у блоці `if` обчислюється у ціле число, а вираз у блоці `else` обчислюється у стрічку. Це не працює, оскільки змінна мусить мати лише один тип. Rust має точно знати під час компіляції тип змінної `number`, щоб перевірити, що цей тип коректний усюди, де змінна `number` використовується. Rust не зможе зробити це, якщо тип `number` буде визначений після запуску програми; компілятор був би складнішим і надавав би менше гарантій стосовно коду, якби мусив стежити за чисельними можливими типами кожної змінної.

Повторення з Циклами

Часто трапляється, що блок коду треба виконати більше одного разу. Для цього Rust надає декілька *циклів*. Цикл виконує весь код тіла циклу до кінця, після чого починає спочатку. Для експериментів з циклами зробимо новий проєкт під назвою *loops*.

У Rust є три види циклів: `loop`, `while` та `for`. Спробуємо кожен з них.

Повторне Виконання Коду з `loop`

Ключове слово `loop` каже Rust виконувати блок коду знову і знову без кінця або ж доки не буде прямо сказано зупинитися.

Наприклад, змініть вміст файлу `src/main.rs` в теці `loops`, щоб він виглядав так:

Файл: `src/main.rs`

```
fn main() {  
    loop {  
        println!("again!");  
    }  
}
```

Якщо запустити цю програму, ми побачимо, що `again!` виводиться неперервно раз у раз, доки ми не зупинимо програму вручну. Більшість терміналів підтримують клавіатурне скорочення `ctrl+c`, яке зупиняє програму, що застрягла у нескінченному циклі. Спробуйте:

```
$ cargo run  
Compiling loops v0.1.0 (file:///projects/loops)  
Finished dev [unoptimized + debuginfo] target(s) in 0.29s  
Running `target/debug/loops`  
again!  
again!  
again!  
again!  
^Cagain!
```

Символ `^C` позначає, де ви натиснули `ctrl-c`. Слово `again!` може вивестися після `^C` чи ні, залежно від того, в який саме момент виконання коду був надісланий сигнал зупинки.

На щастя, Rust надає також інший, більш надійний спосіб перервати цикл за допомогою коду. Ви можете розмістити в циклі ключове слово `break`, щоб сказати програмі, коли припинити виконувати цикл. Згадайте, що ми вже його використовували у грі "Відгадай число" в підрозділі "Вихід після вдалої здогадки" Розділу 2, щоб вийти з програми, коли користувач виграв у грі, відгадавши правильне число.

Ми також використовували у цій грі `continue`, який у циклі каже програмі пропустити будь-який код, що лишився в цій ітерації циклу і перейти до наступної ітерації.

Повернення Значень з Циклів

Одне з застосувань циклу `loop` - повторні спроби здійснити операцію, яка може зазнати невдачі, такої як перевірка, чи завершив певний процес свою роботу. Вам може бути потрібно при цьому передати результат цієї операції з циклу для використання в подальшому коді. Для цього, ви можете дописати значення, що ви хочете повернути, після виразу `break`, яким ви зупиняєте цикл; це значення повернеться з циклу і ви зможете використати його, як показано тут:

```
fn main() {  
    let mut counter = 0;  
  
    let result = loop {  
        counter += 1;  
  
        if counter == 10 {  
            break counter * 2;  
        }  
    };  
  
    println!("The result is {result}");  
}
```

Перед циклом ми проголошуємо змінну `counter` і ініціалізуємо її в `0`. Потім ми проголошуємо змінну `result`, що отримає значення, повернуте з циклу. На кожній ітерації циклу ми додаємо `1` до змінної `counter`, а потім перевіряємо, чи дорівнює `counter` `10`. Коли так, ми використовуємо ключове слово `break` зі значенням `counter * 2`. Після циклу ми ставимо крапку з комою, щоб завершити інструкцію, що присвоює значення змінній `result`. Наприкінці ми виводимо значення `result`, у цьому випадку `20`.

Мітки Циклів для Розрізнення Між Декількома Циклами

Якщо у вас є цикли, вкладені в інші цикли, `break` і `continue` стосуються найглибшого циклу в точці, де вони застосовані. Ви можете за потреби вказати на циклі *мітку циклу*, що її можна потім використати в інструкціях `break` та `continue`, щоб вказати, що ці ключові слова стосуються циклу з міткою, а не найглибшого циклу. Мітки циклу починаються з одинарної лапки. Ось приклад із двома вкладеними циклами:

```
fn main() {
    let mut count = 0;
    'counting_up: loop {
        println!("count = {count}");
        let mut remaining = 10;

        loop {
            println!("remaining = {remaining}");
            if remaining == 9 {
                break;
            }
            if count == 2 {
                break 'counting_up;
            }
            remaining -= 1;
        }

        count += 1;
    }
    println!("End count = {count}");
}
```

Зовнішній цикл має мітку `'counting_up'`, і він лічить від 0 до 2. Внутрішній цикл без мітки лічить навпаки від 10 до 9. Перший `break`, без указання мітки, виходить лише з внутрішнього циклу. Інструкція `break 'counting_up;` вийде з зовнішнього циклу. Цей код виведе:

```
$ cargo run
Compiling loops v0.1.0 (file:///projects/loops)
Finished dev [unoptimized + debuginfo] target(s) in 0.58s
Running `target/debug/loops`
count = 0
remaining = 10
remaining = 9
count = 1
remaining = 10
remaining = 9
count = 2
remaining = 10
End count = 2
```

Умовні Цикли з `while`

Програмі часто потрібно обчислювати умову в циклі. Доки умова `true`, цикл виконується. Коли умова припиняє бути `true`, програма викликає `break`, щоб зупинити цикл. Подібну поведінку можна реалізувати за допомогою комбінації `loop`, `if`, `else` та `break`; якщо бажаєте, можете спробувати зробити це зараз. Утім, цей шаблон настільки поширений, що Rust має вбудовану конструкцію для цього, що зветься циклом `while`. У Блоці коду 3-3

ми використовуємо `while`, щоб повторити програму тричі, зменшуючи кожного разу відлік, і потім, після циклу, вивести повідомлення і завершитися.

Файл: `src/main.rs`

```
fn main() {  
    let mut number = 3;  
  
    while number != 0 {  
        println!("{number}!");  
  
        number -= 1;  
    }  
  
    println!("LIFTOFF!!!");  
}
```

Блок коду 3-3: використання циклу `while` для виконання коду, поки умова лишається істинною

Ця конструкція мови усуває складні вкладені конструкції, які були б потрібні, якби ви використовували `loop`, `if`, `else` та `break`, і вона зрозуміліша. Поки умова `true`, код виконується; в іншому разі, виходить з циклу.

Циклічний Прохід по Колекції з `for`

Ви можете скористатися конструкцією `while`, щоб зробити цикл по елементах колекції, такої, як масив. Наприклад, цикл у Блоці коду 3-4 виводить кожен елемент масиву `a`.

Файл: `src/main.rs`

```
fn main() {  
    let a = [10, 20, 30, 40, 50];  
    let mut index = 0;  
  
    while index < 5 {  
        println!("the value is: {}", a[index]);  
  
        index += 1;  
    }  
}
```

Блок коду 3-4: перебір елементів колекції за допомогою циклу `while`

Тут код перелічує всі елементи в масиві. Він починає з індексу `0`, а потім повторює, доки не досягне останнього індексу масиву (тобто коли `index < 5` вже не буде `true`).

Виконання цього коду виведе всі елементи масиву:

```
$ cargo run
Compiling loops v0.1.0 (file:///projects/loops)
Finished dev [unoptimized + debuginfo] target(s) in 0.32s
Running `target/debug/loops`
the value is: 10
the value is: 20
the value is: 30
the value is: 40
the value is: 50
```

Всі п'ять значень з масиву з'являються в терміналі, як і очікувалося. Хоча `index` досягне значення `5`, виконання циклу припиняється до спроби отримати шосте значення з масиву.

Але такий підхід вразливий до помилок; ми можемо викликати паніку в програмі некоректним індексом чи умовою продовження. Скажімо, якщо ви зміните визначення масиву `a` так, щоб він мав чотири елементи, і забудете змінити умову на `while index < 4`, це код викличе паніку. Також він повільний, оскільки компілятор додає код для перевірки коректності індексу кожного елементу на кожній ітерації циклу.

Як стислішу альтернативу можна використати цикл `for`, який виконує код для кожного елементу колекції. Цикл `for` виглядає так, як показано в Блоці коду 3-5.

Файл: `src/main.rs`

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

    for element in a {
        println!("the value is: {element}");
    }
}
```

Блок коду 3-5: перебір елементів колекції за допомогою циклу `for`

Запустивши цей код, ми побачимо такий самий вивід, як і в Блоці коду 3-4. Що важливіше, ми збільшили безпеку коду та усунули можливість помилок - тепер неможливо, що код перейде за кінець масиву чи завершиться зарано, пропустивши кілька значень.

При використанні циклу `for` вам не треба пам'ятати, що треба змінити якийсь інший код, якщо ви змінили кількість значень у масиві, як це потрібно за методу, застосованого в Блоці коду 3-4.

Безпечність і лаконічність циклів `for` робить їх найпоширенішою конструкцією циклів у Rust. Навіть у ситуаціях, де треба виконати певний код визначену кількість разів, як у прикладі відліком в циклі `while` з Блоку коду 3-3, більшість рашацеанців скористаються

циклом `for`. Для цього треба буде скористатися типом `Range` ("діапазон"), який надається стандартною бібліотекою і генерує послідовно всі числа, починаючи з одного і закінчуючись перед іншим.

Ось як виглядає відлік, що використовує цикл `for` і ще один метод, про який ми ще не говорили, `rev`, для обернення діапазону:

Файл: `src/main.rs`

```
fn main() {  
    for number in (1..4).rev() {  
        println!("{number}!");  
    }  
    println!("LIFTOFF!!!");  
}
```

Виглядає трохи краще, правда ж?

Підсумок

Нарешті закінчили! Це був величенький розділ: ви вивчили змінні, скалярні та складені типи даних, функції, коментарі, вирази `if`, та ще цикли! Якщо ви хочете повправлятися з концепціями, обговореними у цьому розділі, спробуйте написати програми, що роблять таке:

- Конвертує температуру між шкалами Фаренгейта та Цельсія.
- Обчислює n -е число Фібоначчі.
- Виводить слова англійської різдвяної пісні "Дванадцять днів Різдва" з використанням повторень у пісні (якщо хочете - можете спробувати вивести казку "Ріпка").

Коли будете готові продовжувати, ми поговоримо про концепцію мови Rust, якої *немає* серед поширених в інших мовах програмування - володіння. [ch02-00-guessing-game-tutorial.html#comparing-the-guess-to-the-secret-number](https://pavloslav.github.io/RustBookUkr/print.html#comparing-the-guess-to-the-secret-number) [ch02-00-guessing-game-tutorial.html#quitting-after-a-correct-guess](https://pavloslav.github.io/RustBookUkr/print.html#quitting-after-a-correct-guess)

Розуміння Володіння

Володіння є найунікальнішою особливістю мови Rust, що має глибокий вплив на решту мови. Воно дозволяє Rust гарантувати безпечну роботу з пам'яттю без потреби у збирачі сміття, і тому важливо розуміти, як володіння працює. У цьому розділі ми поговоримо про володіння і декілька пов'язаних особливостей: позичання, слайси, і як Rust розташовує дані в пам'яті.

Що Таке Володіння?

Володіння - це набір правил, які регулюють, як програма на Rust керує пам'яттю. Усі програми мають керувати тим, як вони використовують пам'ять комп'ютера під час роботи. Деякі мови мають збирача сміття, який постійно шукає пам'ять, що її вже не використовують, під час роботи програми; в інших мовах програміст має явно виділяти та звільняти пам'ять. Rust використовує третій підхід: пам'ять управляється системою володіння з набором правил, які перевіряє компілятор. Якщо якесь із правил порушується, програма не скомпілюється. Жодна з особливостей володіння не сповільнить вашу програму під час виконання.

Оскільки володіння - нова концепція для багатьох програмістів, потрібен деякий час, щоб звикнути до нього. Добра новина - що досвідченішим ви ставатимете в Rust і правилах системи володіння, тим легшим для вас буде природно писати безпечний і ефективний код. Не здавайтеся!

Коли ви зрозумієте володіння, ви матимете надійну основу для розуміння особливостей, що роблять Rust унікальною мовою. В цьому розділі ви вивчите володіння, працюючи з прикладами, що зосереджуються на добре відомих структурах даних: рядках.

Стек і Купа

У багатьох мовах програмування програміст нечасто має думати про стек і купу. Але в системних мовах, таких, як Rust, розташування значення в стеку чи в купі впливає на поведінку програми й змушує вас ухвалювати певні рішення. Деталі володіння будуть описані стосовно стека та купи пізніше у цьому розділі, а тут коротке пояснення для підготовки.

І стек, і купа - частини пам'яті, до яких ваш код має доступ під час виконання, але вони мають різну структуру. Стек зберігає значення в порядку, в якому їх отримує, і видаляє їх у зворотньому порядку. Це зветься *останнім надійшов, першим пішов (last in, first out)*. Стек можна уявити, як стос тарілок: коли ви додаєте тарілки, треба ставити їх зверху, а коли треба зняти тарілку, то доводиться брати теж зверху. Додавання чи прибирання тарілок з середини чи знизу стосу матимуть значно гірший наслідок! Додавання даних також зветься *заштовхуванням у стек (push)*, а видалення - відповідно, *виштовхуванням (pop)*. Усі дані, що зберігаються в стеку, мають бути відомого і незмінного розміру. Дані, розмір яких невідомий під час компіляції, або може змінитися, мають зберігатися в купі.

Купа менш організована: коли ви розміщуєте дані в купі, то запитуєте певний обсяг місця. Програма-розподільувач знаходить достатньо велику порожню ділянку в купі, позначає, що вона використовується, і повертає *вказівник*, тобто адресу цього місця. Цей процес зветься *розподілом у купі*, що іноді скорочується до простого *розподілом* (заштовхування значень до стека не вважається розподілом). Оскільки вказівник на купу має відомий, постійний розмір, ви можете зберегти цей вказівник у стеку, але коли вам потрібні дані, вам треба перейти за вказівником. Уявіть собі столи в ресторані. Коли ви входите до ресторану, вам треба сказати кількість людей, що прийшли з вами, тоді офіціант знайде вам порожній стіл, за який всі зможуть сісти, і відведе вас до нього. Якщо хтось спізнився, він зможе спитати, де вас розмістили, щоб приєднатися.

Заштовхування до стека швидше за розподіл у купі, оскільки розподільувачу не треба шукати місце для нових даних, бо це місце завжди є вершиною стека. Розподіл місця у купі вимагає порівняно більше роботи, бо розподільувач має спершу знайти достатньо місця для даних, а потім провести облік місця, щоб приготуватися до наступного розподілу.

Доступ до даних у купі повільніший, ніж у стеку, бо треба переходити за вказівником, щоб дістатися туди. Сучасні процесори швидше працюють, якщо відбувається менше переходів у пам'яті. Розвинемо аналогію: уявімо офіціанта у ресторані, який приймає замовлення з багатьох столів. Найефективніше буде > прийняти всі замовлення з одного столу перед тим, як переходити до наступного. Приймати замовлення зі столу А, потім зі столу В, потім знову з А і знову з В буде значно повільніше. З тієї ж причини процесор краще працює з даними, розташованими поруч (як у стеку), ніж далеко (як може статися в купі).

Коли ваш код викликає функцію, значення, що передаються у функцію (включно з, можливо, вказівниками на дані у купі) і локальні змінні функції заштовхуються у стек. Коли функція завершується, ці значення виштовхуються зі стека.

Відстеження, які частини коду використовують які дані в купі, мінімізація дублювання даних у купі та очищення даних у купі, що вже не потрібні, щоб не скінчилося місце - ось ті завдання, які покликане розв'язати володіння. Коли ви зрозумієте концепцію володіння, вам більше не треба буде постійно думати про стек і купу, але знання, що причина існування володіння - управління даними у купі, допоможе вам зрозуміти, чому воно працює саме так.

Правила Володіння

По-перше, познайомимося із правилами володіння. Тримайте ці правила на увазі, поки ми працюватимемо із прикладами, що їх ілюструють:

- Кожне значення в Rust має *власника*.
- У кожен момент може бути лише один власник.
- Коли власник виходить зі зони видимості, значення буде скинуто.

Область Видимості Змінної

Тепер, оскільки ми вже знайомі з основами синтаксису Rust, більше не будемо включати всі ці `fn main() {` у приклади, тому, щоб випробувати їх, вам доведеться помістити ці приклади до функції `main` самостійно. Завдяки цьому приклади стануть лаконічнішими і дозволять зосередитися на важливих деталях, а не на шаблонному коді.

У першому прикладі володіння ми розглянемо *область видимості* деяких змінних. Область видимості - це фрагмент програми, в якому з елементом можна працювати. Нехай ми маємо ось таку змінну:

```
let s = "hello";
```

Змінна `s` посилається на рядковий літерал, де значення рядка закодовано у текст програми. Зі змінною можна працювати з моменту її проголошення до кінця поточної *області видимості*. Коментарі у Блоці коду 4-1 підказують, де змінна `s` доступна.

```
{           // s is not valid here, it's not yet declared
  let s = "hello"; // s is valid from this point forward

  // do stuff with s
}           // this scope is now over, and s is no longer valid
```

Блок коду 4-1: Змінна і область видимості, де вона доступна

Іншими словами, є два важливі моменти часу:

- Коли `s` потрапляє в *область видимості*, вона стає доступною.
- Вона лишається доступною, доки не вийде з *області видимості*.

Поки що стосунки між областями видимості та доступністю змінних такі ж самі, як і в інших мовах програмування. Тепер, спираючися на це розуміння, будемо розвиватися, додавши тип `String`.

Тип `String`

Щоб проілюструвати правила володіння, нам знадобиться тип даних, складніший за `ti`, що ми вже розглянули у підрозділі “[Типи даних](#)” Розділу 3. Всі типи даних, які ми розглядали раніше, мають заздалегідь відомий розмір, можуть зберігатися в стеку і виштовхуватися звідти, коли їхня область видимості закінчується, і їх можна швидко і просто скопіювати, щоб зробити новий, незалежний екземпляр, коли інша частина коду потребує використати те саме значення в іншій області видимості. Але тепер ми розглянемо дані, що зберігаються в купі та подивимося, як Rust дізнається, коли ці дані треба вичищати, і тип `String` є чудовим прикладом.

Ми зосередимося на особливостях `String`, що стосуються володіння. Ці аспекти також застосовуються до інших складних типів даних, які надає стандартна бібліотека або ви створюєте самі. Ми поговоримо про `String` більш детально в [Розділі 8](#).

Ми вже бачили рядкові літерали, у яких значення рядка закодоване в нашу програму. Рядкові літерали зручні, але не завжди підходять для різних ситуацій, де виникає потреба скористатися текстом. Одна з причин полягає в тому, що вони є сталими. Інша - що не кожне значення рядка є відомим під час написання коду: наприклад, як взяти те, що ввів користувач, і зберегти його? Для цих ситуацій, Rust має другий рядковий тип, `String`. Цей тип керує даними, розподіленими в купі й, відтак, може зберігати текст, обсяг якого невідомий під час компіляції. Можна створити `String` з рядкового літерала за допомогою функції `from`, ось так:

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

Оператор подвійна двокрапка `::` дозволяє доступ до простору імен, що надає нам можливість використати, в цьому випадку, функцію `from` з типу `String`, щоб не довелося використовувати назву на кшталт `string_from`. Цей синтаксис детальніше обговорюється у підрозділі “[Синтаксис методів](#)” Розділу 5 і в обговоренні просторів імен в модулях у “[Способи звертання до елементу в модульному дереві](#)” Розділу 7.

Цей тип рядків може бути зміненим:

```
let mut s = String::from("hello");  
  
s.push_str(", world!"); // push_str() appends a literal to a String  
  
println!("{}", s); // This will print `hello, world!`
```

У чому ж різниця? Чому `String` може бути зміненим, але літерали - ні? Різниця полягає в тому, як ці два типи працюють із пам'яттю.

Пам'ять та її Виділення

У випадку рядкового літерала ми знаємо його вміст під час компіляції, тому текст жорстко заданий прямо у виконуваному файлі, що робить рядкові літерали швидкими і ефективними. Але ці властивості впливають з незмінності літерала. На жаль, ми не можемо розмістити у двійковому файлі по шмату пам'яті для кожного фрагменту тексту, розмір якого ми не знаємо під час компіляції й чий розмір може змінитися під час виконання програми.

Для типу `String`, задля підтримки несталого шматка тексту, що може зростати, нам потрібно розподілити певну кількість пам'яті в купі, невідому під час компіляції, для зберігання вмісту. Це означає:

- Пам'ять має бути запитана в розподільвача під час виконання.
- Нам потрібен спосіб повернення цієї пам'яті до розподільвача, коли ми закінчили працювати з нашим `String`.

Першу частину робимо ми самі: коли ми викликаємо `String::from`, її реалізація запитує потрібну пам'ять. Так роблять практично всі мови програмування.

Але друга частина відбувається інакше. У мовах зі *збирачем сміття* (*garbage collector, GC*), саме GC стежить і очищує пам'ять, що більше не використовується, і ми, як програмісти, більше можемо не думати про неї. У більшості мов без GC це наша відповідальність - визначити, яка пам'ять більше не потрібна та викликати код для її повернення, так само як до того ми її запитали. Правильно це робити історично є складною задачею у програмуванні. Якщо ми забудемо, ми змарнуємо пам'ять. Якщо ми це зробимо зарано, ми матимемо некоректну змінну. Якщо ми це зробимо двічі, це теж буде помилкою. Потрібно забезпечити, щоб на кожен **розподіл** було рівно одне **звільнення** пам'яті.

Rust іде іншим шляхом: пам'ять автоматично повертається, щойно змінна, що нею володіла, іде з області видимості. Ось версія нашого прикладу з Блоку коду 4-1 із використанням `String` замість рядкового літерала:

```
{
    let s = String::from("hello"); // s is valid from this point forward

    // do stuff with s

}                                // this scope is now over, and s is no
                                // longer valid
```

Існує точка, де природно можна повернути пам'ять, використану нашим рядком, розподільвачу: коли `s` іде з області видимості. Коли змінна виходить з області видимості, Rust викликає для нас спеціальну функцію. Ця функція зветься `drop`, і саме там автор

`String` може розмістити код для повернення пам'яті. Rust викликає `drop` автоматично на закриваючій фігурній дужці.

Примітка: в C++ цей шаблон звільнення ресурсів наприкінці життя об'єкта іноді зветься *Отримання ресурсу є ініціалізація (Resource Acquisition Is Initialization, RAII)*. Функція Rust `drop` має бути знайома вам, якщо ви користувалися шаблонами RAII.

Цей шаблон має глибокий вплив на спосіб написання коду Rust. Він наразі може виглядати простим, але поведінка коду може бути неочікуваною у складніших ситуаціях, коли ми працюватимемо із декількома змінними, що використовують дані, розподіленими в купі. Тепер дослідимо деякі з цих ситуацій.

Взаємодія Змінних з Даними: Переміщення

Різні змінні у Rust можуть взаємодіяти з одними й тими ж даними у різні способи. Подивимося на приклад, що використовує ціле число, у Блоці коду 4-2.

```
let x = 5;  
let y = x;
```

Блок коду 4-2: Присвоєння цілого значення змінної `x` змінній `y`

Ми, мабуть, можемо здогадатися, що робить цей код: "прив'язати значення `5` до `x`; потім зробити копію значення у `x` і прив'язати її до `y`". Тепер ми маємо дві змінні, `x` та `y`, і обидві дорівнюють `5`. І дійсно це так і відбувається, бо цілі - прості значення із відомим, фіксованим розміром, і ці два значення `5` заштовхуються у стек.

Тепер подивимося на версію зі `String`:

```
let s1 = String::from("hello");  
let s2 = s1;
```

Це виглядає дуже схожим на попередній код, тому ми можемо припустити, що воно працює так само, тобто другий рядок створить копію значення з `s1` і прив'яже її до `s2`. Але тут відбувається щось трохи інше.

Поглянемо на Рисунок 4-1, щоб зрозуміти, що відбувається всередині `String`. `String` складається з трьох частин, показаних ліворуч: вказівника на пам'ять, що зберігає вміст рядка, довжину та місткість. Цей набір даних зберігається в стеку. Праворуч показана пам'ять у купі, що зберігає вміст.

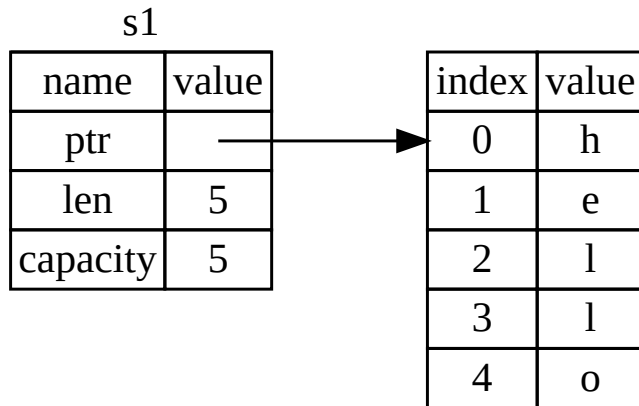


Рисунок 4-1: Представлення в пам'яті `String` зі значенням `"hello"`, прив'язаної до `s1`

Довжина - це кількість пам'яті, в байтах, що вміст `String` наразі використовує. Місткість - це загальний обсяг пам'яті в байтах, що `String` отримала від розподільвача. Різниця між довжиною та місткістю має значення, але не в цьому контексті, тому поки що місткість можна спокійно ігнорувати.

Коли ми присвоюємо значення `s1` змінній `s2`, дані `String` копіюються - тобто копіюється вказівник, довжина і місткість, що знаходяться в стеку. Ми не копіюємо даних у купі, на які посилається вказівник. Іншими словами, представлення даних у пам'яті виглядає як на Рисунку 4-2.

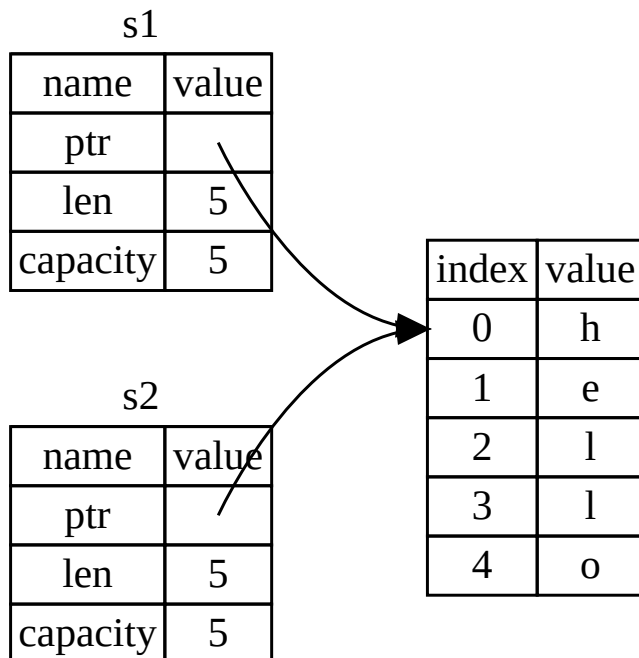


Рисунок 4-2: Представлення в пам'яті змінної `s2`, що містить копію вказівника, довжини та місткості з `s1`

Представлення *не* виглядає, як показано на Рисунку 4-3, як було б якби Rust дійсно копіювала також і дані в купі. Якби Rust так робила, операція `s2 = s1` була б потенційно дуже витратною з точки зору швидкості виконання, якщо в купі було б багато даних.

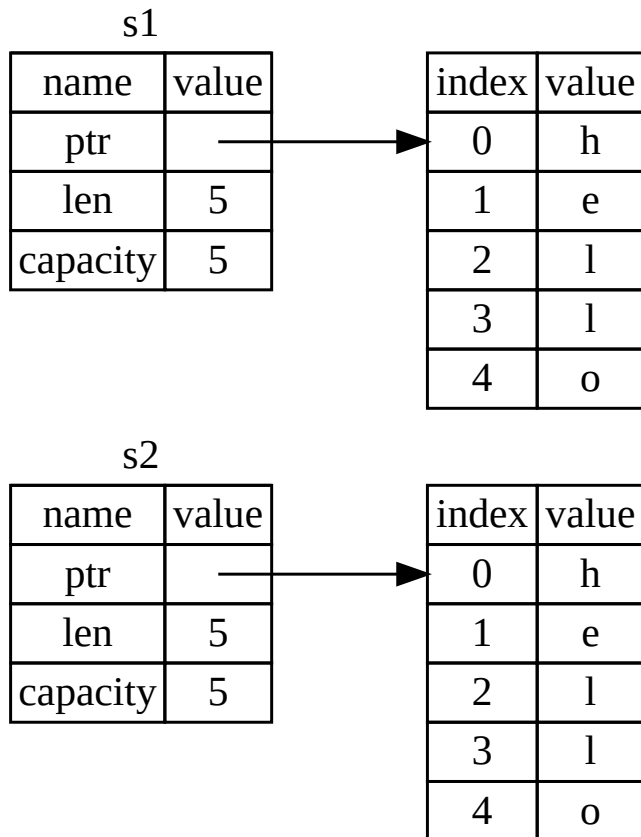


Рисунок 4-3: Інша можливість того, що могло б робити `s2 = s1`, якби Rust копіювала також дані в купі

Раніше ми казали, що коли змінна виходить з області видимості, Rust автоматично викликає функцію `drop` і очищає пам'ять цієї змінної в купі. Але Рисунок 4-2 показує, що обидва вказівники вказують на одне й те саме місце. Це створює проблему: коли `s2` і `s1` вийдуть з області видимості, вони удвох спробують звільнити одну й ту саму пам'ять. Це зветься помилкою *подвійного звільнення*, і ми про неї вже згадували. Звільнення пам'яті двічі може призвести до пошкодження пам'яті, і, потенційно, до вразливостей у безпеці.

Для убезпечення пам'яті після рядка `let s2 = s1` Rust розглядає змінну `s1` як більше не коректну. Відтак, Rust тепер не буде нічого звільняти, коли `s1` вийде з області видимості. Перевірте, що станеться, коли ви спробуєте використати `s1` після створення `s2`; це не спрацює:


```
let s1 = String::from("hello");  
let s2 = s1;  
  
println!("{}", world!", s1);
```

Ви отримаєте помилку на кшталт цієї, бо Rust не допускає використання некоректних посилань:

```
$ cargo run  
  Compiling ownership v0.1.0 (file:///projects/ownership)  
error[E0382]: borrow of moved value: `s1`  
--> src/main.rs:5:28  
   |  
2  |     let s1 = String::from("hello");  
   |         -- move occurs because `s1` has type `String`, which does not  
implement the `Copy` trait  
3  |     let s2 = s1;  
   |         -- value moved here  
4  |  
5  |     println!("{}", world!", s1);  
   |                               ^^ value borrowed here after move  
   |  
= note: this error originates in the macro `$crate::format_args_nl` which comes  
from the expansion of the macro `println` (in Nightly builds, run with -Z macro-  
backtrace for more info)  
  
For more information about this error, try `rustc --explain E0382`.  
error: could not compile `ownership` due to previous error
```

Якщо ви чули терміни *пласка копія* та *глибока копія* ("shallow copy" та "deep copy"), коли працювали з іншими мовами, поняття копіювання вказівника, довжини та місткості без копіювання даних виглядають для вас схожими на пласку копію. Але оскільки Rust також унепридатнює першу змінну, це зветься не пласкою копією, а *переміщенням*. У цьому прикладі ми кажемо, що `s1` було *переміщено* в `s2`. Що фактично відбувається, показано на Рисунок 4-4.

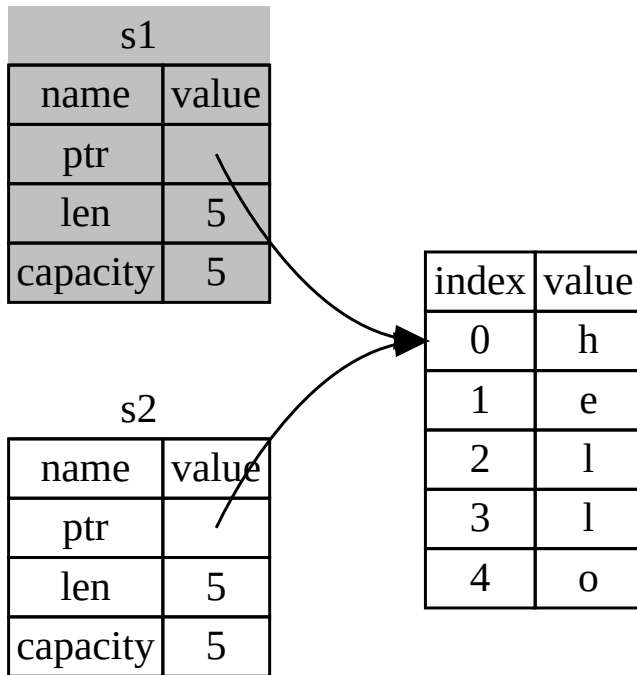


Рисунок 4-4: Представлення в пам'яті після унепридатнення `s1`

Це розв'язує нашу проблему! Якщо коректним залишилося лише `s2`, коли воно вийде з області видимості, то саме звільнить пам'ять, і готово.

На додачу, такий дизайн мови неявно гарантує, що Rust ніколи не буде автоматично створювати "глибокі" копії ваших даних. Таким чином, будь-яке *автоматичне* копіювання може вважатися недорогим з точки зору продуктивності під час виконання.

Взаємодія Змінних з Даними: Клонування

Якщо ми хочемо зробити глибоку копію даних `String` у купі, а не лише в стеку, ми можемо використати загальний метод, що зветься `clone`. Синтаксис використання методів буде обговорено в Розділі 5, але оскільки методи є загальною особливістю багатьох мов програмування, ви, швидше за все, вже бачили їх.

Ось приклад застосування методу `clone`:

```
let s1 = String::from("hello");
let s2 = s1.clone();

println!("s1 = {}, s2 = {}", s1, s2);
```

Це чудово працює і явно продукує поведінку, показану на Рисунку 4-3, де дані в купі *дійсно* копіюються.

Коли ви бачите виклик `clone`, ви знаєте, що виконується певний визначений код і цей код може коштувати продуктивності. Це візуальний індикатор, що відбувається певна операція.

Стекові Дані: Копіювання

Є ще одна дрібниця, про яку ми ще не говорили. Цей код, що використовує цілі числа, частина якого вже була показана раніше в Блоці коду 4-2, коректно працює:

```
let x = 5;
let y = x;

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

But this code seems to contradict what we just learned: we don't have a call to `clone`, but `x` is still valid and wasn't moved into `y`.

Причина у тому, що типи на кшталт цілих, що мають відомий розмір часу компіляції, зберігаються повністю в стеку, тому копіювання їхніх значень відбувається швидко. Це означає, що нема підстав запобігати коректності `x` після створення змінної `y`. Іншими словами, тут немає різниці між глибокою та пласкою копією, і виклик `clone` не зробить нічого відмінного від звичайного плаского копіювання, тож можна його не викликати.

Rust має спеціальне позначення, що зветься трейтом `Copy`, який можна додати до типів на кшталт цілих, що зберігаються в стеку (детальніше трейти обговорюються в [Розділі 10](#)). Якщо тип реалізовує трейт `Copy`, змінні, що використовують його не переміщуються, а банально копіюються, що робить їх коректними після присвоєння іншій змінній.

Rust не дозволить позначити тип трейтом `Copy`, якщо тип, чи якась з його частин, реалізовує `трейт`Drop`. Якщо тип потребує чогось особливого, коли змінна виходить з області видимості, і ми додаємо позначення `Copy` до цього типу, ми отримаємо помилку часу компіляції. Щоб дізнатися про те, як додати позначку `Copy` до вашого типу для реалізації трейта, див. ["Придатні до успадкування трейти"](#) у Додатку C.

Тож які типи реалізовують трейт `Copy`? Можна перевірити документацію до певного типу, щоб бути певним, але загальне правило таке: будь-яка група простих скалярних значень може реалізовувати `Copy`, і нічого з того, що потребує розподілу пам'яті чи є ресурсом, не є `Copy`. Ось кілька типів, що реалізовують `Copy`:

- Всі цілі типи, на кшталт `u32`.
- Булевий тип, `bool`, значення якого `true` та `false`.
- Всі типи з рухомою комою, на кшталт `f64`.
- Символьний тип, `char`.

- Кортежі, якщо вони містять лише типи, що реалізують `Copy`. Скажімо, `(i32, i32)` реалізовує `Copy`, але `(i32, String)` - ні.

Володіння та Функції

Механіка передачі значень функції подібна до присвоювання значення змінній. Передача змінної функції є переміщенням чи копією, як і присвоювання. Блок коду 4-7 містить приклад з певними поясненнями, що розкривають, де змінні входять і виходять з області видимості.

Файл: `src/main.rs`

```
fn main() {
    let s = String::from("hello"); // s comes into scope

    takes_ownership(s);             // s's value moves into the function...
                                    // ... and so is no longer valid here

    let x = 5;                       // x comes into scope

    makes_copy(x);                  // x would move into the function,
                                    // but i32 is Copy, so it's okay to still
                                    // use x afterward
} // Here, x goes out of scope, then s. But because s's value was moved, nothing
// special happens.

fn takes_ownership(some_string: String) { // some_string comes into scope
    println!("{}", some_string);
} // Here, some_string goes out of scope and `drop` is called. The backing
// memory is freed.

fn makes_copy(some_integer: i32) { // some_integer comes into scope
    println!("{}", some_integer);
} // Here, some_integer goes out of scope. Nothing special happens.
```

Listing 4-3: Functions with ownership and scope annotated

Якби ми спробували використати `s` після виклику `takes_ownership`, Rust повідомив би про помилку часу компіляції. Ці статичні перевірки захищають нас від помилок. Спробуйте додати в `main` код, що використовує `s` та `x`, щоб побачити, де їх можна використовувати, а де правила володіння запобігають цьому.

Повернення Значень та Область Видимості

Повернення значень також передає володіння. Блок коду 4-4 містить приклад функції, що повертає значення, зі схожими на Блок коду 4-3 поясненнями.

Файл: src/main.rs

```
fn main() {  
    let s1 = gives_ownership();           // gives_ownership moves its return  
                                         // value into s1  
  
    let s2 = String::from("hello");      // s2 comes into scope  
  
    let s3 = takes_and_gives_back(s2);   // s2 is moved into  
                                         // takes_and_gives_back, which also  
                                         // moves its return value into s3  
} // Here, s3 goes out of scope and is dropped. s2 was moved, so nothing  
  // happens. s1 goes out of scope and is dropped.  
  
fn gives_ownership() -> String {        // gives_ownership will move its  
                                         // return value into the function  
                                         // that calls it  
  
    let some_string = String::from("yours"); // some_string comes into scope  
  
    some_string                          // some_string is returned and  
                                         // moves out to the calling  
                                         // function  
}  
  
// This function takes a String and returns one  
fn takes_and_gives_back(a_string: String) -> String { // a_string comes into  
                                                         // scope  
  
    a_string // a_string is returned and moves out to the calling function  
}
```

Блок коду 4-4: Передача володіння значенням, що повертається

Володіння змінними завше дотримується однакової схеми: присвоєння значення іншій змінній переміщує його. Коли змінна, що включає дані в купі, виходить з області видимості, якщо дані не були переміщені у володіння іншої змінної, значення буде очищене викликом `drop`.

Хоча так код працює, все ж взяття володіння і повернення володіння в кожній функції дещо втомлює. Що, як ми хочемо дозволити функції використати значення, але не брати володіння? Потреба повертати все, що ми передаємо в функції, щоб його можна було знову використовувати, разом із даними, утвореними в результаті роботи функції, дратує.

Можна повертати багато значень кортежем, як показано в Блоці коду 4-5.

Файл: src/main.rs

```
fn main() {  
    let s1 = String::from("hello");  
  
    let (s2, len) = calculate_length(s1);  
  
    println!("The length of '{}' is {}.", s2, len);  
}  
  
fn calculate_length(s: String) -> (String, usize) {  
    let length = s.len(); // len() returns the length of a String  
  
    (s, length)  
}
```

Блок коду 4-5: Повернення володіння параметрами

Але це б давало забагато ритуальних рухів і зайвої роботи для концепції, що має бути загальноновживаною. На щастя для нас, Rust має засіб для використання змінної без передачі володіння, що зветься *посиланнями*.

Посилання та Позичання

Проблема з кодом, що використовує кортежі, з Блоку коду 4-5 полягає в тому, що ми маємо повертати `String` у функцію, що викликає, щоб можна було використовувати `String` після виклику `calculate_length`, бо `String` переміщується до `calculate_length`. Натомість ми можемо надати посилання на значення `String`. *Посилання* - це як вказівник, тобто адреса, за якою можна перейти, щоб отримати дані, збережені за цією адресою; ці дані є володінням якоїсь іншої змінної. На відміну від вказівника, посилання гарантовано вказує на коректне значення певного типу весь час існування цього посилання.

Ось як ви маєте визначити і використовувати функцію `calculate_length`, що має параметром посилання на об'єкт замість перебирання володіння значенням:

Файл: `src/main.rs`

```
fn main() {  
    let s1 = String::from("hello");  
  
    let len = calculate_length(&s1);  
  
    println!("The length of '{}' is {}.", s1, len);  
}  
  
fn calculate_length(s: &String) -> usize {  
    s.len()  
}
```

По-перше, зауважте, що весь код із кортежами при визначенні змінної та поверненні з функції зник. По-друге, зауважте, що ми передаємо `&s1` у `calculate_length`, а у визначенні функції ми приймаємо `&String` замість `String`. Ці амперсанди представляють *посилання*, і вони дозволяють нам посилатися на певне значення, не перебираючи володіння ним. Рисунок 4-5 описує цю концепцію.

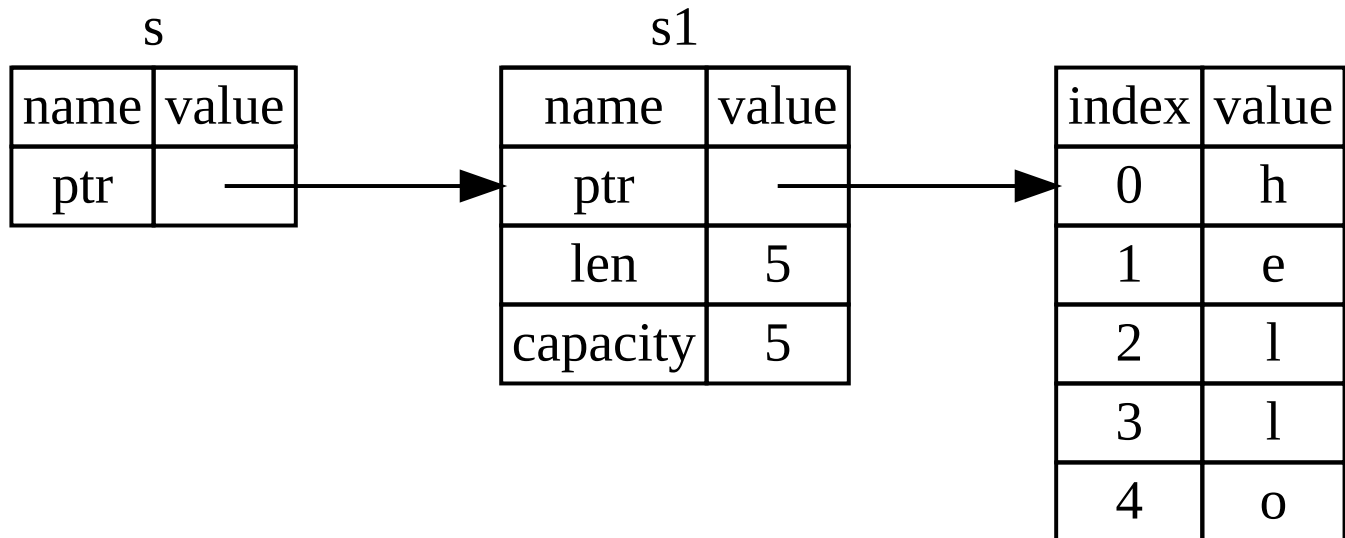


Рисунок 4-5: Діаграма, як `&String s` вказує на `String s1`

Примітка: операція, зворотна до посилання `&`, зветься *розіменуванням*, і виконується оператором розіменування `*`. Ми побачимо деякі застосування оператора розіменування в Розділі 8 і обговоримо подробиці розіменування у Розділі 15.

Розглянемо детальніше виклик функції:

```
let s1 = String::from("hello");
let len = calculate_length(&s1);
```

Запис `&s1` створює посилання, що *посилається* на значення `s1`, але не володіє ним. Оскільки володіння немає, значення, на яке воно вказує, не буде знищене, коли посилання вийде з області видимості.

Так само, сигнатура функції використовує `&`, щоб показати, що тип параметра `s` - посилання. Додамо трохи коментарів для пояснення:

```
fn calculate_length(s: &String) -> usize { // s is a reference to a String
    s.len()
} // Here, s goes out of scope. But because it does not have ownership of what
// it refers to, it is not dropped.
```

Область видимості, де змінна `s` є коректною, така сама, як і у будь-якого параметра функції, але значення, на що вказує посилання, не припиняє свого існування коли `s` припиняє використовуватися, бо `s` ним не володіє. Коли функції мають параметри -

посилання замість значень, нам не треба повертати значення, щоб повернути володіння, бо ми й не мали володіння.

Операція створення посилання зветься *позичанням*. Як і в справжньому житті, якщо особа володіє чимось, ви можете це позичити у неї, а коли річ вам стане не потрібна, треба її віддати. Бо вона вам не належить.

Що ж станеться, якщо ми спробуємо змінити щось, що ми позичили? Спробуйте запустити код з Блоку коду 4-6. Обережно, спойлер: він не працює!

Файл: src/main.rs

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

    change(&s);
}

fn change(some_string: &String) {
    some_string.push_str(", world");
}
```

Блок коду 4-6: Спроба змінити позичене значення

Ось помилка:

```
$ cargo run
   Compiling ownership v0.1.0 (file:///projects/ownership)
error[E0596]: cannot borrow `*some_string` as mutable, as it is behind a `&`
reference
--> src/main.rs:8:5
   |
7 | fn change(some_string: &String) {
   |                               ----- help: consider changing this to be a mutable
reference: `&mut String`
8 |     some_string.push_str(", world");
   |     ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^ `some_string` is a `&` reference, so the
data it refers to cannot be borrowed as mutable

For more information about this error, try `rustc --explain E0596`.
error: could not compile `ownership` due to previous error
```

Посилання, так само як і змінні, за умовчання є немутабельними. Ми не можемо змінити щось, на що ми маємо посилання.

Мутабельні Посилання

We can fix the code from Listing 4-6 to allow us to modify a borrowed value with just a few small tweaks that use, instead, a *mutable reference*:

Файл: src/main.rs

```
fn main() {  
    let mut s = String::from("hello");  
  
    change(&mut s);  
}  
  
fn change(some_string: &mut String) {  
    some_string.push_str(", world");  
}
```

По-перше, треба змінити `s`, щоб він став `mut`. Потім ми створюємо мутабельне посилання за допомогою `&mut s` там, де викликаємо функцію `change`, і змінюємо сигнатуру функції, щоб вона приймала мутабельне посилання за допомогою `some_string: &mut String`. Це явно показує, що функція `change` змінить позичене значення.

Мутабельні посилання мають одне суттєве обмеження: якщо ви маєте мутабельне посилання на значення, то більше не можете мати інших посилань на це значення. Цей код, що намагається створити два мутабельні посилання на `s`, не спрацює:

Файл: src/main.rs

```
let mut s = String::from("hello");  
  
let r1 = &mut s;  
let r2 = &mut s;  
  
println!("{}", r1, r2);
```

Ось помилка:

```

$ cargo run
  Compiling ownership v0.1.0 (file:///projects/ownership)
error[E0499]: cannot borrow `s` as mutable more than once at a time
--> src/main.rs:5:14
|
4 |     let r1 = &mut s;
|               ----- first mutable borrow occurs here
5 |     let r2 = &mut s;
|               ^^^^^^^ second mutable borrow occurs here
6 |
7 |     println!("{}", {}, r1, r2);
|                       -- first borrow later used here

For more information about this error, try `rustc --explain E0499`.
error: could not compile `ownership` due to previous error

```

Помилка каже, що цей код некоректний, бо ми не можемо позичити `s` як мутабельне значення більш ніж один раз. Перше мутабельне позичання знаходиться в `r1` має існувати, доки не буде використане в `println!`, але між створенням цього мутабельного посилання і його використанням, ми намагалися створити ще одне мутабельне посилання в `r2`, що позичає ті ж дані, що й `r1`.

Це обмеження, що забороняє кілька мутабельних посилань на одні й ті самі дані в один час, дозволяє їх змінювати, але під пильним контролем. Це те, із чим борються початківці-растацеанці, бо більшість мов дозволяють вам змінювати дані коли завгодно. Перевага цього обмеження в тому, що Rust може запобігти гонитві даних під час компіляції. Гонитва даних подібна до стану гонитви й стається, коли мають місце такі три умови:

- Два чи більше вказівників мають доступ до одних даних у один і той самий час.
- Щонайменше один зі вказівників використовується для запису даних.
- Не застосовується жодних механізмів синхронізації доступу до даних.

Гонитви даних викликають невизначену поведінку і їх може бути складно діагностувати та виправляти, коли ви намагаєтесь відстежити їх під час роботи програми; Rust запобігає проблемі, відмовляючись компілювати код з гонитвою даних!

As always, we can use curly brackets to create a new scope, allowing for multiple mutable references, just not *simultaneous* ones:

```

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

{
    let r1 = &mut s;
} // r1 goes out of scope here, so we can make a new reference with no
problems.

let r2 = &mut s;

```

Rust застосовує схоже правило для змішування мутабельних і немутабельних посилань. Цей код призводить до помилки:

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

let r1 = &s; // no problem
let r2 = &s; // no problem
let r3 = &mut s; // BIG PROBLEM

println!("{}", {}, and {}", r1, r2, r3);
```

Ось помилка:

```
$ cargo run
   Compiling ownership v0.1.0 (file:///projects/ownership)
error[E0502]: cannot borrow `s` as mutable because it is also borrowed as
immutable
--> src/main.rs:6:14
4 |     let r1 = &s; // no problem
   |               -- immutable borrow occurs here
5 |     let r2 = &s; // no problem
6 |     let r3 = &mut s; // BIG PROBLEM
   |               ^^^^^^ mutable borrow occurs here
7 |
8 |     println!("{}", {}, and {}", r1, r2, r3);
   |                                   -- immutable borrow later used here

For more information about this error, try `rustc --explain E0502`.
error: could not compile `ownership` due to previous error
```

Хух! Не виходить *також* мати мутабельне посилання, коли в нас є немутабельне посилання на це ж значення.

Користувачі немутабельного посилання не очікують, що його значення несподівано зміниться прямо під час використання. Втім, багато немутабельних посилань допустимі, бо жоден з тих, хто просто читає дані, не може вплинути на те, що інші теж читають ці ж дані.

Зверніть увагу, що область видимості посилання починається з місця його проголошення і продовжується до останнього разу, коли посилання використовується. Наприклад, цей код компілюється тому, що останнє використання немутабельних посилань у `println!` відбувається до проголошення мутабельного посилання:

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

let r1 = &s; // no problem
let r2 = &s; // no problem
println!("{}", r1, r2);
// variables r1 and r2 will not be used after this point

let r3 = &mut s; // no problem
println!("{}", r3);
```

Області видимості немутабельних посилань `r1` і `r2` завершуються після `println!`, де вони востаннє використані, тобто перед створенням мутабельного посилання `r3`. Ці області видимості не перекриваються, тож цей код є коректним: компілятор може сказати, що посилання більше не використовується, навіть якщо це місце до завершення області видимості.

Хоча ці помилки часом і дратують, пам'ятайте, що це компілятор Rust вказує на потенційний баг завчасно (під час компіляції замість часу виконання) і точно вказує, де полягає проблема, замість змушувати вас відстежувати, чому іноді ваші дані не такі, як ви очікували.

Висячі Посилання

У мовах із вказівниками легко можна помилково створити *висячий вказівник* - вказівник, що посилається на місце в пам'яті, що було виділене комусь іще - звільнивши пам'ять, але залишивши вказівник на цю пам'ять. У Rust натомість компілятор гарантує, що посилання ніколи не стануть висячими: якщо ви маєте посилання на певні дані, компілятор пересвідчиться, що дані не вийдуть з області видимості раніше за посилання на ці дані.

Спробуймо створити висяче посилання, чому Rust запобіжить помилкою часу компіляції:

Файл: src/main.rs

```
fn main() {
    let reference_to_nothing = dangle();
}

fn dangle() -> &String {
    let s = String::from("hello");

    &s
}
```

Ось помилка:

```
$ cargo run
  Compiling ownership v0.1.0 (file:///projects/ownership)
error[E0106]: missing lifetime specifier
  --> src/main.rs:5:16
   |
5  | fn dangle() -> &String {
   |               ^ expected named lifetime parameter
   = help: this function's return type contains a borrowed value, but there is no value for it to be borrowed from
   help: consider using the `'static` lifetime
5  | fn dangle() -> &'static String {
   |               ++++++

For more information about this error, try `rustc --explain E0106`.
error: could not compile `ownership` due to previous error
```

Це повідомлення про помилку посилається на особливість, про яку ми ще не розповідали: час існування. Ми обговоримо часи існування детальніше у Розділі 10. Але, якщо опустити частини про час існування, повідомлення містить ключ до того, чому цей код містить проблему:

this function's return type contains a borrowed value, but there is no value for it to be borrowed from (тип, що повертає ця функція, містить позичене значення, але немає значення, яке воно може позичити)

Подивімося ближче, що саме відбувається на кожному кроці нашого коду `dangle`:

Файл: src/main.rs

```
fn dangle() -> &String { // dangle returns a reference to a String

    let s = String::from("hello"); // s is a new String

    &s // we return a reference to the String, s
} // Here, s goes out of scope, and is dropped. Its memory goes away.
// Danger!
```

Оскільки `s` було створено всередині `dangle`, коли код `dangle` завершується, `s` буде вивільнено. Але ми намагаємося повернути посилання на нього. Це означає, що це посилання буде вказувати на некоректний `String`. Так не можна! І Rust цього не допустить.

Рішення тут - повертати `String` безпосередньо:

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

Це працює без проблем. Володіння переміщується з функції, і нічого не звільняється.

Правила Посилань

Ще раз повторимо, що ми обговорили про посилання:

- У будь-який час можна мати *або* одне мутабельне посилання, *або* будь-яку кількість немутабельних посилань.
- Посилання завжди мають бути коректними.

Далі ми поглянемо на інший тип посилань: слайси.

Слайси

Слайси дозволяють вам посилатися на неперервні послідовності елементів у колекції замість усієї колекції. Слайс - це посилання, тому він не володіє даними.

Ось проста задача з програмування: написати функцію, що приймає стрічку зі слів, розділених пробілами, і повертає перше слово, яке знаходиться в цій стрічці. Якщо функція не знайде пробіл у стрічці, це означає, що вся стрічка є одним словом і, відтак, функція має повернути всю стрічку.

Let's work through how we'd write the signature of this function without using slices, to understand the problem that slices will solve:

```
fn first_word(s: &String) -> ?
```

Ця функція, `first_word`, приймає параметром `&String`. Нам не потрібне володіння, тому це нормально. Але що ми маємо повернути? У нас немає способу, що виразити *частину* стрічки. Однак ми можемо повернути індекс кінця слова, позначений пробілом. Спробуємо зробити це у Блоці коду 4-7.

Файл: src/main.rs

```
fn first_word(s: &String) -> usize {  
    let bytes = s.as_bytes();  
  
    for (i, &item) in bytes.iter().enumerate() {  
        if item == b' ' {  
            return i;  
        }  
    }  
  
    s.len()  
}
```

Listing 4-7: The `first_word` function that returns a byte index value into the `String` parameter

Оскільки нам треба пройти через `String` елемент за елементом і перевірити, чи значення є пробілом, ми перетворимо наш `String` на масив байтів за допомогою методу `as_bytes`.

```
let bytes = s.as_bytes();
```


Далі ми створюємо ітератор по масиву байтів за допомогою методу `iter`:

```
for (i, &item) in bytes.iter().enumerate() {
```

Ітератори будуть детальніше обговорені в [Розділі 13](#). Поки що достатньо знати, що `iter` - метод, що повертає кожен елемент у колекції, а метод `enumerate` обгортає результат `iter` у кортеж. Перший елемент кортежу, що його повертає `enumerate` - індекс, а другий - посилання на елемент. Це трохи зручніше, ніж обчислювати індекс самостійно.

Оскільки метод `enumerate` повертає кортеж, ми можемо скористатися шаблонами для деструктуризації цього кортежу. Ми ще будемо обговорювати шаблони в [Розділі 6](#). В циклі `for` ми визначаємо шаблон, що складається з індексу `i` і байту `&item` в кортежі. Оскільки ми отримуємо посилання на елемент від `.iter().enumerate()`, то використовуємо в шаблоні `&`.

У циклі `for` ми шукаємо байт, що представляє пробіл, за допомогою байтового літералу. Коли знаходимо пробіл, ми повертаємо його індекс. Якщо цього не сталося, повертаємо довжину стрічки за допомогою методу `s.len()`.

```
    if item == b' ' {
        return i;
    }
}

s.len()
```

Тепер ми маємо спосіб знайти індекс кінця першого слова у стрічці, але є проблема. Ми повертаємо одне значення `usize`, але це значення має сенс лише в контексті нашої стрічки `&String`. Іншими словами, оскільки це значення не пов'язане із зі стрічкою, немає гарантії, що воно буде коректним надалі. Розглянемо програму у Блоці коду 4-8, що використовує функцію `first_word` з Блоку коду 4-7.

Файл: `src/main.rs`

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

    let word = first_word(&s); // word will get the value 5

    s.clear(); // this empties the String, making it equal to ""

    // word still has the value 5 here, but there's no more string that
    // we could meaningfully use the value 5 with. word is now totally invalid!
}
```

Блок коду 4-8: Збереження результату виклику функції `first_word` і наступна зміна вмісту стрічки

Ця програма компілюється без помилок, і також скомпілювалася б, якби ви використали `word` після виклику `s.clear()`. `word` ніяк не пов'язане зі станом `s`, і тому `word` міститиме значення `5`. Ми можемо використати це значення `5` зі змінною `s`, щоб спробувати видобути з неї перше слово, але це буде помилкою, бо вміст `s` змінився відколи ми зберегли `5` до `word`.

Необхідність дбати про актуальність індексу в `word` відносно даних в `s` нудна і може спровокувати помилки! Керування такими індексами стає ще більш ламким, якщо ми напишемо функцію `second_word`. Її сигнатура буде виглядати так:

```
fn second_word(s: &String) -> (usize, usize) {
```

Тепер ми відстежуємо початковий і кінцевий індекси, і ми маємо ще більше значень, обчислених з даних у конкретному стані, але ніяк не прив'язаних до цього стану. Тепер ми маємо три непов'язані змінні, підвішені в повітрі, які нам треба тримати синхронізованими.

На щастя, у Rust є розв'язання цієї проблеми: стрічкові слайси.

Стрічкові Слайси

Стрічковий слайс - це посилання на частину `String`, і виглядає він так:

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

let hello = &s[0..5];
let world = &s[6..11];
```

Замість того, щоб посилатися на всю `String`, `hello` посилається на частину `String`, указану у фрагменті `[0..5]`. Ми створюємо слайси за допомогою діапазону з квадратними дужками, вказуючи `[starting_index..ending_index]`, де `starting_index` - це перша позиція в слайсі, а `ending_index` - позиція на одну більша за останню позицію в слайсі. В середині структура даних слайсу насправді зберігає початкову позицію і довжину слайсу, що відповідає `ending_index` мінус `starting_index`. Тому в прикладі `let world = &s[6..11];`, `world` буде слайсом, що складається зі вказівника на байт з індексом 6 у `s` і довжини `5`.

Рисунок 4-6 показує це у формі діаграми.

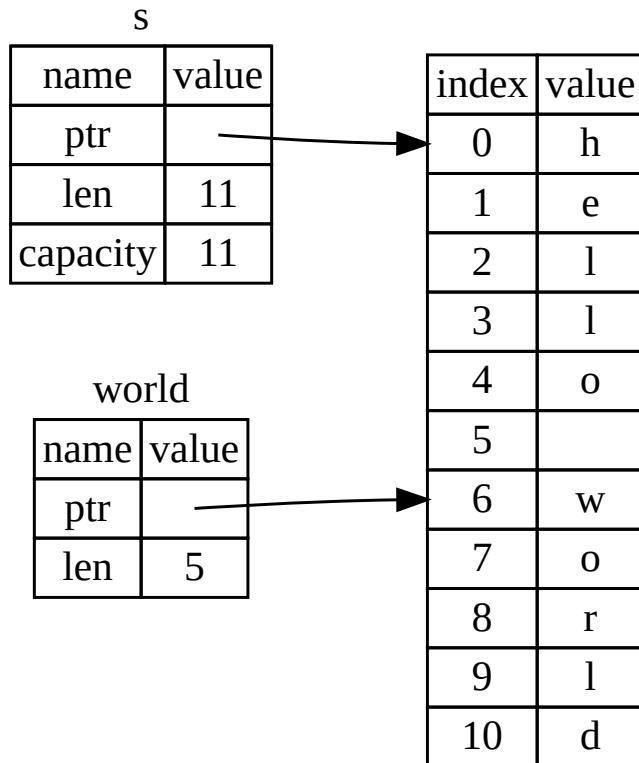


Рисунок 4-6: стрічковий слайс, що посилається на частину `String`

Синтаксис діапазонів `..` у Rust дозволяє, якщо ви хочете почати слайс на індексі 0, пропустити значення перед крапками. Іншими словами, ці рядки тотожні:

```
let s = String::from("hello");
let slice = &s[0..2];
let slice = &s[..2];
```

Так само, якщо ваш слайс включає останній байт `String`, ви можете пропустити останнє число. Таким чином, ці рядки також тотожні:

```
let s = String::from("hello");
let len = s.len();
let slice = &s[3..len];
let slice = &s[3..];
```

Також можна пропустити обидва значення, щоб взяти слайс з усієї стрічки. Це також тотожні рядки. Це також тотожні рядки:

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

let len = s.len();

let slice = &s[0..len];
let slice = &s[..];
```

Примітка: Індеси діапазону слайсу стрічки мають бути коректними границями символів UTF-8. Якщо ви спробуєте створити слайс стрічки посеред багатобайтового символу, ваша програма завершиться з помилкою. Заради ознайомлення зі слайсами стрічок, ми припускаємо в цьому розділі, що стрічка буде складатися лише з ASCII; ретельніше обговорення обробки UTF-8 міститься в підрозділі [“Зберігання тексту, кодованого в UTF-8, у стрічках”](#) Розділу 8.

Враховуючи всю цю інформацію, перепишімо `first_word`, щоб вона повертала слайс. Тип, що позначає слайс стрічки, записується як `&str`:

Файл: src/main.rs

```
fn first_word(s: &String) -> &str {
    let bytes = s.as_bytes();

    for (i, &item) in bytes.iter().enumerate() {
        if item == b' ' {
            return &s[0..i];
        }
    }

    &s[..]
}
```

Ми отримуємо індекс кінця слова тим же чином, що й у Блоці коду 4-7, пошуком першого стрічного пробілу. Коли ми знаходимо пробіл, ми повертаємо слайс стрічки за допомогою початку стрічки та індексу пробілу як початкового і кінцевого індексів.

Тепер при виклику `first_word` ми отримаємо одне значення, пов'язане з даними. Це значення складається з посилання на початкову точку слайсу і кількість елементів у ньому.

Повернення слайсу також спрацює для функції `second_word`:

```
fn second_word(s: &String) -> &str {
```

Тепер ми маємо нехитрий API, з яким значно складніше потрапити в халепу, оскільки компілятор забезпечить коректність посилань на `String`. Пам'ятаєте помилку в програмі з Блоці коду 4-8, коли ми мали індекс кінця першого слова, але очистили стрічку, чим зробили наш індекс некоректним? Цей код мав логічну помилку, але не призводив до жодних негайних помилок. Проблеми з'явилися б надалі, якби ми спробували використовувати індекс першого слова з порожньою стрічкою. Слайси унеможливають цю помилку і дають знати про проблему в коді значно раніше. Використання слайсової версії `first_word` призведе до помилки під час компіляції:

Файл: `src/main.rs`

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

    let word = first_word(&s);

    s.clear(); // error!

    println!("the first word is: {}", word);
}
```

Ось помилка компілятора:

```
$ cargo run
Compiling ownership v0.1.0 (file:///projects/ownership)
error[E0502]: cannot borrow `s` as mutable because it is also borrowed as
immutable
--> src/main.rs:18:5
16 |     let word = first_word(&s);
   |                          -- immutable borrow occurs here
17 |
18 |     s.clear(); // error!
   |     ^^^^^^^^^ mutable borrow occurs here
19 |
20 |     println!("the first word is: {}", word);
   |                                           ---- immutable borrow later used here

For more information about this error, try `rustc --explain E0502`.
error: could not compile `ownership` due to previous error
```

Пригадаємо, що за правилами позичання, якщо ми маємо немутабельне посилання на щось, ми не можемо робити мутабельне посилання на це ж. Оскільки `clear` має скоротити `String`, він намагається взяти мутабельне посилання. `println!` після виклику `clear` використовує посилання в `word`, так що немутабельне посилання все ще має бути активним в цій точці. Rust забороняє водночас мутабельне посилання в `clear` і

немутабельне посилання у `word`, і компіляція зазнає невдачі. Rust не тільки робить наш API простішим у використанні, а ще й усуває під час компіляції цілий клас помилок!

Стрічкові Літерали як Слайси

Згадайте, що ми говорили про стрічкові літерали, збережені у двійковому файлі. Оскільки тепер ми вже знаємо про слайси, ми можемо як слід зрозуміти стрічкові літерали:

```
let s = "Hello, world!";
```

Типом `s` є `&str`: це слайс, що вказує на конкретне місце у двійковому файлі. Це також є причиною, чому стрічкові літерали є немутабельними; `&str` - немутабельне посиланням.

Стрічкові Слайси як Параметри

Знання того, що можна брати слайси як літералів, так і `String` веде нас до ще одного поліпшення `first_word`. Її сигнатура наразі така:

```
fn first_word(s: &String) -> &str {
```

Більш досвідчений растеаєанин напише замість цього такий рядок, бо він дозволяє нам використовувати одну й ту саму функцію і для `String` і для `&str`:

```
fn first_word(s: &str) -> &str {
```

Блок коду 4-9: поліпшення функції `first_word`, використовуючи слайс стрічки як тип параметра `s`

Якщо у нас є слайс стрічки, ми можемо передати його прямо. Якщо у нас є `String`, ми можемо передати слайс цього `String` чи посилання на `String`. Ця гнучкість є можливою завдяки *приведенню при розіменуванні*, особливості, про яку ми розкажемо в підрозділі “Неявні приведення при розіменуваннях у функціях та методах” Розділу 15.

Визначення функції, що приймає слайс стрічки замість посилання на `String` робить наш API більш загальним і корисним без втрати функціональності:

Файл: `src/main.rs`

```
fn main() {  
    let my_string = String::from("hello world");  
  
    // `first_word` works on slices of `String`s, whether partial or whole  
    let word = first_word(&my_string[0..6]);  
    let word = first_word(&my_string[..]);  
    // `first_word` also works on references to `String`s, which are equivalent  
    // to whole slices of `String`s  
    let word = first_word(&my_string);  
  
    let my_string_literal = "hello world";  
  
    // `first_word` works on slices of string literals, whether partial or whole  
    let word = first_word(&my_string_literal[0..6]);  
    let word = first_word(&my_string_literal[..]);  
  
    // Because string literals are string slices already,  
    // this works too, without the slice syntax!  
    let word = first_word(my_string_literal);  
}
```

Інші Слайси

Слайси стрічок, як можна зрозуміти, пов'язані зі стрічками. Але є також і більш загальний тип слайсів. Розгляньмо такий масив:

```
let a = [1, 2, 3, 4, 5];
```

Так само як ми можемо захотіти послатися на частину стрічки, ми можемо захотіти послатися на частину масиву. Це робиться так:

```
let a = [1, 2, 3, 4, 5];  
  
let slice = &a[1..3];  
  
assert_eq!(slice, &[2, 3]);
```

Цей слайс має тип `&[i32]`. Він працює тим же чином, що й слайси стрічок, зберігаючи посилання на перший елемент і довжину. Цей тип слайсів можна використовувати для всіх інших видів колекцій. Ми поговоримо про ці колекції детальніше, коли будемо обговорювати вектори в Розділі 8.

Підсумок

Концепції власності, позичання і слайсів - це те, що гарантує безпеку роботи із пам'яттю в програмах на Rust під час компіляції. Мова Rust надає вам контроль над використанням пам'яті так само як і інші системні мови програмування, але те, що наявність власника даних автоматично призводить до очищення даних, коли власник виходить з області видимості, означає, що вам не треба писати і зневаджувати додатковий код, щоб отримати цей контроль.

Власність впливає на те, як працює велика кількість інших частин Rust, тому ми говоритимемо про ці концепції й надалі у цій книзі. Перейдімо далі до наступного розділу і погляньмо на групування окремих даних до купи в структури `struct`.

Використання Структур для Групування Пов'язаних Даних

struct, або *структура* - це спеціальний тип даних, що дозволяє вам збирати в одну сутність зі спільною назвою декілька пов'язаних значень, що складають значущу групу. Якщо ви знайомі з якоюсь об'єктноорієнтованою мовою, *struct* - це як атрибути даних об'єкта. У цьому розділі ми порівняємо і покажемо різницю між кортежами та структурами, щоб було на що спертися у навчанні й продемонструємо, коли структури є кращим засобом для об'єднання даних.

Ми продемонструємо, як визначати і створювати структури. Ми обговоримо, як визначати асоційовані функції, особливо вид асоційованих функцій, що зветься *методами*, для визначення поведінки, пов'язаної із типом структури. Структури та енуми (про які йдеться в Розділі 6) є будівельними блоками для створення нових типів у вашій програмі, які дозволяють по повній скористатись перевіркою типів часу компіляції Rust.

Визначення та Створення Екземпляра Структури

Структури подібні до кортежів, про які ми говорили в підрозділі “Тип кортеж” Розділу 3, бо обидва складаються з кількох пов'язаних значень. Як і у кортежах, частини структур можуть бути різних типів. На відміну від кортежів, у структурі ви називаєте кожен елемент даних, щоб було зрозуміло, що ці значення означають. Завдяки цим іменам структури гнучкіші за кортежі: ви не мусите покладатися на порядок даних, щоб визначати чи отримувати доступ до значень екземпляра.

Для визначення структури, ми вводимо ключове слово `struct` і називаємо всю структуру. Ім'я структури має описувати сенс групування цих елементів даних. Потім, у фігурних дужках, ми визначаємо імена і типи елементів даних, які зветься *полями*. Наприклад, Блок коду 5-1 показує структуру, що зберігає інформацію про обліковий запис користувача.

Файл: src/main.rs

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

Блок коду 5-1: Визначення структури `User`

Щоб скористатися структурою по визначенню, ми створюємо *екземпляр* цієї структури, визначаючи конкретні значення для кожного поля. Ми створюємо екземпляр, вказуючи назву структури, а потім додаємо фігурні дужки, що містять пари *ключ: значення*, де ключі - це імена полів, а значення - дані, які ми хочемо зберігати в цих полях. Поля не обов'язково вказувати у тому ж порядку, в якому вони були проголошені в структурі. Іншими словами, визначення структури - це загальний шаблон типу, а екземпляри заповнюють цей шаблон конкретними даними, щоб створити значення цього типу. Наприклад, ми можемо проголосити конкретного користувача, як показано в Блоці коду 5-2.

Файл: src/main.rs

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

Listing 5-2: Creating an instance of the `User` struct

Щоб отримати конкретне значення зі структури, використовують записом через точку. Якщо ми хочемо отримати адресу електронної пошти користувача, ми можемо написати `user1.email`. Якщо екземпляр є мутабельним, ми можемо змінити значення за допомогою запису через точку і присвоюванням конкретному полю. Блок коду 5-3 показує, як змінити значення поля `email` мутабельного екземпляра `User`.

Файл: `src/main.rs`

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

Блок коду 5-3: Зміна значення поля `email` екземпляру `User`

Зверніть увагу, що мутабельним має бути весь екземпляр; Rust не дозволяє позначати лише окремі поля як мутабельні. Як і з будь-яким виразом, ми можемо написати новий екземпляр останнім виразом у тілі функції, щоб неявно повернути цей новий екземпляр.

Блок коду 5-4 демонструє функцію `build_user`, що повертає екземпляр `User` зі встановленими адресою та ім'ям. Поле `active` отримує значення `true`, а `sign_in_count` - значення `1`.

Файл: `src/main.rs`

```
fn build_user(email: String, username: String) -> User {  
    User {  
        active: true,  
        username: username,  
        email: email,  
        sign_in_count: 1,  
    }  
}
```

Listing 5-4: A `build_user` function that takes an email and username and returns a `User` instance

Має сенс називати аргументи такої функції тими ж іменами, що й імена відповідних полів структури, але необхідність повторювати імена полів `email` та `username` утомлює. Якщо у структурі більше полів, повторення кожного імені дратує ще більше. На щастя, є зручне скорочення!

Скорочена Ініціалізація Полів

Оскільки назви параметрів та полів структури є абсолютно однаковими у Блоці коду 5-4, ми можемо використати синтаксис *скороченої ініціалізації полів* для переписування `build_user` так, щоб він поведився точно так само, але не містив повторення `username` і `email`, як показано у Блоці коду 5-5.

Файл: `src/main.rs`

```
fn build_user(email: String, username: String) -> User {  
    User {  
        active: true,  
        username,  
        email,  
        sign_in_count: 1,  
    }  
}
```

Блок коду 5-5: функція `build_user`, що використовує скорочену ініціалізацію полів, бо параметри `username` і `email` мають такі самі назви, як і поля структури

Ми створюємо новий екземпляр структури `User`, яка має поле з назвою `email`. Ми хочемо встановити значення поля `email` у значення параметра `email` функції `build_user`. Оскільки поле `email` і параметр `email` мають одну назву, можна писати скорочено `email` замість `email: email`.

Створення Екземплярів Структур з Інших Екземплярів Використовуючи Синтаксис Оновлення

Часто буває корисним створити новий екземпляр структури, що бере більшу частину даних з екземпляра, що вже існує, проте деякі змінює. Ви можете зробити так за допомогою *синтаксису оновлення структури*.

Для початку, Блок коду 5-6 показує, як створити новий екземпляр `User`, що зветься `user2`, без синтаксису оновлення. Ми виставляємо нове значення поля `email`, проте решта полів використовує значення зі структури `user1`, створеної у Блоці коду 5-2.

Файл: `src/main.rs`

```
fn main() {  
    // --snip--  
  
    let user2 = User {  
        active: user1.active,  
        username: user1.username,  
        email: String::from("another@example.com"),  
        sign_in_count: user1.sign_in_count,  
    };  
}
```

Блок коду 5-6: Створення нового екземпляру `User` з деякими значеннями з `user1`

Синтаксис оновлення структури дає той самий результат із меншою кількістю коду, як показано у Блоці коду 5-7. Запис `..` позначає, що решта полів, що їх не було явно виставлено, отримують ті значення, що були в заданому екземплярі.

Файл: `src/main.rs`

```
fn main() {  
    // --snip--  
  
    let user2 = User {  
        email: String::from("another@example.com"),  
        ..user1  
    };  
}
```

Блок коду 5-7: використання синтаксису оновлення структури для встановлення нового значення `email` у екземплярі `User`, але використовуючи решту значень з `user1`

Код у Блоці коду 5-7 також створює екземпляр `user2`, що має відмінне значення `email`, але має ті ж значення `username`, `active` та `sign_in_count`, що й `user1`. Запис `..user1` має бути останнім, щоб позначити, що решта полів отримують значення з відповідних

полів у `user1`, але ми можемо зазначати значення для будь-якої кількості полів у будь-якому порядку, без урахування того, як вони йдуть у визначенні структури.

Зверніть увагу, що синтаксис оновлення структури використовує `=`, як при присвоєнні; це тому, що він переміщує дані, як показано в підрозділі "Способи взаємодії змінних і даних: переміщення". У цьому прикладі, ми більше не зможемо використовувати `user1` після створення `user2`, бо `String` з поля `username` структури `user1` було переміщено у `user2`. Якби ми надали `user2` нові значення типу `String` для обох `email` і `username` і таким чином використали тільки значення `active` і `sign_in_count` з `user1`, тоді `user1` все ще був би коректним після створення `user2`. Типи полів `active` і `sign_in_count` реалізують трейт `Copy`, тому застосовується поведінка, яку ми обговорювали в підрозділі "Дані в стеку: копіювання".

Використання Структур-Кортежів без Названих Полів для Створення Нових Типів

Rust також підтримує структури, які виглядають схожими на кортежі, що зветься *структури-кортежі* (tuple struct). Структури-кортежі надають значення структурі, бо мають назву, але не мають назв полів, тільки типи. Структури-кортежі корисні, коли ви хочете дати кортежу ім'я і зробити кортеж окремим типом, але називати кожне поле, як у звичайній структурі, буде надто багатослівним чи надмірним.

Щоб визначити структуру-кортеж, треба вказати ключове слово `struct` і ім'я структури, а потім типи в кортежі. Наприклад, ось визначення і приклади застосування двох структур-кортежів, що зветься `Color` і `Point`:

Файл: `src/main.rs`

```
struct Color(i32, i32, i32);
struct Point(i32, i32, i32);

fn main() {
    let black = Color(0, 0, 0);
    let origin = Point(0, 0, 0);
}
```

Зауважте, що значення `black` та `origin` мають різні типи, бо вони є екземплярами різних структур-кортежів. Кожна визначена нами структура має свій власний тип, навіть якщо поля структур мають однакові типи. Наприклад, функція, що приймає параметр типу `Color`, не може прийняти аргументом `Point`, хоча обидва типи складаються з трьох значень `i32`. В іншому ж структури-кортежі поведуться як кортежі: ви можете

деструктуризувати їх на окремі шматки, і ви можете використовувати `.` з індексом, щоб отримати доступ до окремого значення.

Одиничні Структури без Полів

Також можна визначати структури без жодних полів! Вони зветься *одинично-подібні структури* (unit-like struct), бо поводяться аналогічно до `()`, одиничного типу, згаданого в підрозділі “Тип кортеж”. Одинично-подібні структури можуть бути корисними в ситуаціях, коли вам потрібно реалізувати трейт на якомусь типі, але у вас немає потреби зберігати якісь дані в цьому типі. Про трейти ми поговоримо в Розділі 10. Ось приклад проголошення та створення одиничної структури під назвою `AlwaysEqual`:

Файл: src/main.rs

```
struct AlwaysEqual;

fn main() {
    let subject = AlwaysEqual;
}
```

Щоб визначити `AlwaysEqual`, ми використовуємо ключове слово `struct`, назву структури та крапку з комою. Дужки не потрібні - ані звичайні, ані фігурні! Тоді ми можемо створити екземпляр з `AlwaysEqual` у змінній `subject` у схожий спосіб: використовуючи ім'я, яке ми визначили, без будь-яких - звичайних чи фігурних - дужок. Уявімо, що згодом ми реалізуємо поведінку для цього типу, так що кожен екземпляр `AlwaysEqual` завжди дорівнює будь-якому екземпляру будь-якого іншого типу, скажімо, щоб мати завжди відомий результат для тестування. Нам не потрібні жодні дані для реалізації такої поведінки! У Розділі 10 ви побачите, як визначити трейти і реалізувати їх на будь-якому типі, включно з одинично-подібними структурами.

Володіння Даними Структури

У структурі `User` з Блоку коду 5-1 ми використовували тип `String`, що має володіння, а не стрічковий слайс `&str`. Це свідомий вибір, бо ми хочемо, щоб екземпляри цієї структури володіли всіма своїми даними і щоб ці дані були коректними, поки структури в цілому коректна.

Структура також може зберігати посилання на дані, якими володіє хтось інший, але це потребує використання *часу життя*, особливості Rust, що обговорюється у Розділі 10. Час життя гарантує, що дані, на які посилається структура, будуть

коректними весь час існування структури. Наприклад, якщо ви спробуєте зберегти посилання у структурі без уточнення часу життя, ось так, то дістанете помилку:

Файл: src/main.rs

```
struct User {  
    active: bool,  
    username: &str,  
    email: &str,  
    sign_in_count: u64,  
}  
  
fn main() {  
    let user1 = User {  
        active: true,  
        username: "someusername123",  
        email: "someone@example.com",  
        sign_in_count: 1,  
    };  
}
```

Компілятор поскаржиться, що потрібно зазначити час існування:


```
$ cargo run
Compiling structs v0.1.0 (file:///projects/structs)
error[E0106]: missing lifetime specifier
--> src/main.rs:3:15
|
3 |     username: &str,
|               ^ expected named lifetime parameter
help: consider introducing a named lifetime parameter
|
1 ~ struct User<'a> {
2 |     active: bool,
3 ~     username: &'a str,
|

error[E0106]: missing lifetime specifier
--> src/main.rs:4:12
|
4 |     email: &str,
|           ^ expected named lifetime parameter
help: consider introducing a named lifetime parameter
|
1 ~ struct User<'a> {
2 |     active: bool,
3 |     username: &str,
4 ~     email: &'a str,
|

For more information about this error, try `rustc --explain E0106`.
error: could not compile `structs` due to 2 previous errors
```

Ми обговоримо, як це виправити, щоб можна було зберігати посилання у структурах, у Розділі 10, а поки що будемо виправляти подібні помилки за допомогою типів, що володіють своїми даними, на кшталт `String`, замість посилань на кшталт `&str`.

Приклад Програми з Використанням Структур

Щоб зрозуміти, де можна використовувати структури, напишімо програму, що обчислює площу прямокутника. Почнемо з окремих змінних, а потім рефакторизуємо її так, щоб вона використовувала структури.

За допомогою Cargo створімо двійковий проєкт програми, що зветься *rectangles*, яка прийматиме ширину і висоту прямокутника в пікселях і обчислюватиме його площу. Блок коду 5-8 показує коротку очевидну програму, що робить саме те, що треба, у *src/main.rs* нашого проєкту.

Файл: *src/main.rs*

```
fn main() {  
    let width1 = 30;  
    let height1 = 50;  
  
    println!(  
        "The area of the rectangle is {} square pixels.",  
        area(width1, height1)  
    );  
}  
  
fn area(width: u32, height: u32) -> u32 {  
    width * height  
}
```

Блок коду 5-8: Обчислення площі прямокутника, заданого шириною та висотою в окремих змінних

Тепер запусимо програму командою `cargo run`:

```
$ cargo run  
Compiling rectangles v0.1.0 (file:///projects/rectangles)  
Finished dev [unoptimized + debuginfo] target(s) in 0.42s  
Running `target/debug/rectangles`  
The area of the rectangle is 1500 square pixels.
```

This code succeeds in figuring out the area of the rectangle by calling the `area` function with each dimension, but we can do more to make this code clear and readable.

Проблема в цьому коді очевидна, якщо поглянути на сигнатуру функції `area`:

```
fn area(width: u32, height: u32) -> u32 {
```

Функція `area` має обчислювати площу одного прямокутника, але функція, яку ми написали, приймає два параметри, і з коду зовсім не ясно, що ці параметри пов'язані. Для кращої читаності та керованості буде краще згрупувати ширину і висоту разом. Ми вже обговорювали один зі способів, як це зробити, у підрозділі "Тип кортеж" Розділу 3: за допомогою кортежів.

Рефакторинг з Кортежами

Блок коду 5-9 показує версію нашої програми із кортежами.

Файл: `src/main.rs`

```
fn main() {  
    let rect1 = (30, 50);  
  
    println!(  
        "The area of the rectangle is {} square pixels.",  
        area(rect1)  
    );  
}  
  
fn area(dimensions: (u32, u32)) -> u32 {  
    dimensions.0 * dimensions.1  
}
```

Блок коду 5-9: Ширину та висоту прямокутника задані кортежем

З одного боку, ця програма краща. Кортежі додають трохи структурованості, і тепер ми передаємо лише один аргумент. Але з іншого боку ця версія менш зрозуміла: кортежі не мають назв для своїх елементів, тому тепер доводиться індексувати частини кортежу, що робить наші обчислення менш очевидними.

Не має значення, якщо ми переплутаємо ширину і висоту при обчисленні площі, але якщо ми захочемо намалювати прямокутник на екрані, це матиме значення! Нам доведеться пам'ятати, що `ширина` має індекс `0` у кортежі, а `висота` має індекс `1`. Для когось іще буде ще складніше розібратися в цьому і пам'ятати, якщо він буде використовувати наш код. Оскільки ми не показали сенс наших даних у коді, тепер легше припускатися помилок.

Рефакторинг зі Структурами: Додаємо Більшого Сенсу

Ми використовуємо структури, щоб додати сенс за допомогою "ярликів" до даних. Ми можемо перетворити наш кортеж на тип даних з іменами як для цілого, так і для частин, як показано в Блоці коду 5-10.

Файл: src/main.rs

```
struct Rectangle {  
    width: u32,  
    height: u32,  
}  
  
fn main() {  
    let rect1 = Rectangle {  
        width: 30,  
        height: 50,  
    };  
  
    println!(  
        "The area of the rectangle is {} square pixels.",  
        area(&rect1)  
    );  
}  
  
fn area(rectangle: &Rectangle) -> u32 {  
    rectangle.width * rectangle.height  
}
```

Блок коду 5-10: визначення структури `Rectangle`

Тут ми визначили структуру і назвали її `Rectangle`. Всередині фігурних дужок ми визначили поля `width` та `height`, обидва типу `u32`. Далі в `main` ми створюємо конкретний екземпляр `Rectangle` з шириною `30` і висотою `50`.

Наша функція `area` тепер має визначення з одним параметром, який ми назвали `rectangle`, тип якого - немутабельне позичення екземпляра структури `Rectangle`. Як ми вже казали в Розділі 4, ми можемо позичити структуру замість перебирати володіння ним. Таким чином `main` зберігає володіння і може продовжувати використовувати `rect1`, тому ми застосовуємо `&` у сигнатурі функції та при її виклику.

Функція `area` звертається до полів `width` та `height` екземпляру `Rectangle` (зверніть увагу, що доступ до полів позиченого екземпляру структури не переміщує значення полів, ось чому ви часто бачитимете позичання структур). Сигнатура функції `area` тепер каже саме те, що ми мали на увазі: обчислити площу `Rectangle` за допомогою полів `width` та `height`. Це сповіщає, що ширина і висота пов'язані одна з іншою, і дає змістовні імена значенням замість індексів кортежу `0` та `1`. Це вигравш для ясності.

Додаємо Корисний Функціонал Похідними Трейтами

Було б непогано мати змогу виводити екземпляр нашого `Rectangle` при зневадженні програми та бачити значення його полів. Блок коду 5-11 намагається вжити макрос `println!` так само як це було в попередніх розділах. Але цей код не працює.

Файл: `src/main.rs`

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

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

    println!("rect1 is {}", rect1);
}
```

Listing 5-11: Attempting to print a `Rectangle` instance

Якщо скомпілювати цей код, ми дістанемо помилку із головним повідомленням:

```
error[E0277]: `Rectangle` doesn't implement `std::fmt::Display`
```

Макрос `println!` може виконувати багато різних видів форматувань, і за замовчанням фігурні дужки кажуть `println!` використати форматування, відоме як `Display`: вивести те, що призначене для читання кінцевим споживачем. Примітивні типи, з яким ми досі стикалися, реалізують `Display` за замовчанням, оскільки є лише один спосіб, яким можна показати `1` чи якийсь інший примітивний тип користувачу. Але зі структурами вже не настільки очевидно, як `println!` має формувати вивід, оскільки є багато можливостей виведення: потрібні коми чи ні? Чи треба виводити фігурні дужки? Чи всі поля слід показувати? Через цю невизначеність, Rust не намагається відгадати, чого ми хочемо, і структури не мають підготовленої реалізації `Display`, яку можна було б використати у `println!` за допомогою заовнювача `{}`.

Якщо ми подивимося помилки далі, то знайдемо цю корисну примітку:

```
= help: the trait `std::fmt::Display` is not implemented for `Rectangle`
= note: in format strings you may be able to use `{:?}` (or `{:#?}` for pretty-print) instead
```

Спробуймо це зробити! Виклик макросу `println!` тепер виглядає так: `println!("rect1 = {:?}", rect1);`. Додавання специфікатора `?:` у фігурні дужки каже `println!`, що ми хочемо використати формат виведення, що зветься `Debug`. Трейт `Debug` дозволяє вивести нашу структуру у спосіб, зручний для розробників, щоб дивитися її значення під час зневадження коду.

Скомпілюймо змінений код. Трясця! Все одно помилка:

```
error[E0277]: `Rectangle` doesn't implement `Debug`
```

Але знову компілятор дає нам корисну примітку:

```
= help: the trait `Debug` is not implemented for `Rectangle`
= note: add `#[derive(Debug)]` to `Rectangle` or manually `impl Debug for Rectangle`
```

Rust має функціонал для виведення інформації для зневадження, та нам доведеться увімкнути його у явний спосіб, щоб зробити доступним для нашої структури. Щоб зробити це, додамо зовнішній атрибут `#[derive(Debug)]` прямо перед визначенням структури, як показано в Блоці коду 5-12.

Файл: `src/main.rs`

```
#[derive(Debug)]
struct Rectangle {
    width: u32,
    height: u32,
}

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

    println!("rect1 is {:?}", rect1);
}
```

Блок коду 5-12: Додавання атрибута для успадкування трейта `Debug` і виведення екземпляра `Rectangle` за допомогою форматування для зневадження

Коли ми знову запустимо програму, то не отримаємо жодної помилки й побачимо таке:

```
$ cargo run
Compiling rectangles v0.1.0 (file:///projects/rectangles)
Finished dev [unoptimized + debuginfo] target(s) in 0.48s
Running `target/debug/rectangles`
rect1 is Rectangle { width: 30, height: 50 }
```

Чудово! Це не найкрасивіший вивід, але він показує значення всіх полів цього екземпляру, що точно допоможе при зневадженні. Коли у нас будуть більші структури, корисно мати зручніший для читання вивід; в цих випадках, ми можемо використати `{:#?}` замість `{:?}` у стрічці `println!`. Якщо скористатися стилем `{:#?}` у цьому прикладі, вивід виглядатиме так:

```
$ cargo run
Compiling rectangles v0.1.0 (file:///projects/rectangles)
Finished dev [unoptimized + debuginfo] target(s) in 0.48s
Running `target/debug/rectangles`
rect1 is Rectangle {
  width: 30,
  height: 50,
}
```

Інший спосіб вивести значення у форматі `Debug` - скористатися макросом `dbg!`, який перебирає володіння виразом (на відміну від `println!`, що приймає посилання), виводить файл і номер рядка, де був у вашому коді викликаний макрос `dbg!` і обчислене значення виразу, і повертає володіння значенням.

Примітка: виклик макроса `dbg!` виводить до стандартного потоку помилок у консолі (`stderr`), на відміну від `println!`, що виводить до стандартного потоку виводу консолі (`stdout`). Ми поговоримо більше про `stderr` і `stdout` у підрозділі "Виведення повідомлень про помилки до стандартного потоку помилок замість стандартного вихідного потоку" Розділу 12.

Here's an example where we're interested in the value that gets assigned to the `width` field, as well as the value of the whole struct in `rect1`:

```
#[derive(Debug)]
struct Rectangle {
    width: u32,
    height: u32,
}

fn main() {
    let scale = 2;
    let rect1 = Rectangle {
        width: dbg!(30 * scale),
        height: 50,
    };

    dbg!(&rect1);
}
```

Ми можемо написати `dbg!` навколо виразу `30 * scale` і, оскільки `dbg!` повертає володіння виразу, поле `width` отримає це саме значення, як ніби й не було виклику `dbg!`. Ми не хочемо, щоб `dbg!` перебирав володіння `rect1`, так що ми використовуємо посилання на `rect1` при наступному виклику. Ось як виглядає те, що виводить цей приклад:

```
$ cargo run
  Compiling rectangles v0.1.0 (file:///projects/rectangles)
  Finished dev [unoptimized + debuginfo] target(s) in 0.61s
  Running `target/debug/rectangles`
[src/main.rs:10] 30 * scale = 60
[src/main.rs:14] &rect1 = Rectangle {
    width: 60,
    height: 50,
}
```

Ми бачимо що перший фрагмент був виведений з рядка 10 `src/main.rs`, де ми зневаджуємо вираз `30 * scale`, а його обчислене значення `60` (реалізація форматування `Debug` для цілих чисел виводить самі їхні значення). Виклик `dbg!` у рядку 14 `src/main.rs` виводить значення `&rect1`, яке дорівнює структурі `Rectangle`. Виведення використовує покращене форматування `Debug` для типу `Rectangle`. Макрос `dbg!` може бути дійсно корисним, коли ви намагаєтеся розібратися, що робить ваш код!

На додачу до трейту `Debug`, Rust надає нам ряд трейтів, що можна використовувати з атрибутом `derive`, які можуть додати корисну поведінку до наших власних типів. Ці трейти та їхня поведінка перераховані в [Додатку С](#). Ми розглянемо, як реалізувати ці трейти з кастомізованою поведінкою і як створювати свої власні трейти в Розділі 10. Також існує багато атрибутів, відмінних від

`derive`; для отримання додаткової інформації дивіться [розділ "Атрибути" Довідника Rust](#).

Функція `area` дуже конкретна: вона розраховує лише площу прямокутників. Було б корисно прив'язати цю поведінку до нашої структури `Rectangle`, оскільки вона не буде працювати з жодним іншим типом. Подивімося, як ми можемо продовжувати рефакторизовувати цей код, перетворивши функцію `area` на *метод* `area`, визначений на нашому типі `Rectangle`.

Синтаксис Методів

Методи подібні до функцій: вони проголошуються ключовим словом `fn` і іменем, можуть мати параметри та повертати значення, і містять код, що виконується, коли їх викликають з іншого місця. На відміну від функцій, методи визначаються в контексті структури (або енума чи трейтового об'єкта, про які йтиметься в [Розділі 6](#) і [Розділі 17](#), відповідно), і їхній перший параметр - це завжди `self`, який представляє екземпляр структури, для якого викликається метод.

Визначення Методів

Let's change the `area` function that has a `Rectangle` instance as a parameter and instead make an `area` method defined on the `Rectangle` struct, as shown in Listing 5-13.

Файл: `src/main.rs`

```
#[derive(Debug)]
struct Rectangle {
    width: u32,
    height: u32,
}

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

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

    println!(
        "The area of the rectangle is {} square pixels.",
        rect1.area()
    );
}
```

Listing 5-13: Defining an `area` method on the `Rectangle` struct

Щоб визначити функцію в контексті `Rectangle`, ми починаємо блок `impl` (від implementation, "реалізація") для `Rectangle`. Все в цьому блоці `impl` буде пов'язано з типом `Rectangle`. Потім ми переносимо функцію `area` до фігурних дужок після `impl` і замінюємо перший (а в цьому випадку єдиний) параметр на `self` у сигнатурі та повсюди в тілі. У `main`, де ми викликали функцію `area` і передавали аргументом `rect1`, тепер використаємо *синтаксис виклику метода*, щоб викликати метод `area` нашого екземпляра `Rectangle`. Синтаксис виклику методу записується після екземпляру: ми додаємо крапку, за якою - ім'я методу, дужки, і параметри, якщо такі є.

У сигнатурі `area` ми використовуємо `&self` замість `rectangle: &Rectangle`. `&self` є насправді скороченням для `self: &Self`. У середині блоку `impl` тип `Self` є псевдонімом для типу, для якого призначено цей блок `impl`. Методи мусять мати перший параметр на ім'я `self` типу `Self`, тому Rust дозволяє вам скоротити це до лише імені `self` на місці першого параметра. Зверніть увагу, що нам все ще потрібно використовувати `&` перед скороченням `self`, щоб вказати, що цей метод позичає екземпляр `Self`, так само як ми це зробили в `rectangle: &Rectangle`. Методи можуть перебирати володіння над `self`, позичати `self` немутабельно, як у цьому випадку, чи позичати `self` мутабельно, як і будь-який інший параметр.

Ми обрали `&self` з тих самих причин, що й `&Rectangle` у версії з функцією: ми не хочемо брати володіння, ми хочемо просто читати дані структури, не писати їх. Якби ми хотіли змінити екземпляр, для якого викликали метод, десь у методі, то перший параметр мав би бути `&mut self`. Методи, що беруть володіння над екземпляром за допомогою просто `self`, зустрічаються нечасто; ця техніка зазвичай використовується, коли метод перетворює `self` у щось інше і ми не хочемо, щоб оригінальний екземпляр використовувався після трансформації.

Основна перевага використання методів замість функцій, окрім використання синтаксису виклику метода та відсутності необхідності повторювати тип `self` у сигнатурі кожного метода - це організація коду. Ми збираємо все, що ми можемо зробити з екземпляром типа, в один блок `impl`, не примушуючи майбутніх користувачів нашого коду шукати можливостей використання `Rectangle` у різних місцях у нашій бібліотеці.

Зверніть увагу, що ми можемо вирішити назвати метод так само як зветься одне з полів структури. Наприклад, ми можемо визначити метод `Rectangle`, що також зватиметься `width`:

Файл: `src/main.rs`

```
impl Rectangle {
    fn width(&self) -> bool {
        self.width > 0
    }
}

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

    if rect1.width() {
        println!("The rectangle has a nonzero width; it is {} ", rect1.width);
    }
}
```

Тут ми вирішили, що метод `width` має повертати `true`, якщо значення у полі екземпляра `width` більше за `0`, і `false`, якщо його значення `0`: ми можемо як завгодно використати поле в методі з тою самою назвою. У `main`, коли ми пишемо `rect1.width` з дужками, Rust знає, що ми маємо на увазі метод `width`. Коли ми не використовуємо дужки, Rust знає, що ми маємо на увазі поле `width`.

Часто, але не завжди, коли ми даємо методам ім'я, що має поле, ми хочемо, щоб цей метод лише повертав значення поля і більше нічого не робив. Такі методи називаються *геттерами*, і Rust не реалізує їх автоматично для полів структур, як деякі інші мови. Геттери є корисними, бо дозволяють зробити поле приватним, а метод публічним, і таким чином уможливити доступ лише для читання як частину публічного API цього типу. Ми поговоримо про публічне та приватне і як визначити поле чи метод публічним чи приватним у [Розділі 7](#).

А Де Оператор `->`?

У C та C++ використовуються два різні оператори для виклику методів: `.`, якщо метод викликається для об'єкта безпосередньо, і `->`, якщо ви викликаєте метод для вказівника на об'єкт і спершу вказівник слід розіменувати. Іншими словами, якщо `object` - це вказівник, то `object->something()` робить те саме, що й `(*object).something()`.

Rust не має еквівалента оператора `->`; натомість, Rust має особливість, що зветься *автоматичне посилання і розіменування* (automatic referencing and dereferencing). Виклик методів - це одне з небагатьох місць у Rust з такою поведінкою.

Ось як це працює: коли ви викликаєте метод з `object.something()`, Rust автоматично додає `&`, `&mut`, або `*`, щоб `object` відповідав сигнатурі методу. Іншими словами, наступними вирази означають одне й те саме:

```
p1.distance(&p2);
(&p1).distance(&p2);
```

Але перший вираз є значно яснішим. Ці автоматичні посилання працюють, бо методи мають чітко заданого отримувача - тип `self`. Знаючи отримувача і назву метода, Rust може однозначно з'ясувати, чи цей метод для читання (`&self`), змін (`&mut self`) чи поглинання (`self`). Те, що Rust робить позичання неявним для отримувача метода є суттєвою частиною того, що робить володіння ергономічним на практиці.

Методи з Більшою Кількістю Параметрів

Попрактикуймося використовувати методи, створивши другий метод для структури `Rectangle`. Цього разу ми хочемо, щоб екземпляр `Rectangle` прийняв інший екземпляр `Rectangle` і повернув `true`, якщо другий `Rectangle` може повністю поміститися в межах `self` (першого `Rectangle`); інакше він повинен повернути `false`. Тобто після визначення метода `can_hold`, ми хочемо мати можливість написати програму, показану в Блоці коду 5-14.

Файл: `src/main.rs`

```
fn main() {
    let rect1 = Rectangle {
        width: 30,
        height: 50,
    };
    let rect2 = Rectangle {
        width: 10,
        height: 40,
    };
    let rect3 = Rectangle {
        width: 60,
        height: 45,
    };

    println!("Can rect1 hold rect2? {}", rect1.can_hold(&rect2));
    println!("Can rect1 hold rect3? {}", rect1.can_hold(&rect3));
}
```

Роздрук 5-14: Демонстрація використання поки що ненаписаного методу `can_hold`

Очікуване виведення буде виглядати наступним чином, оскільки обидва виміри `rect2` менші за розміри `rect1`, але `rect3` ширший, ніж `rect1`:

```
Can rect1 hold rect2? true
Can rect1 hold rect3? false
```

Ми знаємо, що хочемо визначити метод, тож він буде написаний у блоці `impl` `Rectangle`. Метод буде зватися `can_hold`, і буде приймати параметром немутабельне позичання іншого `Rectangle`. Ми можемо зрозуміти, якого типу буде параметр, подивившись на код, що викликає метод: `rect1.can_hold(&rect2)` передає `&rect2`, тобто немутабельно позичає `rect2`, екземпляр `Rectangle`. Це зрозуміло, бо нам треба лише читати `rect2` (а не писати, бо тоді б було потрібне мутабельне позичання), і ми хочемо, щоб `main` залишав собі володіння `rect2`, щоб його можна було використовувати після виклику методі `can_hold`. Значення, що повертає `can_hold`, буде булевого типу, а реалізація перевірить, чи ширина та висота `self` більші за відповідно ширину та висоту іншого `Rectangle`. Додамо метод `can_hold` до блоку `impl` з Блоку коду 5-13, як показано в Блоці коду 5-15.

Файл: `src/main.rs`

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

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

Listing 5-15: Implementing the `can_hold` method on `Rectangle` that takes another `Rectangle` instance as a parameter

Коли ми запустимо цей код з функції `main` у Блоці коду 5-14, ми отримаємо вивід, який хотіли. Методи можуть приймати багато параметрів, які ми додаємо до сигнатури після параметру `self`, і ці параметри працюють так само як у функціях.

Асоційовані Функції

Усі функції, визначені в блоці `impl`, зветься *асоційованими функціями*, бо вони асоційовані з типом, названим після `impl`. Ми можемо визначити асоційовані функції, що не мають першим параметром `self` (і відтак не є методами), і вони не потребують екземпляра

типа, щоб із ним працювати. Ми вже користалися такою асоційованою функцією, а саме функцією `String::from`, визначеною на типі `String`.

Асоційовані функції, що не є методами, часто використовуються як конструктори, що повертають новий екземпляр структури. Вони часто називаються `new`, але `new` не є спеціальним ім'ям і не вбудовано в мову. Наприклад, ми можемо написати асоційовану функцію `square`, що матиме один параметр розміру і використовуватиме його і як ширину, і як висоту, щоб створити таким чином квадратний `Rectangle`, не вказуючи одне й те саме значення двічі:

Файл: src/main.rs

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

The `Self` keywords in the return type and in the body of the function are aliases for the type that appears after the `impl` keyword, which in this case is `Rectangle`.

Щоб викликати асоційовану функцію, ми використовуємо запис `::` з іменем структури, наприклад `let sq = Rectangle::square(3);`. Ця функція включена до простору імен структури: запис `::` використовується і для асоційованих функцій, і для просторів імен, створених модулями. Ми будемо обговорювати модулі у [Розділі 7](#).

Кілька Однакових Блоків `impl`

Кожна структура може мати кілька блоків `impl`. Наприклад, Блок коду 5-15 тотожний коду, показаному в Блоці коду 5-16, де кожен метод знаходиться у власному блоці `impl`.

```
impl Rectangle {  
    fn area(&self) -> u32 {  
        self.width * self.height  
    }  
}  
  
impl Rectangle {  
    fn can_hold(&self, other: &Rectangle) -> bool {  
        self.width > other.width && self.height > other.height  
    }  
}
```

Блок коду 5-16: Блок коду 5-15, переписаний у кількох блоках `impl`

Тут немає підстав розділяти ці методи у декілька блоків `impl`, але це коректний синтаксис. Ми побачимо випадок, де кілька блоків `impl` можуть бути корисні, у Розділі 10, де ми поговоримо про узагальнені типи і трейти.

Підсумок

Структури дозволяють вам створювати власні типи, що мають значення для предметної області програми. Використовуючи структури, ми можемо зберігати пов'язані між собою фрагменти даних разом і давати ім'я кожному фрагменту, щоб зробити наш код зрозумілим. У блоках `impl` ви можете визначити функції, асоційовані з вашим типом, а методи - це різновид асоційованих функцій, що дозволяють визначити поведінку, яку мають екземпляри ваших структур.

Але структури не є єдиним способом, яким можна створювати власні типи: перейдімо до такої особливості Rust, як енуми, щоб додати ще одне знаряддя до нашого інструментарію.

Енуми та Зіставлення зі Шаблоном

У цьому розділі ми розглянемо *перелічені типи* (enumeration), також відомі, як *енуми*. Енуми дозволяють вам визначити тип, перелічивши всі його можливі *варіанти*. Спершу ми визначимо і використаємо еnum, щоб показати, як він кодує значення разом із даними. Далі, ми дослідимо особливо корисний еnum, що зветься `Option`, який виражає, що значення може бути або чимось або нічим. Потім ми подивимося на те, як зіставлення з шаблоном у виразі `match` полегшує виконання різних кодів для різних значень еnumа. Нарешті, ми розкриємо, як конструкція `if let` зручно і дозволяє вам зручно та лаконічно використовувати енуми у вашому коді.

Визначення Енума

Якщо структури надають спосіб групування пов'язаних полів і даних, як `Rectangle` з його `width` і `height`, то енуми дають вам спосіб виразити значення, що є одним з можливого набору значень. Скажімо, ми хочемо сказати, що `Rectangle` є однією з можливих фігур, які також включають `Circle` (круг) і `Triangle` (трикутник). Для цього Rust надає нам можливість закодувати ці варіанти у енул.

Розгляньмо ситуацію, яку ми можемо захотіти виразити в коді, і побачимо, чому енуми корисні і краще за структури підходять для цієї ситуації. Нехай нам потрібно працювати із IP-адресами. Наразі використовується два стандарти IP-адрес, четверта та шоста версії. Оскільки це єдині можливі IP-адреси, які наша програма може зустріти, ми можемо *перелічити* (enumerate) усі можливі варіанти, звідси й назва для енумів.

Будь-яка IP-адреса може бути або версії чотири, або версії шість, але не одночасно. Ця властивість IP-адрес робить енул відповідним засобом для вираження цієї ситуації, бо значення енума можуть бути лише одним із його варіантів. Адреси як четвертої, так і шостої версій засадничо є саме IP-адресами, і з ними можна працювати як з одним типом, коли код стосується ситуацій, де можуть використовуватися обидва види адрес.

Цю концепцію можна виразити, визначивши енул `IpAddrKind` і перерахувавши можливі види IP-адрес, `V4` та `V6`. Це зветься варіантами енума:

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

`IpAddrKind` тепер є користувацьким типом даних, яким ми можемо користуватися деінде в нашому коді.

Значення Енума

Ми можемо створити екземпляри обох варіантів `IpAddrKind` таким чином:

```
let four = IpAddrKind::V4;  
let six = IpAddrKind::V6;
```

Зверніть увагу, що варіанти енума знаходяться у просторі імен його ідентифікатора, і для з'єднання ми використовуємо подвійну двокрапку. Це корисно, бо значення `IpAddrKind::V4` і `IpAddrKind::V6` належать до одного типу `IpAddrKind`. Тепер можна, скажімо, визначити функцію, що приймає `IpAddrKind`:

```
fn route(ip_kind: IpAddrKind) {}
```

І ми можемо викликати цю функцію для будь-якого з варіантів:

```
route(IpAddrKind::V4);  
route(IpAddrKind::V6);
```

Але використання еnum-ів дає ще більше переваг. Наразі ми не маємо способу зберігати власне *дані* IP-адреси; ми знаємо лише її *вид*. Оскільки ви щойно дізналися про структури в Розділі 5, у вас може виникнути спокуса розв'язати цю проблему структурами, як показано у Блоці коду 6-1.

```
enum IpAddrKind {  
    V4,  
    V6,  
}  
  
struct IpAddr {  
    kind: IpAddrKind,  
    address: String,  
}  
  
let home = IpAddr {  
    kind: IpAddrKind::V4,  
    address: String::from("127.0.0.1"),  
};  
  
let loopback = IpAddr {  
    kind: IpAddrKind::V6,  
    address: String::from("::1"),  
};
```

Блок коду 6-1: зберігання даних і варіанту `IpAddrKind` IP-адреси за допомогою `struct`

Тут ми визначили структуру `IpAddr`, що має два поля: `kind` (вид) типу `IpAddrKind` (щойно визначений нами еnum) та `address` типу `String`. Ми маємо два екземпляри цієї структури. Перший, `home`, має значення `IpAddrKind::V4` в полі `kind` і прив'язані дані адреси `127.0.0.1`. Другий екземпляр, `loopback`, має значенням поля `kind` інший варіант `IpAddrKind` - `V6`, і має прив'язану адресу `::1`. Ми використали структуру, щоб пов'язати значення `kind` та `address` разом, таким чином варіант тепер прив'язаний до значення.

Але цю концепцію можна представити у коротший спосіб за допомогою самого енума, а не енума всередині структури, розмістивши дані безпосередньо в кожному варіанті енума. Це нове визначення енума `IpAddr` каже, що обидва варіанти `V4` та `V6` мають прив'язані значення `String`:

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

let home = IpAddr::V4(String::from("127.0.0.1"));

let loopback = IpAddr::V6(String::from("::1"));
```

Ми причепили дані безпосередньо до кожного варіанту енума, і тепер нема потреби в додатковій структурі. Тепер легше побачити ще одну деталь роботи еnumів: назва кожного варіанту енума, визначеного нами, стає також функцією, що конструює екземпляр енума. Тобто `IpAddr::V4()` - це виклик функції, що приймає аргументом `String` і повертає екземпляр типу `IpAddr`. Ми автоматично отримуємо функцію-конструктор, визначену в результаті визначення енума.

Є ще одна перевага у використанні енума перед структурою: кожен варіант може мати різні типи та об'єм прив'язаних даних. IP-адреси четвертої версії завжди складаються з чотирьох числових компонентів зі значеннями між 0 та 255. Якби ми хотіли зберігати адреси `V4` як чотири значення `u8`, але все ще представляти `V6` як єдине значення типу `String`, то структурою ми б цього зробити не змогли. Натомість еnumи легко впораються із цим:

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

let home = IpAddr::V4(127, 0, 0, 1);

let loopback = IpAddr::V6(String::from("::1"));
```

Ми представили кілька різних способів визначення структури даних для зберігання IP-адрес версій чотири та шість. Однак, як виявляється, бажання зберігати IP-адреси і кодувати їхній вид настільки поширене, що [стандартна бібліотека вже містить визначення, яке можна використати!](#) Подивімося, як стандартна бібліотека визначає `IpAddr`: там є точно такий `enum` і варіанти, як і ті, що ми визначили, але дані адрес усередині варіантів представлені двома різними структурами, які визначені окремо для кожного варіанту:

```

struct Ipv4Addr {
    // --snip--
}

struct Ipv6Addr {
    // --snip--
}

enum IpAddr {
    V4(Ipv4Addr),
    V6(Ipv6Addr),
}

```

Цей код показує, що ми можемо помістити будь-який вид даних усередину варіанту енума: стрічки, числові типи, структури тощо. Можна навіть вкласти інший еnum! Також типи стандартної бібліотеки часто не набагато складніші за те, що ви б могли самі придумати.

Зверніть увагу, що хоча стандартна бібліотека містить визначення `IpAddr`, ми можемо створити і користуватися нашим власним визначенням без конфлікту, бо ми не ввели визначення зі стандартної бібліотеки до області видимості програми. Ми ще поговоримо про введення до області видимості в Розділі 7.

Let's look at another example of an enum in Listing 6-2: this one has a wide variety of types embedded in its variants.

```

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

```

Блок коду 6-2: Еnum `Message` (повідомлення), варіанти якого містять різну кількість та типи значень

Цей еnum має чотири варіанти різних типів:

- `Quit` ("вийти") не має пов'язаних даних.
- `Move` ("перейти") має всередині анонімний struct.
- `Write` ("написати") включає один `String`.
- `ChangeColor` ("змінити колір") включає три значення `i32`.

Визначення енума з варіантами, схожими на наведені у Блоці коду 6-2, нагадує визначення різних видів структур, але еnum не використовує ключового слова `struct` і

всі варіанти згруповані разом в одному типі `Message`. Наступні структури могли б зберігати ті самі дані, що й варіанти попереднього енума:

```
struct QuitMessage; // unit struct
struct MoveMessage {
    x: i32,
    y: i32,
}
struct WriteMessage(String); // tuple struct
struct ChangeColorMessage(i32, i32, i32); // tuple struct
```

Але якби ми використали різні структури, кожна з яких має свій тип, ми не змогли визначити функцію, яка б приймала будь-який з цих типів повідомлень, як можна робити з енумом `Message`, визначеним у Блоці коду 6-2, що є одним типом.

Енуми та структури мають ще одну спільну рису: як за допомогою `impl` ми можемо оголошувати методи на структурах, ми можемо так само їх оголошувати на енумах. Ось метод, що зветься `call`, який можна визначити на нашому енумі `Message`:

```
impl Message {
    fn call(&self) {
        // method body would be defined here
    }
}

let m = Message::Write(String::from("hello"));
m.call();
```

Тіло методу використовує `self`, щоб отримати значення, для кого було викликано метод. У цьому прикладі ми створили змінну `m`, що має значення `Message::Write(String::from("hello"))`, і саме цей `self` буде в тілі методу `call`, коли буде виконано `m.call()`.

Let's look at another enum in the standard library that is very common and useful: `Option`.

Енум `Option` і Його Переваги над Null-Значеннями

Цей підрозділ розглядає використання `Option`, ще одного енума, визначеного в стандартній бібліотеці. Тип `Option` кодує дуже поширену ситуацію, де значення може бути чи його може не бути.

Наприклад, якщо ви запитуєте перший елемент з непорожнього списку, ви отримаєте значення. Якщо ж ви запитаете перший елемент порожнього списку, ви не отримаєте нічого. Те, що ця концепція виражена в системі типів, означає, що компілятор може

перевірити, що ви обробили всі можливі варіанти, які потребують обробки; цей функціонал запобігає вкрай поширеним в інших мовах програмування вадам.

Дизайн мови програмування часто оцінюють за тим, який функціонал у ній є; але функціонал, який свідомо уникли, також важливий. Rust не має такої особливості, як `null`, що є в багатьох інших мовах. *Null* - це значення, що означає відсутність значення. У мовах із `null` змінні завжди можуть бути в одному з двох станів: `null` і `не-null`.

In his 2009 presentation “Null References: The Billion Dollar Mistake,” Tony Hoare, the inventor of `null`, has this to say:

Я називаю це своєю помилкою на мільярд доларів. У той час я розробляв першу всеосяжну систему типів для посилань в об'єктноорієнтованій мові. Моєю метою було переконатися, що всі використання посилань будуть абсолютно безпечними, з автоматичною перевіркою компілятором. Але я не міг опиратися спокусі додати нульове посилання просто тому, що його було так легко реалізувати. Це призвело до незлічених помилок, вразливостей і системних збоїв, які, мабуть, коштували мільярд доларів болю і шкоди за останні 40 років.

Проблема з `null`-значеннями полягає в тому, що якщо ви спробуєте використовувати значення, яке є `null`, ніби це не `null`, ви дістанете помилку. А оскільки ця властивість є поширеною, стає неймовірно просто помилитися таким чином.

However, the concept that `null` is trying to express is still a useful one: a `null` is a value that is currently invalid or absent for some reason.

Проблема насправді не в самій концепції, а в конкретній реалізації. Відтак Rust не має `null`-значень, але має еnum, що представляє концепцію присутнього чи відсутнього значення. Цей еnum - `Option<T>`, і він визначений у стандартній бібліотеці ось так:

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

Enum `Option<T>` настільки корисний, що він включений у прелюдію; вам не потрібно явно вводити його в область видимості програми. Його варіанти також введені у прелюдію: ви можете використовувати `Some` та `None` напряму без префіксу `Option::`. Утім `Option<T>` - це лише звичайний еnum, а `Some(T)` та `None` - лише варіанти типу `Option<T>`.

Запис `<T>` - особливість Rust, про яку ми ще не говорили. Це параметр узагальненого типу, і детальніше ми розглянемо узагальнення в Розділі 10. Поки що все, що вам слід

знати - що `<T>` означає, що варіант `Some` енума `Option` може вміщати одне значення даних будь-якого типу, і що конкретний тип, підставлений на місце `T`, робить весь вираз `Option<T>` окремим типом. Ось деякі приклади використання значень `Option` для зберігання числових типів і стрічкових типів:

```
let some_number = Some(5);
let some_char = Some('e');

let absent_number: Option<i32> = None;
```

Тип `some_number` - `Option<i32>`. Типом `some_char` є `Option<char>`, тобто інший тип. Rust може вивести ці типи, бо ми вказали значення всередині варіанту `Some`. А для `absent_number` Rust вимагає, щоб ми анотували весь тип `Option`: компілятор не може вивести тип відповідного варіанту `Some`, що міститиме значення, за самим лише значенням `None`. Тут ми вказуємо Rust, що хочемо, аби `absent_number` мав тип `Option<i32>`.

Коли у нас є значення `Some`, ми знаємо, що значення наявне, і значення зберігається в варіанті `Some`. Коли є значення `None`, у певному сенсі, це означає те саме, що й `null`: ми не маємо придатного значення. То чим же `Option<T>` кращий за значення `null`?

Одним словом, оскільки `Option<T>` і `T` (де `T` може бути будь-яким типом) - різні типи, компілятор не дозволить нам використовувати значення `Option<T>` так, ніби ми маємо коректне значення. Наприклад, цей код не скомпілюється, бо він намагається додати `i8` до `Option<i8>`:

```
let x: i8 = 5;
let y: Option<i8> = Some(5);

let sum = x + y;
```

Якщо ми запусимо цей код, ми дістанемо повідомлення про помилку на кшталт цього:


```

$ cargo run
   Compiling enums v0.1.0 (file:///projects/enums)
error[E0277]: cannot add `Option<i8>` to `i8`
  --> src/main.rs:5:17
   |
5  |         let sum = x + y;
   |                     ^ no implementation for `i8 + Option<i8>`
   |
= help: the trait `Add<Option<i8>>` is not implemented for `i8`
= help: the following other types implement trait `Add<Rhs>`:
      <&'a f32 as Add<f32>>
      <&'a f64 as Add<f64>>
      <&'a i128 as Add<i128>>
      <&'a i16 as Add<i16>>
      <&'a i32 as Add<i32>>
      <&'a i64 as Add<i64>>
      <&'a i8 as Add<i8>>
      <&'a isize as Add<isize>>
      and 48 others

For more information about this error, try `rustc --explain E0277`.
error: could not compile `enums` due to previous error

```

Сильно! Насправді це повідомлення про помилку означає, що Rust не розуміє, як додати `i8` та `Option<i8>`, оскільки вони різних типів. Коли у нас у Rust є значення типу на кшталт `i8`, компілятор гарантує, що у нас завжди є коректне значення. Ми можемо діяти впевнено без потреби у перевірці на null перш ніж використовувати це значення. Тільки тоді, коли у нас є `Option<i8>` (або будь-який тип чи значення, з яким ми працюємо), ми маємо турбуватися про те, що, можливо, значення не буде, і компілятор переконується, що ми обробляємо цей випадок, перш ніж використовувати значення.

Іншими словами, перед тим, як виконувати операції, які можна робити з `T`, треба перетворити значення `Option<T>` на `T`. В цілому це допомагає перехопити одну з найпоширеніших проблем із null - припущення, що щось не є null, коли насправді воно null.

Відсутність потреби турбуватися про некоректне припущення про не-null значення допомагає вам бути певнішим у власному коді. Щоб значення могло бути null, вам треба явно це вказати зробивши тип цього значення `Option<T>`. Потім, коли ви використовуєте це значення, від вас вимагається явно обробити випадок, коли це значення null. Всюди, де значення має тип, відмінний від `Option<T>`, ви *можете* безпечно припустити, що це значення не null. Це свідоме рішення при розробці Rust для обмеження передавання null і збільшення безпеки коду Rust.

Але як же отримати значення `T` з варіанту `Some`, коли ви маєте значення типу `Option<T>`, щоб його використати? Енум `Option<T>` має велику кількість методів, зручних у

різноманітних ситуаціях; ви можете подивитися їх у [документації](#). Ознайомлення з методами `Option<T>` буде вкрай корисним для вашого вивчення Rust.

В цілому, щоб скористатися значенням `Option<T>`, ми хочемо мати код, що обробить обидва варіанти. Ми хочемо, щоб певний код виконувався лише для значень `Some(T)`, і цей код міг використовувати внутрішнє `T`. І ми хочемо, щоб інший код виконувався лише коли ми маємо значення `None`, і цей код не має доступу до значення `T`. Вираз `match` - це конструкція управління, що саме це й робить, коли використовується з енумами: воно виконає різний код залежно від варіанту енума, і цей код може використовувати дані всередині відповідного значення.

Конструкція Потoku Виконання `match`

Rust має вкрай потужну конструкцію управління, що зветься `match`, яка дозволяє нам порівнювати значення із кількома шаблонами та потім виконати код, виходячи з того, який шаблон відповідає цьому значенню. Шаплони можуть складатися з літералів, імен змінних, символів узагальнення і багатьох інших речей; [Розділ 18](#) розповідає про всі різні види шаблонів і що вони роблять. Потужність `match` походить від виразності шаблонів і того факту, що компілятор перевіряє, щоб усі можливі ситуації були оброблені.

Вираз `match` можна уявити собі як сортувальну машину для монет: монети ковзають жолобом з отворами різних розмірів, і кожна монета падає крізь перший отвір, в який вона проходить. Так само значення проходить крізь кожен шаблон в `match`, і на першому шаблоні, якому воно відповідає, значення "провалюється" в пов'язаний блок коду, де може бути використане при його виконанні.

Оскільки ми згадали монети, використаємо їх як приклад використання `match`! Ми можемо написати функцію, що приймає невідому монету США і, так само як і лічильна машина, визначає, яка це монета і повертає її значення в центах, як показано в Блоці коду 6-3.

```
enum Coin {
    Penny,
    Nickel,
    Dime,
    Quarter,
}

fn value_in_cents(coin: Coin) -> u8 {
    match coin {
        Coin::Penny => 1,
        Coin::Nickel => 5,
        Coin::Dime => 10,
        Coin::Quarter => 25,
    }
}
```

Блок коду 6-3: Енум і вираз `match` із варіантами енума як шаблонами

Розберімо `match` у функції `value_in_cents`. По-перше, ми пишемо ключове слово `match`, за яким іде вираз, у цьому випадку - значення `coin`. Це дуже схоже на вираз, що використовується в `if`, але є велика відмінність: в `if` вираз має повертати булеве

значення, а тут значення може бути будь-якого типу. Тип `coin` у цьому прикладі - еnum `Coin`, який ми визначили у першому рядку.

Далі йдуть рукави `match`. Рукав має дві частини: шаблон і код. Перший рукав має шаблон, що є значенням `Coin::Penny`, після чого оператор `=>` відокремлює шаблон і код, що буде виконано. Код у цьому випадку - просто значення `1`. Кожен рукав відокремлений від наступного комою.

Коли виконується вираз `match`, значення по черзі порівнюється із шаблоном кожного рукава. Якщо шаблон відповідає значенню, виконується пов'язаний із цим шаблоном код. Якщо шаблон не відповідає значенню, виконання передається наступному рукаву, як монетка в сортувальній машині. Рукавів може бути стільки, скільки нам потрібно: у Блоці коду 6-3 `match` має чотири рукави.

Код, пов'язаний з кожним рукавом, є виразом, і кінцеве значення виразу у відповідному рукаві є значенням, яке повертається для всього виразу `match`.

Фігурні дужки зазвичай не використовуються, якщо код рукава `match` невеликий, як у Блоці коду 6-3, де кожен рукав просто повертає значення. Якщо ви хочете виконати багато рядків коду у рукаві `match`, то маєте скористатися фігурними дужками, кома після яких в такому разі не обов'язкова. Наприклад, наступний код виводитиме "Lucky penny!" кожного разу, коли метод викличуть для `Coin::Penny`, але також поверне останнє значення блоку, тобто `1`:

```
fn value_in_cents(coin: Coin) -> u8 {  
    match coin {  
        Coin::Penny => {  
            println!("Lucky penny!");  
            1  
        }  
        Coin::Nickel => 5,  
        Coin::Dime => 10,  
        Coin::Quarter => 25,  
    }  
}
```

Шаблони, які Прив'язуються до Значень

Інша корисна властивість рукавів `match` полягає в тому, що вони можуть зв'язуватися з частинами значення, що відповідає шаблону. Таким чином ми можемо дістати значення з варіантів еnumів.

Наприклад, змінімо один з варіантів еnumа, щоб він мав дані усередині. З 1999 по 2008 роки Сполучені Штати карбували четвертаки з різними дизайнами для кожного з 50

штатів на одному боці. Інші монети не мають окремих дизайнів для штатів, тому лише четвертаки мають таке додаткове значення. Ми можемо додати цю інформацію до нашого `enum`, змінивши варіант `Quarter`, аби він містив у собі значення `UsState`, що й зроблено в Блоці коду 6-4.

```
#[derive(Debug)] // so we can inspect the state in a minute
enum UsState {
    Alabama,
    Alaska,
    // --snip--
}

enum Coin {
    Penny,
    Nickel,
    Dime,
    Quarter(UsState),
}
```

Listing 6-4: A `Coin` enum in which the `Quarter` variant also holds a `UsState` value

Уявімо, що наш друг намагається зібрати всі 50 четвертаків різних штатів. Сортиючи дріб'язок по типах монет, ми також будемо називати назви штатів, пов'язаних з кожним четвертаком, щоб, якщо такого наш друг такого не має, він зміг би додати його до своєї колекції.

У виразі `match` у цьому коді ми додаємо змінну, що зветься `state` до шаблону, що відповідає значенню варіанту `Coin::Quarter`. Коли шаблон `Coin::Quarter` буде відповідним до виразу, змінна `state` зв'яжеться зі значенням штату цього четвертака. Тоді ми можемо використати `state` у коді цього рукава, ось так:

```
fn value_in_cents(coin: Coin) -> u8 {
    match coin {
        Coin::Penny => 1,
        Coin::Nickel => 5,
        Coin::Dime => 10,
        Coin::Quarter(state) => {
            println!("State quarter from {:?}!", state);
            25
        }
    }
}
```

Якщо ми викличемо `value_in_cents(Coin::Quarter(UsState::Alaska))`, значення `coin` буде `Coin::Quarter(UsState::Alaska)`. Коли ми порівняємо це значення з усіма рукавами, то не підійде жоден, поки ми не дістанемося `Coin::Quarter(state)`. У цьому місці `state` буде зв'язане зі значенням `UsState::Alaska`. Ми зможемо тоді скористатися цим

зв'язуванням у виразі `println!`, отримавши таким чином внутрішнє значення штату з енума `Coin` для варіанту `Quarter`.

Зіставлення з `Option<T>`

У попередньому підрозділі ми хотіли дістати внутрішнє значення типу `T` з варіанту `Some`, коли працювали з `Option<T>`; з `Option<T>` ми теж можемо скористатися конструкцією `match`, так само як робили з енумом `Coin`! Замість монет ми порівнюватимемо варіанти `Option<T>`, але вираз `match` при цьому працює тим самим чином.

Хай, скажімо, ми хочемо написати функцію, що приймає `Option<i32>`, і якщо він містить значення, додає один до цього значення. А якщо там немає значення всередині, функція має повертати значення `None` і не намагатися виконати жодних дій.

This function is very easy to write, thanks to `match`, and will look like Listing 6-5.

```
fn plus_one(x: Option<i32>) -> Option<i32> {  
    match x {  
        None => None,  
        Some(i) => Some(i + 1),  
    }  
}  
  
let five = Some(5);  
let six = plus_one(five);  
let none = plus_one(None);
```

Блок коду 6-5: функція, що застосовує вираз `match` до `Option<i32>`

Розгляньмо детальніше перше виконання `plus_one`. Коли ми викликаємо `plus_one(five)`, змінна `x` у тілі `plus_one` матиме значення `Some(5)`. Далі ми порівнюємо це значення з кожним рукавом `match`:

```
None => None,
```

Значення `Some(5)` не відповідає шаблону `None`, тому ми переходимо до наступного рукава:

```
Some(i) => Some(i + 1),
```

Чи відповідає `Some(5)` шаблону `Some(i)`? Так! Ми маємо той самий варіант. Змінна `i` зв'язується зі значенням, що міститься в `Some`, тобто `i` набуває значення `5`. Далі

виконується код у рукаві `match`, тобто додається один до значення `i` і створюється нове значення `Some` із результатом `6` всередині.

Тепер розгляньмо другий виклик `plus_one` у Блоці коду 6-5, де `x` дорівнює `None`. Ми входимо в `match` і порівнюємо перший рукав:

```
None => None,
```

Підходить! Немає значення, до якого треба додавати, і програма зупиняється і повертає значення `None`, що стоїть праворуч від `=>`. Оскільки перший рукав відповідає значенню, решта рукавів не перевіряються.

Комбінування `match` і еnumів корисне в багатьох ситуаціях. Ви часто бачитимете цей шаблон у коді Rust: `match` із еnumом, зв'язування змінної з даними усередині, і виконання коду відповідно до цього. Це спершу трохи мудровано, але шойно ви звикнете до цього, то бажатимете мати таку конструкцію в усіх мовах. Ця конструкція - незмінний улюбленець користувачів Rust.

Зіставлення є Вичерпними

Є ще один бік `match`, що ми маємо обговорити: шаблони рукавів мають покривати всі можливості. Розгляньте таку версію нашої функції `plus_one`, в якій є вада і вона не скомпілюється:

```
fn plus_one(x: Option<i32>) -> Option<i32> {  
    match x {  
        Some(i) => Some(i + 1),  
    }  
}
```

Ми не обробили варіанту `None`, тому цей код призводить до вади. На щастя, Rust знає, як виявляти такі вади. Якщо ми спробуємо скомпілювати цей код, то отримаємо таке повідомлення про помилку:

```

$ cargo run
   Compiling enums v0.1.0 (file:///projects/enums)
error[E0004]: non-exhaustive patterns: `None` not covered
  --> src/main.rs:3:15
   |
3  |         match x {
   |             ^ pattern `None` not covered
   |
note: `Option<i32>` defined here
   = note: the matched value is of type `Option<i32>`
help: ensure that all possible cases are being handled by adding a match arm with
a wildcard pattern or an explicit pattern as shown
   |
4  ~             Some(i) => Some(i + 1),
5  ~             None => todo!(),
   |

For more information about this error, try `rustc --explain E0004`.
error: could not compile `enums` due to previous error

```

Rust знає, що ми не покрили усі можливі випадки, і навіть знає, який саме шаблон ми забули! Match в Rust *вичерпні*: ми маємо вичерпати всі можливі ситуації, щоб код був коректним. Особливо у випадку з `Option<T>`, коли Rust, запобігаючи тому, щоб ми забули явно обробити випадок `None`, захищає нас від припущення, що ми маємо значення, коли ми можемо мати null, таким чином припускаючись помилки на мільярд доларів, про яку ми говорили вище.

Шаблони для Всіх Випадків і Заповнювач

При роботі з енумами нам може знадобитися особлива дія для кількох конкретних значень, а для всіх інших значень - одна дія за замовчуванням. Уявіть, що ми розробляємо гру, де, якщо ви викинули 3 на кубіку, ваш гравець не рухається, а отримує нового модного капелюха. Якщо ви викинете 7, ваш гравець втратить модного капелюха. Для всіх інших значень, ваш гравець рухається на цю кількість клітинок на ігровому полі. Ось `match`, що реалізовує цю логіку, де результат кидання кубіка жорстко задано замість випадкового значення, і решта логіки представлена функціями без тіл, бо ми насправді реалізуємо їх поза областю видимості цього прикладу:


```
let dice_roll = 9;
match dice_roll {
    3 => add_fancy_hat(),
    7 => remove_fancy_hat(),
    other => move_player(other),
}

fn add_fancy_hat() {}
fn remove_fancy_hat() {}
fn move_player(num_spaces: u8) {}
```

У перших двох рукавів шаблони - літерали зі значеннями `3` і `7`. У останнього рукава, що покриває всі інші можливі значення, шаблон - це змінна, яку ми вирішили назвати `other`. Код, що виконується в цьому рукаві, використовує змінну `other`, передаючи її у функцію `move_player`.

Цей код компілюється, хоча ми не перерахували усі можливі значення, яких може набути `u8`, бо останній шаблон відповідає всім значенням, які не були вказані окремо. Цей шаблон для всіх випадків задовольняє вимозі вичерпності `match`. Зверніть увагу, що шаблон для всіх випадків розміщується останнім, бо шаблони обчислюються послідовно. Якщо ми розмістимо рукав для всіх випадків раніше, решта рукавів ніколи не запусяться, тому Rust попередить нас, якщо ми після нього додамо ще рукави!

Rust також має шаблон, яким можна скористатися, коли нам потрібно обробити всі випадки, але ми не хочемо *використовувати* значення у шаблоні для всіх випадків: `_` є спеціальним шаблоном, що відповідає будь-якому значенню і не зв'язується із цим значенням. Це каже Rust, що ми не збираємося використовувати це значення, і тому він не попереджатиме про невикористану змінну.

Змінімо правила гри: тепер, якщо ви викинете щось, крім 3 чи 7, то маєте кидати кубик знову. Нам більше не потрібне значення для всіх випадків, тож ми можемо змінити наш код і скористатися `_` замість змінної на ім'я `other`:

```
let dice_roll = 9;
match dice_roll {
    3 => add_fancy_hat(),
    7 => remove_fancy_hat(),
    _ => reroll(),
}

fn add_fancy_hat() {}
fn remove_fancy_hat() {}
fn reroll() {}
```

This example also meets the exhaustiveness requirement because we're explicitly ignoring all other values in the last arm; we haven't forgotten anything.

Нарешті, змінимо правила гри ще раз, так щоб нічого не ставалося на вашому ході, якщо ви викинули щось інше, крім 3 чи 7. Це можна виразити одиничним значенням (тип порожнього кортежу, який ми згадували у підрозділі [“Тип кортеж”](#)) у коді, що знаходиться у рукаві `_`:

```
let dice_roll = 9;
match dice_roll {
    3 => add_fancy_hat(),
    7 => remove_fancy_hat(),
    _ => (),
}

fn add_fancy_hat() {}
fn remove_fancy_hat() {}
```

Тут ми явно говоримо Rust, що не збираємося використовувати жодного іншого значення, яке б не відповідало шаблону якогось з попередніх рукавів, і ми не хочемо виконувати жодного коду в цьому випадку.

Більш детально про шаблони і зіставлення з ними йдеться у [Розділі 18](#). Ну а поки що ми перейдемо до конструкції `if let`, яка може бути корисною в ситуаціях, де вираз `match` буде надто багатослівним.

Лаконічний Потік Виконання з `if let`

Конструкція `if let` дозволяє вам комбінувати `if` та `let` менш багатослівно, щоб обробляти значення, що відповідають одному шаблону, і ігнорувати інші. Розглянемо програму у Блоці коду 6-6, що працює зі значенням `Option<u8>` у змінній `config_max`, але хоче виконувати код лише коли значення є варіантом `Some`.

```
let config_max = Some(3u8);
match config_max {
    Some(max) => println!("The maximum is configured to be {}", max),
    _ => (),
}
```

Блок коду 6-6: `match`, що виконує код лише зі значенням `Some`

Якщо значення є `Some`, ми виводимо значення у варіанті `Some`, зв'язавши у шаблоні це значення зі змінною `max`. Ми не хочемо нічого робити зі значенням `None`. Щоб задовольнити вираз `match`, нам доводиться додати `_ => ()` після обробки лише одного варіанту, що є набридливо надлишковим.

Натомість ми можемо записати це коротше за допомогою `if let`. Наступний код робить те саме, що й `match` з Блоку коду 6-6:

```
let config_max = Some(3u8);
if let Some(max) = config_max {
    println!("The maximum is configured to be {}", max);
}
```

Конструкція `if let` бере шаблон і вираз, розділені знаком рівності. Вона працює так само як і `match`, де вираз стоїть після `match`, а шаблон є його першим рукавом. У цьому випадку шаблоном буде `Some(max)`, і `max` зв'язується зі значенням всередині `Some`. Тепер ми можемо використати `max` у тілі блоку `if let` так само як ми використали `max` у відповідному рукаві `match`. Код у блоці `if let` не буде виконано, якщо значення не відповідає шаблону.

Використання `if let` означає, що вам треба менше друкувати, менше ставити відступів і писати менше зайвого коду. Разом з тим, ми втрачаємо перевірку на вичерпність, до якої зобов'язує `match`. Вибір між `match` та `if let` залежить від того, що ви робите у конкретній ситуації та чи лаконічність варта втрати перевірки на вичерпність.

In other words, you can think of `if let` as syntax sugar for a `match` that runs code when the value matches one pattern and then ignores all other values.

У `if let` можна також додати `else`. Блок, що іде після `else` - це той самий блок, що був би у випадку `_` у виразу `match`, еквівалентному нашому `if let` та `else`. Згадаємо визначення енума `Coin` у Блоці коду 6-4, де варіант `Quarter` також включав значення `UsState`. Якби ми захотіли полічити усе, крім чвертаків, і водночас виводити штат з чвертаків, ми могли б зробити це за допомогою десь такого виразу `match`:

```
let mut count = 0;
match coin {
    Coin::Quarter(state) => println!("State quarter from {:?}!", state),
    _ => count += 1,
}
```

Або ж ми могли б скористатися виразом `if let` та `else` ось таким чином:

```
let mut count = 0;
if let Coin::Quarter(state) = coin {
    println!("State quarter from {:?}!", state);
} else {
    count += 1;
}
```

If you have a situation in which your program has logic that is too verbose to express using a `match`, remember that `if let` is in your Rust toolbox as well.

Підсумок

Ми щойно розібрали, як використовувати енуми для створення власних типів, які можуть набувати одне з множини перелічених значень. Ми показали, як тип `Option<T>` зі стандартної бібліотеки допомагає використовувати систему типів для уникання помилок. Коли значення енума мають дані всередині, можна скористатися `match` чи `if let`, щоб витягти та використати ці значення, залежно від того, скільки різних варіантів вам треба обробити.

Ваші програми Rust тепер можуть виражати концепції з проблемної області за допомогою структур та еnumів. Створення власних типів для використання у вашому API гарантує безпеку типів: компілятор забезпечить, що ваші функції отримають лише значення тих типів, на які ці функції очікують.

In order to provide a well-organized API to your users that is straightforward to use and only exposes exactly what your users will need, let's now turn to Rust's modules.

Керування Щораз Більшими Проєктами із Пакетами, Крейтами та Модулями

Що більші програми ви пишете, то більшого значення набуває організація коду. Групуючи пов'язаний функціонал і розділяючи код з не пов'язаними функціями, ви вносите ясність, де шукати код, що реалізовує певний функціонал, і де вносити зміни до нього.

Програми, які ми написали раніше, поки знаходилися в одному модулі в єдиному файлі. У міру зростання проєкту вам слід організовувати код, розбиваючи його на кілька модулів і декілька файлів. Пакет може містити багато двійкових крейтів і, можливо, один бібліотечний крейт. Зі зростанням пакета ви можете виділяти його частини в окремі крейти, що стають зовнішніми залежностями. Цей розділ висвітлює усі ці техніки. Для дуже великих проєктів, які містять взаємопов'язані пакети, що розвиваються разом, Cargo надає *робочі простори*, які будуть висвітлені у підрозділі "[Робочі простори Cargo](#)" у Розділі 14.

Ми також обговоримо інкапсуляцію деталей реалізації, що дозволяє повторно використовувати код на більш високому рівні: щойно ви реалізували операцію, інший код може викликати ваш код через публічний інтерфейс, навіть не знаючи деталей реалізації. Ваш підхід до написання коду визначає, які частини програми є публічними для використання іншим кодом, а які є приватними, деталі реалізації яких ви б хотіли приховати. Це ще один спосіб обмеження кількості деталей, які вам потрібно тримати в голові.

Пов'язане поняття - область видимості (scope): вкладений контекст, у якому написаний код, має набір назв, про які кажуть, що вони "в області видимості." При читанні, написанні і компілюванні коду, програмісти та компілятори мають знати, чи певна назва в певному місці стосується змінної, функції, структури, енуму, модулю, константи або іншого елемента і що саме цей елемент означає. Ви можете створювати області видимості та визначати, які імена належать до них, а які ні. Але не можна мати два елементи з однаковою назвою в одній області видимості. Існують інструменти для вирішення конфліктів імен.

Rust має ряд функцій, що дозволяють керувати організацією коду, як-от тим, які деталі робити публічними, які приватними, і які імена будуть в кожній області видимості вашої програми. Ці функції, які інколи називають *модульною системою*, охоплюють:

- **Пакети:** функціонал Cargo, що дозволяє збирати, тестувати і поширювати крейти
- **Крейти:** дерево модулів, що створює бібліотеку або виконуваний файл

- **Модулі** та **use**: дозволяють керувати організацією, областю видимості та приватністю шляхів
- **Шляхи**: спосіб іменування елемента, як-то структура, функція або модуль

У цьому розділі ми розглянемо весь цей функціонал, подивимось, як він взаємодіє і пояснимо, як його використовувати для керування областю видимості. В результаті у вас має бути ґрунтовне розуміння модульної системи та здатність працювати з областями видимості на рівні професіоналів!

Пакети та Крейти

Першими частинами модульної системи, які ми охопимо, будуть пакети та крейти.

Крейт - це найменша кількість коду, яку компілятор Rust розглядає за один раз. Навіть, якщо ви запускаєте `rustc`, а не `cargo` і передаєте єдиний файл з вихідним кодом (як ми це робили у секції "Написання і запуск програми на Rust" Розділу 1), компілятор розглядає цей файл як крейт. Крейти можуть містити модулі, і модулі можуть бути визначені в інших файлах, які компілюються з крейтом, як ми побачимо у наступних підрозділах.

Крейт може бути представленим у двох формах: двійковий крейт або бібліотечний крейт. *Двійкові крейти* - це програми, які ви можете скомпілювати у виконуваний файл, що можна запустити, такі як, наприклад, програму командного рядка чи сервер. Кожен з них має містити функцію із назвою `main`, яка визначає, що відбувається, коли виконуваний файл запускається. Усі крейти, що ми поки що створили, є двійковими.

Бібліотечні крейти не мають функції `main` і вони не компілюються у виконуваний файл. Замість цього вони визначають функціонал, призначений для спільного використання у кількох проєктах. Наприклад, крейт `rand`, який ми використовували у [Розділі 2](#), забезпечує функціонал, що генерує випадкові числа. У більшості випадків, коли Растаціанці говорять "крейт", вони мають на увазі саме бібліотечний крейт, і вони використовують "крейт" взаємозамінно із загальною концепцією програмування "бібліотека".

Корінь крейта - це вихідний файл, з якого компілятор Rust розпочинає роботу і створює кореневий модуль вашого крейта (про модулі ми розкажемо детальніше у підрозділі ["Визначення модулів для контролю області видимості та приватності"](#)).

Пакети - це набір одного або більше крейтів, які забезпечують набір функціоналів. Пакет містить файл `Cargo.toml`, який описує, як зібрати ці крейти. Cargo - це, по суті, пакет, який містить двійковий крейт для інструменту командного рядка, який ви вже використовували для збірки вашого коду. Пакет Cargo також містить бібліотечний крейт, від якого залежить двійковий крейт. Інші проєкти також можуть залежати від бібліотечного крейта Cargo, щоб використовувати таку ж саму логіку, яку використовують інструмент командного рядка Cargo.

Пакет може містити стільки двійкових крейтів, скільки ви захочете, проте не більше одного бібліотечного. Пакет повинен містити принаймні один крейт, бібліотечний чи двійковий.

Розгляньмо, що відбувається, коли ми створюємо пакет. Спочатку ми вводимо команду

```
cargo new:
```

```
$ cargo new my-project
    Created binary (application) `my-project` package
$ ls my-project
Cargo.toml
src
$ ls my-project/src
main.rs
```

Після того, як ми запустили `cargo new`, ми використовуємо `ls` для того, щоб побачити, що Cargo створює. У каталозі проєкту знаходиться файл *Cargo.toml*, який дає нам пакет. Також, є ще каталог *src*, який містить *main.rs*. Відкрийте *Cargo.toml* вашому текстовому редактору й зверніть увагу, що там немає жодної згадки про *src/main.rs*. Cargo слідує домовленості, що *src/main.rs* є коренем двійкового крейта із тою ж самою назвою, що має пакунок. Окрім цього, Cargo знає, що якщо каталог пакунків містить *src/lib.rs*, пакунок містить бібліотечний крейт із тою ж назвою, що й пакунок, і *src/lib.rs* є коренем цього крейта. Cargo передає файли кореня крейта до `rustc` для збірки бібліотеки чи двійкового файлу.

Отже, ми маємо пакет, який містить лише *src/main.rs*, що означає, що тут міститься тільки двійковий крейт з назвою `my-project`. Якщо пакет містить *src/main.rs* і *src/lib.rs*, він складається з двох крейтів: двійкового і бібліотечного, обидва із такою ж назвою, як і пакет. У пакеті можна мати кілька двійкових крейтів, розмістивши файли у каталозі *src/bin*: кожен файл буде окремим двійковим крейтом.

Визначення Модулів для Контролю Області Видимості та Приватності

В цьому розділі ми поговоримо про модулі та інші частини модульної системи, а саме: про *шляхи*, що дозволяють іменувати елементи; про ключове слово `use`, яке додає шлях в область видимості; та про ключове слово `pub`, що робить елементи публічними. Ми також розглянемо ключове слово `as`, зовнішні пакети та оператор `glob`.

Ми почнемо з переліку правил, до яких вам було б зручно повертатися в якості довідки в майбутньому при організації коду. Потім ми детально пояснимо кожне з правил.

Шпаргалка на Тему Модулів

У цьому місці ми дамо короткий огляд того, як модулі, шляхи, ключові слова `use` та `pub` працюють у компіляторі, та як більшість розробників організовують свій код. В цьому розділі ми також розберемо приклади кожного з цих правил. Цей розділ буде прекрасним місцем, куди варто звертатися для нагадування про те, як працюють модулі.

- **Починайте з кореня крейту:** Компілюючи крейт, компілятор спочатку дивиться в кореневий файл крейту в пошуках коду для компіляції. Зазвичай це `src/lib.rs` для бібліотечного крейту або `src/main.rs` для бінарного.
- **Оголошення модулів:** Ви можете оголошувати нові модулі в кореновому файлі крейту. Скажімо, ви хочете оголосити модуль "garden" як `mod garden;`. Компілятор шукатиме код даного модуля в наступних місцях:
 - Локально в цьому файлі всередині фігурних дужок, які замінюють крапку з комою після `mod garden`
 - У файлі `src/garden.rs`
 - У файлі `src/garden/mod.rs`
- **Оголошення підмодулів:** Ви можете оголошувати підмодулі в будь-якому файлі, не лише в корені крейту. Наприклад, ви можете оголосити `mod vegetables;` в `src/garden.rs`. Компілятор шукатиме код підмодуля в теці з іменем батьківського модуля в наступних місцях:
 - Локально в цьому файлі, одразу після `mod vegetables`, всередині фігурних дужок замість крапки з комою
 - У файлі `src/garden/vegetables.rs`
 - У файлі `src/garden/vegetables/mod.rs`
- **Шляхи до коду в модулях:** Після того як модуль став частиною вашого крейту, ви можете звертатися до його коду з будь-якого місця даного крейту за допомогою

шляху до коду, якщо дозволяють правила приватності. Наприклад, тип `Asparagus` в модулі `garden vegetables` буде знайдений за шляхом

```
crate::garden::vegetables::Asparagus.
```

- **Приватність або публічність:** Код всередині модуля є приватним від його батьківських модулів за замовчуванням. Аби зробити модуль публічним, оголосіть його за допомогою `pub mod` замість `mod`. Аби зробити елементи всередині публічного модуля публічними також, використовуйте `pub` перед їх оголошенням.
- **Ключове слово `use`:** Всередині області видимості ключове слово `use` створює псевдоніми для елементів аби прибрати необхідність повторювати довгі шляхи. В будь якій області видимості, де необхідно звертатися до `crate::garden::vegetables::Asparagus` ви можете створити псевдонім `use crate::garden::vegetables::Asparagus;` і після цього просто писати `Asparagus` для використання цього типу в даній області видимості.

Аби продемонструвати ці правила, створимо бінарний крейт `backyard`. Тека крейту, яка також називається `backyard`, містить такі файли та теки:

```
backyard
├── Cargo.lock
├── Cargo.toml
└── src
    ├── garden
    │   └── vegetables.rs
    ├── garden.rs
    └── main.rs
```

Кореневий файл крейту в цьому випадку це `src/main.rs`. Його вміст:

Файл: `src/main.rs`

```
use crate::garden::vegetables::Asparagus;

pub mod garden;

fn main() {
    let plant = Asparagus {};
    println!("I'm growing {:?}", plant);
}
```

Рядок `pub mod garden;` каже компілятору підключити код, який знайдений в `src/garden.rs`:

Файл: `src/garden.rs`

```
pub mod vegetables;
```

Тут `pub mod vegetables;` означає, що код в `src/garden/vegetables.rs` також буде підключений. Цей код:

```
#[derive(Debug)]  
pub struct Asparagus {}
```

Тепер давайте розглянемо ці правила детальніше і продемонструємо їх в роботі!

Групування Пов'язаного Коду в Модулі

Модулі дозволяють організувати код в крейт для читабельності та простоти повторного використання. Модулі також дозволяють контролювати *приватність* елементів, оскільки код всередині модуля є приватним за замовчуванням. Приватні елементи являють собою внутрішні деталі реалізації, недоступні для використання ззовні. Ми можемо зробити модулі і елементи всередині них публічними, що дозволить сторонньому коду використовувати їх і залежати від них.

В якості прикладу давайте напишемо бібліотечний крейт, що реалізує функціонал ресторану. Ми визначимо сигнатури функцій, проте залишимо їх вміст пустим для того, щоб сконцентруватися на організації коду, а не на деталях імплементації ресторану.

В ресторанній справі вирізняють такі частини ресторану як *внутрішня кухня (back of house)* та *зал (front of house)*. Зал це те, де сидять відвідувачі. Тут знаходяться місця для клієнтів, офіціанти приймають замовлення і оплату, а бармени роблять напої. Внутрішня кухня - це те, де шеф-кухарі і повари працюють на кухні, посудомийники миють посуд, а менеджери виконують адміністративну роботу.

Для того аби структурувати наш крейт правильним чином, можемо організувати його функції у вкладених модулях. Створіть нову бібліотеку з іменем `restaurant`, виконавши `cargo new restaurant --lib`; тоді наберіть код з Блоку коду 7-1 в `src/lib.rs`, аби визначити деякі модулі та сигнатури функцій. Далі йде секція для зали:

Файл: `src/lib.rs`

```
mod front_of_house {
    mod hosting {
        fn add_to_waitlist() {}

        fn seat_at_table() {}
    }

    mod serving {
        fn take_order() {}

        fn serve_order() {}

        fn take_payment() {}
    }
}
```

Блок коду 7-1: Модуль `front_of_house`, що містить інші модулі, які своєю чергою містять функції

Ми визначаємо модуль за допомогою ключового слова `mod`, після якого йде назва модуля (в цьому випадку `front_of_house`). Тіло модуля розміщується всередині фігурних дужок. Модулі можуть містити інші модулі, як в нашому випадку це зроблено з модулями `hosting` та `serving`. Також в модулях можуть знаходитися визначення інших елементів, таких як структури, переліки, константи, трейти, і - як у Блоці коду 7-1 - функції.

Використовуючи модулі, ми можемо групувати пов'язані визначення між собою і показувати, чому саме вони пов'язані. Програмісти, що використовують цей код, можуть орієнтуватись в коді на рівні функцій замість того, аби бути змушеними читати всі визначення в коді. Це робить задачу пошуку необхідних елементів набагато простішою. Додаючи новий функціонал до коду, програмісти знають де розмістити певний код аби підтримувати порядок і організацію в програмі.

Раніше ми згадували, що `src/main.rs` та `src/lib.rs` називаються коренями крейту. Причина такого іменування в тому, що вміст будь-якого з цих двох файлів утворює модуль з іменем `crate` в корені структури модуля крейту, яка також відома як *дерево модулів*.

Блок коду 7-2 демонструє дерево модулів для структури з Блоку коду 7-1.

```
crate
├── front_of_house
│   ├── hosting
│   │   ├── add_to_waitlist
│   │   └── seat_at_table
│   └── serving
│       ├── take_order
│       ├── serve_order
│       └── take_payment
```

Блок коду 7-2: дерево модулів для коду в Блоці коду 7-1

Це дерево показує, як одні модулі вкладені в інші. Наприклад, `hosting` вкладений в `front_of_house`. Дерево також показує, що деякі модулі є *братами* (*siblings*) один для одного, що означає, що вони визначені в одному модулі. `hosting` та `serving` є братами, визначеними всередині `front_of_house`. Якщо модуль A міститься всередині модуля B, ми кажемо, що модуль A є *нащадком* (*child*) модуля B, а модуль B є *батьком* (*parent*) модуля A. Зверніть увагу, що батьком усього дерева модулів є неявний модуль з назвою `crate`.

Дерево модулів може нагадувати вам дерево тек і файлів файлової системи на вашому комп'ютері. Це дуже влучне порівняння! Ви можете використовувати модулі для організації коду точно так само, як ви використовуєте теки у файловій системі. І так само, як у випадку з файлами в теці, нам потрібен спосіб пошуку необхідних модулів.

Шлях для Доступу до Елементів у Дереві Модулів

Щоб вказати Rust, де шукати елемент у дереві модулів, ми використовуємо шляхи так само як ми використовуємо шляхи для навігації по файловій системі. Щоб викликати функцію, ми повинні знати її шлях.

Шлях може приймати дві форми:

- *Абсолютний шлях* це повний шлях, що починається в кореневій директорії крейту; для коду від зовнішнього крейту, абсолютний шлях починається з назви крейту, і для коду з поточного ящика починається з рядка `crate`.
- *Відносний шлях* починається у поточному модулі і використовує `self`, `super` чи ідентифікатор поточного модуля.

І абсолютні, і відносні шляхи складаються з одного чи кількох ідентифікаторів, розділених подвійною двокрапкою (`::`).

Повернімося до Блоку коду 7-1. Скажімо, ми хочемо викликати функцію `add_to_waitlist`. Це те саме, що й запитати: який шлях до функції `add_to_waitlist`? Блок коду 7-3 містить Блок коду 7-1, але деякі з модулів та функцій прибрані.

Ми покажемо два способи викликати функцію `add_to_waitlist` з нової функції `eat_at_restaurant`, визначеної в корені крейта. Ці шляхи є правильними, але залишилася інша проблема, яка перешкоджає компілюванню цього прикладу "як є". Ми пояснимо, чому, трохи пізніше.

Функція `eat_at_restaurant` є частиною публічного API нашого бібліотечного крейта, тому ми позначимо її ключевим словом `pub`. Детальніше про `pub` йтиметься у підрозділі "Надання доступу до шляхів за допомогою ключового слова `<1>pub</1>` .

Файл: `src/lib.rs`

```
mod front_of_house {
    mod hosting {
        fn add_to_waitlist() {}
    }
}

pub fn eat_at_restaurant() {
    // Absolute path
    crate::front_of_house::hosting::add_to_waitlist();

    // Relative path
    front_of_house::hosting::add_to_waitlist();
}
```

Блок коду 7-3: виклик функції `add_to_waitlist` за допомогою абсолютного та відносного шляхів

Коли ми вперше ми викликаємо функцію `add_to_waitlist` в `eat_at_restaurant`, то використовуємо абсолютний шлях. Функція `add_to_waitlist` визначена у тому ж крейті, що й `eat_at_restaurant`, тобто ми можемо використати ключове слово `crate` на початку абсолютного шляху. Потім ми додаємо кожен з вкладених модулів, доки не вкажемо весь шлях до `add_to_waitlist`. Уявіть собі файлову систему з такою ж структурою: ми повинні вказати шлях `/front_of_house/hosting/add_to_waitlist`, щоб запустити програму `add_to_waitlist`; використання назви `crate`, щоб почати з кореня, схожий на використання `/`, щоб почати шлях з кореня файлової системи у вашій оболонці.

Коли ми вдруге викликаємо `add_to_waitlist` у `eat_at_restaurant`, то використовуємо відносний шлях. Шлях починається з `front_of_house`, назви модуля, визначеного на тому ж рівні дерева модулів, що й `eat_at_restaurant`. Тут аналогом з файлової системи буде використання шляху `front_of_house/hosting/add_to_waitlist`. Початок з назви модуля означає, що шлях є відносним.

Рішення, використовувати відносний або абсолютний шлях, вам доведеться робити, виходячи з від вашого проєкту, і залежить від того, чи код, що визначає елемент, окремо від коду, що використовує його, чи разом. Наприклад, якщо ми перемістимо модуль `front_of_house` і функцію `eat_at_restaurant` у модуль `customer_experience`, нам знадобиться оновити абсолютний шлях до `add_to_waitlist`, але відносний шлях усе ще буде коректним. Однак, якби ми перенесли функцію `eat_at_restaurant` окремо до модуля з назвою `dining`, абсолютний шлях до виклику `add_to_waitlist` залишаться таким самим, але відносний шлях треба буде оновити. Загалом, ми вважаємо за краще вказувати абсолютні шляхи, тому що з більшою ймовірністю ми захочемо перемістити код визначення та виклики елементів незалежно один від одного.

Спробуймо скомпілювати Блок коду 7-3 і дізнатися, чому він досі не компілюється! Помилка, що ми отримуємо, показана у Блоці коду 7-4.


```

$ cargo build
  Compiling restaurant v0.1.0 (file:///projects/restaurant)
error[E0603]: module `hosting` is private
  --> src/lib.rs:9:28
   |
9  |         crate::front_of_house::hosting::add_to_waitlist();
   |                                ^^^^^^^^ private module
   |
note: the module `hosting` is defined here
  --> src/lib.rs:2:5
   |
2  |     mod hosting {
   |     ^^^^^^^^^^^^^

error[E0603]: module `hosting` is private
  --> src/lib.rs:12:21
   |
12 |         front_of_house::hosting::add_to_waitlist();
   |                     ^^^^^^^^ private module
   |
note: the module `hosting` is defined here
  --> src/lib.rs:2:5
   |
2  |     mod hosting {
   |     ^^^^^^^^^^^^^

For more information about this error, try `rustc --explain E0603`.
error: could not compile `restaurant` due to 2 previous errors

```

Блок коду 7-4: помилки компілятора при збірці коду в Блоці коду 7-3

Повідомлення про помилки кажуть, що модуль `hosting` є приватним. Іншими словами, ми маємо коректні шляхи для модуля `hosting` і функції `add_to_waitlist`, але Rust не дозволяє нам використовувати їх, бо немає доступу до приватних частин. У Rust усі елементи (функції, методи, структури, енуми, модулі і константи) за замовчуванням є приватними в батьківських модулях. Якщо ви хочете зробити елемент на кшталт функції чи структури приватним, то розміщуєте його у модулі.

Елементи батьківського модуля не можуть використовувати приватні елементи дочірніх модулів, але елементи дочірніх модулів можуть використовувати елементи у модулях-предках. Так зроблено, щоб дочірні модулі огортали і ховали деталі своєї реалізації, але дочірні модулі можуть бачити контекст, у якому їх визначено. Щоб розвинути нашу метафору, уявіть собі правила приватності як бек-офіс ресторану: те, що там відбувається, недоступно для клієнтів ресторану, але менеджери можуть бачити і робити все у ресторані, яким вони керують.

У Rust вирішено зробити модульну систему, де деталі реалізації є прихованими за замовчуванням. Таким чином, ви знаєте, які частини внутрішнього коду ви можете

змінити, не зламавши зовнішній код. Однак Rust надає вам можливість виставити внутрішні частини коду дочірніх модулів для зовнішніх модулів-предків за допомогою ключового слова `pub`, щоб зробити елемент публічним.

Надання Публічного Доступу до Шляхів із Ключовим Словом `pub`

Повернімося до помилки у Блоці коду 7-4, яка каже нам, що модуль `hosting` є приватним. Ми хочемо, щоб функція `eat_at_restaurant` в батьківському модулі мала доступ до функції `add_to_waitlist` в дочірньому модулі, тож ми позначили модуль `hosting` за допомогою ключового слова `pub`, як показано в Блоці коду 7-5.

Файл: `src/lib.rs`

```
mod front_of_house {
    pub mod hosting {
        fn add_to_waitlist() {}
    }
}

pub fn eat_at_restaurant() {
    // Absolute path
    crate::front_of_house::hosting::add_to_waitlist();

    // Relative path
    front_of_house::hosting::add_to_waitlist();
}
```

Блок коду 7-5: проголошення модуля `hosting` як `pub`, щоб використовувати його з `eat_at_restaurant`

На жаль, код у Блоці коду 7-5 все ще призводить до помилки, як це показано в Блоці коду 7-6.

```

$ cargo build
   Compiling restaurant v0.1.0 (file:///projects/restaurant)
error[E0603]: function `add_to_waitlist` is private
  --> src/lib.rs:9:37
  |
9 |         crate::front_of_house::hosting::add_to_waitlist();
  |                                     ^^^^^^^^^^^^^^^^^^^^^ private function
  |
note: the function `add_to_waitlist` is defined here
  --> src/lib.rs:3:9
  |
3 |         fn add_to_waitlist() {}
  |         ^^^^^^^^^^^^^^^^^^^^^
  |

error[E0603]: function `add_to_waitlist` is private
  --> src/lib.rs:12:30
  |
12 |         front_of_house::hosting::add_to_waitlist();
  |                                ^^^^^^^^^^^^^^^^^^^^^ private function
  |
note: the function `add_to_waitlist` is defined here
  --> src/lib.rs:3:9
  |
3 |         fn add_to_waitlist() {}
  |         ^^^^^^^^^^^^^^^^^^^^^
  |

For more information about this error, try `rustc --explain E0603`.
error: could not compile `restaurant` due to 2 previous errors

```

Блок коду 7-6: помилки компілятора від збірки коду у Блоці коду 7-5

Що сталося? Додавання ключового слова `pub` перед `mod hosting` робить модуль публічним. Після цієї зміни, якщо ми маємо доступ `front_of_house`, ми можемо отримати доступ до `hosting`. Але *вміст* `hosting` все ще є приватним; зробивши модуль публічним, ми робимо публічним його вміст. Ключове слово `pub` для модуля дозволяє коду в модулях-предках тільки посилатися на нього, а не мати доступ до його внутрішнього коду. Оскільки модулі є контейнерами, ми багато не зробимо, лише зробивши модуль публічним; ми маємо піти далі і також зробити ще один або більше елементів модуля публічними.

Помилки у Блоці коду 7-6 кажуть, що функція `add_to_waitlist` є приватною. Правила приватності застосовуються до структур, еnumів, функцій і методів, як і до модулів.

Також зробимо публічною функцію `add_to_waitlist`, додавши ключове слово `pub` перед її визначенням, як у Блоці коду 7-7.

Файл: `src/lib.rs`

```
mod front_of_house {
    pub mod hosting {
        pub fn add_to_waitlist() {}
    }
}

pub fn eat_at_restaurant() {
    // Absolute path
    crate::front_of_house::hosting::add_to_waitlist();

    // Relative path
    front_of_house::hosting::add_to_waitlist();
}
```

Блок коду 7-7: Додавання ключового слова `pub` до `mod hosting` і `fn add_to_waitlist` дозволяє нам викликати функцію з `eat_at_restaurant`

Тепер код скомпілюється! Щоб побачити, чому додавання ключового слова `pub` дозволяє нам використовувати ці шляхи у `add_to_waitlist` відповідно до правил приватності, розгляньмо абсолютні та відносні шляхи.

Абсолютний шлях ми починаємо з `crate`, кореня дерева модулів нашого крейта. Модуль `front_of_house` визначено в корені крейта. Оскільки функція `eat_at_restaurant` визначена в тому ж модулі, що й `front_of_house` (тобто, `eat_at_restaurant` та `front_of_house` є сестрами), то поки `front_of_house` не є публічним, ми можемо посилатися на `front_of_house` лише з `eat_at_restaurant`. Наступний модуль `hosting` позначений як `pub`. Ми маємо доступ до батьківського модуля `hosting`, тож маємо доступ до `hosting`. Нарешті, функція `add_to_waitlist` позначена як `pub` і ми маємо доступ до її батьківського модуля, тож виклик функції працює!

У відносному шляху логіка така ж сама як і в абсолютному, окрім першого кроку: замість починати з кореня крейта, шлях починається з `front_of_house`. Модуль `front_of_house` визначено в тому ж модулі, що й `eat_at_restaurant`, тому відносний шлях, що починається з модуля, в якому визначено `eat_at_restaurant`, працює. Потім, оскільки `hosting` і `add_to_waitlist` позначені як `pub`, решта шляху працює, і цей виклик функції - коректний!

Якщо ви плануєте поділитися своєю бібліотекою, щоб інші проекти могли використовувати ваш код, ваш публічний API - це ваш контракт з користувачами вашого крейта, який визначає, як вони можуть взаємодіяти з вашим кодом. Існує багато міркувань щодо управління змінами у вашому публічному API для полегшення залежності від Вашого крейта. Ці міркування не лежать за межами цієї книжки; якщо вам цікава ця тема, дивіться [Керівництво з API Rust](https://pavloslav.github.io/RustBookUkr/print.html).

Кращі Практики для Пакетів з Двійковим Крейтом і Бібліотекою

Ми згадували, що пакунок може містити одночасно корінь як двійкового крейта `src/main.rs`, так і корінь бібліотечного крейта `src/lib.rs`, і обидва крейти матимуть за замовчуванням назву пакету. Зазвичай, пакунки, створені за таким шаблоном, з бібліотекою і двійковим крейтом, матимуть у двійковому крейті лише код, потрібний для запуску виконуваного коду з бібліотечного крейта. Це дозволяє іншим проектам отримувати максимум функціоналу, який надає пакунок, бо бібліотечний крейт можна використовувати спільно.

Дерево модулів має бути визначеним в `src/lib.rs`. Тоді будь-які публічні елементи можна використовувати у двійковому крейті, починаючи шлях з назви пакунку. Двійковий крейт стає таким самим користувачем бібліотечного крейта, як і абсолютно зовнішній крейт, що використовує бібліотечний крейт: він може користуватися лише публічним API. Це допомагає вам розробити хороший API; ви не лише його автор, але також і користувач!

У [Розділі 12](#) ми покажемо цю практику організації крейта у програмі командного рядка, що міститиме як двійковий крейт, так і бібліотечний крейт.

Початок Відносних Шляхів з `super`

Ми можемо створювати відносні шляхи, які починаються в батьківському модулі, а не в поточному чи корені крейта, застосувавши `super` на початку шляху. Це схоже на `..` на початку шляху в файловій системі. За допомогою `super` ми можемо посилатися на елемент, що, як ми знаємо, знаходиться в батьківському модулі, що спрощує зміну дерева модулів, коли модуль є тісно пов'язаним із батьком, але батьківський елемент може бути переміщений в інше місце дерева модулів.

Розглянемо код у Блоці коду 7-8, який моделює ситуацію, в якій шеф-кухар виправляє неправильне замовлення і особисто приносить його клієнту. Функція `fix_incorrect_order`, визначена у модулі `back_of_house` викликає функцію `deliver_order`, визначену в батьківському модулі, вказавши шлях до `deliver_order`, починаючи з `super`:

Файл: `src/lib.rs`

```
fn deliver_order() {}

mod back_of_house {
    fn fix_incorrect_order() {
        cook_order();
        super::deliver_order();
    }

    fn cook_order() {}
}
```

Блок коду 7-8: виклик функції за допомогою відносного шляху, що починається з `super`

Функція `fix_incorrect_order` знаходиться в модулі `back_of_house`, тож ми можемо використати `super`, щоб перейти до батьківського модуля `back_of_house`, який у цьому випадку є коренем, `crate`. Звідси ми шукаємо `deliver_order` і знаходимо її. Успіх! Ми гадаємо, що модуль `back_of_house` і функція `deliver_order` найімовірніше залишатимуться у такому відношенні одне до одного і будуть переміщені разом, якщо ми вирішимо реорганізувати дерево модулів крейта. Таким чином, ми скористалися `super`, щоб мати менше місць, де треба буде для оновлювати код у майбутньому, якщо цей код перемістять в інший модуль.

Робимо Структури та Енуми Публічними

Також ми можемо використовувати `pub` для визначення структур та еnumів публічними, але є додаткові особливості використання `pub` зі структурами та еnumами. Якщо ми використовуємо `pub` перед визначенням структури, ми робимо структуру публічною, але поля структури все одно будуть приватними. Ми можемо зробити публічним чи ні кожне поле окремо в кожному конкретному випадку. У Блоці коду 7-9 ми визначили публічну структуру `back_of_house::Breakfast` з публічним полем `toast`, але приватним полем `seasonal_fruit`. Це моделює ситуацію в ресторані, коли покупець може обрати тип хліба, що додається до їжі, але кухар вирішує, які фрукти йдуть до їжі залежно від сезону і наявності. Доступні фрукти швидко змінюються, тому клієнти не можуть вибрати фрукти і навіть побачити, які фрукти вони отримають.

Файл: `src/lib.rs`

```

mod back_of_house {
    pub struct Breakfast {
        pub toast: String,
        seasonal_fruit: String,
    }

    impl Breakfast {
        pub fn summer(toast: &str) -> Breakfast {
            Breakfast {
                toast: String::from(toast),
                seasonal_fruit: String::from("peaches"),
            }
        }
    }
}

pub fn eat_at_restaurant() {
    // Order a breakfast in the summer with Rye toast
    let mut meal = back_of_house::Breakfast::summer("Rye");
    // Change our mind about what bread we'd like
    meal.toast = String::from("Wheat");
    println!("I'd like {} toast please", meal.toast);

    // The next line won't compile if we uncomment it; we're not allowed
    // to see or modify the seasonal fruit that comes with the meal
    // meal.seasonal_fruit = String::from("blueberries");
}

```

Блок коду 7-9: структура, деякі поля якої є публічними, а деякі приватними

Оскільки поле `toast` у структурі `back_of_house::Breakfast` є публічним, у `eat_at_restaurant` ми можемо писати та читати поле `toast`, використовуючи точку. Зверніть увагу, що ми не можемо використовувати поле `seasonal_fruit` у `eat_at_restaurant`, тому що `seasonal_fruit` є приватним. Спробуйте розкоментувати рядок, що змінює значення поля `seasonal_fruit`, щоб подивитися, яку помилку ви отримуєте!

Крім того, зауважте, що оскільки `back_of_house::Breakfast` має приватне поле, структура має надавати публічну асоційовану функцію, що створює екземпляр `Breakfast` (тут ми назвали її `summer`). Якби `Breakfast` не мав такої функції, ми не могли б створити екземпляр `Breakfast` у `eat_at_restaurant`, бо не могли б виставити значення приватного поля `seasonal_fruit` у `eat_at_restaurant`.

На відміну від цього, якщо ми робимо еnum публічним, усі його варіанти є публічними. Потрібно лише одне ключове слово `pub` перед `enum`, як показано в Блоці коду 7-10.

Файл: `src/lib.rs`

```
mod back_of_house {  
    pub enum Appetizer {  
        Soup,  
        Salad,  
    }  
}  
  
pub fn eat_at_restaurant() {  
    let order1 = back_of_house::Appetizer::Soup;  
    let order2 = back_of_house::Appetizer::Salad;  
}
```

Блок коду 7-10: позначення енума публічним робить публічними усі його варіанти

Оскільки ми зробили еnum `Appetizer` публічним, то можемо використовувати варіанти `Soup` та `Salad` у `eat_at_restaurant`.

Енуми не дуже корисні, коли їхні варіанти не є публічними; було б набридливим анотувати всі варіанти енуму як `pub` у будь-якому випадку, то за замовчуванням варіанти переліку є публічними. Структури часто є корисними без публічних полів, тож поля структур слідує загальному правилу, що все є приватним за замовчуванням, якщо не анотовано як `pub`.

Є ще одна ситуація, пов'язана з `pub`, про яку ми не розповіли, і це остання деталь системи модулів: ключове слово `use`. Ми спершу розповімо про `use`, а потім покажемо, як комбінувати `pub` і `use`.

Введення Шляхів до Області Видимості з Ключовим Словом `use`

Необхідність переписувати шляхи для виклику функцій може здатися незручною та повторюваною. У Блоці коду 7-7 незалежно від того, чи ми вказували абсолютний чи відносний шлях до функції `add_to_waitlist`, для того, щоб її викликати, ми кожного разу мали також вказувати `front_of_house` та `hosting`. На щастя, існує спосіб спростити цей процес: достатньо один раз створити ярлик (shortcut) для шляху за допомогою ключового слова `use` і потім використовувати коротке ім'я будь-де в області видимості.

У Блоці коду 7-11 ми підключаємо модуль `crate::front_of_house::hosting` до області видимості функції `eat_at_restaurant`, отже нам лишається лише вказати `hosting::add_to_waitlist` для виклику функції `add_to_waitlist` всередині `eat_at_restaurant`.

Файл: `src/lib.rs`

```
mod front_of_house {
    pub mod hosting {
        pub fn add_to_waitlist() {}
    }
}

use crate::front_of_house::hosting;

pub fn eat_at_restaurant() {
    hosting::add_to_waitlist();
}
```

Блок коду 7-11: введення модуля до області видимості за допомогою `use`

Додання `use` та шляху до області видимості схоже на створення символічного посилання (symbolic link) у файловій системі. При додаванні `use crate::front_of_house::hosting` в корені крейта, `hosting` стає коректним ім'ям в цій області видимості, так як би модуль `hosting` був визначений в корені крейта. Шляхи, додані до області видимості за допомогою `use`, також перевіряються на приватність, як і будь-які інші.

Зауважте, що `use` лише створює ярлик для конкретної області видимості, в якій знаходиться цей самий `use`. Лістинг 7-12 переносить функцію `eat_at_restaurant` до нового дочірнього модуля `customer`, що має відмінну від `use` область видимості, а отже, тіло функції зкомпільовано не буде:

Файл: src/lib.rs

```
mod front_of_house {
    pub mod hosting {
        pub fn add_to_waitlist() {}
    }
}

use crate::front_of_house::hosting;

mod customer {
    pub fn eat_at_restaurant() {
        hosting::add_to_waitlist();
    }
}
```

Блок коду 7-12: інструкція `use` стосується лише області видимості, в якій вона знаходиться

Помилка компілятора показує, що даний ярлик більше не дійсний в модулі `customer`:

```
$ cargo build
   Compiling restaurant v0.1.0 (file:///projects/restaurant)
error[E0433]: failed to resolve: use of undeclared crate or module `hosting`
  --> src/lib.rs:11:9
   |
11 |         hosting::add_to_waitlist();
   |         ^^^^^^^ use of undeclared crate or module `hosting`

warning: unused import: `crate::front_of_house::hosting`
  --> src/lib.rs:7:5
   |
 7 | use crate::front_of_house::hosting;
   |     ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
   |
   = note: `#[warn(unused_imports)]` on by default

For more information about this error, try `rustc --explain E0433`.
warning: `restaurant` (lib) generated 1 warning
error: could not compile `restaurant` due to previous error; 1 warning emitted
```

Зверніть увагу також на попередження компілятора, що `use` не використовується у власній області видимості! Для вирішення цієї проблеми треба перемістити `use` до модуля `customer`, або послатися на його ярлик у батьківському модулі за допомогою `super::hosting` всередині дочірнього модуля `customer`.

Створення Ідіоматичних Шляхів `use`

У Блоці коду 7-11 у вас могло виникнути питання, чому ми вказали `use crate::front_of_house::hosting` і потім викликали `hosting::add_to_waitlist` в `eat_at_restaurant` замість вказання в `use` повного шляху до функції `add_to_waitlist` для отримання того самого результату, що й у Блоці коду 7-13.

Файл: `src/lib.rs`

```
mod front_of_house {
    pub mod hosting {
        pub fn add_to_waitlist() {}
    }
}

use crate::front_of_house::hosting::add_to_waitlist;

pub fn eat_at_restaurant() {
    add_to_waitlist();
}
```

Блок коду 7-13: Додання функції `add_to_waitlist` до області видимості за допомогою `use`, що не є ідіоматичним способом

Хоча Блоки коду 7-11 та 7-13 і виконують одну й ту саму задачу, Блок коду 7-11 є ідіоматичним способом додавання функції до області видимості за допомогою `use`. Щоб додати батьківський модуль функції до області видимості з `use` треба його вказати при виклику функції. Вказання батьківського модуля при виклику функції явно показує, що функція не оголошена локально, але разом з тим це зводить до мінімуму необхідність повторень повного шляху. З коду в Блоці коду 7-13 не ясно, де саме визначено `add_to_waitlist`.

З іншого боку при додаванні структур, переліків та інших елементів за допомогою `use`, вказання повного шляху є ідіоматичним. Блок коду 7-14 демонструє ідіоматичний спосіб для додавання стандартної структури з бібліотеки `HashMap` до області видимості бінарного крейту.

Файл: `src/main.rs`

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

fn main() {
    let mut map = HashMap::new();
    map.insert(1, 2);
}
```

Listing 7-14: Bringing `HashMap` into scope in an idiomatic way

За цією ідіомою немає якоїсь вагомої причини: це просто згода серед програмістів на Rust, які звикли писати і читати код саме таким чином.

Винятком з цієї ідіоми є випадок, коли треба підключити два елементи з однаковими іменами до області видимості з оператором `use`, оскільки Rust не дозволяє зробити це. Лістинг 7-15 демонструє як підключити до області видимості два типи `Result`, що мають однакове ім'я, але різні батьківські модулі, та як до них звертатися.

Файл: `src/lib.rs`

```
use std::fmt;
use std::io;

fn function1() -> fmt::Result {
    // --snip--
}

fn function2() -> io::Result<()> {
    // --snip--
}
```

Блок коду 7-15: Підключення до області видимості двох типів з одним ім'ям вимагає вказання їх батьківських модулів.

Як ви можете бачити, використання батьківських модулів розрізняє два типи `Result`. Якщо б натомість ми вказали `use std::fmt::Result` та `use std::io::Result`, ми б мали два типи `Result` в одній області видимості та Rust не знав би, який з них ми маємо на увазі, пишучи `Result`.

Надання Нових Назв з Ключовим Словом `as`

Існує також інше рішення проблеми використання двох типів з одним ім'ям в одній області видимості з `use`: після шляху можна вказати `as` та нове локальне ім'я, або *alias* для даного типу. Лістинг 7-16 показує інший спосіб написання коду з Лістинга 7-15, перейменувавши один з двох типів `Result` за допомогою `as`.

Файл: `src/lib.rs`

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

fn function1() -> Result {
    // --snip--
}

fn function2() -> IoResult<()> {
    // --snip--
}
```

Блок коду 7-16: Перейменування типу при його додаванні до області видимості з ключовим словом `as`

У другому операторі `use` ми вказали нове ім'я `IoResult` для типу `std::io::Result`, що не конфліктуватиме з типом `Result` з `std::fmt`, що ми ойго також додали до області видимості. Підходи з Блоків коду 7-15 та 7-16 вважаються ідіоматичними. Отже, вибір за вами!

Реекспорт Назв із `pub use`

При внесенні імені до області видимості із ключовим словом `use`, ім'я, доступне в новій області видимості, є приватним. Аби код міг посилатися на це ім'я так, ніби воно визначене в його області видимості, ми можемо комбінувати `pub` та `use`. Ця техніка називається *re-exporting*. тому що ми не лише додаємо елемент до області видимості, а ще й робимо його доступним для підключення в інші області видимості.

Блок коду 7-17 показує код з Блока коду 7-11, в якому `use` в кореневому модулі замінено на `pub use`.

Файл: `src/lib.rs`

```
mod front_of_house {
    pub mod hosting {
        pub fn add_to_waitlist() {}
    }
}

pub use crate::front_of_house::hosting;

pub fn eat_at_restaurant() {
    hosting::add_to_waitlist();
}
```

Блок коду 7-17: Робимо назву доступною для використання будь-яким кодом з нової області видимості за допомогою `pub use`

До цієї заміни зовнішній код повинен був викликати функцію `add_to_waitlist`, використовуючи шлях `restaurant::front_of_house::hosting::add_to_waitlist()`. Тепер, коли використання `pub use` дозволило реекспортувати модуль `hosting` з кореневого модуля, зовнішній код може натомість використовувати шлях `restaurant::hosting::add_to_waitlist()`.

Реекспорт є корисним, коли внутрішня структура коду відрізняється від того, як програмісти, що викликають ваш код, думають про предметну область. Наприклад, в нашій ресторанній метафорі люди, що керують рестораном, сприймають його як внутрішню кухню та зал. В той час як відвідувачі ресторану, можливо, не сприймають ресторан в таких само термінах. Із `pub use` ми можемо писати код у вигляді однієї структури, проте виставляти його назовні у вигляді іншої. Завдяки цьому наша бібліотека лишається добре організованою для програмістів, які будуть з нею працювати. Ми також розглянемо інший приклад використання `pub use` і як це впливає на вашу документацію крейту в частині “Експорт зручного публічного API із `pub use`” розділу 14.

Використання Зовнішніх Пакетів

У Розділі 2 ми написали гру у вгадування чисел, яка використовувала зовнішній пакет під назвою `rand` для отримання випадкових чисел. Для використання `rand` в нашому проекті ми додали наступний рядок до *Cargo.toml*:

Файл: Cargo.toml

```
rand = "0.8.5"
```

Вказання `rand` в якості залежності до *Cargo.toml* каже Cargo завантажити пакет `rand` та всі залежності з crates.io та зробити `rand` доступним для нашого проекту.

Потім, для того щоб додати `rand` до області видимості нашого пакету, ми додали рядок `use`, що починався з імені крейту `rand` та перелічили елементи, які ми хочемо додати до області видимості. Згадайте, що в секції “Генерація випадкового числа” розділу 2 ми додали трейт `Rng` до області видимості і викликали функцію `rand::thread_rng`:

```
use rand::Rng;

fn main() {
    let secret_number = rand::thread_rng().gen_range(1..=100);
}
```

Члени Rust спільноти зробили доступними багато пакетів, які доступні на crates.io, і додання будь-якого з них до вашого пакету вимагає тих самих кроків: вказання їх у файлі *Cargo.toml* вашого пакету та використання `use` для додання елементів з їх крейтів до області видимості.

Зверніть увагу, що стандартна бібліотека `std` є також крейтом, що є зовнішнім по відношенню до нашого пакету. Оскільки стандартна бібліотека поставляється в комплекті з мовою Rust, нам не портібно змінювати *Cargo.toml* для додання `std`. Але нам потрібно вказати її за допомогою `use` для того щоб додати її елементи до області видимості нашого пакету. Наприклад, для `HashMap` ми б використовували такий рядок:

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

Це абсолютний шлях, що починається з `std`, імені крейту стандартної бібліотеки.

Використання Вкладених Шляхів для Зменшення Великих Переліків

`use`

Якщо нам треба використовувати багато елементів, визначених в тому самому крейті або модулі, вказання кожного з них на окремому рядку займає багато вертикального простору в файлах. Наприклад, ці два оголошення `use` ми використовували у грі вгадування чисел в Блоці коду 2-4 для додавання до області видимості елементів з `std`:

Файл: `src/main.rs`

```
// --snip--  
use std::cmp::Ordering;  
use std::io;  
// --snip--
```

Натомість, ми можемо використовувати вкладені шляхи для того щоб додати ці елементи до області видимості лише одним рядком. Для цього ми вказуємо спільну частину шляху, за якою йдуть дві двокрапки, а потім фігурні дужки навколо переліку частин шляхів, що відрізняються, як показано в Лістингу 7-18.

Файл: `src/main.rs`

```
// --snip--  
use std::{cmp::Ordering, io};  
// --snip--
```

Блок коду 7-18: Вказання вкладених шляхів для додання до області видимості елементів з однаковими префіксами

У більших програмах додання багатьох елементів до області видимості з одного крейту або модулю за допомогою вкладених шляхів може значно скоротити кількість необхідних використань `use`!

Ми можемо використовувати вкладені шляхи будь-якого рівня вкладеності, що є корисним при комбінуванні двох виразів `use`, що мають спільну частину шляху. Наприклад, Блок коду 7-19 демонструє два оператори `use`: один додає до області видимості `std::io` і один, що додає `std::io::Write`.

Файл: `src/lib.rs`

```
use std::io;
use std::io::Write;
```

Блок коду 7-19: Два оператори `use`, де один є частиною іншого

Спільною частиною цих двох шляхів є `std::io`, і це ж є повним шляхом першого. Для об'єднання цих двох шляхів в один оператор `use` ми можемо використати ключове слово `self` у вкладеному шляху, як показано в Блоці коду 7-20.

Файл: `src/lib.rs`

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

Блок коду 7-20: Комбінування шляхів з Блока коду 7-19 в одному операторі `use`

Цей рядок додає `std::io` та `std::io::Write` до області видимості.

Узагальнений Імпорт

Якщо ми хочемо додати до області видимості *всі* публічні елементи, визначені за певним шляхом, то ми можемо вказати шлях, за яким йтиме символ узагальнення `*`:

```
use std::collections::*;
```

Цей оператор `use` додає до області видимості всі публічні елементи, визначені в `std::collections`. Будьте обережні, використовуючи глобальний оператор! Це може ускладнити сприйняття коду, оскільки стає важче визначити, які імена є в області видимості і де саме було визначено певне ім'я, що використовується у вашій програмі.

Глобальний оператор часто використовується при тестуванні для включення до області видимості всіх елементів з модуля `tests`. Ми поговоримо про це пізніше у секції “Як писати тести” розділу 11. Глобальний оператор також інколи використовується як частина патерну Прелюдія (prelude): див. [документацію по стандартній бібліотеці](#) для отримання додаткової інформації по цьому патерну.

Розподіл Модулів на Різні Файли

Поки що всі приклади в цій главі визначали декілька модулів у одному файлі. Коли модулі стають великими, ви можете захотіти перемістити їх визначення в окремі файли, щоб спростити навігацію по коду.

Наприклад, почнімо з коду із Лістинга 7-17, у якому було декілько модулів ресторану. Ми будемо вилучати модулі у файли замість того, щоб визначати всі модулі в кореновому модулі крейта. У нашому випадку кореневий модуль крейта - *src/lib.rs*, але цей розподіл також працює з бінарними крейтами, у яких кореневий модуль крейта - *src/main.rs*.

Спочатку ми вилучимо модуль `front_of_house` в свій власний файл. Видаліть код всередині фігурних дужок для модуля `front_of_house`, залишив тільки визначення `mod front_of_house;` так щоб тепер *src/lib.rs* містив код, показаний в Блоці коду 7-21. Зверніть увагу, що цей варіант не скомпілюється, поки ми не створимо файл *src/front_of_house.rs* з Лістинга 7-22.

Файл: *src/lib.rs*

```
mod front_of_house;

pub use crate::front_of_house::hosting;

pub fn eat_at_restaurant() {
    hosting::add_to_waitlist();
}
```

Блок коду 7-21: Визначення модуля `front_of_house` чий вміст буде у *src/front_of_house.rs*

Далі, розмістимо код, котрий був у фігурних дужках, у новий файл з ім'ям *src/front_of_house.rs*, як показано у Блоці коду 7-22. Компілятор знає, що потрібно шукати у цьому файлі, тому що він натрапив у кореновому модулі крейту на визначення модуля з ім'ям `front_of_house`.

Файл: *src/front_of_house.rs*

```
pub mod hosting {
    pub fn add_to_waitlist() {}
}
```

Блок коду 7-22: Визначення вмісту модуля `front_of_house` у файлі *src/front_of_house.rs*

Зверніть увагу, що вам потрібно тільки **один** раз завантажити файл за допомогою оголошення *mod* у вашому дереві модулів. Як тільки компілятор дізнається, що файл є частиною проекту (та дізнається, де в дереві модулів знаходиться код за допомогою того, де ви розмістили оператор **mod**), інші файли у вашому проекті повинні посилатися на код завантаженого файлу, використовуючи шлях до місця, де він був оголошений, як описано у секції [Шляхи для посилання на елемент у дереві модулів](#). Іншими словами, **mod** - це *не* операція “включення”, яку ви могли бачати в інших мовах програмування.

Далі ми вилучимо модуль **hosting** в його власний файл. Процес трохи відрізняється, тому що **hosting** є дочірнім модулем для **front_of_house**, а не кореневого модуля. Ми помістимо файл для **hosting** в нову директорію, який буде іменований на ім'я його предка в дереві модулів, у цьому випадку це *src/front_of_house/*.

Щоб почати перенесення **hosting**, ми змінюєм *src/front_of_house.rs* таким чином, щоб він одержав тільки визначення модуля **hosting**:

Файл: *src/front_of_house.rs*

```
pub mod hosting;
```

Далі ми створюємо директорію *src/front_of_house* та файл *hosting.rs*, у якому будуть визначення у модулі **hosting**:

Файл: *src/front_of_house/hosting.rs*

```
pub fn add_to_waitlist() {}
```

Якщо замість цього ми розмістимо *hosting.rs* у директорію *src*, компілятор буде думати, що код в *hosting.rs* це модуль **hosting**, визначений у корні крейта, а не визначений як дочірній модуль **front_of_house**. Правила компілятора для перевірки того, які файли містять код яких модулів, припускають, що директорії та файли точно відповідають дереву модулів.

Альтернативні Шляхи до Файлів

Досі ми розглядали найбільш ідіоматичні шляхи до файлів, які використовуються компілятором Rust, але Rust також підтримує старий стиль шляхів до файлу. Для модуля з ім'ям **front_of_house** визначеного в кореновому модулі крейту, компілятор буде шукати код модуля в:

- *src/front_of_house.rs* (стиль, що ми розглядали)

- `src/front_of_house/mod.rs` (старий стиль, який все ще підтримується)

Для модуля з ім'ям `hosting`, який є підмодулем `front_of_house`, компілятор буде шукати код модуля в:

- `src/front_of_house/hosting.rs` (стиль, що ми розглядали)
- `src/front_of_house/hosting/mod.rs` (старий стиль, який все ще підтримується)

Якщо ви використовуєте обидва стилі для одного й того ж модуля, ви отримаєте помилку компілятора. Використання суміші обох стилів для різних модулів у одному проекті дозволено, але це може збивати з пантелику людей, що переміщаються по вашому проекту.

Основним недоліком стиля, в якому використовуються файли з іменами `mod.rs` є те, що у вашому проекті можуть опинитися багато файлів з іменами `mod.rs`, що можуть призвести до плутанини, якщо ви одночасно відкриєте їх в редакторі.

Ми перенесли код кожного модуля в окремий файл, а дерево модулів залишилось без змін. Виклики функцій в `eat_at_restaurant` будуть працювати без яких-небудь змін, не дивлячись на те, що визначення знаходяться у різних файлах. Цей метод дозволяє переміщати модулі в нові файли в міру збільшення їх розмірів.

Зверніть увагу, що оператор `pub use crate::front_of_house::hosting` у `src/lib.rs` також не змінився, та `use` не впливає на те, які файли компілюються як частина крейта. Ключове слово `mod` визначає модулі, і Rust шукає в файлі з таким же ім'ям, що й у модуля, який входить у цей модуль.

Підсумок

Rust дозволяє розбити пакет на декілька крейтів, та крейт - на модулі, таким чином ви маєте змогу посилатися на елементи, визначенні в одному модулі, з іншого модуля. Це можна робити за допомогою вказання абсолютних чи відносних шляхів. Ці шляхи можна додати в область видимості оператором `use`, тому ми можете користуватися коротшими шляхами для багаторазового використання елементів у цій області видимості. Код модуля за замовчуванням є приватним, але можна зробити визначення загальнодоступними, додавши ключове слово `pub`.

У наступній главі ми подивимося деякі колекції структур даних із стандартної бібліотеки, які ви можете використовувати в своєму охайно організованому коді.

Звичайні Колекції

Стандартна бібліотека Rust містить декілька дуже корисних структур даних, що звуться *колекції*. Більшість інших типів даних представляють одне певне значення, але колекції можуть містити багато значень. На відміну від вбудованих типів масив і кортеж, дані, на які вказують ці колекції, зберігаються на купі, тобто кількість даних не має бути обов'язково відомою під час компіляції і може збільшуватися або скорочуватися під час виконання програми. Кожен вид колекції має різні можливості і недоліки, і вибір відповідної колекції для поточної ситуації - це вміння, що ви розвиваєте з часом. У цьому розділі ми обговоримо три колекції, які дуже часто використовуються в програмах Rust:

- *Вектор* дозволяє зберігати змінну кількість значень поруч одне з одним.
- *Стрічка* є колекцією символів. Ми вже згадували тип `String`, але в цьому розділі ми поговоримо про нього глибше.
- *Хеш-мапа* дозволяє пов'язати значення з певним ключем. Це конкретна реалізація більш загальної структури даних, що називається *відображенням* (`map`).

Щоб дізнатися про інші види колекцій стандартної бібліотеки зверніться до [документації](#).

Ми обговоримо, як створювати та оновлювати вектори, стрічки, хеш-мапи, а також те, що робить їх особливими.

Зберігання Списків Значень з Векторами

Перший тип колекцій, який ми розглянемо - це `Vec<T>`, також відомий як *вектор*. Вектори дозволяють вам зберігати більше одного значення в єдиній структурі даних, що розташовує ці значення поруч один з одним у пам'яті. Вектор може зберігати лише значення одного типу. Вони корисні, коли ви маєте список предметів, наприклад рядки тексту у файлі або ціни на товари у кошику.

Створення Нового Вектора

Щоб створити новий порожній вектор, ми викликаємо `Vec::new`, як показано в Блоці коду 8-1.

```
let v: Vec<i32> = Vec::new();
```

Блок коду 8-1: створення нового порожнього вектора для зберігання значень типу `i32`

Зауважте, що тут ми додали анотації типу. Оскільки ми не вставляємо жодного значення в цей вектор, Rust не знає, які елементи ми маємо намір зберігати. Це важлива деталь. Вектори реалізовані за допомогою узагальнень; ми розкажемо, як використовувати узагальнення з вашими власними типами в Розділі 10. Наразі треба знати лише, що тип `Vec<T>`, наданий стандартною бібліотекою, може містити будь-який тип. Коли ми створюємо вектор, що міститиме певний тип, ми можемо зазначити тип у кутових дужках. У Блоці коду 8-1 ми кажемо Rust, що `Vec<T>` у `v` міститиме елементи типу `i32`.

Зазвичай ви створюватимете `Vec<T>` з початковими значеннями, і Rust виведе тип значень, які ви хочете зберігати, тож вам нечасто буде потрібно додавати таку анотацію типу. Rust для зручності надає макрос `vec!`, який створює новий вектор, який містить ваші значення. Блок коду 8-2 створює новий `Vec<i32>`, що містить значення `1`, `2`, і `3`. Тип цілих - `i32`, бо це тип цілих за замовчуванням, як ми вже говорили в підрозділі “Типи даних” Розділу 3.

```
let v = vec![1, 2, 3];
```

Блок коду 8-2: створення нового вектора, що містить значення

Оскільки ми надали початкові значення `i32`, Rust може вивести, що типом `v` є `Vec<i32>` і анотація типу тут не потрібна. Далі ми поглянемо, як змінити вектор.

Оновлення Вектора

Щоб створити вектор і додати до нього елементи ми можемо використати метод `push`, як показано в Блоці коду 8-3.

```
let mut v = Vec::new();

v.push(5);
v.push(6);
v.push(7);
v.push(8);
```

Блок коду 8-3: використання методу `push` для додавання значень у вектор

Як і для будь-якої змінної, якщо ми хочемо змінювати її значення, ми повинні зробити його мутабельним за допомогою ключового слова `mut`, як говорилося в Розділі 3. Числа, як ми розміщуємо у векторі, мають тип `i32`, і Rust виводить це з даних, тож нам не потрібна анотація `Vec<i32>`.

Читання Елементів Векторів

Існує два способи послатися на значення, що зберігається у векторі: через індексацію або використовуючи метод `get`. У наступних прикладах ми анотували типи значень, які повертаються з цих функцій, для додаткової ясності.

Блок коду 8-4 показує обидва методи доступу до значення у векторі - синтаксис індексування і метод `get`.

```
let v = vec![1, 2, 3, 4, 5];

let third: &i32 = &v[2];
println!("The third element is {third}");

let third: Option<&i32> = v.get(2);
match third {
    Some(third) => println!("The third element is {third}"),
    None => println!("There is no third element."),
}
```

Блок коду 8-4: використання синтаксису індексів або методу `get` для доступу до елементів вектора

Зверніть тут увагу на декілька деталей. Ми використовуємо значення індексу `2`, щоб отримати третій елемент, бо вектори індексуються числами, починаючи з нуля.

Використання `&i` надає нам посилання на елемент за значенням індексу. Коли ми

використовуємо метод `get` з індексом, переданим аргументом, то отримуємо `Option<T>`, який ми можемо використати у `match`.

Rust надає ці два способи посилання на елемент, щоб ви могли вибрати, як програма поводитися при спробі використовувати значення індексу поза діапазоном наявних елементів. Як приклад, подивімося, що станеться, коли ми матемо вектор з п'яти елементів, а потім спробуємо отримати доступ до елемента з індексом 100 за допомогою кожної техніки, як показано в Блоці коду 8-5.

```
let v = vec![1, 2, 3, 4, 5];  
  
let does_not_exist = &v[100];  
let does_not_exist = v.get(100);
```

Блок коду 8-5: спроба доступу до елемента з індексом 100 у векторі, що містить п'ять елементів

Коли ми запустимо цей код, перший метод `[]` призведе до паніки програми через те, що він посилається на елемент, якого не існує. Цей метод найкраще використовувати, коли ви хочете, щоб програма аварійно завершилася, якщо сталася спроба отримати доступ до елемента за кінцем вектора.

Коли методу `get` передається індекс, що знаходиться поза вектором, він повертає `None` без паніки. Цей метод краще використовувати, якщо доступ до елемента за межами вектора може ставатися час від часу за нормальних умов. Ваш код тоді міститиме логіку обробки як `Some(&element)`, так і `None`, як пояснюється в Розділі 6. Наприклад, індекс може бути отримано від людини, що вводить число. Якщо хтось випадково введе завелике число і програма отримає значення `None`, чи можете повідомити користувачеві, скільки елементів є у векторі надати йому ще одну спробу ввести коректне значення. Це буде більш дружньо до користувача, ніж аварійне завершення програми через хибодрук!

Коли у програми є посилання, borrow checker забезпечує правила володіння і позичання (про які йдеться у Розділі 4), забезпечуючи, що це посилання та будь-які інші посилання на вміст вектора залишаються коректними. Згадайте правило, яке каже, що не можна мати мутабельні і немутабельні посилання в одній області видимості. Це правило застосовується в Блоці коду 806, де ми тримаємо немутабельне посилання на перший елемент вектора і намагаємося додати елемент у кінець. Ця програма не спрацює, якщо ми спробуємо звернутися до цього елемента пізніше у функції:


```
let mut v = vec![1, 2, 3, 4, 5];

let first = &v[0];

v.push(6);

println!("The first element is: {first}");
```

Блок коду 8-6: спроба додати елемент до вектора, тримаючи посилання на його елемент

Компіляція цього коду завершиться з такою помилкою:

```
$ cargo run
   Compiling collections v0.1.0 (file:///projects/collections)
error[E0502]: cannot borrow `v` as mutable because it is also borrowed as
immutable
--> src/main.rs:6:5
   |
4  |     let first = &v[0];
   |                  - immutable borrow occurs here
5  |
6  |     v.push(6);
   |     ^^^^^^^^^ mutable borrow occurs here
7  |
8  |     println!("The first element is: {first}");
   |                                           ----- immutable borrow later used here

For more information about this error, try `rustc --explain E0502`.
error: could not compile `collections` due to previous error
```

Код у Блоці коду 8-6, можливо, має вигляд, ніби він повинен працювати: чому посилання на перший елемент має турбуватися про зміни в кінці вектора? Ця помилка відбувається через те, як працюють вектори: оскільки вектори тримають значення поруч одне з одним у пам'яті, додавання нового елемента в кінець вектора може вимагати виділення нової пам'яті та копіювання старих елементів у нове місце, якщо там, де наразі зберігається вектор, недостатньо місця, щоб тримати всі елементи один біля одного. У такому разі посилання на перший елемент вказуватиме на звільнену пам'ять. Правила позичання перешкоджають програмі опинитися в такій ситуації.

Примітка: Для деталей імплементації типу `Vec<T>` дивіться ["Растономікон"](#).

Ітерування над Значеннями Вектора

Для доступу до кожного елемента вектора по черзі ми ітеруємо по всіх елементах замість використання індексів для доступу по одному за раз. Блок коду 8-7 показує, як використовувати цикл `for`, щоб отримати немутабельні посилання на кожен елемент вектора значень `i32` і вивести їх.

```
let v = vec![100, 32, 57];
for i in &v {
    println!("{i}");
}
```

Блок коду 8-7: виведення кожного елемента вектора ітеруванням по елементах у циклі `for`

Ми також можемо ітерувати по мутабельних посиланнях на кожен елемент у мутабельному векторі, щоб змінити всі елементи. Цикл `для` у Блоці коду 8-8 додасть `50` до кожного елемента.

```
let mut v = vec![100, 32, 57];
for i in &mut v {
    *i += 50;
}
```

Блок коду 8-8: Ітерування по мутабельних посиланнях на елементи у векторі

Щоб змінити значення, на яке посилається мутабельне посилання, нам потрібно скористатися оператором розіменування `*` для отримання значення в `i` до того, як ми зможемо використовувати оператор `+=`. Ми поговоримо більше про оператора розіменування у підрозділі "[Перехід за вказівником до значення](#)" Розділу 15.

Ітерування по вектору, мутабельне чи немутабельне, є безпечним завдяки правилам borrow checker. Якби ми спробували вставити або видалити елементи в циклі `for` у Блоці коду 8-7 і Блоці коду 8-8, то отримали б помилку компілятора, схожу на той, що ми отримали з кодом у Блоці коду 8-6. Посилання на вектор, яке тримає цикл `for`, запобігає одночасній зміні усього вектора.

Використання Енума для Зберігання Декількох Типів

Вектор може зберігати лише значення одного типу. Це може бути незручно; точно існують випадки, коли є потреба у зберіганні списку елементів різних типів. На щастя, варіанти енума визначені як один тип, тож коли нам потрібен один тип для представлення елементів різних типів, ми можемо визначити і використовувати еnum!

Наприклад, нехай ми хочемо отримати значення з рядка в таблиці, у якій деякі стовпці в рядку містять цілі числа, деякі — числа з рухомими точками, а деякі — рядки. Ми можемо визначити еnum, варіанти якого будуть містити різні типи значень, і всі варіанти еnumа будуть вважатися одним і тим же типом — еnumом. Тоді ми можемо створити вектор, який міститиме цей еnum і, зрештою, міститиме різні типи. Ми продемонстрували це у Блоці коду 8-9.

```
enum SpreadsheetCell {  
    Int(i32),  
    Float(f64),  
    Text(String),  
}  
  
let row = vec![  
    SpreadsheetCell::Int(3),  
    SpreadsheetCell::Text(String::from("blue")),  
    SpreadsheetCell::Float(10.12),  
];
```

Блок коду 8-9: Визначення `enum` для зберігання значень різних типів у одному векторі

Rust має знати, які типи будуть у векторі, під час компіляції, щоб знати, скільки саме пам'яті у купі буде потрібно для зберігання кожного елемента. Ми також маємо явно зазначити, які типи можуть бути в цьому векторі. Якби Rust дозволив вектору містити будь-який тип, була б імовірність, що один або кілька з типів призведуть до помилок при виконанні операцій на елементах вектора. Використання еnumу і виразу `match` означає, що Rust гарантує під час компіляції, що усі можливі випадки буде оброблено, як обговорювалося в Розділі 6.

Якщо ви не маєте вичерпного списку типів, з якими програма працюватиме під час виконання для зберігання у векторі, техніка еnumів не спрацює. Натомість ви можете скористатися трейтовими об'єктами, про які йдеться у Розділі 17.

Тепер, коли ми обговорили деякі найпоширеніші способи використання векторів, обов'язково подивитися [документацію API](#) щоб дізнатися про багато інших корисних методів, визначених для `Vec<T>` у стандартній бібліотеці. Наприклад, на додачу до методу `push`, метод `pop` видаляє і повертає останній елемент.

Очищення Вектора Очищує Його Елементи

Як і будь-яка інша `struct`, вектор вивільняється, коли виходить з області видимості, як підписано в Блоці коду 8-10.

```
{  
    let v = vec![1, 2, 3, 4];  
  
    // do stuff with v  
} // <- v goes out of scope and is freed here
```

Блок коду 8-10: демонстрація, де саме вектор і його елементи очищуються

Коли вектор очищується, також очищується і його вміст, тобто цілі числа, які він містить, будуть очищені. Borrow checker гарантує, що будь-які посилання на вміст вектора використовуються лише поки сам вектор є коректним.

Перейдімо до наступного типу колекцій: **String**!

Зберігання Тексту у Кодуванні UTF-8 в Стрічках

Ми говорили про стрічки в Розділі 4, але тепер ми розглянемо їх глибше. Нові растацеанці зазвичай застрягають на стрічках через поєднання трьох причин: схильність Rust до виявлення можливих помилок, стрічки є складнішою структурою даних, ніж вважають багато програмістів, та UTF-8. Ці фактори комбінуються таким чином, що може здатися вам складним, коли ви переходите з інших мов програмування.

Ми обговорюємо стрічки в контексті колекцій, оскільки стрічки реалізовані як колекція байтів, плюс деякі методи для надання корисної функціональності, коли ці байти розглядаються як текст. У цьому підрозділі ми поговоримо про операції на `String`, які має кожен тип колекцій, такі як створення, змінна та читання. Ми також обговоримо, чим `String` відрізняється від інших колекцій, а саме чому індексування `String` ускладнене відмінністю між тим, як люди і комп'ютери розглядають дані у `String`.

Що Таке Стрічка?

Спочатку ми визначимо те, що ми маємо на увазі під терміном *стрічка*. Rust має лише один стрічковий тип у ядрі мови, а саме стрічковий слайс `str`, який зазвичай буває у запозиченій формі `&str`. У Розділі 4 ми говорили про *стрічкові слайси*, які є посиланням на деякі дані, закодовані в UTF-8, які зберігаються деінде. Наприклад, стрічкові літерали зберігаються в двійковому файлі програми і, отже, є стрічковими слайсами.

Тип `String`, який надається стандартною бібліотекою Rust, а не закодовано в ядро мови, може зростати, бути мутабельним, володіє своїми даними і кодований в UTF-8. Коли растцеанці посилається на "стрічки" в Rust, то можуть посилатись або на тип `String`, або на стрічковий слайс `&str`, а не лише один із цих типів. Хоча цей розділ багато в чому стосується `String`, обидва типи щедро використовуються в стандартній бібліотеці Rust, і як `String`, так і стрічкові слайси мають кодування UTF-8.

Створення Нової Стрічки

Багато операцій, доступних для `Vec<T>`, також доступні для `String`, тому що `String` фактично реалізований як обгортка навколо вектора байтів з деякими додатковими гарантіями, обмеженнями і можливостями. Приклад функції, яка працює однаково як і з `Vec<T>` і `String`, це функція `new`, що створює екземпляр, як показано в Блоці коду 8-11.

```
let mut s = String::new();
```

Блок коду 8-11: Створення нової порожньої `String`

Цей рядок створює нову порожню стрічку, що називається `s`, в яку ми надалі можемо завантажити дані. Часто ми матимемо деякі початкові дані, які ми хочемо одразу розмістити в стрічці. Для цього ми використовуємо метод `to_string`, який доступний для будь-якого типу, що реалізує трейт `Display`, як, зокрема, стрічкові літерали. Блок коду 8-12 показує два приклади.

```
let data = "initial contents";

let s = data.to_string();

// the method also works on a literal directly:
let s = "initial contents".to_string();
```

Блок коду 8-12: використання методу `to_string` для створення `String` зі стрічкового літерала

Цей код створює стрічку, що містить `початковий вміст`.

Також ми можемо скористатися функцією `String::from`, щоб створити `String` зі стрічкового літерала. Код у Блоці коду 8-13 еквівалентний коду зі Блоку коду 8-12, який використовує `to_string`.

```
let s = String::from("initial contents");
```

Блок коду 8-13: використання функції `String::from` для створення `String` зі стрічкового літерала

Оскільки стрічки використовуються для великої кількості речей, ми можемо використати багато різних узагальнених API для стрічок, що надають нам багато варіантів. Деякі з них можуть здатись надлишковими, але всі вони мають своє власне місце! У цьому випадку `String::from` і `to_string` роблять те саме, тому, що ви оберете є питанням стилю і читабельності.

Пам'ятайте, що стрічки мають кодування UTF-8, тож ми можемо включити у них будь-які правильно закодовані дані, як показано в Блоці коду 8-14.

```
let hello = String::from("السلام عليكم");
let hello = String::from("Dobry den");
let hello = String::from("Hello");
let hello = String::from("הי");
let hello = String::from("नमस्ते");
let hello = String::from("こんにちは");
let hello = String::from("안녕하세요");
let hello = String::from("你好");
let hello = String::from("Olá");
let hello = String::from("Привіт");
let hello = String::from("Hola");
```

Блок коду 8-14: збереження вітань різними мовами у стрічках

Усе це коректні значення `String`.

Оновлення Стрічки

`String` може зростати і її вміст може змінюватися, як вміст `Vec<T>`, якщо ви додасте у неї ще дані. На додачу, ви можете для зручності використовувати оператор `+` чи макрос `format!` для конкатенації значень `String`.

Додавання до Стрічки Методами `push_str` і `push`

Ми можемо збільшити `String` за допомогою методу `push_str`, додавши стрічковий слайс, як показано в Блоці коду 8-15.

```
let mut s = String::from("foo");
s.push_str("bar");
```

Блок коду 8-15: додавання стрічкового слайсу до `String` за допомогою методу `push_str`

Після цих двох рядків `s` міститиме `foobar`. Метод `push_str` приймає стрічковий слайс, бо ми не обов'язково хочемо приймати володіння параметром. Наприклад, у коді з Блоку коду 8-16, ми хочемо мати змогу використовувати `s2` після додавання його вмісту до `s1`.

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

Блок коду 8-16: використання стрічкового слайсу після додавання його вмісту до `String`

Якби метод `push_str` взяв володіння `s2`, ми б не змогли вивести його значення в останньому рядку. Але цей код працює, як ми очікуємо!

Метод `push` приймає параметром один символ і додає його до `String`. Блок коду 8-17 додає букву "l" до `String` за допомогою методу `push`.

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

Блок коду 8-17: Додавання одного символу до значення `String` за допомогою `push`

У результаті, `s` міститиме `lol`.

Конкатенація Оператором `+` або Макросом `format!`

Часто вам треба поєднати дві стрічки. Один зі способів зробити це - використати оператор `+`, як показано в Блоці коду 8-18.

```
let s1 = String::from("Hello, ");  
let s2 = String::from("world!");  
let s3 = s1 + &s2; // note s1 has been moved here and can no longer be used
```

Блок коду 8-18: використання оператора `+` для об'єднання двох значень `String` у нову `String`

Стрічка `s3` міститиме `Hello, world!`. Причина, чому `s1` більше не дійсна після додавання, і причина, чому ми використовували посилання на `s2`, стосується сигнатури методу, викликаного, коли ми скористалися оператором `+`. Оператор `+` використовує метод `add`, сигнатура якого виглядає приблизно так:

```
fn add(self, s: &str) -> String {
```

У стандартній бібліотеці ви побачите `add`, визначений за допомогою узагальнених і асоційованих типів. Тут ми підставили конкретні типи, що й стається, коли ми викликаємо цей метод зі значеннями `String`. Узагальнені типи ми обговоримо у Розділі 10. Ця сигнатура дає нам підказки, потрібні для розуміння тонких місць оператора `+`.

По-перше, `s2` має `&`, що означає, що ми додаємо *посилання* на другу стрічку до першої стрічки. Так зроблено через параметр `s` у функції `add`: ми можемо лише додати `&str` до `String`; ми не можемо скласти разом два значення `String`. Але чекайте — типом `&s2` є `&String`, а не `&str`, як зазначено в другому параметрі `add`. То чому ж Блок коду 8-18 компілюється?

Причина, з якої ми можемо використовувати `&s2` у виклику `add` полягає в тому, що компілятор може *привести* аргумент `&String` до `&str`. Коли ми викликаємо метод `add`, Rust використовує *приведення розіменування*, що тут перетворює `&s2` у `&s2[..]`. Ми обговоримо приведення розіменування глибше в Розділі 15. Оскільки `add` не перебирає володіння параметром `s`, `s2` все ще буде коректною `String` після цієї операції.

По-друге, як ми бачимо в сигнатурі, `add` бере володіння над `self`, бо `self` не має `&`. Це означає, що `s1` у Блоці коду 8-18 буде перенесено у виклику `add` і після цього більше не буде дійсним. Таким чином, хоч `let s3 = s1 + &s2;` виглядає, ніби він копіює обидві стрічки і створює нову, ця інструкція насправді бере володіння `s1`, додає копію вмісту `s2`, а потім повертає володіння результатом. Іншими словами, він виглядає, ніби створює багато копій, але насправді ні; реалізація ефективніша за копіювання.

Якщо нам потрібно об'єднати декілька стрічок, поведінка оператора `+` стає громіздкою:

```
let s1 = String::from("tic");
let s2 = String::from("tac");
let s3 = String::from("toe");

let s = s1 + "-" + &s2 + "-" + &s3;
```

У цьому місці `s` буде `tic-tac-toe`. За усіма цими символами `+` і `"` стає важко побачити, що відбувається. Для складнішого комбінування стрічок ми можемо замість цього скористатися макросом `format!`:

```
let s1 = String::from("tic");
let s2 = String::from("tac");
let s3 = String::from("toe");

let s = format!("{s1}-{s2}-{s3}");
```

Цей код також надає `s` значення `tic-tac-toe`. Макрос `format!` працює подібно до `println!`, але замість виведення на екран, повертає `String` з відповідним вмістом. Версію коду, що використовує `format!`, значно простіше читати, і код, що згенеровано макросом `format!`, використовує посилання, тому цей виклик не перебирає володіння жодним зі своїх параметрів.

Індексація Стрічок

У багатьох інших мовах програмування доступ до окремих символів у стрічці за допомогою індексу є припустимою і поширеною операцією. Однак, якщо ви спробуєте отримати шматки `String` за допомогою синтаксису індексів у Rust, то отримаєте помилку. Розглянемо неправильний код у Блоці коду 8-19.

```
let s1 = String::from("hello");
let h = s1[0];
```

Блок коду 8-19: спроба використати синтаксис індексів для String

Цей код призведе до наступної помилки:

```
$ cargo run
   Compiling collections v0.1.0 (file:///projects/collections)
error[E0277]: the type `String` cannot be indexed by `{integer}`
--> src/main.rs:3:13
   |
3  |     let h = s1[0];
   |               ^^^^^ `String` cannot be indexed by `{integer}`
   |
= help: the trait `Index<{integer}>` is not implemented for `String`
= help: the following other types implement trait `Index<Idx>`:
       <String as Index<RangeFrom<usize>>>
       <String as Index<RangeFull>>
       <String as Index<RangeInclusive<usize>>>
       <String as Index<RangeTo<usize>>>
       <String as Index<RangeToInclusive<usize>>>
       <String as Index<std::ops::Range<usize>>>
       <str as Index<I>>

For more information about this error, try `rustc --explain E0277`.
error: could not compile `collections` due to previous error
```

Помилка і примітка кажуть самі за себе: стрічки у Rust не підтримують індексацію. Але чому ні? Щоб відповісти на це запитання, нам потрібно обговорити, як Rust зберігає стрічки в пам'яті.

Внутрішнє Представлення

`String` є обгорткою `Vec<u8>`. Подивімося на деякі зразки правильно кодованих у UTF-8 стрічок з Блоку коду 8-14. Спершу, цей:

```
let hello = String::from("Hola");
```

У цьому випадку `len` буде 4, це означає, що вектор, що зберігає стрічку "Hola", має довжину 4 байти. Кожна з цих літер займає 1 байт в кодуванні в UTF-8. Однак наступний рядок може вас здивувати.

```
let hello = String::from("Привіт");
```

Якщо вас спитати, якої довжини стрічка, ви можете сказати 6. Насправді Rust відповість, що 12 - це число байтів, потрібних, щоб закодувати "Привіт" у UTF-8, оскільки кожен скалярне значення Unicode у цій стрічці займає 2 байти пам'яті. Таким чином, індекс по байтах стрічки не завжди буде співвідноситися з припустимим скалярним значенням Unicode. Для демонстрації, розгляньмо цей некоректний код Rust:

```
let hello = "Привіт";
let answer = &hello[0];
```

Ви вже знаєте, що `answer` не буде `п`, першою літерою. У кодуванні UTF-8 перший байт `п` буде `208`, а другий `159`, тож здається, що `answer` має бути `208`, але `208` сам по собі не є допустимим символом. Швидше за все, користувач, що запитує першу літеру у стрічці, не хоче отримати `208`; однак за індексом 0 Rust має лише ці дані. Користувачі, як правило, не хочуть отримувати значення байтів, навіть якщо стрічка містить лише латинські літери: якби `&"hello"[0]` було коректним кодом, що повертає значення байта, він усе одно поверне `104`, а не `h`.

Отже, відповідь полягає в тому, щоб уникнути повертання неочікуваного значення, що спричинить помилки, які можуть не бути виявлені негайно, і Rust узагалі не компілює цей код, запобігаючи непорозумінням на ранньому етапі процесу розробки.

Байти, Скалярні Значення та Кластери Графем! Божечки!

Ще одна особливість UTF-8 полягає у тому, що насправді існує три способи поглянути на стрічки з точки зору Rust: як на байти, скалярні значення та графеми кластери (найближче до того, що ми називаємо *літерами*).

Скажімо, слово мовою Гінді “नमस्ते”, записане письмом деванагарі, зберігається як вектор значень `u8`, що виглядає ось так:

```
[224, 164, 168, 224, 164, 174, 224, 164, 184, 224, 165, 141, 224, 164, 164,
224, 165, 135]
```

Це 18 байтів і так комп'ютери кінець-кінцем зберігають ці дані. Якщо ми подивимось на них як на скалярні значення Unicode, тобто те, чим є тип `char` у Rust, ці байти виглядають наступним чином:

```
['न', 'म', 'स्', 'ते', '।', '।']
```

Тут є шість значень `char`, але четвертий і шостий — не літери: це діакритичні знаки, які самостійно не мають жодного сенсу. Нарешті, якщо ми подивимось на них як на кластери

графем, то отримаємо те, що людина назвала б чотирма літерами, які складають слово на Гінді:

```
["न", "म", "स्", "ते"]
```

Rust надає різні способи інтерпретації необроблених даних стрічок, збережених комп'ютером, щоб кожна програма могла обрати потрібну їй інтерпретацію, неважливо якою людською мовою є ці дані.

Останньою причиною, чому Rust не дозволяє нам індексувати `String` для отримання символу, є те, що очікується, що операція індексації завжди займає постійний час ($O(1)$). Але неможливо гарантувати таку продуктивність для `String`, тому що Rust повинен проглянути вміст з початку до індексу, щоб визначити, скільки там було валідних символів.

Створення Стрічкових Слайсів

Індексація стрічки часто є поганою ідеєю, бо не зрозуміло, яке значення має повертати операція індексування: байт, символ, кластер графем чи стріковий слайс. Тому, якщо вам дійсно треба використати індекси для створення стрічкових слайсів, Rust просить вас бути більш конкретним.

Замість індексування за допомогою `[]` з одним числом, ви можете використовувати `[]` з діапазоном, щоб створити стрічковий слайс, що містить певні байти:

```
let hello = "Привіт";  
  
let s = &hello[0..4];
```

Тут `s` буде `&str`, що містить перші 4 байти стрічки. Раніше ми згадували, що кожен з цих символів має 2 байти, а це означає, що `s` буде `Пр`.

Якби ми спробували створити слайс з частини байтів символу чимось на кшталт `&hello[0..1]`, то Rust запанікував би під час виконання, так само, якби ми задали неправильний індекс для доступу до вектора:

```
$ cargo run
Compiling collections v0.1.0 (file:///projects/collections)
Finished dev [unoptimized + debuginfo] target(s) in 0.43s
Running `target/debug/collections`
thread 'main' panicked at 'byte index 1 is not a char boundary; it is inside '3'
(bytes 0..2) of `Здравствуйтє`', src/main.rs:4:14
note: run with `RUST_BACKTRACE=1` environment variable to display a backtrace
```

Ви маєте використовувати діапазони для створення стрічкових слайсів з обережністю, тому що так може зламати програму.

Методи для Ітерації над Стрічками

Найкращий спосіб оперувати фрагментами стрічок - це чітко визначити, потрібні вам символи чи байти. Для роботи з окремими скалярними значеннями Unicode використовуйте метод `chars`. Виклик `chars` для "Пр" виділяє і повертає два значення типу `char`, і ви можете ітерувати результатом для доступу до кожного елемента:

```
for c in "Пр".chars() {
    println!("{c}");
}
```

Цей код виводить на екран наступне:

```
П
р
```

Альтернатива, метод `bytes` повертає кожен необроблений байт, що може бути більш придатним для вашої задачі:

```
for b in "Пр".bytes() {
    println!("{b}");
}
```

Цей код виведе чотири байти, з яких складається ця стрічка:

```
208
159
209
128
```

Але не забувайте, що припустимі скалярні значення Unicode можуть бути складені з більш ніж 1 байту.

Отримання кластерів графом зі стрічок, як у письмі деванагарі, є складним, тому ця функціональність не надається стандартною бібліотекою. На crates.io є відповідні крейти, якщо ви потребуєте такого функціонала.

Стрічки Не Такі Прості

Підсумуємо: стрічки є складними. Різні мови програмування роблять різні вибори стосовно того, як представити цю складність програмісту. Rust обрав коректну обробку даних у `String` поведінкою за замовчуванням для всіх програм Rust, що означає, що програмісти повинні краще продумувати заздалегідь обробку даних UTF-8. Цей компроміс розкриває більшу частину складності стрічок, ніж зазвичай в інших мовах програмування, але це убезпечує вас від помилок із символами поза межами ASCII пізніше у життєвому циклі розробки.

Доброю новиною є те, що стандартна бібліотека пропонує велику кількість функціонала, побудованого на типах `String` і `&str`, щоб допомогти правильно впоратися з цими складними ситуаціями. Обов'язково перевірте документацію про корисні методи, такі як `contains` для пошуку в стрічці і `replace` на заміну частин стрічки на іншу стрічку.

Перейдемо до чогось менш складного: хеш-мапи!

Зберігання Ключів з Асоційованими Значеннями у Хеш-Мапах

Остання з поширених колекцій - це *хеш-мапа*. Тип `HashMap<K, V>` зберігає відображення ключів типу `K` на значення типу `V`, використовуючи *функцію хешування*, яка визначає, як розмістити ці ключі та значення у пам'яті. Багато мов програмування підтримують таку структуру даних, але часто використовують іншу назву, таку як хеш, відображення, хеш-таблиця, словник або асоціативний масив, це тільки декілька назв.

Хеш-мапи корисні, коли ви хочете шукати дані не за індексом, як у векторах, а за допомогою ключа довільного типу. Наприклад, у грі ви можете відстежувати результат кожної команди за допомогою хеш-мапи, в якій кожен ключ - це назва команди, а значення - її рахунок. Маючи назву команди, ви можете отримати її рахунок.

У цьому розділі ми розглянемо базовий API хеш-мап, але у функціях, визначених на `HashMap<K, V>` стандартною бібліотекою, ховається ще багато цікавого. Як завжди, зверніться до документації стандартної бібліотеки для додаткової інформації.

Створення Нової Хеш-Мапи

Один зі способів створення порожньої хеш-мапи є використання `new` і додавання елементів за допомогою `insert`. У Блоці коду 8-20 ми відстежуємо рахунки двох команд, що називаються *Синя* та *Жовта*. Синя команда починає з 10 очками, а Жовта - з 50.

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

let mut scores = HashMap::new();

scores.insert(String::from("Blue"), 10);
scores.insert(String::from("Yellow"), 50);
```

Блок коду 8-20: Створення нової хеш-мапи і вставлення деяких ключів та значень

Зверніть увагу, що нам, по-перше, треба зробити `use` для `HashMap` з розділу колекцій стандартної бібліотеки. З трьох наших загальних колекцій ця використовується найрідше, тож вона не включена до функціонала, який автоматично додається до області видимості у прелюдії. Хеш-мапи також мають меншу підтримку від стандартної бібліотеки; наприклад, немає вбудованого макросу для їх побудови.

Так само як і вектори, хеш-мапи зберігають свої дані у купі. Цей `HashMap` має ключі типу `String` і значення типу `i32`. Як і вектори, хеш-мапи є однорідними: усі ключі мають бути одного типу, і всі значення мають бути одного типу.

Доступ до Значень Хеш-Мапи

Ми можемо отримати значення з хеш-мапи, надавши її ключ методу `get`, як показано у Блоці коду 8-21.

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

let mut scores = HashMap::new();

scores.insert(String::from("Blue"), 10);
scores.insert(String::from("Yellow"), 50);

let team_name = String::from("Blue");
let score = scores.get(&team_name).copied().unwrap_or(0);
```

Блок коду 8-21: Доступ до рахунку Синьої команди, що зберігається у хеш-мапі

Тут `score` буде мати значення, пов'язане з Синьою командою, і результат буде `10`. Метод `get` повертає `Option<V>`; якщо у хеш-мапі для цього ключа немає відповідного значення, `get` поверне `None`. Ця програма обробляє `Option` викликом `copied`, отримуючи `Option<i32>`, а не `Option<&i32>`, а тоді `unwrap_or`, щоб встановити `score` у нуль, якщо `scores` не має запису для цього ключа.

Ми можемо ітерувати по кожній парі ключ/значення в хеш-мапі подібно до того, як ми це робимо з векторами, використовуючи цикл `for`:

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

let mut scores = HashMap::new();

scores.insert(String::from("Blue"), 10);
scores.insert(String::from("Yellow"), 50);

for (key, value) in &scores {
    println!("{key}: {value}");
}
```

Цей код виведе в консолі кожну пару в довільному порядку:

```
Yellow: 50
Blue: 10
```


Хеш-Мапи та Володіння

Для типів, що реалізують трейт `Copy`, таких як `i32`, значення копіюються у хеш-мапу. Для значень, які мають володіння, таких як `String`, значення будуть переміщені і хешмапа буде володіти цими значеннями, як це показано у Блоці коду 8-22.

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

let field_name = String::from("Favorite color");
let field_value = String::from("Blue");

let mut map = HashMap::new();
map.insert(field_name, field_value);
// field_name and field_value are invalid at this point, try using them and
// see what compiler error you get!
```

Блок коду 8-22: Демонстрація, що хеш-мапа володіє ключем та значенням після їх вставлення

Ми не можемо використовувати змінні `field_name` і `field_value` після того, як вони були переміщені до хеш-мапи за допомогою виклику `insert`.

Якщо ми вставляємо посилання на значення до хешмапи, ці значення не будуть переміщені до хешмапи. Значення, на які вказують посилання, мають бути коректними щонайменше стільки ж, скільки існує хешмапа. Ми поговоримо про це докладніше у підрозділі "Перевірка коректності посилань за допомогою часів існування" Розділу 10.

Оновлення Хеш-Мапи

Хоча кількість пар ключів і значень зростає, кожен унікальний ключ може мати тільки одне значення, пов'язане з ним, в кожен момент (але не навпаки: наприклад, команда Синя і Жовта могли мати значення 10, збережене в хешмапі `scores`).

Коли ви хочете змінити дані в хешмапі, вам необхідно вирішити, як обробляти випадок, коли ключ уже має присвоєне значення. Ви можете замінити старе значення на нове значення, повністю проігнорувати старе значення. Ви можете зберегти старе значення і проігнорувати нове значення, і лише коли ключ *не має* значення, додавати нове. Або ж ви можете поєднати старе значення і нове значення. Подивімося, як зробити кожен варіант!

Перезапис Значення

Якщо ми вставляємо ключ і значення до хешмапи і тоді вставляємо той самий ключ із іншим значенням, то значення, асоційоване з цим ключем, буде замінено. Попри те, що код у Блоці коду 8-23 викликає `insert` двічі, хешмапа міститиме лише одну пару ключ/значення, оскільки ми обидва рази вставляємо значення для ключа Синьої команди.

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

let mut scores = HashMap::new();

scores.insert(String::from("Blue"), 10);
scores.insert(String::from("Blue"), 25);

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

Блок коду 8-23: заміна значення, збереженого з певним ключем

Цей код виведе `{"Blue": 25}`. Початкове значення `10` було перезаписане.

Додавання Ключа та Значення Тільки якщо Ключ Відсутній

Доволі поширено перевіряти, чи певний ключ уже присутній у хешмапі зі значенням, а тоді якщо ключ існує, то наявне значення має залишатися таким, яким воно є. Якщо ж ключ відсутній, вставити його і значення для нього.

Хешмапи мають спеціальне API для цього, що зветься `entry`, яке приймає параметром ключ, який ви хочете перевірити. Значення, що повертається з методу `entry` - це еnum, що зветься `Entry`, який представляє значення, що може існувати або не існувати. Скажімо, ми хочемо перевірити, чи ключ для Жовтої команди має пов'язане з ним значення. Як ні, ми хочемо вставити значення 50, і те саме для Синьої команди. За допомогою API `entry`, код стає схожим на Блок коду 8-24.

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

let mut scores = HashMap::new();
scores.insert(String::from("Blue"), 10);

scores.entry(String::from("Yellow")).or_insert(50);
scores.entry(String::from("Blue")).or_insert(50);

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

Блок коду 8-24: використання методу `entry` для вставляння лише якщо ключ ще не має відповідного значення

Метод `or_insert` для `Entry` за визначенням повертає мутабельне посилання на відповідний ключ `Entry`, якщо ключ існує, а як ні, то вставляє параметр як нове значення для цього ключа і повертає мутабельне посилання на нове значення. Ця техніка набагато чистіша, ніж написання логіки самостійно і ще, крім того, краще працює з borrow checker.

Запуск коду у Блоці коду 8-24 надрукує `{"Yellow": 50, "Blue": 10}`. Перший виклик `entry` вставить ключ для Жовтої команди зі значенням 50, бо Жовта команда ще не має свого значення. Другий виклик `entry` не змінить хешмапу, бо Синя команда вже має значення 10.

Оновлення Значення на Основі Старого Значення

Інший поширений сценарій використання хешмап - пошук значення ключа і оновлення його на основі старого значення. Наприклад, Блок коду 8-25 показує код, який підраховує, скільки разів кожне слово з'являється в певному тексті. Ми використовуємо хешмапу з ключами - словами і збільшуємо значення, щоб відстежувати, скільки разів ми бачили це слово. Якщо ми зустрічаємо слово уперше, то спершу вставляємо значення 0.

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

let text = "hello world wonderful world";

let mut map = HashMap::new();

for word in text.split_whitespace() {
    let count = map.entry(word).or_insert(0);
    *count += 1;
}

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

Блок коду 8-25: підрахунок кількості слів за допомогою хешмапи слів, що зберігає слова і кількість

Цей код виведе `{"world": 2, "hello": 1, "wonderful": 1}`. Ви можете побачити ті ж пари ключів/значень, виведені в іншому порядку: згадайте з підрозділу ["Доступ до значень у хешмапі"](#), що ітерування по хешмапі відбувається у довільному порядку.

Метод `split_whitespace` повертає ітератор по підслайсах, розділених пробілами, значення у `text`. Метод `or_insert` повертає мутабельне посилання (`&mut V`) на значення для вказаного ключа. Тут ми зберігаємо це мутабельне посилання у змінній `count`, тож для того, щоб присвоїти цьому значенню, нам необхідно спочатку розіменувати `count` за допомогою зірочки (`*`). Мутабельне посилання виходить з

області видимості в кінці циклу `for`, тож всі ці зміни є безпечними та дозволеними правилами позичання.

Функції Хешування

За замовчуванням, `HashMap` використовує функцію хешування під назвою *SipHash*, яка забезпечує стійкість до атак на відмову в обслуговуванні (Denial of Service, DoS) з використанням хеш-таблиць ¹. Це не найшвидший з доступних алгоритмів хешування, але покращення безпеки, отримане з падінням продуктивності, того варте. Якщо ви профілюєте свій код і виявите, що функція хешування за замовчуванням є надто повільною для ваших потреб, ви можете перейти на іншу функцію, вказавши інший хешер. *Хешер* - це тип, який реалізує трейт `BuildHasher`. Ми поговоримо про трейти і як їх реалізовувати у Розділі 10. Вам не обов'язково потрібно реалізувати власний хеш з нуля; crates.io містить бібліотеки, надані іншими користувачами Rust, що забезпечують реалізацію багатьох поширених алгоритмів хешування.

Підсумок

Вектори, стрічки та хешмапи забезпечать значну частину функціональності, необхідної для програм під час зберігання, доступу і зміни даних. Ось деякі вправи, які ви вже маєте бути в змозі виконати:

- Дано список цілих чисел; використайте вектор і поверніть медіану (значення посередині після сортування) та моду (значення, яке зустрічається найчастіше; тут стане в пригоді хешмапа) цього списку.
- Перетворіть рядки на "поросячу латину". Перший приголосний кожного слова переноситься в кінець слова і додається "ay", так що "first" стає "irst-fay." До слів, що починаються на голосну, натомість додається в кінці "hay" ("apple" стає "apple-hay"). Не забувайте, що використовується кодування UTF-8!
- За допомогою хешмапи і векторів створіть текстовий інтерфейс, що дозволить користувачеві додати імена співробітників у відділ компанії. Наприклад, "Add Sally to Engineering" чи "Add Amir to Sales." Тоді дайте користувачеві можливість отримати список усіх людей у відділі чи всіх людей у компанії по відділах, відсортованих за алфавітом.

Документація API стандартної бібліотеки описує методи, які мають вектори, стрічки і хешмапи, які будуть корисними для цих вправ!

Ми переходимо до складніших програм, у яких операції можуть зазнавати невдачу, тому зараз ідеальний час для обговорення обробки помилок. Цим ми й займемося! ch10-03-

lifetime-syntax.html#validating-references-with-lifetimes

¹ <https://en.wikipedia.org/wiki/SipHash>

Обробка Помилки

Помилки є частиною звичайного життя в програмному забезпеченні, тому Rust має ряд можливостей для обробки ситуацій, коли щось йде не так. У багатьох випадках Rust вимагає від вас визнати можливість помилки та вжити певних дій перед тим, як ваш код буде скомпільовано. Ця вимога робить вашу програму більш надійною, забезпечуючи що ви будете виявляти помилки та поводитися з ними належним чином, перш ніж ви розгорнете свій код у production!

Rust групує помилки у дві основні категорії: *виправні* та *невиправні* помилки. Для виправних помилок, таких як помилка *файл не знайдено*, ми, швидше за все, захочемо просто повідомити про проблему користувачеві й спробувати знову повторити операцію. Невиправні помилки завжди є ознаками вад, наприклад спроба отримати доступ до місця в пам'яті після закінчення масиву, і тому ми хочемо негайно зупинити програму.

Більшість мов не розрізняють ці два види помилок і обробляють їх однаково, використовуючи механізми, такі як виняткові ситуації. У Rust немає виняткових ситуацій. Натомість вона має тип `Result<T, E>` для виправних помилок та макрос `panic!`, що зупиняє виконання, коли програма зіткнулася з невірною помилкою. Цей розділ спершу охоплює виклик `panic!`, а потім розповідає про повернення значень `Result<T, E>`. Крім того, ми розглянемо міркування при ухваленні рішення про те, чи намагатися відновитися після помилки, чи зупинити виконання.

Невідновлювані Помилки з `panic!`

Іноді погані речі трапляються у вашому коді з якими ви нічого не можете зробити. У цих випадках Rust має макрос `panic!`. Є два практичних способи викликати паніку: зробивши дію, яка призведе до паніки (наприклад отримати доступ до елемента масиву за його межами) або явно викликавши макрос `panic!`. В обох випадках ми викличемо паніку в нашій програмі. За замовчуванням, ці паніки виведуть в консолі повідомлення про помилку, розгорнуть та очистять стек та закриють програму. За допомогою змінної середовища ви також можете скерувати Rust показати стек викликів, коли виникне паніка, щоб полегшити відстеження її джерела.

Розгортання Стека або Переривання у Відповідь на Паніку

За замовчуванням, коли виникає паніка, програма запускає *розгортання*. Це означає, що Rust проходиться по стеку та очищає дані всіх зустрічних функцій. Проте ця розгортка та очищення це багато роботи. Отже, Rust дозволяє вибрати альтернативу: негайно завершувати програму без її очищення.

Пам'ять, яку використовувала програма, тоді буде очищена операційною системою. Якщо у вашому проєкті вам потрібно зробити кінцевий бінарний файл якомога менше, ви можете змінити поведінку програми при паніці з розгортки стеку на негайне переривання додавши `panic = 'abort'` у відповідну секцію `[profile]` у вашому файлі *Cargo.toml*. Наприклад, якщо ви хочете негайне переривання паніки у режимі збірки, додайте наступне:

```
[profile.release]
panic = 'abort'
```

Спробуймо викликати `panic!` у простій програмі:

Filename: src/main.rs

```
fn main() {
    panic!("crash and burn");
}
```

Коли ви запускаєте програму, ви побачите щось на зразок цього:

```
$ cargo run
  Compiling panic v0.1.0 (file:///projects/panic)
  Finished dev [unoptimized + debuginfo] target(s) in 0.25s
  Running `target/debug/panic`
thread 'main' panicked at 'crash and burn', src/main.rs:2:5
note: run with `RUST_BACKTRACE=1` environment variable to display a backtrace
```

Виклик `panic!` призвів до повідомлення про помилку, що міститься в останніх двох рядках. Перший рядок показує наше повідомлення про паніку і місце в нашому початковому коді, де вона сталася: `src/main.rs:2:5` вказує, що це другий рядок, п'ятий символ нашого файлу `src/main.rs`.

У цьому випадку зазначений рядок є частиною нашого коду, і якщо ми перейдемо до цього рядка, ми побачимо виклик макроса `panic!`. В інших випадках виклик `panic!` може бути в коді, який викликає наш код, а назва файлу і номер рядка, зведений повідомленням про помилку, буде чужим кодом, де викликається макрос `panic!`, а не рядок нашого коду, який зрештою призвів до `panic!`. Ми можемо використати бектрейс функції з якої прийшов виклик `panic`, щоб дізнатися, яка частина нашого коду викликає проблему. Ми обговоримо бектрейс у деталях пізніше.

Використання Бектрейсу `panic!`

Розглянемо ще один приклад, коли виклик `panic!` йде з бібліотеки через помилку в нашому коді, а не через прямий виклик макроса нашим кодом. Блок коду 9-1 намагається отримати доступ до елемента вектора поза меж діапазону припустимих індексів.

Файл: `src/main.rs`

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

    v[99];
}
```

Блок коду 9-1: Спроба отримати доступ до елемента вектора за межами його кінця викличе `panic!`

Тут ми намагаємося отримати доступ до 100-го елемента нашого вектора (який знаходиться за індексом 99, бо індексування починається з нуля), але вектор має всього 3 елементи. В цій ситуації Rust панікуватиме. Використання `[]` повинно повернути елемент, але якщо передати не валідний індекс, то не буде елементу, який Rust може повернути, що було б правильним.

У C спроба прочитати за межами кінця структури даних це не визначена поведінка або *undefined behaviour*. Ви можете отримати те, що розташоване в місці в пам'яті та відповідає цьому елементу структури даних, навіть якщо пам'ять не належить цій структурі. Це називається *читання поза межами буфера* або *buffer overread* і може стати причиною появи уразливостей в безпеці, якщо нападник здатен маніпулювати індексом таким чином, щоб прочитати дані які зберігаються поза структурою даних до яких він не має права на доступ.

Щоб захистити вашу програму від такого роду уразливості, при спробі прочитати елемент за індексом, якого не існує, Rust зупинить виконання програми. Спробуймо:

```
$ cargo run
Compiling panic v0.1.0 (file:///projects/panic)
Finished dev [unoptimized + debuginfo] target(s) in 0.27s
Running `target/debug/panic`
thread 'main' panicked at 'index out of bounds: the len is 3 but the index is 99',
src/main.rs:4:5
note: run with `RUST_BACKTRACE=1` environment variable to display a backtrace
```

Ця помилка вказує на рядок 4 нашого файлу `main.rs`, де ми намагаємося отримати доступ до елемента за індексом 99. Наступний рядок каже нам, що ми можемо встановити змінну середовища `RUST_BACKTRACE` для отримання бектрейсу того, що саме стало причиною помилки. *Бектрейс* це список усіх функцій які були викликані до появи помилки. Бектрейси в Rust працюють так само як і в інших мовах: Читати бекстрейс потрібно зверху вниз й читати доти, доки ви не побачите назви ваших файлів. Ось місце, де виникла проблема. Рядки зверху це те, що викликано вашим кодом; рядки знизу це код, який викликає код зверху. Ці "до-та-після" рядки можуть включати код ядра Rust, код стандартної бібліотеки, або крейтів, що ви використовуєте. Спробуймо отримати бектрейс встановивши змінну середовища `RUST_BACKTRACE` будь-яке значення окрім 0. Блок коду 9-2 показує вивід схожий на те, що ви побачите.

```

$ RUST_BACKTRACE=1 cargo run
thread 'main' panicked at 'index out of bounds: the len is 3 but the index is 99',
src/main.rs:4:5
stack backtrace:
 0: rust_begin_unwind
      at
/rustc/e092d0b6b43f2de967af0887873151bb1c0b18d3/library/std/src/panicking.rs:584:5
 1: core::panicking::panic_fmt
      at
/rustc/e092d0b6b43f2de967af0887873151bb1c0b18d3/library/core/src/panicking.rs:142:
14
 2: core::panicking::panic_bounds_check
      at
/rustc/e092d0b6b43f2de967af0887873151bb1c0b18d3/library/core/src/panicking.rs:84:5
 3: <usize as core::slice::index::SliceIndex<[T]>>::index
      at
/rustc/e092d0b6b43f2de967af0887873151bb1c0b18d3/library/core/src/slice/index.rs:24
2:10
 4: core::slice::index::<impl core::ops::index::Index<I> for [T]>::index
      at
/rustc/e092d0b6b43f2de967af0887873151bb1c0b18d3/library/core/src/slice/index.rs:18
:9
 5: <alloc::vec::Vec<T,A> as core::ops::index::Index<I>::index
      at
/rustc/e092d0b6b43f2de967af0887873151bb1c0b18d3/library/alloc/src/vec/mod.rs:2591:
9
 6: panic::main
      at ./src/main.rs:4:5
 7: core::ops::function::FnOnce::call_once
      at
/rustc/e092d0b6b43f2de967af0887873151bb1c0b18d3/library/core/src/ops/function.rs:2
48:5
note: Some details are omitted, run with `RUST_BACKTRACE=full` for a verbose
backtrace.

```

Блок коду 9-2: Бектрейс утворений викликом `panic!` показує, коли змінна середовища `RUST_BACKTRACE` була встановлена

Це багато виводу! Точний вивід може відрізнятись в залежності від версії операційної системи та версії Rust. Для того, щоб отримати бектрейс із цією інформацією, мають бути увімкнені дебаг символи. Дебаг символи увімкнуті за замовчуванням коли ви використовуєте `cargo build` або `cargo run` без позначки `--release`, як ми зробили тут.

У виводі Блока коду 9-2, рядок 6 бектрейсу вказує на наш рядок який спричиняє проблему: рядок 4 файлу `src/main.rs`. Якщо ми не хочемо, щоб наша програма панікувала, ми повинні почати наше розслідування з першого рядка, де згадується написаний нами файл. В Блоці коду 9-1, де ми навмисно написали код, який викличе паніку, спосіб виправлення паніки це не запитувати елемент поза межами діапазону індексів вектора.

Надалі коли ваш код панікуватиме, вам потрібно буде з'ясувати, які дії виконує код із якими значеннями щоб спричинити паніку і що код повинен робити натомість.

Ми ще повернемося до `panic!` і до того, коли нам слід і не слід використовувати `panic!` для обробки умов помилок у секції "To `panic!` or Not to `panic!`" пізніше у цьому розділі. Далі ми розглянемо, як відновляти помилки за допомогою `Result`. [ch09-03-to-panic-or-not-to-panic.html#to-panic-or-not-to-panic](https://pavloslav.github.io/RustBookUkr/print.html#to-panic-or-not-to-panic)

Відновлювані Помилки з `Result`

Більшість помилок не є достатньо серйозними, щоб вимагати повної зупинки виконання програми. Іноді, якщо функція не спрацьовує, то це можна досить просто пояснити й відреагувати певним чином. Наприклад, якщо ви намагаєтесь відкрити файл і ця операція зазнає невдачі, тому що такого файлу немає, то замість завершення процесу ви б захотіли створити такий файл.

Нагадаємо з підрозділу [Керування потенційною невдачею за допомогою типу `Result`](#) в Розділі 2, що

`Result` визначається як еnum, що має два можливих значення `Ok` та `Err`, наступним чином:

```
enum Result<T, E> {  
    Ok(T),  
    Err(E),  
}
```

The `T` and `E` це параметри, що відносять до узагальнених типів, які ми розглянемо більш детально у розділі 10. Все, що вам необхідно знати на даний момент, що `T` представляє тип значення, яке буде повернуто результатом успішного виконання як вміст варіанту `Ok`, а `E` представляє тип помилки, що буде повернуто як вміст варіанту `Err` у випадку невдачі. Оскільки `Result` містить ці узагальнені типи параметрів, ми можемо використовувати тип `Result` і функції, що визначені для нього у стандартній бібліотеці, для різних випадків, коли значення успішного виконання і значення невдачі, які ми хочемо повернути, можуть відрізнятися.

Спробуймо викликати функцію, яка повертає значення типу `Result`, оскільки ця функція може не спрацювати. В блоці коду 9-3 ми спробуємо відкрити файл.

Файл: `src/main.rs`

```
use std::fs::File;  
  
fn main() {  
    let greeting_file_result = File::open("hello.txt");  
}
```

Блок коду 9-3: Відкривання файлу

Типом, який повертає `File::open` є `Result<T, E>`. Узагальнений параметр `T` був визначений в реалізації `File::open`, як тип значення при успіху при обробці файлу, а саме `std::fs::File`. Тип `E`, що використовується, як значення помилки, визначений як `std::io::Error`. Цей тип значення, що повертається, означає, що виклик `File::open` може бути успішним і повернути обробник файлу, за допомогою якого ми можемо його зчитувати, або записувати. Також виклик функції може завершитися не успішно, наприклад файлу може ще не існувати, або у нас не буде дозволів на обробку цього файлу. Функція `File::open` має мати можливість сповістити нас, чи виклик був успішним чи ні, і дати нам або обробник файлу, або інформацію про помилку. Ця інформація і є безпосередньо тим, що являє собою ену́м `Result`.

У випадку, коли виклик `File::open` був успішним, значенням змінної `greeting_file_result` буде екземпляр `Ok`, що містить обробник файлу. А у випадку помилки, значенням `greeting_file_result` буде екземпляр `Err`, який містить інформацію про цю помилкову ситуацію, що сталася.

Ми повинні розширити блок коду 9-3, щоб зрозуміти різні підходи в залежності від значення, яке повертає `File::open`. Блок коду 9-4 демонструє один із способів обробки `Result`, використовуючи базові підхід, такий як вираз співставлення зі зразком (`match`), що розглядався у розділі 6.

Файл: `src/main.rs`

```
use std::fs::File;

fn main() {
    let greeting_file_result = File::open("hello.txt");

    let greeting_file = match greeting_file_result {
        Ok(file) => file,
        Err(error) => panic!("Problem opening the file: {:?}", error),
    };
}
```

Блок коду 9-4: Використання `match` виразу для обробки варіантів `Result`

Звернуть увагу, що аналогічно до ену́ма `Option`, ену́м `Result` і його варіанти вже введені в область видимості прелюдії, тому немає необхідності вказувати `Result::` перед варіантами `Ok` і `Err` в руках виразу співставлення зі зразком `match`.

Коли результат буде рівний `Ok`, нам необхідно повернути внутрішнє значення `file` з варіанту `Ok`, таким чином при присвоєнню значення обробника файлу змінній `greeting_file`. Після виразу `match` ми можемо використовувати обробник файлу для запису чи зчитування.

Другий рукав виразу `match` обробляє випадок, коли отримуємо значення `Err` результатом виконання `File::open`. В нашому прикладі ми вирішили викликати макрос `panic!`. Якщо в поточному каталозі немає файлу з іменем *hello.txt* і ми запустимо наш код, то завдяки макрокоманді `panic!` побачимо наступний вивід:

```
$ cargo run
Compiling error-handling v0.1.0 (file:///projects/error-handling)
Finished dev [unoptimized + debuginfo] target(s) in 0.73s
Running `target/debug/error-handling`
thread 'main' panicked at 'Problem opening the file: 0s { code: 2, kind: NotFound, message: "No such file or directory" }', src/main.rs:8:23
note: run with `RUST_BACKTRACE=1` environment variable to display a backtrace
```

Як завжди, цей вивід розкаже нам, що саме пішло не так.

Зіставлення з Різними Помилками

Код у блоці 9-4 буде завершитися панікою, незалежно від того, чому виклик `File::open` не спрацював. Однак, ми б хотіли виконати різні дії для різних причин неуспішного виконання: якщо `File::open` не відпрацьовує, оскільки файл не існує, ми б хотіли створити такий файл і повернути обробник для цього нового файлу. Якщо ж `File::open` не спрацював через будь-які інші причини, наприклад, у нас немає дозволів для відкриття файлу, ми б все ж таки хотіли викликати `panic!` таким самим чином, як це було в блоці коду 9-4. Для цього ми додамо вкладений вираз `match`, як показано у блоці коду 9-5.

Файл: `src/main.rs`

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

fn main() {
    let greeting_file_result = File::open("hello.txt");

    let greeting_file = match greeting_file_result {
        Ok(file) => file,
        Err(error) => match error.kind() {
            ErrorKind::NotFound => match File::create("hello.txt") {
                Ok(fc) => fc,
                Err(e) => panic!("Problem creating the file: {:?}", e),
            },
            other_error => {
                panic!("Problem opening the file: {:?}", other_error);
            }
        },
    };
}
```

Блок коду 9-5: Обробка різних типів помилок різним способом

Тип значення, який повертає `File::open` всередині варіантів `Err` є `io::Error`, який в свою чергу є структурою, що поставляється стандартною бібліотекою. Ця структура має метод `kind`, при виклику якого отримаємо `io::ErrorKind` значення. Енум `io::ErrorKind` поставляється у стандартній бібліотеці і має варіанти, які представляють різні типи помилок, що можуть бути результатом операції `io`. Варіант, який ми хочемо використати, це `ErrorKind::NotFound`. Цей варіант сигналізує нам, що файлу, який ми намагаємось відкрити, не існує. Тому ми застосовуємо вираз `match` у `greeting_file_result`, а також ми маємо вкладений вираз `match` для `error.kind()`.

У внутрішньому виразі `match` ми хочемо перевірити, чи значення, що повертає метод `error.kind()` є варіантом `NotFound` енума `ErrorKind`. Якщо ж так і є, ми пробуємо створити такий файл за допомогою методу `File::create`. Однак, оскільки метод `File::create` може також завершитися не успішно, нам треба ще один рукав всередині вкладеного виразу `match`. Якщо файл не може бути створено, то виводимо інше повідомлення про помилку. Другий рукав зовнішнього виразу `match` залишається незмінним, тому програма підіймає паніку на будь-які інші помилки за виключенням помилки відсутнього файлу.

Альтернативи Використанню Виразу `match` для Значень Типу `Result<T, E>`

Схоже, що у нас забагато `match`! Вираз `match` дуже корисний, проте дуже примітивний. У розділі 13 ми будемо вивчати замикання, які використовуються у комбінації з багатьма методами, які визначені для типу `Result<T, E>`. Ці методи можуть бути більш виразними за використання виразу `match`, коли працюємо зі значеннями `Result<T, E>` у своєму коді.

Прикладом може бути інший спосіб описати таку ж саму логіку, що показана у блоці коду 9-5, але з використанням замикань і методу `unwrap_or_else`:

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

fn main() {
    let greeting_file = File::open("hello.txt").unwrap_or_else(|error| {
        if error.kind() == ErrorKind::NotFound {
            File::create("hello.txt").unwrap_or_else(|error| {
                panic!("Problem creating the file: {:?}", error);
            })
        } else {
            panic!("Problem opening the file: {:?}", error);
        }
    });
}
```

Хоча цей код має таку ж поведінку, що і код у блоці 9-5, він не містить жодного виразу `match` і зрозуміліший для читання. Повертайтесь до цього прикладу після того, як прочитаєте розділ 13 і познайомтесь з методом `unwrap_or_else` у документації до стандартної бібліотеки. Також багато інших методів можуть допомогти справитися з великою кількістю вкладених між собою виразів `match`, при роботі з помилками.

Скорочення для Паніки на Помилках: `unwrap` і `expect`

Використання виразу `match` працює достатньо добре, але може бути занадто багатослівним і не завжди добре передавати наші наміри. Тип `Result<T, E>` має багато допоміжних методів, які визначені для того, щоб здійснити більш специфічні обробки. Метод `unwrap` є скороченням імплементації виразу `match`, як це було зроблено у блоці коду 9-4. Якщо значення `Result` є варіантом `Ok`, метод `unwrap` поверне значення, як міститься всередині `Ok`. Якщо ж `Result` є варіантом `Err`, то метод `unwrap` викличе макрос `panic!` для нас. Ось приклад методу `unwrap` у дії:

Файл: `src/main.rs`

```
use std::fs::File;

fn main() {
    let greeting_file = File::open("hello.txt").unwrap();
}
```

Якщо ми виконаємо код без існуючого файлу `hello.txt`, ми отримаємо повідомлення про помилку із виклику `panic!`, який здійснить метод `unwrap`:


```
thread 'main' panicked at 'called `Result::unwrap()` on an `Err` value: Os {
code: 2, kind: NotFound, message: "No such file or directory" }',
src/main.rs:4:49
```

Аналогічний метод `expect` дозволяє додатково нам вибрати повідомлення про помилку для макрокоманди `panic!`. Використання методу `expect` замість `unwrap` разом із заданням хороших повідомлень про помилку допоможе краще передати ваші наміри й спростить відстежування причин такої паніки. Синтаксис методу `expect` виглядає наступним чином:

Файл: src/main.rs

```
use std::fs::File;

fn main() {
    let greeting_file = File::open("hello.txt")
        .expect("hello.txt should be included in this project");
}
```

Ми використовуємо `expect` в такий самий спосіб, як і `unwrap`, щоб повернути обробник файлу або викликати макрос `panic!`. Повідомленням про помилку, що використовує метод `expect` у виклику макросу `panic!`, буде параметром, який ми передаємо у `expect`, замість стандартного повідомлення макросу `panic!`, яку використовує `unwrap`. Ось як це виглядає:

```
thread 'main' panicked at 'hello.txt should be included in this project: Os {
code: 2, kind: NotFound, message: "No such file or directory" }',
src/main.rs:5:10
```

Частіше програмісти на Rust віддають перевагу у своєму коді методу `expect` аніж `unwrap`, додаючи більше контексту з роз'ясненням, чому дана операція має бути завжди успішною для виконання. Таким чином, навіть якщо ваші припущення були не до кінця точними у такій ситуації, у вас буде більше інформації під час відлагодження.

Розповсюдження Помилки

Коли ви пишете функцію, імплементація якої може викликати щось, що може не спрацювати, тоді замість обробки помилки всередині функції, ви можете повернути помилку тій частині коду, що її викликала. І тоді вже ця частина коду буде вирішувати, що робити з цією помилкою. Це називається *розповсюдженням* помилки й дає більше контролю викликаючому коду, де, можливо, є більше інформації або логіки, які диктують, як ця помилка має бути оброблена, ніж те що є доступним в контексті вашого коду функції.

Для прикладу блок коду 9-6 показує функцію, яка зчитує username з файлу. Якщо ж файл не існує або його неможливо прочитати, то цю функцію поверне ці помилки в код, який викликає дану функцію.

Файл: src/main.rs

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

fn read_username_from_file() -> Result<String, io::Error> {
    let username_file_result = File::open("hello.txt");

    let mut username_file = match username_file_result {
        Ok(file) => file,
        Err(e) => return Err(e),
    };

    let mut username = String::new();

    match username_file.read_to_string(&mut username) {
        Ok(_) => Ok(username),
        Err(e) => Err(e),
    }
}
```

Блок коду 9-6: Функція, яка повертає помилки в код, який її викликає за допомогою виразу `match`

Вказану функцію також можливо написати коротшим способом, але ми почнемо з того, що зробимо більшу частину самостійно, для того, щоб познайомитися з обробкою помилок. В кінці ми продемонструємо короткий спосіб. Давайте спочатку розглянемо тип значення, яке повертає функція: `Result<String, io::Error>`. Це означає, що функція повертає значення типу `Result<T, E>`, де узагальнений параметр `T` був підставлений конкретним типом `String`, а узагальнений тип `E` конкретним типом `io::Error`.

Якщо виклик функції відпрацює успішно без жодних проблем, то код, який викликає її, отримає значення типу `Ok`, яке містить `String` - тобто username, який був зчитаним функцією з файлу. Якщо ж виконання функції зіткнеться з якимось проблемами, код, який викликав її отримає значення `Err`, яке містить екземпляр `io::Error`, який, в свою чергу, містить більше інформації стосовно характеру проблем. Ми вибрали `io::Error` як тип значення, що повертається з неї, тому що вона є типом помилок обох функцій, що можуть виконатись не успішно, які ми викликаємо в тілі нашої функції: функція `File::open` і `read_to_string`.

Тіло функції починається з виклику методу `File::open`. Далі ми обробляємо значення `Result` за допомогою виразу `match`, схоже до того, що було у блоці коду 9-4. Якщо виклик

`File::open` буде успішним, тоді обробник файлу буде міститися у змінній `file` виразу співставлення, який стане значення мутабельної змінної `username_file` і виконання функції буде продовжуватися. А у випадку значення `Err`, замість виклику `panic!`, ми використовуємо ключове слово `return` для передчасного виходу з функції з поверненням значення помилки в місце виклику нашої функції, яку отримуємо з виклику `File::open` як внутрішню змінну зіставлення `e`.

Якщо ми маємо обробник файлу у змінній `username_file`, тоді функція створить нове значення `String` у змінній `username` і викличе метод `read_to_string` на обробнику файлу `username_file`, щоб прочитати контент цього файлу у значення змінної `username`. Метод `read_to_string` також повертає `Result`, оскільки може виконатись не успішно, навіть виконання `File::open` було успішним до цього. Тому нам потрібно ще один вираз `match` для обробки цього `Result`: якщо `read_to_string` був успішним, то і виконання нашої функції теж успішне і повертаємо значення `username` з файлу, огорнутим у `Ok`. Якщо `read_to_string` виконалось не успішно, ми просто повертаємо помилку у той самий спосіб, як і у виразі `match`, що обробляв значення виклику `File::open`. Однак нам непотрібно явно використовувати `return`, оскільки це останній вираз нашої функції.

Код, який викликає цей має обробити отримані або значення `Ok`, що містить `username` або значення `Err`, яке містить `io::Error`. Ми не повинні знати, що саме код який викликає буде робити з отриманими значеннями. Якщо він отримає значення `Err`, то може або викликати `panic!` і зупинити виконання програми, або скористатися іменем користувача по замовчуванню, або знайти його де інде. Ми не маємо достатньої інформації стосовно того, що саме код, який викликає буде робити, тому ми поширюємо всю інформацію, як і успішного виконання, так і не успішного вгору, для обробки її належним чином.

Цей патерн поширення помилок є дуже поширеним в Rust, тому Rust має спеціальний оператор знаку питання `?`, для роботи з цим більш зручний спосіб.

Скорочене Поширення помилок Оператором `?`

Блок коду 9-7 демонстрував імплементацію функції `read_username_from_file`, яка має таку ж функціональність, як і функція в блоці коду 9-6, але дана реалізація використовувала оператор `?`.

Файл: `src/main.rs`

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

fn read_username_from_file() -> Result<String, io::Error> {
    let mut username_file = File::open("hello.txt")?;
    let mut username = String::new();
    username_file.read_to_string(&mut username)?;
    Ok(username)
}
```

Listing 9-7: Функція, яка повертає помилки коду, який її викликає за допомогою оператора `?`

Якщо розмістити оператор `?` після значення `Result`, то він буде працювати таким самим чином, як і вираз `match`, який ми визначали для обробки значення `Result` в блоці коду 9-6. Якщо значення `Result` є `Ok`, то значення, що знаходиться всередині `Ok`, буде повернутим як результат виразу і програма продовжить виконання. Якщо ж значення є `Err`, то в цілому з функції буде повернуто `Err`, так ніби ми використали ключове слово `return` і значення помилки буде передано функції, що викликала даний код.

Є певна різниця між тим, що виконує вираз `match` з блоку коду 9-6 і тим, що виконує оператор `?`. Значення помилок, при яких викликається оператор `?` проходять через виклик функції `from`, яка визначена на трейті `From` стандартної бібліотеки і використовується для конвертації значень із одного типу в інший. Коли оператор `?` викликає функцію `from`, отриманий тип помилки конвертується в тип помилки, який був визначений типом, що повертається з поточної функції. Це корисно, коли функція повертає один тип помилки, який являє собою всі можливі шляхи, при яких функція може виконатись не успішно, навіть якщо її частини можуть завершуватись не успішно з різних причин.

Для прикладу, ми б могли змінити функцію `read_username_from_file` в блоці коду 9-7, щоб вона повертала кастомізований тип помилки визначений нами, який б називався `OurError`. Якщо ми також визначимо імплементацію `impl From` для типу `OurError` при створенні екземпляру `OurError` із `io::Error`, тоді виклик оператора `?` в тілі функції `read_username_from_file` викличе метод `<0>from</0>` і здійснить конвертацію типу помилки без необхідності додавання жодного коду у нашу функцію.

В контексті блоку коду 9-7, оператор `?` в кінці виклику функції `File::open` поверне значення значення всередині `Ok` у змінну `username_file`. Якщо ж помилка виникне, то оператор `?` припинить виконання функції заздалегідь і поверне якесь значення `Err` коду, який її викликав. Те ж саме буде справедливим для оператора `?` в кінці виклику методу `read_to_string`.

Оператор `?` дозволяє уникнути надлишкового коду у функціях і робить їх імплементацію простішою. Ми можемо навіть ще більше скоротити код, об'єднуючи виклики методів в ланцюжок відразу після оператора `?`, як це показано в блоці коду 9-8.

Файл: src/main.rs

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

fn read_username_from_file() -> Result<String, io::Error> {
    let mut username = String::new();

    File::open("hello.txt")?.read_to_string(&mut username)?;

    Ok(username)
}
```

Блок коду 9-8: Ланцюжок викликів методів після оператора `?`

Ми перенесли створення нового екземпляру `String` в `username` на початок функції. Замість створення змінної `username_file`, ми приєднали ланцюжком виклик `read_to_string` прямо до результату виклику `File::open("hello.txt")?`. Ми все ще маємо оператор `?` в кінці виклику `read_to_string` і все ще повертаємо значення `Ok`, яке містить `username`, якщо обидва виклики `File::open` і `read_to_string` завершаться успішно, а не повертаємо помилки. Ця функціональність знову ж таки аналогічна тій, що представлена у блоках коду 9-6 і 9-7 з однією тільки відмінністю, що такий шлях більш ергономічний для написання.

Блок коду 9-9 демонструє ще коротший шлях використання `fs::read_to_string`.

Файл: src/main.rs

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

fn read_username_from_file() -> Result<String, io::Error> {
    fs::read_to_string("hello.txt")
}
```

Блок коду 9-9: Використання методу `fs::read_to_string` замість того, щоб відкривати файл і потім виконувати його зчитування

Зчитування файлу в стрічку є досить типовою операцією, тому стандартна бібліотека надає зручнішу функцію `fs::read_to_string`, яка відкриває файл, створює новий екземпляр `String`, зчитує вміст фалу, поміщає вміст в створений екземпляр стрічки `String` і повертає його. Звичайно, використання методу `fs::read_to_string` не дає

можливості пояснити всі підходи до обробки помилок, тому ми спочатку пішли більш довгим шляхом.

Де Можна Використовувати Оператор `?`

Оператор `?` може бути використаним тільки у функціях, які повертають тип, який сумісний зі значенням, яке він може обробити. Це тому, що оператор `?` створений для обробки раннього повернення значення з функції в такий самий спосіб, як і вираз `match`, описаний у блоці коду 9-6. Тут вираз `match` використовує значення `Result` і повертає значення `Err(e)` по рукаву раннього виходу. Типом, що повертається з функції має бути тип `Result`, що є сумісним цим `return`.

Давайте розглянемо в блоці коду 9-10 помилку, яку отримаємо, якщо використаємо оператор `?` у функції `main`, яка має повертати тип, що не сумісний з типом значення, яке ми використовуємо з оператором `?`:

Файл: `src/main.rs`

```
use std::fs::File;

fn main() {
    let greeting_file = File::open("hello.txt");
}
```

Блок коду 9-10: Спроба використати оператор `?` всередині функції `main`, яка не скомпілюється, оскільки має повертати несумісний тип `()`

Цей код відкриває файл, тому ця операція може виконатись не успішно. Оператор `?` слідує за значенням `Result`, який повертає `File::open`, але функція `main` має повертати тип `()`, а не тип `Result`. Коли ми спробуємо скомпілювати цей код, ми отримаємо наступне повідомлення про помилку компілювання:

```
$ cargo run
Compiling error-handling v0.1.0 (file:///projects/error-handling)
error[E0277]: the `?` operator can only be used in a function that returns
`Result` or `Option` (or another type that implements `FromResidual`)
--> src/main.rs:4:48
|
3 | fn main() {
| ----- this function should return `Result` or `Option` to accept `?`
4 |     let greeting_file = File::open("hello.txt")?;
|                                                    ^ cannot use the `?` operator
in a function that returns `()`
|
= help: the trait `FromResidual<Result<Infallible, std::io::Error>>` is not
implemented for `()`

For more information about this error, try `rustc --explain E0277`.
error: could not compile `error-handling` due to previous error
```

Ця помилка компілювання вказує на те, що ми можемо використовувати оператор `?` тільки у функціях, які повертають `Result`, `Option`, або інший тип, який імplementує `FromResidual`.

Для виправлення цієї помилки ми маємо два шляхи. Перший полягає в тому, щоб змінювати тип значення, яке повертаємо для нашої функції, щоб бути сумісним по типу зі значенням, яке використовуємо з оператором `?` до того моменту, поки немає жодних інших обмежень для цього. Інший полягає в тому, щоб використовувати вираз `match` або один із методів визначених для типу `Result<T, E>`, щоб обробити значення `Result<T, E>` більш підходящим способом.

Помилка компілювання також говорить, що оператор `?` також можна використовувати зі значенням `Option<T>`. Як і з використанням `?` на `Result`, ми можемо використовувати оператор `?` на `Option` у функціях, які повертають `Option`. Поведінка оператора `?`, коли викликаємо його на `Option<T>` є подібною до випадку з `Result<T, E>`: якщо значення `None`, то це значення буде повернуто достроково з функції. Якщо ж значення `Some`, то значення всередині `Some` буде значенням результату виразу і виконання функції буде продовжуватися далі. Блок коду 9-11 є прикладом функції, що знаходить останній символ в отриманому тексті:

```
fn last_char_of_first_line(text: &str) -> Option<char> {
    text.lines().next()?.chars().last()
}
```

Блок коду 9-11: Використання оператора `?` на значенні `Option<T>`

Ця функція повертає `Option<char>`, тому що як є можливість, що символ може бути, так і можливість, що символу не буде. Ця функція приймає як аргумент стрічковий слайс `text`

і викликає метод `lines` на ньому, який повертає ітератор над рядками у стрічці. Оскільки ця функція має отримати перший рядок, вона викликає метод `next` на ітераторі, щоб отримати перше значення з ітератора. Якщо `text` буде пустою стрічкою, то виклик `next` поверне значення `None`, для цього випадку ми використовуємо оператор `?`, щоб достроково зупинити виконання і повернути `None` з функції `last_char_of_first_line`. Якщо ж `text` не порожня стрічка, виклик `next` поверне значення `Some`, яке буде містити стрічковий слайс із першим рядком у `text`.

Оператор `?` вилучає цей стрічковий слайс, і ми можемо далі викликати метод `chars` на цьому зрізі, щоб отримати ітератор символів. Ми зацікавлені в останньому символі першої стрічки, тому ми викликаємо метод `last`, для останнього елементу ітератора. Це значення також `Option`, оскільки можливо, що цей перший рядок є пустою стрічкою, наприклад, якщо `text` починається з пустого рядку, але має символи на наступних рядках, як, до прикладу, у `"\nhi"`. Однак, якщо є останній символ у першому рядку, то він повернеться загорнутим у `Some`. Оператор `?` посередині дає нам виразний спосіб описати логіку, змушуючи нас реалізовувати тіло функції в один рядок. Якщо б ми не використовували оператор `?` на `Option`, то довелося би реалізовувати логіку з використанням більшої кількості викликів методів та виразів `match`.

Варто зазначити, що ми можемо використовувати оператор `?` на `Result` всередині функцій, які повертають `Result`, а також можемо використовувати на `Option` у функціях, які повертають `Option`, але ми не можемо їх змішувати і порівнювати. Оператор `?` не може автоматично конвертувати `Result` в `Option` або навпаки. В цих випадках слід використовувати методи на зразок `ok` на `Result`, або `ok_or` на `Option`, для здійснення явного конвертування.

Поки що всі функції `main`, які ми використовували повертали значення типу `()`. Функція `main` є спеціальною, тому що є вхідною та вихідною точкою для запуску програм і має строгі обмеження до типу значення, яке вона повертає, щоб програма поводити себе так, як очікується.

На щастя, `main` може також повертати значення типу `Result<(), E>`. Блок 9-12 містить код з блоку 9-10, але тут ми змінили тип значення, яке повертається з `main` на `Result<(), Box<dyn Error>>` і повернули значення `Ok(() )` в кінці тіла функції. Цей код буде тепер компілюватися:


```
use std::error::Error;
use std::fs::File;

fn main() -> Result<(), Box<dyn Error>> {
    let greeting_file = File::open("hello.txt"?);

    Ok(())
}
```

Блок коду 9-12: Зміна функції `main` щоб повертати `Result<(), E>` і мати можливість використання оператора `?` на значеннях `Result`

Тип `Box<dyn Error>` є об'єктом типу, про який ми будемо говорити у секції «Використання трейт об'єктів, які допускають значення різних типів» розділу 17. На тепер ми можемо читати це, як `Box<dyn Error>`, що означає «будь який тип помилок». Використання оператора `?` на значенні `Result` всередині функції `main` з помилкою типу `Box<dyn Error>` є допустимим, оскільки допустимими є будь-які значення `Err` для дострокового повернення. Навіть якщо тіло цієї функції `main` буде повертати помилки типу `std::io::Error`, спеціалізована `Box<dyn Error>` сигнатура буде залишатися коректною, навіть якщо додамо більше коду в тіло функції `main`, який може повертати помилки іншого типу.

Коли функція `main` повертає `Result<(), E>`, виконання закінчиться зі значенням `0`, якщо `main` поверне `Ok(())` і закінчиться з ненульовим значенням, якщо `main` поверне значення `Err`. Виконувані файли, написані на C повертають цілі числа коли завершуються: програми які виконалися успішно повертають ціле число `0`, а програми що виконалися з помилкою повертають цілі числа, відмінні від `0`. Rust також повертає цілі числа з виконуваних файлів, щоб бути сумісним з такою домовленістю.

Функція `main` може повертати довільний тип, який імплементує `std::process::Termination` трейтщо містить функцію `report`, яка повертає `ExitCode`. Зверніться до документації стандартної бібліотеки для отримання додаткової інформації про реалізацію трейта `Termination` для ваших власних типів.

Тепер, коли ми обговорили деталі виклику `panic!` й використанню `Result`, повернімось до теми, яким чином визначати, що з переліченого доцільно використовувати та в яких випадках.

`panic!` чи не `panic!`

Отже, як приймається рішення, коли слід викликати `panic!`, а коли повернути `Result`? При паніці код не може відновити своє виконання. Можна було б викликати `panic!` для будь-якої помилкової ситуації, незалежно від того, чи є спосіб відновлення, чи ні, але з іншого боку, ви приймаєте рішення від імені коду, який викликає, що ситуація необоротна. Коли ви повертаєте значення `Result`, ви делегуєте прийняття рішення коду, що викликає. Код, що викликає, може спробувати виконати відновлення способом, який підходить в даній ситуації, або ж він може вирішити, що з помилки в `Err` не можна відновитися і викличе `panic!`, перетворивши вашу помилку, що виправляється, в не виправну. Тому повернення `Result` є гарним вибором за замовчуванням для функції, яка може дати збій.

У таких ситуаціях як приклади, прототипи та тести, більш доречно писати код, який панікує замість повернення `Result`. Розгляньмо чому, а потім обговоримо ситуації, коли компілятор не може довести, що помилка неможлива, але ви, як людина, можете це зробити. Глава закінчуватиметься деякими загальними керівними принципами про те, як вирішити, чи варто панікувати в коді бібліотеки.

Приклади, Прототипування та Тести

Коли ви пишете приклад, який ілюструє деяку концепцію, наявність гарного коду обробки помилок може зробити приклад менш зрозумілим. В прикладах виклик методу `unwrap`, який може призвести до паніки, є лише позначенням способу обробки помилок у додатку, який може відрізнитися в залежності від того, що робить решта коду.

Так само методи `unwrap` та `expect` є дуже зручними при створенні прототипу, перш ніж ви будете готові вирішити, як обробляти помилки. Вони залишають чіткі маркери в коді до моменту, коли ви будете готові зробити програму надійнішою.

Якщо в тесті відбувається збій при виклику методу, то ви б хотіли, щоб весь тест не пройшов, навіть якщо цей метод не є функціональністю, що тестується. Оскільки виклик `panic!` це спосіб, яким тест позначається як невдалий, використання `unwrap` чи `expect` – саме те, що потрібно.

Випадки, коли у Вас Більше Інформації, ніж у Компілятора

Також було б доцільно викликати `unwrap` або `expect`, коли у вас є якась інша логіка, яка гарантує, що `Result` буде мати значення `Ok`, але вашу логіку не розуміє компілятор. У вас, як і раніше, буде значення `Result`, яке потрібно обробити: будь-яка операція, яку ви викликаєте, все ще має можливість невдачі в цілому, хоча це логічно неможливо у вашій конкретній ситуації. Якщо, перевіряючи код вручну, ви можете переконатися, що ніколи не буде варіанту `Err`, то можна викликати `unwrap`, а ще краще задокументувати причину, з якої ви думаєте, що ніколи не матимете варіант `Err` у тексті `expect`. Ось приклад:

```
use std::net::IpAddr;

let home: IpAddr = "127.0.0.1"
    .parse()
    .expect("Hardcoded IP address should be valid");
```

Ми створюємо екземпляр `IpAddr` шляхом аналізу жорстко заданого рядка. Можна побачити що `127.0.0.1` є дійсною IP-адресою, тому доречно використовувати `expect` тут. Однак наявність жорстко заданого правильного рядка не змінює тип повертаємого значення методу `parse`: ми все ще отримуємо значення `Result`, і компілятор досі змушує нас обробляти `Result` так, ніби варіант `Err` є можливим, тому що компілятор недостатньо розумний, щоб побачити, що цей рядок завжди є дійсною IP-адресою. Якщо рядок IP-адреси надійшов від користувача, а не є жорстко заданим у програмі, він може призвести до помилки, тому ми точно хотіли б обробити `Result` більш надійним способом. Згадка про припущення, що ця IP-адреса жорстко задана, спонукатиме нас до зміни `expect` на кращий код обробки помилок, якщо в майбутньому нам знадобиться отримати IP-адресу з іншого джерела.

Настанови з Обробки Помилки

Бажано, щоб код панікував, якщо він може опинитися в некоректному стані. В цьому контексті *некоректний стан* це такий стан, коли деяке допущення, гарантія, контракт чи інваріант були порушені. Наприклад, коли неприпустимі, суперечливі чи пропущенні значення передаються у ваш код, та інші приклади зі списку нижче:

- Некоректний стан - це щось неочікуване, відмінне від того, що може відбуватися час від часу, наприклад, коли користувач вводить дані у неправильному форматі.
- Ваш код після цієї точки повинен покладатися на те, що він не знаходиться у некоректному стані, замість перевірок наявності проблеми на кожному етапі.
- Немає гарного способу закодувати цю інформацію в типах, які ви використовуєте. Ми подивимося приклад того, що ми маємо на увазі в розділі [“Кодування станів та поведінки на основі типів”](#) розділу 17.

Якщо хтось викликає ваш код та передає значення, які не мають сенсу, краще за все повернути помилку, якщо це можливо, щоб користувач бібліотеки мав змогу вирішити, що йому робити в цьому випадку. Однак, у випадках, коли продовження може бути небезпечним чи шкідливим, найкращим вибором може бути виклик `panic!` для оповіщення користувача бібліотеки, що в його коді є помилка й він може її виправити. Також `panic!` підходить, якщо ви викликаєте зовнішній, неконтрольований вами код, і він повертає неприпустимий стан, який ви не можете виправити.

Однак, якщо очікується збій, краще повернути `Result`, ніж виконати виклик `panic!`. Як приклад можна привести синтаксичний аналізатор, якому передали неправильно сформовані дані чи статус HTTP-запиту, що повернувся, вказує на те, що ви досягли обмеження частоти запитів. У цих випадках повертання `Result` вказує на те, що відмова є очікуваною, такою, яку код, що викликає, повинен вирішити, як саме обробити.

Коли ваш код виконує операцію, яка може бути ризикованою для користувача, якщо використовуються неприпустимі значення, ваш код повинен спочатку перевірити чи вони коректні, та панікувати, якщо це не так. Діяти таким чином рекомендується в основному з міркувань безпеки: спроба оперувати некоректними даними може спричинити вразливість вашого коду. Це основна причина, через що стандартна бібліотека буде викликати `panic!`, якщо спробувати отримати доступ до пам'яті поза межами масиву: доступ до пам'яті, яка не стосується поточної структури даних, є відомою проблемою безпеки. Функції часто мають *контракти*: їх поведінка гарантується, тільки якщо вхідні дані відповідають визначеним вимогам. Паніка при порушенні контракту має сенс, тому що це завжди вказує на дефект з боку коду, що викликає, і це не помилка, яку б ви хотіли, щоб код, що викликає, явно обробляв. Насправді немає розумного способу для відновлення коду, що викликає; *Програмісти*, що викликають ваш код, повинні виправити свій. Контракти для функції, особливо порушення яких викликає паніку, слід описати в документації API функції.

Проте, наявність великої кількості перевірок помилок у всіх ваших функціях було б багатослівним та дратівливим. На радість, можна використовувати систему типів Rust (отже і перевірку типів компілятором), щоб вона зробила множину перевірок замість вас. Якщо ваша функція має визначений тип в якості параметру, ви можете продовжити роботу з логікою коду знаючи, що компілятор вже забезпечив правильне значення. Наприклад, якщо використовується звичайний тип, а не тип `Option`, то ваша програма очікує наявність чогось замість нічого. Ваш код не повинен буде опрацювати обидва варіанти `Some` та `None`: він буде мати тільки один варіант для певного значення. Код, який намагається нічого не передавати у функцію, не буде навіть компілюватися, тому ваша функція не повинна перевіряти такий випадок під час виконання. Інший приклад - це використання цілого типу без знаку, такого як `u32`, який гарантує, що параметр ніколи не буде від'ємним.

Створення Користувацьких Типів для Перевірки

Розвинемо ідею використання системи типів Rust щоб переконатися, що в нас є коректне значення, та розглянемо створення користувацького типа для валідації. Згадаємо гру вгадування числа з розділу 2, в якому наш код просив користувача вгадати число між 1 й 100. Ми ніколи не перевіряли, що припущення користувача знаходяться в межах цих чисел, перед порівнянням з задуманим нами числом; ми тільки перевіряли, що воно додатне. У цьому випадку наслідки були не дуже страшними: наші повідомлення "Забагато" чи "Замало", які виводилися у консоль, все одно були коректними. Але було б краще підштовхувати користувача до правильних догадок та мати різну поведінку для випадків, коли користувач пропонує число за межами діапазону, і коли користувач вводить, наприклад, літери замість цифр.

One way to do this would be to parse the guess as an `i32` instead of only a `u32` to allow potentially negative numbers, and then add a check for the number being in range, like so:

```
loop {  
    // --snip--  
  
    let guess: i32 = match guess.trim().parse() {  
        Ok(num) => num,  
        Err(_) => continue,  
    };  
  
    if guess < 1 || guess > 100 {  
        println!("The secret number will be between 1 and 100.");  
        continue;  
    }  
  
    match guess.cmp(&secret_number) {  
        // --snip--  
    }  
}
```

Вираз `if` перевіряє, чи знаходиться наше значення поза діапазоном, повідомляє користувачу про проблему та викликає `continue`, щоб почати наступну ітерацію циклу й попросити ввести інше число. Після виразу `if` ми можемо продовжити порівняння значення `guess` із задуманим числом, знаючи, що `guess` належить діапазону від 1 до 100.

Однак, це не ідеальне рішення: якби було надзвичайно важливо, щоб програма працювала тільки зі значеннями від 1 до 100 та в ній існувало б багато функцій, з такою вимогою, наявність подібної перевірки у кожній функції було б виснажливим (та мало б змогу вплинути на швидкодію).

Замість цього можна створити новий тип та помістити перевірки у функцію створення екземпляру цього типу, не повторюючи їх повсюди. Таким чином, функції можуть

використовувати новий тип у своїх сигнатурах та бути впевненими у значеннях, які їм передають. Лістинг 9-13 демонструє один зі способів, як визначити тип `Guess`, так щоб екземпляр `Guess` створювався лише при умові, що функція `new` отримує значення від 1 до 100.

```
pub struct Guess {
    value: i32,
}

impl Guess {
    pub fn new(value: i32) -> Guess {
        if value < 1 || value > 100 {
            panic!("Guess value must be between 1 and 100, got {}.", value);
        }

        Guess { value }
    }

    pub fn value(&self) -> i32 {
        self.value
    }
}
```

Блок коду 9-13: Тип `Guess`, який буде створювати екземпляри тільки для значень від 1 до 100

Спочатку ми визначимо структуру з ім'ям `Guess`, яка має поле з іменем `value` типу `i32`. Ось де буде збережено число.

Потім ми реалізуємо асоційовану функцію `new` структури `Guess`, яка створює нові екземпляри значень типу `Guess`. Функція `new` має один параметр `value` типу `i32` та повертає `Guess`. Код у тілі функції `new` перевіряє, що значення `value` знаходиться між 1 та 100. Якщо `value` не проходить цю перевірку, ми викликаємо `panic!`, що сповістить програміста, який написав код, що в його коді є помилка, яку необхідно виправити, оскільки спроба створення `Guess` зі значенням `value` поза заданого діапазону порушує контракт, на який покладається `Guess::new`. Умови, за яких `Guess::new` панікує, повинні бути описані в документації до API; ми розглянемо угоди про документації, що вказують на можливість виникнення `panic!` в документації API, яку ви створите в розділі 14. Якщо `value` проходить перевірку, ми створюємо новий екземпляр `Guess`, у якого значення поля `value` дорівнює значенню параметра `value`, і повертаємо `Guess`.

Потім ми реалізуємо метод з назвою `value`, який запозичує `self`, не має інших параметрів, та повертає значення типу `i32`. Цей метод іноді називають *витягувач* (*getter*), тому що його метою є вилучити дані з полів структури та повернути їх. Цей публічний метод є необхідним, оскільки поле `value` структури `Guess` є приватним. Важливо, щоб поле `value` було приватним, щоб код, який використовує структуру `Guess`, не міг

встановлювати `value` напряму: код зовні модуля *повинен* використовувати функцію `Guess::new` для створення екземпляру `Guess`, таким чином гарантуючи, що у `Guess` немає можливості отримати `value`, не перевірене умовами у функції `Guess::new`.

Функція, яка отримує або повертає тільки числа від 1 до 100, може оголосити у своїй сигнатурі, що вона отримує чи повертає `Guess`, замість `i32`, таким чином не буде необхідності робити додаткові перевірки в тілі такої функції.

Підсумок

Можливості обробки помилок в Rust покликані допомогти написанню більш надійного коду. Макрос `panic!` сигналізує, що ваша програма знаходиться у стані, яке вона не може обробити, та дозволяє сказати процесу щоб він зупинив своє виконання, замість спроби продовжити виконання з некоректними чи невірними значеннями. Перерахунок (enum) `Result` використовує систему типів Rust, щоб повідомити, що операції можуть завершитися невдачею, і ваш код мав змогу відновитися. Можна використовувати `Result`, щоб повідомити коду, що викликає, що він повинен обробити потенціальний успіх чи потенційну невдачу. Використання `panic!` та `Result` правильним чином зробить ваш код більш надійним перед обличчям неминучих помилок.

Тепер, коли ви побачили корисні способи використання узагальнених типів `Option` та `Result` в стандартній бібліотеці, ми поговоримо про те, як працюють узагальненні типи і як ви можете використовувати їх у власному коді.

Узагальнені Типи, Трейти та Часи Існування

Кожна мова програмування має інструменти, щоб уникати повторення концепцій. У мові Rust, одним з таких інструментів є *узагальнені типи*: абстрактні заміники конкретних типів або інших властивостей. Ми можемо описувати поведінку узагальнених типів і їх відношення до інших узагальнених типів, не знаючи, який саме тип буде на їх місці під час компіляції і виконання коду.

Функції можуть приймати параметри певного узагальненого типу замість конкретного типу (наприклад, `i32` або `String`), так само як функції можуть приймати параметри з невідомими значеннями і виконувати той самий код з багатьма конкретними значеннями. Насправді ми вже стикалися з узагальненими типами у розділі 6 (`Option<T>`), розділі 8 (`Vec<T>` та `HashMap<K, V>`) і розділі 9 (`Result<T, E>`). У цьому розділі ми побачимо, як можна визначати ваші власні типи, функції та методи з узагальненими типами!

Спочатку пригадаємо, як виділити код в окрему функцію, щоб зменшити дублювання. Тоді ми використаємо цю техніку, щоб створити узагальнену функцію з двох функцій, які відрізняються лише типами їх параметрів. Також ми пояснимо, як використовувати узагальнені типи для визначення структур і енамів.

Після цього ви навчитесь використовувати *трейти* (від англ. `<0>trait</0>` "властивість, риса"), щоб визначати поведінку в узагальнений спосіб. Ви можете поєднувати трейти з узагальненими типами, щоб обмежити узагальнений тип так, щоб він працював не з будь-якими типами, а лише тими, які мають певну поведінку.

Нарешті ми поговоримо про *часи існування* - підвид узагальнених типів, які дають компілятору інформацію про те, як посилання відносяться одне до одного. Часи існування дозволяють нам давати компілятору достатньо інформації про позичені значення, щоб він міг впевнитись, що посилання будуть дійсними в тих ситуаціях, де компілятор не знав би цього без наших підказок.

Уникнення Повторень Шляхом Відокремлення Коду у Функцію

Узагальнені типи дозволяють використати змінну типу, яка замінює багато типів, а не конкретний тип, щоб уникнути повторень у коді. Перед тим як розглянути синтаксис

узагальнених типів, погляньмо на уникнення повторень без узагальнених типів, а саме виділення функції, яка замінює конкретні значення на змінну, що представляє багато значень. Тоді ми застосуємо той самий підхід, щоб виділити узагальнену функцію! Поглянувши на те, як помітити продубльований код, який можна винести в окрему функцію, ви почнете помічати продубльований код, який може використовувати узагальнені типи.

Почнімо з короткої програми у роздруку 10-1, яка шукає найбільше число у списку.

Файл: src/main.rs

```
fn main() {  
    let number_list = vec![34, 50, 25, 100, 65];  
  
    let mut largest = &number_list[0];  
  
    for number in &number_list {  
        if number > largest {  
            largest = number;  
        }  
    }  
  
    println!("The largest number is {}", largest);  
}
```

Блок коду 10-1: Пошук найбільшого числа у списку

Ми зберігаємо список цілих чисел у змінній `number_list` і присвоюємо змінній `largest` посилання на перше число у списку. Тоді ми проходимося по всіх числах у списку, і якщо поточне число більше за те, яке зберігається у `largest`, то ми замінемо посилання у цій змінній. Проте якщо поточне число менше або рівне поки що найбільшому числу, змінна зберігає своє значення і наш код продовжує з наступного числа у списку. Після того як ми пройшлися по всіх числах у списку, `largest` має містити значення найбільшого числа. У цьому випадку це 100.

Тепер нам дали завдання знайти найбільше число в інших двох списках чисел. Для цього ми можемо продублювати код з роздруку 10-1 і використати ту саму логіку у двох різних місцях програми, як показано у роздруку 10-2.

Файл: src/main.rs

```
fn main() {  
    let number_list = vec![34, 50, 25, 100, 65];  
  
    let mut largest = &number_list[0];  
  
    for number in &number_list {  
        if number > largest {  
            largest = number;  
        }  
    }  
  
    println!("The largest number is {}", largest);  
  
    let number_list = vec![102, 34, 6000, 89, 54, 2, 43, 8];  
  
    let mut largest = &number_list[0];  
  
    for number in &number_list {  
        if number > largest {  
            largest = number;  
        }  
    }  
  
    println!("The largest number is {}", largest);  
}
```

Блок коду 10-2: Програма, яка знаходить найбільше число у двох списках

Хоча цей код працює, дублювання коду виснажливе і збільшує ризик помилок. Також потрібно не забути оновити код у двох місцях, якщо ми хочемо внести будь-які зміни.

Щоб уникнути цього дублювання, ми створимо абстракцію визначивши функцію, що працює з будь-яким списком цілих чисел, переданим як параметр. Це рішення робить наш код більш зрозумілим і дозволяє нам виразити концепцію пошуку найбільшого числа у списку в абстрактний спосіб.

У роздруку 10-3 ми виносимо у функцію `largest` код, який знаходить найбільше число у списку. Тоді ми можемо викликати цю функцію, щоб знайти найбільше число у двох списках з роздруку 10-2. Також ми можемо використати цю функцію на будь-якому іншому списку значень типу `i32`, який ми отримали б у майбутньому.

Файл: src/main.rs

```
fn largest(list: &i32) -> &i32 {
    let mut largest = &list[0];

    for item in list {
        if item > largest {
            largest = item;
        }
    }

    largest
}

fn main() {
    let number_list = vec![34, 50, 25, 100, 65];

    let result = largest(&number_list);
    println!("The largest number is {}", result);

    let number_list = vec![102, 34, 6000, 89, 54, 2, 43, 8];

    let result = largest(&number_list);
    println!("The largest number is {}", result);
}
```

Блок коду 10-3: абстрагований код для пошуку найбільшого числа у двох списках

Функція `largest` має параметр `list`, який представляє будь-який конкретний слайс значень `i32`, який ми могли б передати в цю функцію. Як результат, коли ми викликаємо функцію, код працює з конкретними значеннями, які ми передаємо.

Підсумовуючи, ось кроки, які ми виконали, щоб з Блока коду 10-2 отримати Блок коду 10-3:

1. Виявили код, що повторюється.
2. Винесли цей код у тіло нової функції і вказали вхідні та вихідні значення цього коду у сигнатурі функції.
3. Замінили два повторювані екземпляри коду на виклик функції.

Далі ми використаємо ці самі кроки з узагальненими типами, щоб зменшити кількість повторень у коді. Так само як функція може працювати з абстрактною змінною `list`, а не конкретними значеннями, узагальнені типи дозволяють коду працювати з абстрактними типами.

Наприклад, скажімо, ми маємо дві функції: одна знаходить найбільший елемент у слайсі значень `i32`, а інша — у слайсі значень `char`. Як можна уникнути повторень? Нумо дізнаймося!

Узагальнені Типи Даних

Ми використовуємо узагальнені типи для створення таких речей як сигнатури функцій або структури, які потім можна використовувати з багатьма конкретними типами даних. Погляньмо на те, як визначити функції, структури, енуми та методи, що використовують узагальнені типи. Далі ми поговоримо про те, як узагальнені типи впливають на швидкодію коду.

У Визначеннях Функцій

При визначенні функції, що використовує узагальнені типи, ми розмістимо їх в сигнатурі функції, де зазвичай ми вказуємо типи даних параметрів та результату. Це робить наш код більш гнучким і забезпечує більше функціоналу користувачам нашої функції, водночас запобігаючи дублюванню коду.

Продовжимо з нашою функцією `largest`. Роздрук 10-4 показує дві функції, які шукають найбільше значення у слайсі. Пізніше ми об'єднаємо їх в одну функцію, яка використовує узагальнені типи.

Файл: `src/main.rs`

```

fn largest_i32(list: &[i32]) -> &i32 {
    let mut largest = &list[0];

    for item in list {
        if item > largest {
            largest = item;
        }
    }

    largest
}

fn largest_char(list: &[char]) -> &char {
    let mut largest = &list[0];

    for item in list {
        if item > largest {
            largest = item;
        }
    }

    largest
}

fn main() {
    let number_list = vec![34, 50, 25, 100, 65];

    let result = largest_i32(&number_list);
    println!("The largest number is {}", result);

    let char_list = vec!['y', 'm', 'a', 'q'];

    let result = largest_char(&char_list);
    println!("The largest char is {}", result);
}

```

Роздрук 10-4: Дві функції, які відрізняються виключно іменем та типом у їх сигнатурі

Функція `largest_i32` – це та сама, яку ми винесли у роздрук 10-3, яка шукає найбільше значення типу `i32` у слайсі. Функція `largest_char` шукає найбільше значення типу `char` у слайсі. Тіла функції мають той самий код, тому можна усунути дублювання, ввівши узагальнений параметр-тип в об'єднаній функції.

Щоб параметризувати типи в новій об'єднаній функції, нам потрібно дати ім'я параметру, так само як ми даємо ім'я параметрам-значенням у функції. Ви можете використовувати будь-який ідентифікатор як ім'я параметра-типу. Але ми використаємо `T`, тому що за домовленістю назви параметрів у Rust короткі і часто складаються лише з однієї букви, а імена типів, за домовленістю, слідуєть "camel case" (окремі слова пишуться без пробілів і з великої букви; наприклад, так: "CamelCase"). Оскільки це скорочення від "тип", `T` – це типовий вибір для програмістів на Rust.

Коли ми використовуємо параметр у тілі функції, ми маємо оголосити його ім'я у сигнатурі, щоб компілятор знав, що воно означає. Так само, коли ми використовуємо ім'я параметру-типу у сигнатурі функції, ми маємо оголосити цей параметр-тип перед використанням. Щоб оголосити узагальнену функцію `largest`, вставте оголошення імен типів у кутові дужки, `<>`, між ім'ям функції та списком параметрів, ось так:

```
fn largest<T>(list: &[T]) -> &T {
```

Ми читаємо це визначення так: функція `largest` узагальнена відносно певного типу `T`. Ця функція має один параметр з назвою `list`, який є слайсом значень типу `T`. Функція `largest` поверне посилання на значення того самого типу `T`.

Роздрук 10-5 показує визначення об'єднаної функції `largest` з використанням узагальненого типу в її сигнатурі. Цей приклад також показує, як можна викликати функцію зі слайсом значень `i32` або `char`. Зверніть увагу, що цей код поки не скомпілюється, але ми виправимо це пізніше у цьому розділі.

Файл: `src/main.rs`

```
fn largest<T>(list: &[T]) -> &T {
    let mut largest = &list[0];

    for item in list {
        if item > largest {
            largest = item;
        }
    }

    largest
}

fn main() {
    let number_list = vec![34, 50, 25, 100, 65];

    let result = largest(&number_list);
    println!("The largest number is {}", result);

    let char_list = vec!['y', 'm', 'a', 'q'];

    let result = largest(&char_list);
    println!("The largest char is {}", result);
}
```

Роздрук 10-5: Функція `largest` з використанням узагальнених параметрів-типів; цей код поки не скомпілюється

Якщо ми скомпілюємо цей код зараз, ми отримаємо таку помилку:

```

$ cargo run
  Compiling chapter10 v0.1.0 (file:///projects/chapter10)
error[E0369]: binary operation `>` cannot be applied to type `&T`
  --> src/main.rs:5:17
   |
5  |         if item > largest {
   |             ^ ----- &T
   |             |
   |             &T
   |
help: consider restricting type parameter `T`
   |
1  | fn largest<T: std::cmp::PartialOrd>(list: &[T]) -> &T {
   |             ++++++

```

For more information about this error, try `rustc --explain E0369`.

error: could not compile `chapter10` due to previous error

Текст довідки згадує `std::cmp::PartialOrd`, який є трейтом, але ми будемо обговорювати трейти в наступній секції. На цей час, запам'ятайте, що ця помилка вказує, що тіло `largest` не працюватиме для всіх можливих типів, якими може бути `T`. Оскільки, ми хочемо порівняти значення типу `T` в тілі, ми можемо використовувати лише типи, значення яких можна впорядкувати. Щоб дозволити операції порівняння стандартна бібліотека має трейт `std::cmp::PartialOrd`, який ви можна реалізувати для типів (див. додаток C для деталей щодо цього трейту). Слідуючи підказці, ми обмежуємо припустимі типи `T` до тих, що реалізують `PartialOrd`, і цей приклад компілюється, оскільки стандартна бібліотека реалізує `PartialOrd` для `i32` і `char`.

У Визначеннях Структур

Ми також можемо визначити структури з використанням узагальнених параметрів-типів в одному або декількох полях використовуючи синтаксис з `<>`. Роздрук 10-6 визначає структуру `Point<T>`, яка містить координати `x` та `y`, які можуть бути значеннями будь-якого типу.

Файл: `src/main.rs`

```

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

fn main() {
    let integer = Point { x: 5, y: 10 };
    let float = Point { x: 1.0, y: 4.0 };
}

```

Роздрук 10-6: Структура `Point<T>`, яка містить значення `x` та `y` типу `T`

Синтаксис для використання узагальнених типів у визначеннях структур схожий на той, що використовується в визначеннях функцій. Спочатку ми оголошуємо ім'я параметру типу всередині кутових дужок одразу після назви структури. Далі ми використовуємо узагальнений тип у визначенні структури де б ми інакше вказували конкретні типи даних.

Зауважте, що оскільки ми тільки використовуємо один узагальнений тип, щоб визначити `Point<T>`, це визначення означає, що структура `Point<T>` узагальнена відносно певного типу `T`, і поля `x` та `y` обоє мають той самий тип, яким би він не був. Якщо ми створимо екземпляр `Point<T>` зі значеннями різних типів, як у роздруку 10-7, наш код не буде компілюватися.

Файл: `src/main.rs`

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

fn main() {
    let wont_work = Point { x: 5, y: 4.0 };
}
```

Роздрук 10-7: Поля `x` та `y` мають мати той самий тип, оскільки обоє використовують узагальнений тип `T`.

У цьому прикладі, коли ми присвоюємо ціле значення 5 до `x`, ми повідомимо компілятору що тип `T` буде цілим числом для даного екземпляру `Point<T>`. Потім ми вкажемо 4,0 для `y`, який ми визначили як такий же тип, що й `x`, і отримаємо невідповідність типів таким чином:

```
$ cargo run
   Compiling chapter10 v0.1.0 (file:///projects/chapter10)
error[E0308]: mismatched types
  --> src/main.rs:7:38
   |
7  |         let wont_work = Point { x: 5, y: 4.0 };
   |                                   ^^^ expected integer, found floating-
point number

For more information about this error, try `rustc --explain E0308`.
error: could not compile `chapter10` due to previous error
```

Щоб визначити структуру `Point`, де `x` і `y` є обидва узагальненими, але можуть мати значення різних типів, можна використовувати декілька узагальнених параметрів-типів.

Наприклад, у роздруку 10-8, ми змінюємо визначення `Point` на узагальнене відносно типів `T` та `U`, де `x` має тип `T`, а `y` має тип `U`.

Файл: `src/main.rs`

```
struct Point<T, U> {  
    x: T,  
    y: U,  
}  
  
fn main() {  
    let both_integer = Point { x: 5, y: 10 };  
    let both_float = Point { x: 1.0, y: 4.0 };  
    let integer_and_float = Point { x: 5, y: 4.0 };  
}
```

Listing 10-8: A `Point<T, U>` generic over two types so that `x` and `y` can be values of different types

Тепер всі екземпляри `Point` допускаються! Ви можете використовувати скільки завгодно параметрів-типів у визначенні, але використання декількох робить ваш код складнішим для читання. Якщо ви виявите, що потрібно багато узагальнених типів в коді, то це може означати, що ваш код потребує розбиття на менші частини.

У Визначеннях Енумів

Так само як зі структурами, ми можемо визначати енуми, які містять узагальнені типи даних у своїх варіантах. Давайте ще раз подивимось на еnum `Option<T>`, який надає стандартна бібліотека, яку ми використали в розділі 6:

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

Тепер таке визначення має бути зрозуміліше. Як ви можете бачити, еnum `Option<T>` є узагальненим відносно типу `T` і має два варіанти: `Some`, який містить одне значення типу `T`, і `None`, який не містить жодних значень. Використовуючи `Option<T>`, ми можемо виразити абстрактне поняття необов'язкового значення, і через те, що `Option<T>` є узагальненим, ми можемо використовувати цю абстракцію, незалежно від типу необов'язкового значення.

Енуми також можуть використовувати декілька узагальнених типів. Визначення еnumу `Result`, який ми використовували у розділі 9 є одним з прикладів:

```
enum Result<T, E> {
    Ok(T),
    Err(E),
}
```

Енум `Result` узагальнений відносно двох типів, `T` та `E`, і має два варіанти: `Ok`, який містить значення типу `T`, і `Err`, який містить значення типу `E`. Це визначення робить `Result` зручним для операцій, які можуть мати успішний результат (повернути значення певного типу `T`) або помилку (повернути помилку певного типу `E`). Насправді це те, що ми використовували для відкриття файлу у роздруку 9-3 де `T` був заповнений типом `std::fs::File`, коли файл був успішно відкритий, а `E` був заповнений типом `std::io::Error`, коли виникли проблеми з відкриттям файлу.

When you recognize situations in your code with multiple struct or enum definitions that differ only in the types of the values they hold, you can avoid duplication by using generic types instead.

У Визначеннях Методів

Ми можемо імплементувати методи структур та енамів (як це було у розділі 5), і використовувати у їх визначеннях узагальнені типи. Роздрук 10-9 показує структуру `Point<T>`, яку ми визначили у роздруку 10-6 з імплементованим методом `x`.

Файл: src/main.rs

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

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

fn main() {
    let p = Point { x: 5, y: 10 };

    println!("p.x = {}", p.x());
}
```

Роздрук 10-9: Реалізацію метода з назвою `x` на структурі `Point<T>`, який повертає посилання на поле `x` типу `T`

Тут ми визначили метод з назвою `x` на `Point<T>`, що повертає посилання до даних у полі `x`.

Зверніть увагу, що ми повинні оголосити `T` відразу після `impl`, тож ми можемо використовувати `T`, щоб вказати, що ми застосовуємо методи на типі `Point<T>`. Оголосивши `T` як узагальнений тип після `impl`, Rust може визначити, що тип у кутових дужках у `Point` – узагальнений, а не конкретний тип. Ми могли б вибрати іншу назву, ніж назва параметра з визначення структури, для даного узагальненого параметра, але за домовленістю ми використовуємо ту саму назву. Методи, написані в межах `impl`, який оголошує узагальнений тип, буде визначено в будь-якому екземплярі типу, неважливо, який конкретний тип ми отримаємо, коли підставимо конкретний тип на місце параметра.

Ми також можемо вказати обмеження для узагальнених типів при визначенні методів у типі. Наприклад, ми можемо реалізувати методи лише на екземплярах `Point<f32>`, а не екземплярах `Point<T>` з будь-яким узагальненим типом. У роздрук 10-10 ми використовуємо конкретний тип `f32`, тобто ми не оголошуємо жодних типів після `impl`.

Файл: src/main.rs

```
impl Point<f32> {  
    fn distance_from_origin(&self) -> f32 {  
        (self.x.powi(2) + self.y.powi(2)).sqrt()  
    }  
}
```

Роздрук 10-10: Блок `impl`, який лише застосовується до структури з конкретним значенням типу-параметра `T`

Цей код означає, що тип `Point<f32>` буде мати метод `distance_from_origin`; інші екземпляри `Point<T>`, у яких `T` не є типом `f32` не будуть мати цього методу. Метод вимірює відстань від нашої точки до координати (0,0; 0,0) і використовує математичні операції, які доступні тільки для чисел з рухомою комою.

Типи-параметри у визначеннях структури не завжди такі самі, що й у сигнатурах методів цієї структури. Роздрук 10-11 використовує типи `X1` та `Y1` для структури `Point` і `X2` `Y2` для сигнатури методу `mixup`, щоб краще пояснити цей приклад. Метод створює новий екземпляр `Point` зі значенням `x` з `self` `Point` (з типом `X1`) і значенням `y` з екземпляра `Point`, що передається як параметр (з типом `Y2`).

Файл: src/main.rs

```

struct Point<X1, Y1> {
    x: X1,
    y: Y1,
}

impl<X1, Y1> Point<X1, Y1> {
    fn mixup<X2, Y2>(self, other: Point<X2, Y2>) -> Point<X1, Y2> {
        Point {
            x: self.x,
            y: other.y,
        }
    }
}

fn main() {
    let p1 = Point { x: 5, y: 10.4 };
    let p2 = Point { x: "Hello", y: 'c' };

    let p3 = p1.mixup(p2);

    println!("p3.x = {}, p3.y = {}", p3.x, p3.y);
}

```

Роздрук 10-11: Метод, що використовує типи-параметри, відмінні від тих, що їх має його структура

У `main`, ми визначили `Point`, що має тип `i32` для `x` (зі значенням `5`) і тип `f64` для `y` (зі значенням `10.4`). Змінна `p2` – це структура `Point`, де `x` є слайсом стрічки (зі значенням `"Hello"`), `y` є `char` (зі значенням `c`). Виклик `mixup` на `p1` з аргументом `p2` дає нам `p3`, у якому `x` буде `i32`, тому що `x` береться з `p1`. Змінна `p3` матиме `y` з типом `char`, тому що `y` береться з `p2`. Виклик макроса `println!` виведе в консоль `p3.x = 5, p3.y = c`.

Мета цього прикладу – продемонструвати ситуацію, у якій деякі параметри-типи визначені в `impl`, а деякі у визначенні метода. Тут параметри-типи `X1` і `Y1` оголошені після `impl`, тому що вони відповідають визначенню структури. Параметри-типи `X2` і `Y2` оголошені після `fn mixup`, тому що вони стосуються виключно метода.

Швидкодія Коду з Узагальненими Типами

Можливо, вам цікаво, чи страждає швидкодія, коли ми використовуємо узагальнені типи. Гарна новина в тому, що використання узагальнених типів не зробить вашу програму повільнішою, ніж якби ви використовували конкретні типи.

Rust може досягнути цього за допомогою мономорфізації коду, який використовує узагальнені типи, під час компіляції. *Мономорфізація* – це процес перетворення коду з узагальненими типами в код з конкретними, заповнюючи типів-параметрів конкретними

типами, під час компіляції. У цьому процесі компілятор робить зворотні кроки, до тих, які ми виконали, створюючи узагальнену функцію у роздруку 10-5; компілятор шукає всі місця, де код з узагальненими типами викликається і генерує код для кожного конкретного типу, з яким він викликається.

Погляньмо, як це працює на прикладі узагальненого енуму `Option<T>` зі стандартної бібліотеки:

```
let integer = Some(5);  
let float = Some(5.0);
```

Коли Rust компілює цей код, він виконує мономорфізацію. Під час цього процесу, компілятор читає значення, які були використані в екземплярах `Option<T>` і визначає два види `Option<T>`: один з `i32`, а інший – з `f64`. Таким чином, він розкладає узагальнене визначення `Option<T>` на два визначення, які використовують `i32` і `f64`, замінюючи узагальнене визначення на визначення з конкретизовані.

Мономорфізована версія коду схожа на таке (компілятор використовує інші імена, ніж ті, що ми використовуємо тут для ілюстрації):

Файл: src/main.rs

```
enum Option_i32 {  
    Some(i32),  
    None,  
}  
  
enum Option_f64 {  
    Some(f64),  
    None,  
}  
  
fn main() {  
    let integer = Option_i32::Some(5);  
    let float = Option_f64::Some(5.0);  
}
```

Узагальнене `Option<T>` замінюється на конкретизовані визначення, створені компілятором. Оскільки Rust компілює код з узагальненими типами в код, який вказує тип в кожному випадку, ми не платимо за використання узагальнених типів під час виконання. Коли код запускається, він виконується так само, як і якби ми продублювали кожне визначення вручну. Процес мономорфізації робить узагальнені типи в Rust надзвичайно ефективними під час виконання коду.

Трейти: Визначення Спільної Поведінки

Трейт визначає функціональність, якою володіє визначений тип та якою він може ділитися з іншими типами. Ми можемо використовувати трейти, щоб визначати спільну поведінку абстрактним способом. Також маємо змогу застосувати *обмеження трейту*, щоб вказати, що загальний тип, може бути будь-яким типом, який реалізує певну поведінку.

Note: Traits are similar to a feature often called *interfaces* in other languages, although with some differences.

Визначення Трейту

Поведінка типу визначається тими методами, які ми можемо викликати у цього типу. Різні типи розділяють однакову поведінку, якщо ми можемо викликати одні й ті самі методи цих типів. Визначення трейтів - це спосіб згрупувати сигнатури методів разом, заради того, щоб описати загальну поведінку, необхідну для досягнення певної мети.

For example, let's say we have multiple structs that hold various kinds and amounts of text: a `NewsArticle` struct that holds a news story filed in a particular location and a `Tweet` that can have at most 280 characters along with metadata that indicates whether it was a new tweet, a retweet, or a reply to another tweet.

Ми хочемо створити бібліотечний крейт медіа агрегатору під назвою `aggregator`, який може відображати зведення даних, які збережені в екземплярах структур `NewsArticle` чи `Tweet`. Щоб це зробити, нам треба мати можливість для кожної структури зробити коротке зведення на основі даних, які маємо: для цього треба, щоб обидві структури реалізували загальну поведінку, в нашому випадку це буде виклик методу `summarize` в екземпляра об'єкту. Лістинг 10-12 ілюструє визначення публічного трейту `Summary`, який висловлює таку поведінку.

Файл: `src/lib.rs`

```
pub trait Summary {  
    fn summarize(&self) -> String;  
}
```

Блок коду 10-12: Визначення трейту `Summary`, що містить поведінку, надану методом `summarize`

Тут ми визначаємо трейт, використовуючи ключове слово `trait`, а потім його назву, якою є `Summary` в цьому випадку. Ми також визначили цей трейт як `pub`, щоб крейти, які залежать від цього крейту, також могли використовувати цей трейт, як ми побачимо в декількох прикладах. В середині фігурних дужок визначаються сигнатури методів, які описують поведінку типів, які реалізують цей трейт. У цьому випадку поведінка визначається тільки однією сигнатурою методу: `fn summarize(&self) -> String`.

Після сигнатури методу, замість надання реалізації у фігурних дужках, ми використовуємо крапку з комою. Кожен тип, який реалізує цей трейт, повинен надати свою власну поведінку для цього методу. Компілятор забезпечить, що будь-який тип, який містить трейт `Summary`, буде також мати й метод `summarize` визначений з точно такою сигнатурою.

Трейт може мати декілька методів у описі його тіла: сигнатури методів перераховуються по одній на кожному рядку та повинні закінчуватися крапкою з комою.

Реалізація Трейту для Типів

Тепер, після того, як ми визначили бажану поведінку, використовуючи трейт `Summary`, можна реалізувати його для типів у нашому медіа агрегатору. Лістинг 10-13 показує реалізацію трейту `Summary` для структури `NewsArticle`, яка використовує для створення зведення в методі `summarize` заголовок, автора та місце публікації. Для структури `Tweet` ми визначаємо реалізацію `summarize`, використовуючи користувача та повний текст твіту, вважаючи зміст вже обмеженим 280 символами.

Файл: `src/lib.rs`

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

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

pub struct Tweet {
    pub username: String,
    pub content: String,
    pub reply: bool,
    pub retweet: bool,
}

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

Блок коду 10-13: Реалізація трейту `Summary` для структур `NewsArticle` та `Tweet`

Реалізація трейту для типу аналогічна реалізації звичайних методів. Різниця в тому, що після `impl` ми пишемо ім'я трейту, який ми хочемо реалізувати, після чого використовуємо ключове слово `for`, а потім вказуємо ім'я типу, для якого ми хочемо зробити реалізацію трейту. Всередині блоку `impl` ми розташовуємо сигнатуру методу, яка визначена в трейту. Замість додавання крапки з комою в кінці, після кожної сигнатури використовуються фігурні дужки, та тіло методу заповнюється конкретною поведінкою, яку ми хочемо отримати у методів трейту для конкретного типу.

Тепер, коли в бібліотеці реалізований трейт `Summary` для `NewsArticle` та `Tweet`, користувачі крейту можуть викликати методи трейту для екземплярів `NewsArticle` й `Tweet`, так само як ми викликаємо звичайні методи. Єдина різниця в тому, що користувач повинен ввести в область видимості трейти, а також типи. Ось приклад як бінарний крейт може використовувати наш `aggregator`:


```
use aggregator::{Summary, Tweet};

fn main() {
    let tweet = Tweet {
        username: String::from("horse_ebooks"),
        content: String::from(
            "of course, as you probably already know, people",
        ),
        reply: false,
        retweet: false,
    };

    println!("1 new tweet: {}", tweet.summarize());
}
```

Цей код надрукує: `1 new tweet: horse_ebooks: of course, as you probably already know, people`.

Інші крейти, які залежать від крейту `aggregator`, також можуть взяти `Summary` в область видимості, щоб реалізувати `Summary` для своїх власних типів. Слід зазначити одне обмеження: ми можемо реалізувати трейт для типу тільки в тому випадку, якщо хоча б один трейт чи тип є локальними для нашого крейту. Наприклад, ми можемо реалізувати стандартні бібліотечні трейти, такі як `Display` на користувальницькому типі `Tweet`, як частина функціональності нашого крейту `aggregator`, тому що тип `Tweet` є локальним для нашого крейту `aggregator`. Також ми можемо реалізувати `Summary` для `Vec<T>` в нашому крейті `aggregator`, оскільки трейт `Summary` є локальним для нашого крейту `aggregator`.

Але ми не можемо реалізувати зовнішні трейти на зовнішніх типах. Наприклад, ми не можемо реалізувати трейт `Display` для `Vec<T>` в нашому крейті `aggregator`, тому що `Display` та `Vec<T>` визначені у стандартній бібліотеці та не є локальними для нашого крейту `aggregator`. Це обмеження є частиною властивості, яка називається *узгодженість* (coherence), та, більш конкретно *правило сироти* (orphan rule), яке назвали так, тому що батьківський тип відсутній. Це правило гарантує, що чужий код не може порушити ваш код, та навпаки. Без цього правила два крейти мали б змогу реалізовувати один й той самий трейт для одного й того самого типу, і Rust не знав би, яку реалізацію використовувати.

Реалізація Поведінки за Замовчуванням

Іноді корисно мати поведінку за замовчуванням для деяких чи всіх методів трейту замість того, щоб вимагати реалізації всіх методів у кожному типі, що реалізує цей трейт. Потім, коли ми реалізуємо трейт для певного типу, можна зберегти чи перевизначити поведінку кожного методу за замовчуванням вже всередині типів.

В Блоці коду 10-14 показано, як вказати стрічку за замовчуванням для методу `summarize` з трейту `Summary` замість визначення тільки сигнатури методу, як ми робили в Блоці коду 10-12.

Файл: `src/lib.rs`

```
pub trait Summary {  
    fn summarize(&self) -> String {  
        String::from("(Read more...)")  
    }  
}
```

Блок коду 10-14: Визначення трейту `Summary` з реалізацією методу `summarize` за замовчуванням

Для використання реалізації за замовчуванням під час створення зведення в екземплярах `NewsArticle`, ми вказуємо порожній блок `impl` з `impl Summary for NewsArticle {}`.

Хоча ми більше не визначаємо метод `summarize` безпосередньо в `NewsArticle`, ми надали реалізацію за замовчуванням та вказали, що `NewsArticle` реалізує трейт `Summary`. В результаті ми все ще маємо змогу викликати метод `summarize` в екземпляра `NewsArticle`, наприклад таким чином:

```
let article = NewsArticle {  
    headline: String::from("Penguins win the Stanley Cup Championship!"),  
    location: String::from("Pittsburgh, PA, USA"),  
    author: String::from("Iceburgh"),  
    content: String::from(  
        "The Pittsburgh Penguins once again are the best \  
        hockey team in the NHL."  
    ),  
};  
  
println!("New article available! {}", article.summarize());
```

Цей код виведе в консолі `New article available! (Read more...)`.

Створення реалізації за замовчуванням не вимагає від нас змін чого-небудь у реалізації `Summary` для типу `Tweet` у Блоці коду 10-13. Причина полягає в тому, що синтаксис для перевизначення реалізації за замовчуванням є таким, як синтаксис для реалізації метода трейту, котрий не має реалізації за замовчуванням.

Реалізації за замовчуванням можуть викликати інші методи в тому ж трейті, навіть якщо ці методи не мають реалізації за замовчуванням. Таким чином, трейт може надати багато корисної функціональності, тільки вимагаючи від розробників вказувати невелику його

частину. Наприклад, ми мали змогу б визначити трейт `Summary`, який має метод `summarize_author`, реалізація якого вимагається, а потім визначити метод `summarize`, який має реалізацію за замовчуванням, котра всередині викликає метод `summarize_author`:

```
pub trait Summary {
    fn summarize_author(&self) -> String;

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

Щоб використовувати таку версію трейту `Summary`, потрібно тільки визначити метод `summarize_author`, під час реалізації трейту для типу:

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

Після того, як ми визначимо `summarize_author`, можемо викликати `summarize` для екземплярів структури `Tweet` і реалізація за замовчуванням методу `summarize` буде викликати визначення `summarize_author`, яке ми вже надали. Оскільки ми реалізували метод `summarize_author` трейту `Summary`, то трейт дає нам поведінку метода `summarize` без необхідності писати код.

```
let tweet = Tweet {
    username: String::from("horse_ebooks"),
    content: String::from(
        "of course, as you probably already know, people",
    ),
    reply: false,
    retweet: false,
};

println!("1 new tweet: {}", tweet.summarize());
```

Цей код друкує: `1 new tweet: (Read more from @horse_ebooks...)`.

Зверніть увагу, що неможливо викликати реалізацію за замовчуванням з перевизначеної реалізації того ж самого методу.

Трейти як Параметри

Тепер, коли ви знаєте, як визначати та реалізовувати трейти, можна вивчити, як використовувати трейти, щоб визначити функції, які приймають багато різних типів. Ми будемо використовувати трейт `Summary`, який ми реалізували для типів `NewsArticle` та `Tweet` у Блоці коду 10-13, щоб визначити функцію `notify`, яка викликає метод `summarize` для свого параметра `item`, який є деяким типом, який реалізує трейт `Summary`. Для цього ми використовуємо синтаксис `impl Trait`, наприклад таким чином:

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

Замість конкретного типу в параметрі `item` вказується ключове слово `impl` та ім'я трейту. Цей параметр приймає будь-який тип, який реалізує вказаний трейт. У тілі `notify` ми маємо змогу викликати будь-які методи в екземпляра `item`, які повинні бути визначені при реалізації трейту `Summary`, наприклад можна викликати метод `summarize`. Ми можемо викликати `notify` та передати в нього будь-який екземпляр `NewsArticle` чи `Tweet`. Код, який викликає цю функцію з будь-яким іншим типом, таким як `String` чи `i32`, не буде компілюватися, тому що ці типи не реалізують трейт `Summary`.

Синтаксис Обмеження Трейту

Синтаксис `impl Trait` працює для простих випадків, але насправді є синтаксичним цукром для більш довгої форми, яка називається *обмеження трейту*, це виглядає ось так:

```
pub fn notify<T: Summary>(item: &T) {  
    println!("Breaking news! {}", item.summarize());  
}
```

Ця більш довга форма еквівалентна прикладу в минулому розділі, але вона більш багатослівна. Ми розміщуємо оголошення параметра узагальненого типа з обмеженням трейту після двокрапки всередині кутових дужок.

Синтаксис `impl Trait` зручний та робить більш виразним код у простих випадках, в той час, як більш повний синтаксис обмеження трейту може висловити більшу складність в інших випадках. Наприклад, у нас може бути два параметри, які реалізують трейт `Summary`. Використання синтаксису `impl Trait` виглядає наступним чином:

```
pub fn notify(item1: &impl Summary, item2: &impl Summary) {
```

Використання `impl Trait` доцільно, якщо ми хочемо, щоб ця функція дозволяла `item1` та `item2` мати різні типи (за умовою, що обидва типи реалізують `Summary`). Якщо ми хочемо змусити обидва параметри мати один й той самий тип, ми повинні використовувати обмеження трейту, наприклад, ось так:

```
pub fn notify<T: Summary>(item1: &T, item2: &T) {
```

Узагальнений тип `T` зазначений як тип параметрів `item1` та `item2` обмежує функцію таким чином, що конкретний тип значення переданого як аргумент для `item1` і `item2` має бути однаковим.

Визначення Кількох Обмежень Трейтів із Синтаксисом `+`

Також можна вказати більше одного обмеження трейту. Скажімо, ми хочемо, щоб `notify` використовував форматування відображення, а також `summarize` для `item`: ми вказуємо у визначенні `notify`, що `item` повинен реалізувати `Display` та `Summary` одночасно. Це можна зробити за допомогою синтаксису `+`:

```
pub fn notify(item: &(impl Summary + Display)) {
```

Синтаксис `+` також валідний з обмеженням трейту для узагальнених типів:

```
pub fn notify<T: Summary + Display>(item: &T) {
```

За наявності двох обмежень трейту, тіло методу `notify` може викликати `summarize` та використовувати `{}` для форматування `item` під час його друку.

Чіткіші Межі Трейту з `where`

Використання занадто великої кількості обмежень трейту має свої недоліки. Кожен узагальнений тип має свої межі трейту, тому функції з декількома параметрами узагальненого типу можуть містити багато інформації про обмеження між назвою функції та списком її параметрів, що ускладнює читання сигнатури. З цієї причини в Rust є альтернативний синтаксис для визначення обмежень трейту всередині блок `where` після сигнатури функції. Тому замість того, щоб писати ось так:

```
fn some_function<T: Display + Clone, U: Clone + Debug>(t: &T, u: &U) -> i32 {
```

Можна використати блок `where`, наприклад таким чином:

```
fn some_function<T, U>(t: &T, u: &U) -> i32
where
    T: Display + Clone,
    U: Clone + Debug,
{
```

Сигнатура цієї функції менш захаращена: назва функції, список параметрів, та тип значення, що повертається, знаходяться поруч, а сигнатура не містить в собі множину обмежень трейту.

Повернення Значень Типу, що Реалізує Певний Трейт

We can also use the `impl Trait` syntax in the return position to return a value of some type that implements a trait, as shown here:

```
fn returns_summarizable() -> impl Summary {
    Tweet {
        username: String::from("horse_ebooks"),
        content: String::from(
            "of course, as you probably already know, people",
        ),
        reply: false,
        retweet: false,
    }
}
```

Використовуючи `impl Summary` для типу, що повертається, ми вказуємо, що функція `returns_summarizable` повертає деяких тип, який реалізує трейт `Summary` без позначення конкретного типу. В цьому випадку `returns_summarizable` повертає `Tweet`, але код, який викликає цю функцію, цього не знає.

Можливість повертати тип, який визначається тільки ознакою, яку він реалізує, є особливо корисна в контексті замикань та ітераторів, які ми розглянемо у розділі 13. Замикання та ітератори створюють типи, які відомі тільки компілятору, або типи, які дуже довго визначати. Синтаксис `impl Trait` дозволяє вам лаконічно вказати, що функція повертає деяких тип, що реалізує ознаку `Iterator`, без необхідності вказувати дуже довгий тип.

Проте, `impl Trait` можливо використовувати, якщо ви повертаєте тільки один тип. Наприклад, цей код, який повертає значення типу `NewsArticle` або `Tweet`, але як тип, що повертається, оголошує `impl Summary`, не буде працювати:

```
fn returns_summarizable(switch: bool) -> impl Summary {
    if switch {
        NewsArticle {
            headline: String::from(
                "Penguins win the Stanley Cup Championship!",
            ),
            location: String::from("Pittsburgh, PA, USA"),
            author: String::from("Iceburgh"),
            content: String::from(
                "The Pittsburgh Penguins once again are the best \
                hockey team in the NHL.",
            ),
        }
    } else {
        Tweet {
            username: String::from("horse_ebooks"),
            content: String::from(
                "of course, as you probably already know, people",
            ),
            reply: false,
            retweet: false,
        }
    }
}
```

Повертання або `NewsArticle` або `Tweet` не дозволяється через обмеження того, як реалізований синтаксис `impl Trait` в компіляторі. Ми подивимося, як написати функцію з такою поведінкою у секції [“Використання об'єктів трейтів, які дозволяють використовувати значення різних типів”](#) розділу 17.

Використання Обмежень Трейту для Умовної Реалізації Методів

Використовуючи обмеження трейту з блоком `impl`, який використовує параметри узагальненого типу, можна реалізувати методи умовно, для тих типів, які реалізують вказаний трейт. Наприклад, тип `Pair<T>` у Блоці коду 10-15 завжди реалізує функцію `new` для повертання нового екземпляру `Pair<T>` (нагадаємо з секції [“Визначення методів”](#) розділу 5, що `Self` це псевдонім типу для типа блоку `impl`, який в цьому випадку є `Pair<T>`). Але в наступному блоці `impl`, `Pair<T>` реалізує тільки метод `cmp_display`, якщо його внутрішній тип `T` реалізує трейт `PartialOrd`, який дозволяє порівнювати `i` трейт `Display`, який забезпечує друкування.

Файл: src/lib.rs

```

use std::fmt::Display;

struct Pair<T> {
    x: T,
    y: T,
}

impl<T> Pair<T> {
    fn new(x: T, y: T) -> Self {
        Self { x, y }
    }
}

impl<T: Display + PartialOrd> Pair<T> {
    fn cmp_display(&self) {
        if self.x >= self.y {
            println!("The largest member is x = {}", self.x);
        } else {
            println!("The largest member is y = {}", self.y);
        }
    }
}

```

Лістинг 10-15: Умовна реалізація методів в узагальнених типів в залежності від обмежень трейту

Ми також можемо умовно реалізувати трейт для будь-якого типу, який реалізує інший трейт. Реалізація трейту для будь-якого типу, що задовольняє обмеженням трейту називається *загальною реалізацією* (blanket implementations) й широко використовується в стандартній бібліотеці Rust. Наприклад, стандартна бібліотека реалізує трейт `ToString` для будь-якого типу, який реалізує трейт `Display`. Блок `impl` в стандартній бібліотеці виглядає приблизно так:

```

impl<T: Display> ToString for T {
    // --snip--
}

```

Оскільки стандартна бібліотека має загальну реалізацію, то можна викликати метод `to_string` визначений трейтом `ToString` для будь-якого типу, який реалізує трейт `Display`. Наприклад, ми можемо перетворити цілі числа в їх відповідні `String` значення, тому що цілі числа реалізують трейт `Display`:

```

let s = 3.to_string();

```

Загальні реалізації наведені в документації до трейту в розділі “Implementors”.

Трейти та обмеження трейтів дозволяють писати код, який використовує параметри узагальненого типу для зменшення дублювання коду, а також вказання компілятору, що ми хочемо узагальнений тип, щоб мати визначену поведінку. Потім компілятор може використовувати інформацію про обмеження трейту, щоб перевірити, що всі конкретні типи, які використовуються з нашим кодом, забезпечують правильну поведінку. У динамічно типізованих мовах програмування ми отримали б помилку під час виконання, якби викликали метод для типу, який не реалізує тип визначений методом. Але Rust перекладає ці помилки на час компіляції, тому ми повинні виправити проблеми, перш ніж наш код почне працювати. Крім того, ми не повинні писати код, який перевіряє свою поведінку під час компіляції, тому що це вже перевірено під час компіляції. Це підвищує швидкодію без необхідності відмовлятися від гнучкості узагальнених типів.

Перевірка Коректності Посилань із Часами Існування

Часи існування є ще одним узагальненим типом даних, який ми вже використовували. Замість того щоб гарантувати, що тип має бажану поведінку, часи існування гарантують що посилання є валідним доти, доки воно може бути нам потрібним.

В частині “Посилання і позичання” четвертого розділу ми не згадали про те, що кожне посилання в Rust має свій *час існування*, який обмежує час протягом якого посилання є дійсним. В більшості випадків, часи існування є неявними (implicit) та виведеними (inferred), так само як і типи. Ми зобов’язані додавати анотації лише у випадках коли можливий більше ніж один тип. Відповідно, ми мусимо додавати анотації до часів існування лише якщо останні можуть бути використані у кілька різних способів. Rust зобов’язує нас анотувати зв’язки використовуючи узагальнені параметри часу існування, щоб впевнитися що посилання використані протягом часу виконання програми будуть коректними.

Додавання анотацій для часів існування не є поширеним в інших мовах програмування, тож може бути здаватися дещо складним для сприйняття. Попри те що в цьому розділі ми не розглядатимемо всі деталі часів існування, ми обговоримо їх найбільш загальні способи використання, щоб в подальшому у вас не виникало проблем зі сприйняттям цієї концепції.

Запобігання Висячим Посиланням із Часами Існування

Головною метою використання часів існування є запобігання *висячим посиланням*, які зберігають в пам’яті дані, котрі більше не будуть використані програмою. Розглянемо приклад з блоку коду 10-16, який має внутрішню і зовнішню область видимості.

```
fn main() {  
    let r;  
  
    {  
        let x = 5;  
        r = &x;  
    }  
  
    println!("r: {}", r);  
}
```

Блок коду 10-16: Спроба використати посилання, значення якого лежить за межами області видимості

Примітка: Приклади у Блоках коду 10-16, 10-17 та 10-23 проголошують змінні без надання їм початкового значення, тож, назва змінної існує в зовнішньої області видимості. На перший погляд, може видатися, що це суперечить тому, що Rust не має нульових значень. Однак, якщо ми спробуємо використати змінну перед наданням їй значення, ми отримаємо помилку часу компіляції, що показує, що Rust дійсно не допускає значення `null`.

Зовнішня область видимості проголошує змінну з назвою `r` без початкового значення, а внутрішня область видимості проголошує змінну з назвою `x` з початковим значенням 5. У середині внутрішньої області видимості ми намагаємося встановити значення `r` у посилання до `x`. Тоді внутрішня область видимості закінчується, і ми намагаємося вивести значення `r`. Цей код не компілюється, бо значення, на яке посилається `r`, вийшло з області видимості до того, як ми спробували ним скористатися. Ось повідомлення про помилку:

```
$ cargo run
  Compiling chapter10 v0.1.0 (file:///projects/chapter10)
error[E0597]: `x` does not live long enough
--> src/main.rs:6:13
   |
6  |         r = &x;
   |         ^^ borrowed value does not live long enough
7  |     }
   |     - `x` dropped here while still borrowed
8  |
9  |     println!("r: {}", r);
   |                       - borrow later used here

For more information about this error, try `rustc --explain E0597`.
error: could not compile `chapter10` due to previous error
```

Змінна `x` не "існує достатньо довго". Причина в тому, що `x` вийде з області видимості коли внутрішня область видимості скінчиться у рядку 7. Але змінна `r` все ще валідна у зовнішній області видимості; оскільки її область видимості більша, ми кажемо, що вона "існує довше". Якби Rust дозволив цьому коду працювати, `r` посилався би на пам'ять, що була звільнена, коли `x` вийшов з області видимості, і все, що ми намагатимемося робити з `r`, не працюватиме належним чином. То як Rust визначає, що код є некоректним? Вона використовує borrow checker.

The Borrow Checker

Компілятор Rust має *borrow checker*, який порівнює області видимості і визначає, чи всі позичання валідні. Блок коду 10-17 показує такий самий код, як у Блоці коду 10-16, але з анотаціями, які показують часи існування змінних.

```
fn main() {
    let r;           // -----+-- 'a
                    //
    {
        let x = 5;   // -+-- 'b
        r = &x;      // |
    }               // -+
                    //
    println!("r: {}", r); //
}                  // -----+
```

Блок коду 10-17: анотації часів існування `r` та `x`, що називаються відповідно `'a` та `'b`

Тут ми анотували час існування `r` як `'a` і час існування `x` як `'b`. Як бачите, час існування внутрішнього блоку `'b` є набагато меншим, ніж зовнішнього `'a`. Під час компіляції Rust порівнює розмір двох часів існування і бачить, що `r` має час існування `'a`, але він посилається на пам'ять з часом існування `'b`. Програма буде відхилена через те, що `'b` коротший за `'a`: те, на що посилаються, існує менше, ніж посилання.

Блок коду 10-18 виправляє код, щоб не було всячого посилання, і компілюється без жодних помилок.

```
fn main() {
    let x = 5;       // -----+-- 'b
                    //
    let r = &x;      // --+-- 'a
                    // |
    println!("r: {}", r); // |
                    // --+
                    // -----+
```

Блок коду 10-18: посилання є валідним, бо дані мають довший час існування, ніж посилання

Тут `x` має час існування `'b`, що у цьому випадку більше, ніж `'a`. Це означає, що `r` може посилатись на `x`, тому що Rust знає, що посилання в `r` завжди буде дійсним, поки `x` є дійсним.

Тепер, коли ви знаєте, де знаходяться часи існування посилань і як Rust аналізує часи існування, щоб гарантувати, що посилання завжди будуть валідними, розгляньмо

узагальнені часи існування параметрів та значень, що повертаються, в контексті функцій.

Узагальнені Часи Існування у Функціях

Ми напишемо функцію, яка повертає довший з двох стрічкових слайсів. Ця функція прийматиме два стрічкові слайси і повертатиме один слайс. Після реалізації функції

`longest`, код у Блоці коду 10-19 має вивести `The longest string is abcd`.

Файл: `src/main.rs`

```
fn main() {  
    let string1 = String::from("abcd");  
    let string2 = "xyz";  
  
    let result = longest(string1.as_str(), string2);  
    println!("The longest string is {}", result);  
}
```

Блок коду 10-19: функція `main`, що викликає функцію `longest` щоб знайти довший з двох стрічкових слайсів

Зверніть увагу, що нам потрібно, щоб функція приймала рядки фрагментів, які є посиланнями, а не стрічки, тому що ми не хочемо, щоб функція `longest` брала володіння над своїми параметрами. Зверніться до підрозділу "Стрічкові слайси як параметри" Розділу 4 для детальнішого обговорення, чому ми хочемо використати саме такі параметри в Блоці коду 10-19.

Якщо ми спробуємо реалізувати функцію `longest` як показано у Блоці коду 10-20, вона не скомпілюється.

Файл: `src/lib.rs`

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

Блок коду 10-20: реалізація функції `longest`, що повертає довший з двох стрічкових слайсів, але ще не компілюється

Натомість ми отримуємо наступну помилку, яка говорить про часи існування:

```

$ cargo run
  Compiling chapter10 v0.1.0 (file:///projects/chapter10)
error[E0106]: missing lifetime specifier
  --> src/main.rs:9:33
   |
 9 | fn longest(x: &str, y: &str) -> &str {
   |          ----      ----      ^ expected named lifetime parameter
   |
   = help: this function's return type contains a borrowed value, but the signature
does not say whether it is borrowed from `x` or `y`
help: consider introducing a named lifetime parameter
   |
 9 | fn longest<'a>(x: &'a str, y: &'a str) -> &'a str {
   |          ++++    ++          ++          ++

```

For more information about this error, try `rustc --explain E0106`.

error: could not compile `chapter10` due to previous error

Текст підказки показує, що тип, який повертається, потребує для себе вказаного узагальненого часу існування, оскільки Rust не може сказати, буде повернуте посилання `x` чи `y`. Насправді ми також цього не знаємо, оскільки блок `if` у тілі цієї функції повертає посилання на `x`, а блок `else` повертає посилання на `y`!

Коли ми визначаємо цю функцію, ми не знаємо конкретних значень, які будуть передані в цю функцію, тому ми не знаємо, спрацює випадок `if` чи `else`. Ми також не знаємо конкретного часу існування посилань, які будуть передані, тож ми не можемо подивитися на область видимості, як ми робили у Блоках коду 10-17 та 10-18, щоб визначити, чи посилання, яке ми повертаємо, буде завжди валідним. Borrow checker також не може визначити цього, оскільки він не знає як час існування `x` або `y` стосується часу існування значення, що повертається. Щоб виправити цю помилку, ми додамо узагальнені параметри часу існування, які визначають зв'язок між посиланнями, щоб borrow checker міг його проаналізувати.

Синтаксис Анотації Часу Існування

Анотації часу існування не змінюють як довго існує посилання. Натомість вони описують взаємозв'язок часів існування багатьох посилань між собою, не впливаючи на ці часи існування. Так само як функції можуть приймати будь-який тип, коли сигнатура визначає параметр узагальненого типу, функції можуть приймати посилання з будь-яким часом існування, якщо заданий узагальнений параметр часу існування.

Анотації часу існування мають дещо незвичний синтаксис: імена параметрів часу існування мають починатися на апостроф (`'`) і зазвичай в нижньому регістрі та дуже короткі, як узагальнені типи. Більшість людей використовують ім'я `'a` для першої

анотації часу існування. Ми розміщуємо анотації часу існування параметрів після `&` посилання, використовуючи пробіл для відокремлення анотації від типу посилання.

Ось кілька прикладів: посилання на `i32` без параметру часу існування, посилання на `i32`, що має параметр часу існування `'a`, і мутабельне посилання на `i32`, що також має час існування `'a`.

```
&i32          // посилання
&'a i32       // посилання з явним часом існування
&'a mut i32   // мутабельне посилання з явним часом існування
```

Сам по собі одна анотація часу існування не має великого значення. оскільки анотації мають повідомляти Rust, як узагальнені параметри часу існування багатьох посилань співвідносяться один з одним. Дослідімо, як анотації часу існування співвідносяться одна з одною в контексті функції `longest`.

Анотації Часу Існування у Сигнатурах Функцій

Щоб використовувати анотації часу існування в сигнатурах функцій, ми маємо проголосити узагальнений параметр *часу існування* в кутових дужках між назвою функції і списком параметрів, так, як це робиться з узагальненими параметрами *типу*.

Ми хочемо, щоб сигнатура виражала наступне обмеження: повернуте посилання буде валідним стільки, скільки обидва параметри будуть валідними. Це відношення між часом існування параметрів та значення, що повертається. Ми назовемо час існування `'a`, а тоді додамо його до кожного посилання, як показано в Блоці коду 10-21.

Файл: src/main.rs

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

Блок коду 10-21: Визначення функції `longest` із зазначенням, що всі посилання у сигнатурі повинні мати однаковий час існування `'a`

Цей код має компілюватися і створювати бажаний результат, коли ми використовуємо його у функції `main` у Блоці коду 10-19.

Сигнатура функції тепер повідомляє Rust, що для якогось часу існування `'a` функція приймає два параметри, обидва з яких - стрічкові слайси, які існують принаймні так довго, як час існування `'a`. Сигнатура функції також каже Rust, що стрічковий слайс, повернутий з функції, існуватиме щонайменше так довго, як час існування `'a`. На практиці це означає, що час існування посилання, повернутого функцією `longest`, дорівнює коротшому з часів існування значень, на які посилаються аргументи функції. Ці відносини - це те, що ми хочемо, щоб Rust використовувала під час аналізу цього коду.

Пам'ятайте, що коли ми зазначаємо час існування параметрів на сигнатурі цієї функції, ми не змінюємо час існування жодного значення, які передаються або повертаються. Радше ми уточнюємо, що borrow checker повинен відкидати будь-які значення, які не відповідають цим обмеженням. Зверніть увагу, що функція `longest` не повинна точно знати, як довго `x` і `y` будуть існувати, лише те, що деяка область видимості може бути замінена на `'a` що задовольняє цій сигнатурі.

При анотації часів існування у функції анотації зазначаються у сигнатурі функції, а не в її тілі. Анотації часу існування стають частиною контракту функції, так само, як типи у сигнатурі. Те, що сигнатури функцій містять контракт на час існування, означає, що аналіз, який робить компілятор Rust, буде простішим. Якщо є проблема з тим, як функція анована або як її викликають, помилки компілятора можуть точніше вказувати на частину нашого коду та обмеження. Якщо натомість компілятор Rust робив більше припущень про те, якими ми хотіли б бачити відносини між часами існування, компілятор міг би вказувати лише на використання нашого коду за багато кроків від причини проблеми.

Коли ми передаємо конкретні посилання до `longest`, конкретний час існування, що підставляється в `'a`, є частиною області видимості `x`, що перетинається з областю видимості `y`. Іншими словами, узагальнений час існування `'a` отримає конкретний час існування, що є рівним меншому з часів існування `x` та `y`. Оскільки ми анотували посилання, що повертається, тим самим параметром часу існування `'a`, посилання, що повертається, також буде валідним під час тривалості меншого з часів існування `x` та `y`.

Подивімося, як анотації часу існування обмежують функцію `longest`, передаючи посилання, що мають різне конкретні часи існування. Блок коду 10-22 надає прямолінійний приклад.

Файл: src/main.rs


```
fn main() {  
    let string1 = String::from("long string is long");  
  
    {  
        let string2 = String::from("xyz");  
        let result = longest(string1.as_str(), string2.as_str());  
        println!("The longest string is {}", result);  
    }  
}
```

Блок коду 10-22: використання функції "longest" з посиланнями на значення "String", які мають різний конкретний час існування

У цьому прикладі `string1` буде валідним до кінця зовнішньої області видимості, `string2` до кінця внутрішньої області видимості, а `result` посилається на щось, що є валідним до кінця внутрішньої області видимості. Запустіть цей код, і ви побачите, що borrow checker його приймає; код скомпілюється і виведе `The longest string is long string is long`.

Далі розгляньмо приклад, який показує, що час існування посилання в `result` має бути меншим з тривалостей існування двох аргументів. Ми перемістимо оголошення змінної `result` за межі внутрішньої області видимості, але залишимо присвоєння значення змінній `result` усередині області видимості зі `string2`. Потім ми перемістимо `println!`, який використовує `result`, за межі внутрішньої області видимості, її після закінчення. Код в Блоці коду 10-23 не скомпілюється.

Файл: src/main.rs

```
fn main() {  
    let string1 = String::from("long string is long");  
    let result;  
    {  
        let string2 = String::from("xyz");  
        result = longest(string1.as_str(), string2.as_str());  
    }  
    println!("The longest string is {}", result);  
}
```

Блок коду 10-23: спроба використати `result` після виходу `string2` з області видимості

Якщо ми спробуємо скомпілювати цей код, то отримаємо таку помилку:

```

$ cargo run
   Compiling chapter10 v0.1.0 (file:///projects/chapter10)
error[E0597]: `string2` does not live long enough
  --> src/main.rs:6:44
   |
6  |         result = longest(string1.as_str(), string2.as_str());
   |                                     ^^^^^^^^^^^^^^^^^^^^^^^^^ borrowed value
does not live long enough
7  |     }
   |     - `string2` dropped here while still borrowed
8  |     println!("The longest string is {}", result);
   |                                     ----- borrow later used here

For more information about this error, try `rustc --explain E0597`.
error: could not compile `chapter10` due to previous error

```

Помилка показує, що для того, що `result` був валідним для інструкції `println!`, `string2` має бути валідним до кінця зовнішньої області видимості. Rust знає це, бо ми анотували час існування параметрів функції та значення, що повертається, за допомогою того самого параметра часу існування `'a`.

Як люди, ми можемо подивитися на цей код і побачити, що `string1` довша за `string2` і тому `result` міститиме посилання на `string1`. Оскільки `string1` ще не вийшла з області видимості, посилання на `string1` все ще буде валідним для інструкції `println!`. Однак, компілятор не може побачити, що посилання в цьому випадку валідне. Ми повідомили Rust, що час існування посилання, що повертається функцією `longest`, такий самий, як менший з часів існування переданих їй посилань. Тому borrow checker забороняє код у Блоці коду 10-23 як такий, що потенційно містить неправильне посилання.

Спробуйте провести більше експериментів, які змінюють значення і часи існування посилань, переданих у функцію `longest`, а також використання посилання, що повертається. Робіть припущення про те, чи пройдуть ваші експерименти borrow checker до компіляції; потім перевірте, щоб побачити, чи маєте ви рацію!

Мислення в Термінах Часів Існування

Спосіб, яким треба позначати параметри часу існування, залежить від того, що саме робить ваша функція. Наприклад, якби ми змінили реалізацію функції `longest`, щоб та завжди повертала перший параметр замість найдовшої стрічки, то нам не потрібно було б вказувати час існування для параметра `y`. Цей код буде скомпільований:

Файл: src/main.rs

```
fn longest<'a>(x: &'a str, y: &str) -> &'a str {
    x
}
```

Ми зазначили параметр часу існування `'a` для параметра `x` та типу, що повертається, але не для параметра `y`, оскільки час існування `y` у не має стосунку до часу існування `x` або значення, що повертається.

При поверненні посилання з функції, параметр часу існування для типу, що повертається, має збігатися з параметром часу існування для одного з параметрів. Якщо повернуте посилання *не* посилається на один з параметрів, воно повинне посилатись на значення, створене всередині цієї функції. Однак це буде всяке посилання, тому що значення вийде з області видимості в кінці функції. Розглянемо спробу реалізації функції `longest`, яка не компілюється:

Файл: src/main.rs

```
fn longest<'a>(x: &str, y: &str) -> &'a str {
    let result = String::from("really long string");
    result.as_str()
}
```

Тут, хоча ми й зазначили параметр часу існування `'a` для типу, що повертається, ця реалізація не буде скомпільованою, оскільки час існування значення, що повертається, взагалі не пов'язаний з часом існування параметрів. Ось яке повідомлення про помилку ми отримаємо:

```
$ cargo run
   Compiling chapter10 v0.1.0 (file:///projects/chapter10)
error[E0515]: cannot return reference to local variable `result`
  --> src/main.rs:11:5
   |
11 |     result.as_str()
   |     ^^^^^^^^^^^^^^ returns a reference to data owned by the current function

For more information about this error, try `rustc --explain E0515`.
error: could not compile `chapter10` due to previous error
```

Проблема в тому, що `result` виходить з області видимості й очищується в кінці функції `longest`. Ми також намагаємося повернути з функції посилання на `result`. Неможливо вказати параметри часу існування, які змінять всяке посилання, і Rust не дозволяє нам створити всяке посилання. У цьому випадку найкращим виправленням було б повернути тип, що володіє даними, замість посилання, щоб функція, яка викликала, була б відповідальною за очищення значення.

Зрештою, синтаксис часу існування стосується зв'язування часів існування різних параметрів і значень, що повертаються з функцій. Коли вони пов'язуються, Rust має достатньо інформації, щоб дозволити безпечні операції з пам'яттю і забороняти операції, які створювали б висячі вказівники або іншим чином порушували безпеку пам'яті.

Анотації Часів Існування у Визначеннях Структур

Досі всі визначені нами структури містили типи, що володіють даними. Ми можемо визначити структури, що міститимуть посилання, але в цьому разі ми маємо додати анотацію часу існування до кожного посилання у визначенні структури. Блок коду 10-24 демонструє структуру, що зветься `ImportantExcerpt`, що містить стрічковий слайс.

Файл: `src/main.rs`

```
struct ImportantExcerpt<'a> {  
    part: &'a str,  
}  
  
fn main() {  
    let novel = String::from("Call me Ishmael. Some years ago...");  
    let first_sentence = novel.split('.').next().expect("Could not find a '.');  
    let i = ImportantExcerpt {  
        part: first_sentence,  
    };  
}
```

Блок коду 10-24: структура, що містить посилання, яке потребує анотації часу існування

Ця структура має єдине поле `part`, що містить стрічковий слайс, що є посиланням. Як і для узагальнених типів даних, ми проголошуємо назву узагальненого параметру часу існування в кутових дужках після назви структури, щоб ми могли використати цей параметр часу існування у визначенні структури. Ця анотація означає, що екземпляр `ImportantExcerpt` не може існувати довше, ніж посилання, яке він містить у своєму полі `part`.

Функція `main` тут створює екземпляр структури `ImportantExcerpt`, який містить посилання на перше речення `String`, якою володіє змінна `novel`. Дані в `novel` існують до створення екземпляра `ImportantExcerpt`. Крім того, `novel` не виходить з області видимості, доки `ImportantExcerpt` не вийде з області видимості, тож посилання в екземплярі `ImportantExcerpt` є валідним.

Елізія Часу Існування

Ви дізналися, що кожне посилання має час існування і ви маєте зазначити параметри часу існування для функцій та структур, які використовують посилання. Однак у Розділі 4 у нас була функція в Блоці коду 4-9, показана знову у Блоці коду 10-25, яка скомпільовалася без анотацій часу існування.

Файл: src/lib.rs

```
fn first_word(s: &str) -> &str {
    let bytes = s.as_bytes();

    for (i, &item) in bytes.iter().enumerate() {
        if item == b' ' {
            return &s[0..i];
        }
    }

    &s[..]
}
```

Блок коду 10-25: функція, визначена в Блоці коду 4-9, яка компілюється без анотацій часу існування, хоча параметр і тип, що повертається є посиланнями

Причина, чому ця функція компілюється без анотацій часу існування, є історичною: у ранніх версіях (до 1.0) Rust цей код не скомпільовався б тому, що кожне посилання потребувало явного часу існування. На той момент сигнатура функції була б записана так:

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

Після написання великої кількості коду на Rust, команда Rust виявила, що програмісти Rust вводять ті самі анотації часу існування знову і знову в конкретних випадках. Ці ситуації були передбачуваними та відповідали декільком визначеним шаблонам. Розробники запрограмували ці шаблони у код компілятора, щоб borrow checker міг вивести часи існування в цих ситуаціях і не потребував явних анотацій.

Цей шматок історії Rust має значення, оскільки цілком можливо, що буде виявлено ще більше визначених шаблонів і будуть додані до компілятора. Можливо, у майбутньому буде потрібно ще менше анотацій часу існування.

Шаблони, запрограмовані в аналіз посилань Rust, називаються *правилами елізії часів існування*. Це не правила для програмістів; вони є набором певних випадків, які розглядає компілятор, і якщо ваш код відповідає цим випадкам, вам не потрібно явно вказувати часи існування.

Правила елізії не забезпечують повного виведення. Якщо Rust детерміновано застосовує правила, але залишається невизначеність щодо того, які часи існування мають посилання, компілятор не зможе здогадатися, якими мають бути часи існування решти посилань. Замість відгадування компілятор дасть вам помилку, котру ви можете її усунути, додавши анотації часів існування.

Часи існування на параметрах функції чи методу називаються *вхідні часи існування*, а часи існування на значеннях, що повертаються - *вихідні часи існування*.

Компілятор використовує три правила, щоб знайти час існування для посилань, коли немає явних анотацій. Перше правило застосовується до вхідних часів існування, а друге і третє правила до вихідного часу існування. Якщо компілятор досягає закінчення трьох правил все ще є посилання, для яких він не може визначити часи існування, компілятор зупиниться з помилкою. Ці правила застосовуються до проголошень `fn`, а також до блоків `impl`.

Перше правило полягає в тому, що компілятор встановлює параметр часу існування для кожного параметра, що є посиланням. Іншими словами, функція з одним параметром отримує один параметр часу існування: `fn foo<'a>(x: &'a i32)`; функція з двома параметрами отримує два окремі параметри часу існування: `fn foo<'a, 'b>(x: &'a i32, y: &'b i32)`; і так далі.

Друге правило полягає в тому, що якщо існує рівно один вхідний параметр часу існування, цей час існування призначається на всі вихідні параметри часів існування: `fn foo<'a>(x: &'a i32) -> &'a i32`.

Третє правило полягає в тому, що якщо є багато вхідних параметрів часу існування, але один з них є `&self` or `&mut self`, бо це метод, час існування `self` призначається усім вихідним параметрам часу існування. Це третє правило набагато полегшує читання і писання методів, бо потрібно менше символів.

Зробімо вигляд, що ми компілятор. Ми застосуємо ці правила, щоб визначити часи існування посилань у сигнатурі функції `first_word` у Блоці коду 10-25. Сигнатура починається без жодних часів існування, пов'язаних із посиланнями:

```
fn first_word(s: &str) -> &str {
```

Потім компілятор застосовує перше правило, яке визначає, що кожен параметр отримує свій власний час існування. Ми назвемо його `'a`, як зазвичай, тому тепер сигнатура є такою:

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

Друге правило застосовується, бо тут є рівно один вхідний час існування. Друге правило каже, що час існування єдиного вхідного параметра призначається вихідному часу існування, тож сигнатура тепер така:

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

Тепер всі посилання в сигнатурі цієї функції мають часи існування, і компілятор може продовжити свій аналіз, не змушуючи програміста анотувати часи існування в сигнатурі цієї функції.

Розгляньмо інший приклад, цього разу з функцією `longest`, яка не мала параметрів часу існування, коли ми почали працювати з нею у Блоці коду 10-20:

```
fn longest(x: &str, y: &str) -> &str {
```

Застосуємо перше правило: кожен параметр отримує свій час існування. Цього разу ми маємо два параметри замість одного, тому ми маємо два часи існування:

```
fn longest<'a, 'b>(x: &'a str, y: &'b str) -> &str {
```

Як бачите, друге правило не застосовується, тому що є більш ніж один вхідний час існування. Третє правило також не застосовується, тому що `longest` є функцією, а не методом, тож жоден з параметрів не є `self`. Виконавши всі три правила, ми все ще не з'ясували час існування типу, що повертається. Саме тому ми отримали помилку при спробі скомпілювати код у Блоці коду 10-20: компілятор пропрацював правила елізії часів існування, але все ж не зміг з'ясувати всі часи існування посилань у сигнатурі.

Оскільки третє правило застосовується тільки в сигнатурі методів, далі ми подивимося на часи існування в цьому контексті, щоб зрозуміти, чому третє правило означає, що нам не дуже часто доводиться анотувати часи існування у сигнатурах методів.

Анотації Часів Існування у Визначеннях Методів

Коли ми реалізуємо методи для структур з часами існування, то використовуємо той самий синтаксис, що і параметри узагальненого типу, показані у Блоці коду 10-11. Де нам проголошувати і використовувати параметри часу існування, залежить від того, чи стосуються вони полів структури, чи параметрів методу і значення, що повертається.

Назви часу існування полів структур завжди треба проголошувати після ключового слова `impl`, а потім використовувати після назви структури, тому що ці часи існування є частиною типу структури.

У сигнатурах методів всередині блоку `impl` посилання мають бути або прив'язані до часу існування посилань у полях структури, або ж мають бути незалежними. Крім того, правила елізії часів існування часто роблять анотації часів існування непотрібними у сигнатурах методів. Погляньмо на деякі приклади, використовуючи структуру з назвою `ImportantExcerpt`, яку ми визначили у Блоці коду 10-24.

Спершу, ми використовуємо метод з назвою `level`, єдиним параметром якого є посилання на `self`, а значення, що повертається є `i32`, що не є посиланням:

```
impl<'a> ImportantExcerpt<'a> {  
    fn level(&self) -> i32 {  
        3  
    }  
}
```

Декларація параметра часу існування після `impl` і його використання після назви типу є необхідними, але ми не маємо анотувати час існування посилання на `self` через перше правило елізії.

Ось приклад, де застосовується третє правило елізії часу існування:

```
impl<'a> ImportantExcerpt<'a> {  
    fn announce_and_return_part(&self, announcement: &str) -> &str {  
        println!("Attention please: {}", announcement);  
        self.part  
    }  
}
```

Тут є два вхідних часи існування, тож Rust застосовує перше правило елізії часів існування і дає обом `&self` та `announcement` власні часи існування. Тоді, оскільки один з параметрів є `&self`, тип, що повертається, отримує час існування `&self`, і всі часи існування було призначено.

Статичний Час Існування

Ми маємо обговорити один особливий час існування - `'static`, що позначає, що посилання може існувати весь час роботи програми. Всі стрічкові літерали мають час існування `'static`, що ми можемо анотувати таким чином:

```
let s: &'static str = "I have a static lifetime.";
```

Текст цієї стрічки зберігається безпосередньо в двійковому файлі програми, який є завжди доступним. Таким чином, час життя усіх стрічкових літералів є `'static`.

Ви можете побачити пропозиції використовувати час існування `'static` у повідомленнях про помилки. Але перш ніж вказати часом існування посилання `'static`, подумайте, чи дійсно ваше посилання існує впродовж усієї роботи вашої програми чи ні, і чи хочете, щоб воно таким було. У більшості випадків повідомлення про помилку, що пропонує час існування `'static` є результатом спроби створити висяче посилання чи невідповідність наявних часів існування. У таких випадках рішенням є виправити ці проблеми, а не визначити час існування `'static`.

Параметри Узагальненого Типу, Обмеження Трейтів і Часи Існування Разом

Подивімося коротко на синтаксис визначення параметрів узагальненого типу, обмежень трейтів і часів існування разом в одній функції!

```
use std::fmt::Display;

fn longest_with_an_announcement<'a, T>(
    x: &'a str,
    y: &'a str,
    ann: T,
) -> &'a str
where
    T: Display,
{
    println!("Announcement! {}", ann);
    if x.len() > y.len() {
        x
    } else {
        y
    }
}
```

Це функція `longest` зі Блоку коду 10-21, яка повертає довший із двох стрічкових слайсів. Але тепер вона має додатковий параметр з назвою `ann` узагальненого типу `T`, який може бути заповнений будь-яким типом, який реалізує трейт `Display`, як зазначено в клаузі `where`. Цей додатковий параметр буде виведено за допомогою `{}`, то й робить необхідним обмеження трейту `Display`. Оскільки часи існування є узагальненням, проголошення параметра часу існування `'a` і параметру узагальненого типу `T` розміщуються в одному списку в кутових дужках після назви функції.

Підсумок

Ми багато відкрили в цьому розділі! Тепер, коли ви знаєте про параметри узагальненого типу, трейти і трейтові обмеження, і узагальнені параметри часу існування, ви готові написати код без повторень, що працюватиме у багатьох різних ситуаціях. Параметри узагальнених типів дозволяють вам застосовувати один код для різних типів. Трейти та трейтові обмеження гарантують, що навіть якщо типи є узагальненими, вони матимуть поведінку, потрібну коду. Ви вивчили, як використовувати анотації часу існування, щоб убезпечити такий гнучкий код від висячих посилань. І відбувається весь цей аналіз під час компіляції, що не впливає на продуктивність під час виконання!

Вірите ви чи ні, є ще багато чого, що можна дізнатися про обговорені в цьому розділі теми: Розділ 17 обговорює трейтові об'єкти, які є іншим способом використання трейтів. Також є складніші сценарії, що використовують анотації часів існування, що знадобляться вам лише у дуже просунутих сценаріях; для них ви маєте прочитати [Rust Reference](#). Та зараз ви навчитеся писати тести на Rust, щоб переконатися, що ваш код працює так, як треба. [ch04-02-references-and-borrowing.html#references-and-borrowing](#) [ch04-03-slices.html#string-slices-as-parameters](#)

Написання Автоматизованих Тестів

У 1972 році у своєму есе "Скромний програміст" Едсгер Дейкстра сказав "Тестування програми може бути дуже ефективним способом показати наявність помилок, але його зовсім недостатньо, щоб показати їх відсутність." Це не означає, що ми не повинні тестувати стільки, скільки ми можемо!

Коректність наших програм полягає у ступені відповідності того, що вони роблять тому, що ми мали на увазі, коли їх розробляли. Rust розроблений з високим ступенем турботи про коректність програм, але коректність є складною та її не легко довести. Система типів Rust несе величезну частину цього тягаря, але вона не здатна впоратися з усім. Таким чином, Rust включає підтримку написання автоматизованих тестів програмного забезпечення.

Нехай ми пишемо функцію `add_two` яка додає 2 до будь-якого числа, що передано в неї. В сигнатурі цієї функції ми вказуємо ціле число як вхідний параметр, та ціле число, як результат, що повертається. Коли ми реалізуємо та компілюємо цю функцію, Rust робить усі перевірки типів та запозичень, щоб гарантувати, що ми не передаємо `String` або недійсне посилання до цієї функції. Але Rust *не може* перевірити, що ця функція робить безпосередньо те, що ми задумали, що повертає параметр плюс 2, а не, скажімо, параметр плюс 10 або параметр мінус 50! Ось тут і з'являються тести.

Ми можемо написати тести, які підтверджують, що, наприклад, коли ми передаємо `3` до функції `add_two`, то вона повертає значення `5`. Ми можемо запускати ці тести кожного разу, коли вносимо зміни до нашого коду, щоб бути впевненими в тому, що коректна поведінка програми при цьому не змінилася.

Тестування - це складна навичка: хоча ми не можемо в одному розділі охопити усі нюанси того, як створювати гарні тести, ми оглянемо засоби тестування у Rust. Ми поговоримо про анотації та макроси, доступні вам для написання тестів, про поведінку за замовчуванням та параметри для запуску ваших тестів, а також як організувати тестування за допомогою unit- та інтеграційних тестів.

Як Писати Тести

Тести - це функції Rust, які перевіряють, чи тестований код працює у очікуваний спосіб. Тіла тестових функцій зазвичай виконують наступні три дії:

1. Встановити будь-яке потрібне значення або стан.
2. Запустити на виконання код, який ви хочете протестувати.
3. Переконатися, що отримані результати відповідають вашим очікуванням.

Розгляньмо функціонал, наданий Rust спеціально для написання тестів та виконання зазначених дій, що включає атрибут `test`, декілька макросів, а також атрибут `should_panic`.

Анатомія Тестувальної Функції

У найпростішому випадку тест у Rust - це функція, анотована за допомогою атрибута `test`. Атрибути - це метадані фрагментів коду на Rust; прикладом може бути атрибут `derive`, який ми використовували зі структурами у Розділі 5. Для перетворення звичайної функції на тестувальну функцію додайте `#[test]` у рядок перед `fn`. Коли ви запускаєте ваші тести командою `cargo test`, Rust збирає двійковий файл, що запускає анотовані функції та звітує, чи тестові функції пройшли, чи провалилися.

Кожного разу, коли ми створюємо новий бібліотечний проєкт за допомогою Cargo, він автоматично генерує для нас тестовий модуль з тестовими функціями. Цей модуль надає вам шаблон для написання тестів, а отже вам не потрібно кожного разу при створенні нового проєкту уточнювати їхню структуру та синтаксис. Ви можете додати стільки тестових модулів та тестових функцій, скільки забажаєте!

Перш ніж фактично протестувати будь-який код, ми розглянемо деякі аспекти роботи тестів, експериментуючи з шаблоном тесту. Потім ми напишемо декілька реальних тестів, що запускають наш код та підтверджують, що його поведінка є правильною.

Створімо новий бібліотечний проєкт під назвою `adder`, в якому додаються два числа:

```
$ cargo new adder --lib
   Created library `adder` project
$ cd adder
```

Вміст файлу `src/lib.rs` у вашій бібліотеці `adder` має виглядати, як показано в Блоці коду 11-1.

Файл: src/lib.rs

```
#[cfg(test)]
mod tests {
    #[test]
    fn it_works() {
        let result = 2 + 2;
        assert_eq!(result, 4);
    }
}
```

Блок коду 11-1: тестовий модуль і функція, створена автоматично за допомогою `cargo new`

Наразі проігноруймо два верхні рядки та зосередимося на функції. Зверніть увагу на анотацію `#[test]`: цей атрибут вказує на те, що функція є тестувальною, тож функціонал для запуску тестів ставитиметься до неї, як до тесту. Ми також можемо мати нетестувальні функції в модулі `tests`, щоб допомогти налаштувати типові сценарії або виконати загальні операції, тому ми завжди повинні позначати анотаціями, які саме функції є тестувальними.

Тіло функції зі приклада використовує макрос `assert_eq!` для ствердження того, що `result`, який містить результат операції додавання 2 та 2, дорівнює 4. Це ствердження служить типовим зразком формату для тесту. Запустимо його та впевнимось, що тест проходить.

Команда `cargo test` запускає усі тести з нашого проєкту, як показано у Блоці коду 11-2.

```
$ cargo test
Compiling adder v0.1.0 (file:///projects/adder)
Finished test [unoptimized + debuginfo] target(s) in 0.57s
Running unittests src/lib.rs (target/debug/deps/adder-92948b65e88960b4)

running 1 test
test tests::it_works ... ok

test result: ok. 1 passed; 0 failed; 0 ignored; 0 measured; 0 filtered out;
finished in 0.00s

Doc-tests adder

running 0 tests

test result: ok. 0 passed; 0 failed; 0 ignored; 0 measured; 0 filtered out;
finished in 0.00s
```

Блок коду 11-2: виведення після запуску автоматично згенерованого тесту

Cargo скомпілював та запустив тест. Ми бачимо рядок `running 1 test`. Наступний рядок показує назву згенерованої тестувальної функції `it_works`, а також що результат запуску тесту є `ok`. Загальний результат `test result: ok.` означає, що усі тести пройшли, а частина `1 passed; 0 failed` показує загальну кількість тестів що були пройдені та провалилися.

Можна позначити деякі тести як ігноровані, тоді вони не будуть запускатися; ми розглянемо це далі у цьому розділі у підрозділі "Ігнорування окремих тестів без спеціального уточнення". Оскільки ми не позначали жодного тесту для ігнорування, то отримуємо на виході `0 ignored`. Ми також можемо додати до команди `cargo test` аргумент, щоб запускалися лише ті тести, що відповідають певній стрічці; Це називається *фільтрацією* та ми поговоримо про це у підрозділі "Запуск підмножини тестів за назвою". Ми також не маємо відфільтрованих тестів, тому вивід показує `0 filtered out`.

`0 measured` показує статистику бенчмарків, що тестують швидкодію. Бенчмарки на момент написання цієї статті доступні лише у нічних збірках Rust. Дивись [документацію про бенчмарки](#) для більш детальної інформації.

Наступна частина виводу `Doc-tests adder` призначена для результатів документаційних тестів. У нас поки що немає документаційних тестів, але Rust може скомпілювати будь-які приклади коду з документації по нашому API. Ця функція допомагає синхронізувати вашу документацію та код! Ми розглянемо, як писати документаційні тести в підрозділі "Документаційні коментарі як тести" Розділу 14. Зараз ми проігноруємо частину виводу, присвячену `Doc-tests`.

Налаштуймо тест для відповідності нашим потребам. Спочатку змінимо назву тестової функції `it_works` на іншу, наприклад `exploration`, ось так:

Файл: `src/lib.rs`

```
[cfg(test)]
mod tests {
    #[test]
    fn exploration() {
        assert_eq!(2 + 2, 4);
    }
}
```

Далі знову запустимо `cargo test`. Вивід тепер покаже `exploration` замість `it_works`:

```
$ cargo test
Compiling adder v0.1.0 (file:///projects/adder)
Finished test [unoptimized + debuginfo] target(s) in 0.59s
Running unittests src/lib.rs (target/debug/deps/adder-92948b65e88960b4)

running 1 test
test tests::exploration ... ok

test result: ok. 1 passed; 0 failed; 0 ignored; 0 measured; 0 filtered out;
finished in 0.00s

Doc-tests adder

running 0 tests

test result: ok. 0 passed; 0 failed; 0 ignored; 0 measured; 0 filtered out;
finished in 0.00s
```

Тепер ми додамо інший тест, але цього разу він завершиться зі збоєм! Тести завершуються зі збоєм, коли щось у тестувальній функції викликає паніку. Кожний тест запускається в окремому потоці, та коли головний потік бачить, що тестовий потік упав, то тест позначається як такий невдалий. У Розділі 9 ми розглядали найпростіший спосіб викликати паніку за допомогою виклику макросу `panic!`. Створіть новий тест та назвіть тестувальну функцію `another`, щоб ваш файл `src/lib.rs` виглядав як у Блоці коду 11-3.

Файл: `src/lib.rs`

```
[cfg(test)]
mod tests {
    #[test]
    fn exploration() {
        assert_eq!(2 + 2, 4);
    }

    #[test]
    fn another() {
        panic!("Make this test fail");
    }
}
```

Блок коду 11-3: додавання другого тесту, що провалюється, бо ми викликаємо макрос `panic!`

Запустіть тест знову, використовуючи `cargo test`. Вивід виглядатиме схоже на Блок коду 11-4, який показує, що тест `exploration` пройшов, а `another` провалився.

```

$ cargo test
Compiling adder v0.1.0 (file:///projects/adder)
Finished test [unoptimized + debuginfo] target(s) in 0.72s
Running unittests src/lib.rs (target/debug/deps/adder-92948b65e88960b4)

running 2 tests
test tests::another ... FAILED
test tests::exploration ... ok

failures:

---- tests::another stdout ----
thread 'main' panicked at 'Make this test fail', src/lib.rs:10:9
note: run with `RUST_BACKTRACE=1` environment variable to display a backtrace

failures:
    tests::another

test result: FAILED. 1 passed; 1 failed; 0 ignored; 0 measured; 0 filtered out;
finished in 0.00s

error: test failed, to rerun pass `--lib`

```

Блок коду 11-4: результати тестів, коли один тест проходить, а другий провалюється

Замість `ok`, рядок `test tests::another` показує `FAILED`. Дві нові секції з'явилися між результатами окремих тестів та загальними результатами: перша показує детальну причину того, чому тест провалився. У цьому випадку ми отримали те, що тест `another` провалився тому, що `panicked at 'Make this test fail'` у рядку 10 у файлі `src/lib.rs`. У наступній секції наведені назви тестів, що провалилися, і це зручно коли у нас багато таких тестів та багато деталей про провали. Ми можемо використати назву тесту для його подальшого зневадження; ми поговоримо більше про запуск тестів у підрозділі [Керування запуском тестів](#).

У кінці показується підсумковий результат тестування: в цілому результат нашого тесту `FAILED`. У нас один тест пройшов, та один провалився.

Тепер, коли ви побачили, як виглядають результати тесту в різних сценаріях, розгляньмо деякі макроси, крім `panic!`, які корисні в тестах.

Перевірка Результатів із Макросом `assert!`

Макрос `assert!`, що надається стандартною бібліотекою, широко використовується для того, щоб впевнитися, що деяка умова у тесті приймає значення `true`. Ми даємо макросу `assert!` аргумент, який обчислюється як вираз булевого типу. Якщо значення

обчислюється як `true`, нічого поганого не трапляється та тест вважається пройденим. Якщо ж значення буде `false` макрос `assert!` викликає паніку `panic!`, що спричиняє провал тесту. Використання макросу `assert!` допомагає нам перевірити, чи працює наш код в очікуваний спосіб.

У Розділі 5, Блок коду 5-15, ми використовували структуру `Rectangle` та метод `can_hold`, які повторюються у Блоці коду 11-5. Розмістімо цей код у файлі `src/lib.rs`, а далі напишемо декілька тестів, використовуючи макрос `assert!`.

Файл: `src/lib.rs`

```
#[derive(Debug)]
struct Rectangle {
    width: u32,
    height: u32,
}

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

Блок коду 11-5: використання структури `Rectangle` і її методу `can_hold` з Розділу 5

Метод `can_hold` повертає булеве значення, що означає, що це ідеальний варіант використання для макросу `assert!`. У Блоці коду 11-6 ми пишемо тест, який перевіряє метод `can_hold`, створивши екземпляр `Rectangle`, що має ширину 8 і висоту 7, і стверджує, що він може вмістити інший екземпляр `Rectangle`, що має ширину 5 і висоту 1.

Файл: `src/lib.rs`

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

    #[test]
    fn larger_can_hold_smaller() {
        let larger = Rectangle {
            width: 8,
            height: 7,
        };
        let smaller = Rectangle {
            width: 5,
            height: 1,
        };

        assert!(larger.can_hold(&smaller));
    }
}
```

Блок коду 11-6: тест для `can_hold`, що перевіряє, чи більший прямокутник справді може вмістити менший

Зверніть увагу, що ми додали новий рядок всередині модуля `tests`: `use super::*`. Модуль `tests` є звичайним модулем, що слідує звичайним правилам видимості, про які ми розповідали у підрозділі [“Шляхи для посилання на елемент в дереві модулів”](#) Розділу 7. Оскільки модуль `tests` є внутрішнім модулем, нам потрібно ввести код для тестування з зовнішнього модуля до області видимості внутрішнього модуля. Ми використовуємо глобальний режим для того, щоб все, що визначене у зовнішньому модулі, було доступним к модулі `tests`.

Ми назвали наш тест `larger_can_hold_smaller` і створили потрібні нам два екземпляри `Rectangle`. Тоді ми викликали макрос `assert!` і передали йому результат виклику `larger.can_hold(&smaller)`. Цей вираз повинен повернути `true`, тому наш тест повинен пройти. З'ясуймо, чи це так!

```
$ cargo test
Compiling rectangle v0.1.0 (file:///projects/rectangle)
Finished test [unoptimized + debuginfo] target(s) in 0.66s
Running unittests src/lib.rs (target/debug/deps/rectangle-6584c4561e48942e)

running 1 test
test tests::larger_can_hold_smaller ... ok

test result: ok. 1 passed; 0 failed; 0 ignored; 0 measured; 0 filtered out;
finished in 0.00s

Doc-tests rectangle

running 0 tests

test result: ok. 0 passed; 0 failed; 0 ignored; 0 measured; 0 filtered out;
finished in 0.00s
```

Цей тест проходить! Додамо ще один тест, цього разу стверджуючи, що менший прямокутник не може вміститися в більшому:

Файл: src/lib.rs

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

    #[test]
    fn larger_can_hold_smaller() {
        // --snip--
    }

    #[test]
    fn smaller_cannot_hold_larger() {
        let larger = Rectangle {
            width: 8,
            height: 7,
        };
        let smaller = Rectangle {
            width: 5,
            height: 1,
        };

        assert!(!smaller.can_hold(&larger));
    }
}
```

Оскільки правильний результат функції `can_hold` у цьому випадку `false`, нам необхідно обернути результат перед тим, як передати його до макросу `assert!`. В результаті наш

тест пройде, якщо `can_hold` повертає `false`:

```
$ cargo test
Compiling rectangle v0.1.0 (file:///projects/rectangle)
Finished test [unoptimized + debuginfo] target(s) in 0.66s
Running unittests src/lib.rs (target/debug/deps/rectangle-6584c4561e48942e)

running 2 tests
test tests::larger_can_hold_smaller ... ok
test tests::smaller_cannot_hold_larger ... ok

test result: ok. 2 passed; 0 failed; 0 ignored; 0 measured; 0 filtered out;
finished in 0.00s

Doc-tests rectangle

running 0 tests

test result: ok. 0 passed; 0 failed; 0 ignored; 0 measured; 0 filtered out;
finished in 0.00s
```

Проходять вже два тести! Тепер подивімося, що відбувається з результатами тесту, якщо в код додати ваду. Ми змінимо реалізацію методу `can_hold`, замінивши знак більше на менше при порівняння ширин:

```
// --snip--
impl Rectangle {
    fn can_hold(&self, other: &Rectangle) -> bool {
        self.width < other.width && self.height > other.height
    }
}
```

Запуск тестів тепер виводить таке:

```

$ cargo test
Compiling rectangle v0.1.0 (file:///projects/rectangle)
Finished test [unoptimized + debuginfo] target(s) in 0.66s
Running unittests src/lib.rs (target/debug/deps/rectangle-6584c4561e48942e)

running 2 tests
test tests::larger_can_hold_smaller ... FAILED
test tests::smaller_cannot_hold_larger ... ok

failures:

---- tests::larger_can_hold_smaller stdout ----
thread 'main' panicked at 'assertion failed: larger.can_hold(&smaller)',
src/lib.rs:28:9
note: run with `RUST_BACKTRACE=1` environment variable to display a backtrace

failures:
    tests::larger_can_hold_smaller

test result: FAILED. 1 passed; 1 failed; 0 ignored; 0 measured; 0 filtered out;
finished in 0.00s

error: test failed, to rerun pass `--lib`

```

Наші тести викрили ваду! Оскільки `larger.width` дорівнює 8, а `smaller.width` дорівнює 5, порівняння ширин у `can_hold` тепер повертає `false`: 8 не є меншим за 5.

Перевірка Рівності з Макросами `assert_eq!` та `assert_ne!`

Поширеним способом перевірки функціональності є перевірка на рівність між результатом коду, що тестується, і значенням, яке ви очікуєте від коду. Ви можете зробити це за допомогою макросу `assert!`, передавши йому вираз за допомогою оператора `==`. Однак це такий поширений тест, що стандартна бібліотека надає пару макросів — `assert_eq!` та `assert_ne!` — для зручнішого проведення цього тесту. Ці макроси порівнюють два аргументи на рівність або нерівність відповідно. Також вони виводять два значення, якщо ствердження провалюється, що допомагає зрозуміти, чому тест провалився; і навпаки, макрос `assert!` лише вказує на те, що отримав значення `false` для виразу `==`, без виведення значень, що призвели до цього `false`.

У Блоці коду 11-7 ми пишемо функцію з назвою `add_two`, яка додає до свого параметра `2`, а потім тестуємо цю функцію за допомогою макросу `assert_eq!`.

Файл: `src/lib.rs`

```
pub fn add_two(a: i32) -> i32 {
    a + 2
}

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

    #[test]
    fn it_adds_two() {
        assert_eq!(4, add_two(2));
    }
}
```

Блок коду 11-7: Тестування функції `add_two` за допомогою макросу `assert_eq!`

Переконаймося, що вона проходить!

```
$ cargo test
Compiling adder v0.1.0 (file:///projects/adder)
Finished test [unoptimized + debuginfo] target(s) in 0.58s
Running unittests src/lib.rs (target/debug/deps/adder-92948b65e88960b4)

running 1 test
test tests::it_adds_two ... ok

test result: ok. 1 passed; 0 failed; 0 ignored; 0 measured; 0 filtered out;
finished in 0.00s

Doc-tests adder

running 0 tests

test result: ok. 0 passed; 0 failed; 0 ignored; 0 measured; 0 filtered out;
finished in 0.00s
```

Ми передали `4` як аргумент `assert_eq!`, що дорівнює результату виклику `add_two(2)`. Рядок для цього тесту `test tests::it_adds_two ... ok`, і текст `ok` позначає, що наш тест пройшов!

Додамо в наш код ваду, щоб побачити, як виглядає `assert_eq!`, коли тест провалюється. Змініть реалізацію функції `add_two`, щоб натомість додавати `3`:

```
pub fn add_two(a: i32) -> i32 {
    a + 3
}
```

Запустимо тести знову:

```

$ cargo test
  Compiling adder v0.1.0 (file:///projects/adder)
  Finished test [unoptimized + debuginfo] target(s) in 0.61s
  Running unittests src/lib.rs (target/debug/deps/adder-92948b65e88960b4)

running 1 test
test tests::it_adds_two ... FAILED

failures:

---- tests::it_adds_two stdout ----
thread 'main' panicked at 'assertion failed: `(left == right)`
  left: `4`,
 right: `5`', src/lib.rs:11:9
note: run with `RUST_BACKTRACE=1` environment variable to display a backtrace

failures:
  tests::it_adds_two

test result: FAILED. 0 passed; 1 failed; 0 ignored; 0 measured; 0 filtered out;
finished in 0.00s

error: test failed, to rerun pass `--lib`

```

Наш тест виявив ваду! Тест `it_adds_two` провалився, і повідомлення каже нам про те, що провалене ствердження було `assertion failed: `(left == right)`` і значення `left` і `right`. Це повідомлення допомагає нам почати зневадження: аргумент `left` був `4`, але аргумент `right`, де стоїть `add_two(2)`, був `5`. Ви можете собі уявити, що це буде особливо корисно, коли у нас проводиться багато тестів.

Зверніть увагу, що в деяких мовах і тестувальних фреймворках параметри функції ствердження рівності називаються `expected` (очікувалося) і `actual` (фактично), і порядок, в якому ми вказуємо аргументи, важливий. Однак у Rust вони називаються `left` і `right`, і порядок, в якому ми вказуємо значення, яке ми очікуємо, і значення, обчислене кодом, не має значення. Ми могли б записати ствердження у цьому тесті як `assert_eq!(add_two(2), 4)`, що призведе до того ж повідомлення про помилку, яке показує `assertion failed: `(left == right)``.

Макрос `assert_ne!` проходить тест, якщо два значення, які ми даємо, не дорівнюють одне одному, і провалюється, якщо вони рівні. Цей макрос найбільш корисний для випадків, коли ми не впевнені яким значення *буде*, але ми знаємо, яким значення безумовно *не повинне* бути. Наприклад, якщо ми тестуємо функцію, яка гарантовано певним чином змінює вхідні параметри, але те, як змінюється вони змінюються, залежить від дня тижня, коли ми проводимо свої тести, найкраще, що може стверджувати — що вихід функції не дорівнює вводу.

Під капотом макроси `assert_eq!` і `assert_ne!` використовують оператори `==` and `!=`, відповідно. Коли ствердження провалюється, ці макроси друкують свої аргументи з використанням форматування налагодження, тобто, що значення, які порівнюються, повинні реалізовувати трейти `PartialEq` і `Debug`. Всі примітивні типи та більшість типів зі стандартної бібліотеки реалізують ці трейти. Для структур та еnumів, які ви визначаєте самостійно, вам потрібно буде реалізувати `PartialEq`, щоб стверджувати рівність таких типів. Також потрібно буде реалізувати `Debug` для виведення значень, коли ствердження провалюється. Оскільки обидва ці трейти вивідні, як було зазначено в Блоці коду 5-12 у Розділі 5, зазвичай просто треба додати анотацію `#[derive(PartialEq, Debug)]` до визначення вашої структури чи еnumу. Дивіться Додаток C, "Вивідні трейти", для детальнішої інформації про ці та інші вивідні трейти.

Додавання Власних Повідомлень про Збої

Ви також можете додати власне повідомлення, яке буде виведене в консолі разом із повідомленням про помилку, як додаткові аргументи до макросів `assert!`, `assert_eq!`, та `assert_ne!`. Будь-які аргументи, вказані після необхідних аргументів, передаються до макросу `format!` (обговорюється у Розділі 8 у підрозділі "Конкатенація оператором `+` або макросом `format!`"), тож ви можете передати стрічку форматування, що містить заповнювачі `{}` та значення для підставлення в ці заповнювачі. Користувальницькі повідомлення корисні для документування, що означає ствердження; коли тест провалюється, ви будете мати краще розуміння того, що за проблема з кодом.

Наприклад, припустимо, у нас є функція, яка вітає людей на ім'я, і ми хочемо перевірити, що ім'я, яке ми передаємо, з'являється у вихідній стрічці:

Файл: `src/lib.rs`

```
pub fn greeting(name: &str) -> String {
    format!("Hello {}", name)
}

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

    #[test]
    fn greeting_contains_name() {
        let result = greeting("Carol");
        assert!(result.contains("Carol"));
    }
}
```


Вимоги до цієї програми ще не були узгоджені, і ми цілком певні, що текст `Hello,` на початку вітання зміниться. Ми вирішили не оновлювати тест при змінах вимог, отже замість перевірки на точну рівність значенню, яке повернула функція `greeting`, ми просто ствердимо, що результат містить текст вихідного параметра.

Тепер введемо ваду у цей код, змінивши `greeting` з виключивши `name`, щоб побачити, як виглядає тест провалу за замовчуванням:

```
pub fn greeting(name: &str) -> String {  
    String::from("Hello!")  
}
```

Запуск цього тесту виводить таке:

```
$ cargo test  
Compiling greeter v0.1.0 (file:///projects/greeter)  
Finished test [unoptimized + debuginfo] target(s) in 0.91s  
Running unittests src/lib.rs (target/debug/deps/greeter-170b942eb5bf5e3a)  
  
running 1 test  
test tests::greeting_contains_name ... FAILED  
  
failures:  
  
---- tests::greeting_contains_name stdout ----  
thread 'main' panicked at 'assertion failed: result.contains(\"Carol\")',  
src/lib.rs:12:9  
note: run with `RUST_BACKTRACE=1` environment variable to display a backtrace  
  
failures:  
    tests::greeting_contains_name  
  
test result: FAILED. 0 passed; 1 failed; 0 ignored; 0 measured; 0 filtered out;  
finished in 0.00s  
  
error: test failed, to rerun pass `--lib`
```

Цей результат вказує на те, що ствердження провалилося і в якому рядку. Більш корисне повідомлення про помилку може вивести значення з функції `greeting`. Додаймо власне повідомлення про помилку, складене зі стрічки форматування з заповнювачем, заповненим фактичним значенням, яке ми отримали від функції `greeting`:

```
#[test]
fn greeting_contains_name() {
    let result = greeting("Carol");
    assert!(
        result.contains("Carol"),
        "Greeting did not contain name, value was `{}`",
        result
    );
}
```

Тепер, коли ми запустимо тест, ми отримаємо більш інформативне повідомлення про помилку:

```
$ cargo test
Compiling greeter v0.1.0 (file:///projects/greeter)
Finished test [unoptimized + debuginfo] target(s) in 0.93s
Running unittests src/lib.rs (target/debug/deps/greeter-170b942eb5bf5e3a)

running 1 test
test tests::greeting_contains_name ... FAILED

failures:

---- tests::greeting_contains_name stdout ----
thread 'main' panicked at 'Greeting did not contain name, value was `Hello!`,
src/lib.rs:12:9
note: run with `RUST_BACKTRACE=1` environment variable to display a backtrace

failures:
    tests::greeting_contains_name

test result: FAILED. 0 passed; 1 failed; 0 ignored; 0 measured; 0 filtered out;
finished in 0.00s

error: test failed, to rerun pass `--lib`
```

Ми можемо бачити значення, яке ми фактично отримали на виході тесту, що допоможе нам налагодити, що сталося замість того, що ми очікували.

Перевірка Наявності Паніки з `should_panic`

На додаток до перевірки значень, повернених з функцій, важливо перевірити, чи наш код обробляє помилкові стани, які ми очікуємо. Наприклад, розгляньте тип `Guess`, який ми створили у Блоці коду 9-13 у Розділі 9. Інший код, який використовує `Guess`, залежить від гарантії, що екземпляри `Guess` будуть містити лише значення у діапазоні від 1 до 100. Ми

можемо написати тест, який гарантує, що намагання створити екземпляр `Guess` зі значенням поза інтервалом панікує.

Ми робимо це, додаючи атрибут `should_panic` до нашої тестової функції. Тест проходить, якщо код усередині функції панікує; тест провалюється, якщо код усередині функції не запанікував.

Блок коду 11-8 показує тест, який перевіряє, що умови помилки `Guess::new` стаються тоді, коли ми очікуємо на них.

Файл: `src/lib.rs`

```
pub struct Guess {
    value: i32,
}

impl Guess {
    pub fn new(value: i32) -> Guess {
        if value < 1 || value > 100 {
            panic!("Guess value must be between 1 and 100, got {}.", value);
        }

        Guess { value }
    }
}

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

    #[test]
    #[should_panic]
    fn greater_than_100() {
        Guess::new(200);
    }
}
```

Блок коду 11-8: Тестування умови, що призводить до `panic!`

Ми розміщуємо атрибут `#[should_panic]` після `#[test]` перед тестовою функцією, до якої він застосовується. Подивімося на результат, коли цей тест проходить:

```
$ cargo test
Compiling guessing_game v0.1.0 (file:///projects/guessing_game)
Finished test [unoptimized + debuginfo] target(s) in 0.58s
Running unittests src/lib.rs (target/debug/deps/guessing_game-57d70c3acb738f4d)

running 1 test
test tests::greater_than_100 - should panic ... ok

test result: ok. 1 passed; 0 failed; 0 ignored; 0 measured; 0 filtered out;
finished in 0.00s

Doc-tests guessing_game

running 0 tests

test result: ok. 0 passed; 0 failed; 0 ignored; 0 measured; 0 filtered out;
finished in 0.00s
```

Здається, усе гаразд! Тепер додамо вад у наш код, видаливши умову, що функція **new** запанікує, якщо значення більше за 100:

```
// --snip--
impl Guess {
    pub fn new(value: i32) -> Guess {
        if value < 1 {
            panic!("Guess value must be between 1 and 100, got {}.", value);
        }

        Guess { value }
    }
}
```

Коли ми запустимо тест з Блоку коду 11-8, він провалюється:

```
$ cargo test
Compiling guessing_game v0.1.0 (file:///projects/guessing_game)
Finished test [unoptimized + debuginfo] target(s) in 0.62s
Running unittests src/lib.rs (target/debug/deps/guessing_game-57d70c3acb738f4d)

running 1 test
test tests::greater_than_100 - should panic ... FAILED

failures:

---- tests::greater_than_100 stdout ----
note: test did not panic as expected

failures:
    tests::greater_than_100

test result: FAILED. 0 passed; 1 failed; 0 ignored; 0 measured; 0 filtered out;
finished in 0.00s

error: test failed, to rerun pass `--lib`
```

В цьому випадку ми отримуємо не дуже помічне повідомлення, але коли подивитися на тестову функцію, то бачимо, що вона анотована `#[should_panic]`. Отриманий провал означає, що код у тестовій функції не призвів до паніки.

Тести, що використовують `should_panic`, можуть бути неточними. Тест із `should_panic` пройде, навіть якщо тест запанікує з іншої причини, а не очікуваної. Щоб зробити тести із `should_panic` більш точними, ми можемо додати необов'язковий параметр `expected` до атрибута `should_panic`. Тестова оболонка забезпечить, щоб повідомлення про провал містило наданий текст. Наприклад, розгляньте модифікований код `Guess` у Блоці коду 11-9, де функція `new` панікує з різними повідомленнями залежно від того, значення занадто мале або занадто велике.

Файл: `src/lib.rs`

```
// --snip--

impl Guess {
    pub fn new(value: i32) -> Guess {
        if value < 1 {
            panic!(
                "Guess value must be greater than or equal to 1, got {}.",
                value
            );
        } else if value > 100 {
            panic!(
                "Guess value must be less than or equal to 100, got {}.",
                value
            );
        }

        Guess { value }
    }
}

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

    #[test]
    #[should_panic(expected = "less than or equal to 100")]
    fn greater_than_100() {
        Guess::new(200);
    }
}
```

Блок коду 11-9: тестування `panic!` з повідомленням паніки, що містить зазначену підстрічку

Цей тест пройде, оскільки значення, яке ми додали в параметр `expected` атрибуту `should_panic` є підстрічкою повідомлення, з яким панікує функція `Guess::new`. Ми могли б вказати повне повідомлення паніки, яке очікуємо, яке у цьому випадку буде `Guess value must be less than or equal to 100, got 200.` Те, що саме ви зазначите, залежить від того, яка частина повідомлення паніки є унікальною чи динамічним і наскільки точним тест ви хочете зробити. У цьому випадку підстрічки повідомлення паніки достатньо, щоб переконатися, що код у тестовій функції обробляє випадок `else if value > 100`.

Щоб побачити, що трапиться, якщо тест `should_panic` з повідомленням `expected` провалюється, знову додамо ваду у наш код, обмінявши тіла блоків `if value < 1` та `else if value > 100`:

```

    if value < 1 {
        panic!(
            "Guess value must be less than or equal to 100, got {}.",
            value
        );
    } else if value > 100 {
        panic!(
            "Guess value must be greater than or equal to 1, got {}.",
            value
        );
    }
}

```

Цього разу, коли ми запускаємо тест `should_panic`, він провалиться:

```

$ cargo test
  Compiling guessing_game v0.1.0 (file:///projects/guessing_game)
  Finished test [unoptimized + debuginfo] target(s) in 0.66s
  Running unittests src/lib.rs (target/debug/deps/guessing_game-57d70c3acb738f4d)

running 1 test
test tests::greater_than_100 - should panic ... FAILED

failures:

---- tests::greater_than_100 stdout ----
thread 'main' panicked at 'Guess value must be greater than or equal to 1, got 200.', src/lib.rs:13:13
note: run with `RUST_BACKTRACE=1` environment variable to display a backtrace
note: panic did not contain expected string
      panic message: `Guess value must be greater than or equal to 1, got 200.`
    expected substring: `less than or equal to 100`

failures:
    tests::greater_than_100

test result: FAILED. 0 passed; 1 failed; 0 ignored; 0 measured; 0 filtered out;
finished in 0.00s

error: test failed, to rerun pass `--lib`

```

Повідомлення про провал вказує на те, що цей тест дійсно панікував, як ми очікували, але повідомлення про паніку не містить очікувану стрічку `'Guess value must be less than or equal to 100'`. Повідомлення про паніку, яке ми взяли, в цьому випадку було `Guess value must be greater than or equal to 1, got 200.` Тепер ми можемо почати з'ясуоввати, де знаходиться наша помилка!

Використання `Result<T, E>` у Тестах

Всі наші тести поки що панікують, коли провалюються. Ми також можемо написати тести, які використовують `Result<T, E>`! Ось тест зі Блоку коду 11-1, переписаний для використання `Result<T, E>`, який повертає `Err` замість паніки:

```
[cfg(test)]
mod tests {
    #[test]
    fn it_works() -> Result<(), String> {
        if 2 + 2 == 4 {
            Ok(())
        } else {
            Err(String::from("two plus two does not equal four"))
        }
    }
}
```

Функція `it_works` зараз повертає тип `Result<(), String>`. У тілі функції, замість того, щоб викликати макрос `assert_eq!`, ми повертаємо `Ok(())`, коли тест пройшов і `Err` зі `String` всередині, коли тест провалено.

Написання тестів, щоб вони повертали `Result<T, E>`, дозволить вам використовувати оператор знак питання у тілі тестів, що може бути зручним способом писати тести, що мають провалитися, якщо будь-яка операція всередині них повертає варіант `Err`.

Ви не можете використовувати анотацію `#[should_panic]` в тестах, які використовують `Result<T, E>`. Щоб ствердити, що операція повертає варіант `Err`, не використовуйте оператор знак питання значенні на `Result<T, E>`. Натомість, використовуйте `assert!(value.is_err())`.

Тепер, коли ви знаєте кілька способів писати тести, подивімося на те, що відбувається, коли ми запускаємо наші тести і дослідимо різні опції, як ми можемо використовувати з `cargo test`. [ch08-02-strings.html#concatenation-with-the--operator-or-the-format-macro](https://pavloslav.github.io/RustBookUkr/print.html#ch08-02-strings.html#concatenation-with-the--operator-or-the-format-macro) [ch11-02-running-tests.html#controlling-how-tests-are-run](https://pavloslav.github.io/RustBookUkr/print.html#ch11-02-running-tests.html#controlling-how-tests-are-run)

Керування Запуском Тестів

Так само як `cargo run` компілює ваш код і запускає утворений виконуваний файл, `cargo test` компілює ваш код у режимі тестів і запускає утворений тестовий виконуваний файл. За замовчанням виконуваний файл, згенерований `cargo test`, запускає всі тести паралельно і перехоплює вивід, згенерований під час виконання тестів, запобігаючи виведенню на екран і спрощуючи читання виведення, яке стосується результатів тестів. Однак ви можете вказати опції командного рядка, щоб змінити таку поведінку за замовчанням.

Деякі опції командного рядка стосуються `cargo test`, а деякі - вихідного тестового виконуваного файлу. Щоб розділити ці два типи аргументів, треба вказати аргументи, що стосуються `cargo test`, далі розділювач `--`, а потім ті, що стосуються тестового виконуваного файлу. Запуск `cargo test --help` покаже опції, що можна використовувати з `cargo test`, а запуск `cargo test -- --help` покаже опції, що можна вказувати після розділювача.

Запуск Тестів Паралельно чи Послідовно

Коли ви запускаєте декілька тестів, за замовчуванням вони запускаються паралельно у кількох потоках, тобто вони закінчують роботу швидше і ви скоріше отримуєте зворотний зв'язок. Оскільки тести виконуються одночасно, ви повинні переконатися, що ваші тести не залежать один від одного або від будь-якого спільного стану, включно зі спільним середовищем, наприклад поточною робочою текою чи змінною середовища.

Наприклад, нехай кожен з ваших тестів запускає певний код, що створює файл на диску з назвою `test-output.txt` і записує якісь дані в цей файл. Потім кожен тест зчитує дані з цього файлу та перевіряє, що файл містить певне значення, яке різниться в кожному тесті. Оскільки тести виконуються одночасно, один тест може перезаписати файл у час між тим, коли інший тест пише і читає цей файл. Другий тест тоді провалиться - не тому, що код неправильний, але тому, що тести втручалися в роботу один одного під час паралельної роботи. Одне можливе рішення - переконатися, що кожен тест пише в окремий файл; інше рішення - запускати тести по одному за раз.

Якщо ви не хочете запускати тести паралельно, або якщо хочете мати більш докладний контроль над кількістю потоків, ви можете встановити прапорець `--test-threads` і кількість потоків, які Ви хочете використовувати для тестування. Погляньте на наступний приклад:

```
$ cargo test -- --test-threads=1
```

Ми встановили кількість потоків тестів на значення **1**, які повідомляють програмі не використовувати паралелізм. Виконання тестів за допомогою одного потоку займе більше часу, ніж запуск їх паралельно, але тести не будуть втручатися один одного, якщо вони мають спільний стан.

Показ Виводу Функції

За замовчуванням, якщо тест проходить вдало, бібліотека тестування Rust перехоплює все, що виводиться у стандартний вихідний потік. Наприклад, якщо ми викличемо `println!` у тесті й тест проходить, ми не побачимо виведення з `println!` терміналі; ми побачимо тільки рядок, який каже, що тест пройдено. Якщо тест провалено, ми побачимо все, що було виведено до стандартного потоку виведення з рештою повідомлення про помилку.

Як приклад, Блок коду 11-10 має простеньку функцію, яка друкує значення свого параметра і повертає 10, а також і тест, що проходить і тест, що провалюється.

Файл: `src/lib.rs`

```
fn prints_and_returns_10(a: i32) -> i32 {
    println!("I got the value {}", a);
    10
}

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

    #[test]
    fn this_test_will_pass() {
        let value = prints_and_returns_10(4);
        assert_eq!(10, value);
    }

    #[test]
    fn this_test_will_fail() {
        let value = prints_and_returns_10(8);
        assert_eq!(5, value);
    }
}
```

Блок коду 11-10: Тести для функції, що викликає `println!`

Коли ми запусимо ці тести за допомогою `cargo test`, то побачимо таке:

```

$ cargo test
  Compiling silly-function v0.1.0 (file:///projects/silly-function)
  Finished test [unoptimized + debuginfo] target(s) in 0.58s
  Running unittests src/lib.rs (target/debug/deps/silly_function-160869f38cff9166)

running 2 tests
test tests::this_test_will_fail ... FAILED
test tests::this_test_will_pass ... ok

failures:

---- tests::this_test_will_fail stdout ----
I got the value 8
thread 'main' panicked at 'assertion failed: `(left == right)`
  left: `5`,
 right: `10`', src/lib.rs:19:9
note: run with `RUST_BACKTRACE=1` environment variable to display a backtrace

failures:
  tests::this_test_will_fail

test result: FAILED. 1 passed; 1 failed; 0 ignored; 0 measured; 0 filtered out;
finished in 0.00s

error: test failed, to rerun pass `--lib`

```

Зверніть увагу, що у виведеному ніде немає `I got the value 4` - того, що виводиться в тесті, що проходить. Це виведення було перехоплено. Виведення з тесту, що провалився, `I got the value 8`, з'являється в розділі підсумків тесту, де також показана і причина провалу тесту.

Якщо ми хочемо вивести значення і для тестів, що пройшли, ми можемо сказати Rust також показати виведення з успішних тестів за допомогою `--show-output`.

```
$ cargo test -- --show-output
```

Коли ми запустимо тести з Блоку коду 11-10 знову, вказавши прапорець `--show-output`, то побачимо таке:

```
$ cargo test -- --show-output
   Compiling silly-function v0.1.0 (file:///projects/silly-function)
   Finished test [unoptimized + debuginfo] target(s) in 0.60s
   Running unittests src/lib.rs (target/debug/deps/silly_function-160869f38cff9166)

running 2 tests
test tests::this_test_will_fail ... FAILED
test tests::this_test_will_pass ... ok

successes:

---- tests::this_test_will_pass stdout ----
I got the value 4

successes:
    tests::this_test_will_pass

failures:

---- tests::this_test_will_fail stdout ----
I got the value 8
thread 'main' panicked at 'assertion failed: `(left == right)`
  left: `5`,
 right: `10`', src/lib.rs:19:9
note: run with `RUST_BACKTRACE=1` environment variable to display a backtrace

failures:
    tests::this_test_will_fail

test result: FAILED. 1 passed; 1 failed; 0 ignored; 0 measured; 0 filtered out;
finished in 0.00s

error: test failed, to rerun pass `--lib`
```

Запуск Підмножини Тестів за Назвою

Іноді виконання повного набору тестів може тривати довго. Якщо ви працюєте над кодом у певній області, то можете захотіти запускати лише тести, що містять цей код. Ви можете обирати, які тести виконати, передавши `cargo test` ім'я чи імена тесту(ів), які хочете запустити, як аргумент.

Щоб продемонструвати, як запустити частину тестів, ми спершу створимо три тести для нашої функції `add_two`, як показано у Блоці коду 11-11, і оберемо, які з них запустити.

Файл: `src/lib.rs`

```
pub fn add_two(a: i32) -> i32 {
    a + 2
}

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

    #[test]
    fn add_two_and_two() {
        assert_eq!(4, add_two(2));
    }

    #[test]
    fn add_three_and_two() {
        assert_eq!(5, add_two(3));
    }

    #[test]
    fn one_hundred() {
        assert_eq!(102, add_two(100));
    }
}
```

Блок коду 11-11: Три тести з різними іменами

Якщо ми запустимо тести, не передавши жодних аргументів, то, як ми бачили раніше, всі тести запустяться паралельно:

```
$ cargo test
Compiling adder v0.1.0 (file:///projects/adder)
Finished test [unoptimized + debuginfo] target(s) in 0.62s
Running unittests src/lib.rs (target/debug/deps/adder-92948b65e88960b4)

running 3 tests
test tests::add_three_and_two ... ok
test tests::add_two_and_two ... ok
test tests::one_hundred ... ok

test result: ok. 3 passed; 0 failed; 0 ignored; 0 measured; 0 filtered out;
finished in 0.00s

Doc-tests adder

running 0 tests

test result: ok. 0 passed; 0 failed; 0 ignored; 0 measured; 0 filtered out;
finished in 0.00s
```

Запуск Одного Тесту

Ми можемо передати назву будь-якої тестової функції `cargo test`, щоб запустити тільки цей тест:

```
$ cargo test one_hundred
Compiling adder v0.1.0 (file:///projects/adder)
Finished test [unoptimized + debuginfo] target(s) in 0.69s
Running unittests src/lib.rs (target/debug/deps/adder-92948b65e88960b4)

running 1 test
test tests::one_hundred ... ok

test result: ok. 1 passed; 0 failed; 0 ignored; 0 measured; 2 filtered out;
finished in 0.00s
```

Лише тест з ім'ям `one_hundred` було виконано; інші два тести мають невідповідні імена. Вивід тесту дає нам знати, що ми мали більше тестів, що не були запущені, показавши наприкінці `2 filtered out`.

Ми не можемо таким чином вказувати імена кількох тестів; буде використане лише перше значення, передане `cargo test`. Але є спосіб запустити кілька тестів.

Фільтрування для Запуску Кількох Тестів

Ми можемо вказати частину назви тесту, і всі тести, чиї імена відповідають цьому значенню, будуть запущені. Наприклад, оскільки дві назви наших тестів містять `add`, ми можемо виконати ці два тести, запустивши `cargo test add`:

```
$ cargo test add
Compiling adder v0.1.0 (file:///projects/adder)
Finished test [unoptimized + debuginfo] target(s) in 0.61s
Running unittests src/lib.rs (target/debug/deps/adder-92948b65e88960b4)

running 2 tests
test tests::add_three_and_two ... ok
test tests::add_two_and_two ... ok

test result: ok. 2 passed; 0 failed; 0 ignored; 0 measured; 1 filtered out;
finished in 0.00s
```

Ця команда виконала всі тести, що містили `add` у назві й відфільтрувала тест з назвою `one_hundred`. Також зверніть увагу, що модуль, в якому з'являється тест, перетворюється

на частину імені тесту, тож ми можемо запустити усі тести в модулі, відфільтрувавши тести за ім'ям модуля.

Ігнорування Деяких Тестів якщо Немає Спеціального Запиту

Іноді кілька специфічних тестів можуть витрачати дуже багато часу для виконання, так що ви можете захотіти виключити їх під час більшості запусків `cargo test`. Замість того, щоб перелічувати всі тести, які ви хочете запустити, як аргументи, ви можете натомість додати анотацію трудомістких тестів, додавши атрибут `ignore`, щоб виключити їх, як показано тут:

Файл: `src/lib.rs`

```
#[test]
fn it_works() {
    assert_eq!(2 + 2, 4);
}

#[test]
#[ignore]
fn expensive_test() {
    // code that takes an hour to run
}
```

Після `#[test]` ми додаємо рядок `#[ignore]` до тесту, що його ми хочемо виключити. Тепер, коли ми запускаємо наші тести, `it_works` запускається, а `expensive_test` - ні:

```
$ cargo test
Compiling adder v0.1.0 (file:///projects/adder)
Finished test [unoptimized + debuginfo] target(s) in 0.60s
Running unittests src/lib.rs (target/debug/deps/adder-92948b65e88960b4)

running 2 tests
test expensive_test ... ignored
test it_works ... ok

test result: ok. 1 passed; 0 failed; 1 ignored; 0 measured; 0 filtered out;
finished in 0.00s

Doc-tests adder

running 0 tests

test result: ok. 0 passed; 0 failed; 0 ignored; 0 measured; 0 filtered out;
finished in 0.00s
```

Функція `expensive_test` показана як `ignored`. Якби ми захотіли запустити лише ігноровані тести, то могли б запустити `cargo test -- --ignored`:

```
$ cargo test -- --ignored
Compiling adder v0.1.0 (file:///projects/adder)
Finished test [unoptimized + debuginfo] target(s) in 0.61s
Running unittests src/lib.rs (target/debug/deps/adder-92948b65e88960b4)

running 1 test
test expensive_test ... ok

test result: ok. 1 passed; 0 failed; 0 ignored; 0 measured; 1 filtered out;
finished in 0.00s

Doc-tests adder

running 0 tests

test result: ok. 0 passed; 0 failed; 0 ignored; 0 measured; 0 filtered out;
finished in 0.00s
```

Контролюючи, які тести запустити, ви можете забезпечити швидкість результатів `cargo test`. Коли ви дістанетеся до етапу, коли матиме сенс перевірити результати тестів `ignored` і матимете час дочекатися результатів, то зможете натомість запустити `cargo test -- --ignored`. Якщо ви хочете запустити усі тести, ігноровані чи ні, то можете запустити `cargo test -- --include-ignored`.

Організація Тестів

Як зазначено на початку розділу, тестування є складною дисципліною, і різні люди використовують різну термінологію та організацію. Спільнота Rust думає про тести з точки зору двох основних категорій: модульні тести та інтеграційні тести. *Модульні тести* є невеликими та більш сфокусованими, ізольовано тестують один модуль за один раз, і можуть тестувати приватні інтерфейси. *Інтеграційні тести* є повністю зовнішніми до вашої бібліотеки та використовують ваш код так само як будь-який інший зовнішній код, використовуючи тільки публічний інтерфейс і потенційно випробовуючи багато модулів під час тесту.

Написання обох типів тестів є важливим, щоб переконатися, що частини вашої бібліотеки роблять те, на що ви очікували від них, окремо і разом.

Модульні Тести

Мета модульних тестів — перевірити кожну одиницю коду ізольованою від решти коду, щоб швидко визначити точку, де код не працює як очікувалося. Модульні тести розташовуються в теці `src` в кожному файлі коду, який вони тестують. За домовленістю, у кожному файлі, що містить функції для тестування, створюється модуль з назвою `tests`, анотований `cfg(test)`.

Модуль tests і `#[cfg(test)]`

Анотація модуля tests `#[cfg(test)]` каже Rust компілювати і виконувати тестовий код лише коли ви запускаєте `cargo test`, а не `cargo build`. Це зберігає час компіляції, коли ви хочете зібрати бібліотеку, і зберігає місце у отриманому скомпільованому артефакті, бо тести не до нього не включені. Як ви побачите, оскільки інтеграційні тести розміщуються в іншій теці, вони не потребують анотації `#[cfg(test)]`. Однак, оскільки модульні тести розміщуються у тих самих файлах, що й код, вам треба вказувати `#[cfg(test)]`, щоб позначити, що їх не треба включати у результат компіляції.

Згадайте, що коли ми створили новий проєкт `adder` у першому підрозділі цього розділу, Cargo згенерував для нас цей код:

Файл: `src/lib.rs`

```
#[cfg(test)]
mod tests {
    #[test]
    fn it_works() {
        let result = 2 + 2;
        assert_eq!(result, 4);
    }
}
```

Цей код є автоматично згенерованим модульним тестом. Атрибут `cfg` означає *конфігурація* і каже Rust, що наступний елемент має включатися лише з певною опцією конфігурації. У цьому випадку опцією конфігурації є `test`, що надається Rust для компіляції і запуску тестів. Використовуючи атрибут `cfg`, ми вказуємо Cargo компілювати наш тестовий код лише коли ми явно запускаємо тести за допомогою `cargo test`. Це стосується і будь-яких допоміжних функцій, що можуть бути в цьому модулі, на додачу до функцій, анотованих `#[test]`.

Тестування Приватних Функцій

У тестовій спільноті є дискусія про те, чи мають приватні функції тестуватися безпосередньо, і інші мови ускладнюють або унеможливають тестування приватних функцій. Незалежно від того, якої тестової ідеології ви дотримуетесь, правила приватності Rust дозволяють вам тестувати приватні функції. Розгляньте код у Блоці коду 11-12 з приватною функцією `internal_adder`.

Файл: `src/lib.rs`

```
pub fn add_two(a: i32) -> i32 {
    internal_adder(a, 2)
}

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

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

    #[test]
    fn internal() {
        assert_eq!(4, internal_adder(2, 2));
    }
}
```

Блок коду 11-12: тестування приватної функції

Зверніть увагу, що функція `internal_adder` не позначена як `pub`. Тести - це просто код Rust, а модуль `tests` - це просто ще один модуль. Як ми вже говорили в підрозділі “Способи звернутися до елемента в дереві модулів”, елементи дочірніх модулів можуть використовувати елементи своїх батьківських модулів. У цьому тесті, ми вводимо всі елементи батьківського для `test` модуля в область видимості за допомогою `use super::*`, і тоді тест може викликати `internal_adder`. Якщо ви не вважаєте, що приватні функції мають бути протестовані, немає нічого в Rust, що змусить вас це робити.

Інтеграційні Тести

У Rust, інтеграційні тести є цілковито зовнішніми відносно до вашої бібліотеки. Вони використовують вашу бібліотеку так само як це робив би будь-який інший код, що означає, що вони можуть викликати лише функції, які є частиною публічного API вашої бібліотеки. Їхнє призначення - перевірити, чи правильно різні частини вашої бібліотеки працюють разом. Фрагменти коду, які правильно самі по собі працюють, можуть мати проблеми при інтеграції, тому покриття інтегрованого коду тестами також важливе. Для створення інтеграційних тестів вам знадобиться для початку тека `tests`.

Тека `tests`

Ми створимо теку `tests` на верхньому рівні тек нашого проєкту, поруч із `src`. Cargo знає, що файли інтеграційних тестів треба шукати в цій теці. Ми можемо зробити стільки тестових файлів, скільки захочемо, і Cargo скомпілює кожен з файлів як окремий крейт.

Створімо інтеграційний тест. Поки у файлі `src/lib.rs` все ще код з Блоку коду 11-12, створіть теку `tests`, а в ній - новий файл, з назвою `tests/integration_test.rs`. Структура вашої теки має виглядати ось так:

```
adder
├── Cargo.lock
├── Cargo.toml
├── src
│   └── lib.rs
└── tests
    └── integration_test.rs
```

Введіть код з Блоку коду 11-13 у файл `tests/integration_test.rs`:

Файл: `tests/integration_test.rs`

```
use adder;

#[test]
fn it_adds_two() {
    assert_eq!(4, adder::add_two(2));
}
```

Блок коду 11-13: інтеграційний тест функції з крейту `adder`

Кожен файл у теці `tests` є окремим крейтом, тож нам потрібно ввести нашу бібліотеку до області видимості кожного тестового крейту. Саме тому ми додаємо `use adder` на початку коду, чого не робили в модульних тестах.

Нам не треба додавати до коду у `tests/integration_test.rs` анотацію `#[cfg(test)]`. Cargo розглядає теку `tests` окремо і компілює файли у цій теці лише коли ми запускаємо `cargo test`. Запустімо зараз `cargo test`:

```
$ cargo test
Compiling adder v0.1.0 (file:///projects/adder)
Finished test [unoptimized + debuginfo] target(s) in 1.31s
Running unittests src/lib.rs (target/debug/deps/adder-1082c4b063a8fbe6)

running 1 test
test tests::internal ... ok

test result: ok. 1 passed; 0 failed; 0 ignored; 0 measured; 0 filtered out;
finished in 0.00s

Running tests/integration_test.rs (target/debug/deps/integration_test-1082c4b063a8fbe6)

running 1 test
test it_adds_two ... ok

test result: ok. 1 passed; 0 failed; 0 ignored; 0 measured; 0 filtered out;
finished in 0.00s

Doc-tests adder

running 0 tests

test result: ok. 0 passed; 0 failed; 0 ignored; 0 measured; 0 filtered out;
finished in 0.00s
```

Три секції виводу містять модульні тести, інтеграційні тести та документаційні тести. Зверніть увагу, що якщо будь-який тест у секції провалиться, наступна секція не буде запущена. Наприклад, якщо провалиться модульний тест, для інтеграційних і

документаційних тестів не буде виведено нічого, бо ці тести будуть запущені лише якщо всі модульні тести пройдуть.

Перша секція для модульних тестів така сама, яку ми вже бачили: по рядку для кожного модульного тесту (один, що зветься `internal`, який ми додали у Блоці коду 11-12) і далі рядок підсумку для модульних тестів.

Секція інтеграційних тестів починається рядком `Running tests/integration_test.rs`. Далі по рядку для кожної тестової функції у інтеграційному тесті і рядок підсумку для результатів інтеграційних тестів прямо перед початком секції `Doc-tests adder`.

Кожен файл інтеграційного тесту має свою власну секцію, тому якщо ми додамо більше файлів до теки *tests*, буде більше секцій інтеграційних тестів.

Ми все ще можемо запустити певну функцію інтеграційного тесту, вказавши назву тестової функції як аргумент до `cargo test`. Щоб запустити всі тести з певного файлу інтеграційних тестів, вкажіть `cargo test` аргумент `--test` із назвою файлу:

```
$ cargo test --test integration_test
   Compiling adder v0.1.0 (file:///projects/adder)
   Finished test [unoptimized + debuginfo] target(s) in 0.64s
   Running tests/integration_test.rs (target/debug/deps/integration_test-82e7799c1bc62298)

running 1 test
test it_adds_two ... ok

test result: ok. 1 passed; 0 failed; 0 ignored; 0 measured; 0 filtered out;
finished in 0.00s
```

Ця команда виконає лише тести у файлі *tests/integration_test.rs*.

Підмодулі в Інтеграційних Тестах

При додаванні інтеграційних тестів для кращої організації ви можете захотіти створити більше файлів у теці *tests*; наприклад, ви можете згрупувати тестові функції за функціоналом, який вони тестують. Як згадувалося раніше, кожен файл у теці *tests* компілюється як окремий крейт, що є корисним для створення окремих областей видимості для більш ретельного наслідування того, як кінцеві користувачі будуть використовувати ваш крейт. Проте це означає, що файли в теці *tests* не виявляють таку ж поведінку як файли у *src*, як ви дізналися в Розділі 7 щодо того, як відокремити код в модулі та файли.

Відмінна поведінка каталогу *tests* є найбільш помітною, коли ви маєте набір допоміжних функцій, які використовуються в декількох файлах інтеграційних тестів і ви намагаєтесь слідувати крокам з підрозділу "Розподіл модулів на різні файли" Розділу 7, щоб винести їх у спільний модуль. Наприклад, якщо ми створимо *tests/common.rs* і розмістимо там функцію з назвою `setup`, ми можемо додати в цю функцію код, що ми хочемо викликати з декількох тестових функцій у декількох тестових файлах:

Файл: *src/common.rs*

```
pub fn setup() {  
    // setup code specific to your library's tests would go here  
}
```

Коли ми знову запустимо тести, то побачимо нову секцію у виведенні тестів для файлу *common.rs*, хоча цей файл не містить жодних тестових функцій і ми нізвідки не викликали функцію `setup`:

```

$ cargo test
  Compiling adder v0.1.0 (file:///projects/adder)
  Finished test [unoptimized + debuginfo] target(s) in 0.89s
  Running unittests src/lib.rs (target/debug/deps/adder-92948b65e88960b4)

running 1 test
test tests::internal ... ok

test result: ok. 1 passed; 0 failed; 0 ignored; 0 measured; 0 filtered out;
finished in 0.00s

    Running tests/common.rs (target/debug/deps/common-92948b65e88960b4)

running 0 tests

test result: ok. 0 passed; 0 failed; 0 ignored; 0 measured; 0 filtered out;
finished in 0.00s

    Running tests/integration_test.rs (target/debug/deps/integration_test-
92948b65e88960b4)

running 1 test
test it_adds_two ... ok

test result: ok. 1 passed; 0 failed; 0 ignored; 0 measured; 0 filtered out;
finished in 0.00s

Doc-tests adder

running 0 tests

test result: ok. 0 passed; 0 failed; 0 ignored; 0 measured; 0 filtered out;
finished in 0.00s

```

Побачити `common` серед результатів тестів з уточненням `running 0 tests` - ми не цього хотіли. Ми хотіли лише мати код, спільний для кількох файлів інтеграційних тестів.

Щоб `common` не з'являвся в результатах тестів, замість створення `tests/common.rs` ми створимо `tests/common/mod.rs`. Тека проєкту тепер виглядає так:

```

├── Cargo.lock
├── Cargo.toml
├── src
│   └── lib.rs
├── tests
│   ├── common
│   │   └── mod.rs
│   └── integration_test.rs

```

Це давніше правило іменування, яке Rust також розуміє, про яке ми згадували у підрозділі "Альтернативні шляхи файлів" Розділу 7. Те, що файл названо у цей спосіб, каже Rust не розглядати модуль `common` як файл інтеграційного тесту. Коли ми перемістимо код функції `setup` до `tests/common/mod.rs` і видалимо файл `tests/common.rs`, секція для цього файлу більше не показуватиметься. Файли в підтеках теки `tests` не компілюються як окремі крейти і не мають секції в виведенні тестів.

Після того, як ми створили `tests/common/mod.rs`, ми можемо використовувати його з будь-якого з тестових файлів як модуль. Ось приклад виклику функції `setup` з тесту `it_adds_two` в `tests/integration_test.rs`:

Файл: `tests/integration_test.rs`

```
use adder;

mod common;

#[test]
fn it_adds_two() {
    common::setup();
    assert_eq!(4, adder::add_two(2));
}
```

Зверніть увагу, що проголошення `mod common;` - те саме, що й проголошення модуля, продемонстроване в Блоці коду 7-21. Тоді з тестової функції ми можемо викликати функцію `common::setup()`.

Інтеграційні Тести для Двійкових Крейтів

Якщо наш проєкт є двійковим крейтом, що містить лише файл `src/main.rs` і не має файлу `src/lib.rs`, ми не можемо створювати інтеграційні тести у теці `tests` і вводити в область видимості функції, визначені у файлі `src/main.rs`, за допомогою інструкції `use`. Лише бібліотечні крейти надають функції для використання в інших крейтах; двійкові крейти призначені лише для запуску.

Це - одна з причин, чому проєкти Rust, що створюють двійковий файл, мають простий файл `src/main.rs`, що викликає логіку з файлу `src/lib.rs`. За такої структури інтеграційні тести *можуть* тестувати бібліотечний крейт, використовуючи `use`, щоб дістатися до важливого функціоналу. Якщо важливий функціонал працює, невеликий код у файлі `src/main.rs` також працюватиме, і цей невеликий код не треба тестувати.

Підсумок

Можливості тестування Rust надають можливість вказати, як код має працювати, щоб переконатися, що він і надалі працює як очікувалося, навіть якщо ви його зміните. Модульні тести випробовують різні частини бібліотеки окремо і можуть тестувати приватні деталі реалізації. Інтеграційні тести перевіряють, що різні частини бібліотеки коректно працюють разом, і вони використовують публічний API бібліотеки для тестування коду, так само, як це робитиме сторонній код. Попри те, що система типів Rust і правила власності допомагають уникнути деяких видів помилок, тести все одно є важливими для зменшення логічних помилок, які стосуються очікуваної поведінки вашого коду.

Застосуймо усі знання, отримані в цьому та попередніх розділах, щоб попрацювати над проєктом! [ch07-05-separating-modules-into-different-files.html](https://pavloslav.github.io/RustBookUkr/print.html)

Проект з Вводом/Виводом: Створення Програми з Інтерфейсом Командного Рядка

Цей розділ підсумовує багато навичок, які ви отримали досі, та досліджує ще декілька можливостей стандартної бібліотеки. Ми розробимо інструмент командного рядка, що взаємодіятиме введенням/виведенням до файлів і командного рядка, щоб потренуватися з уже вивченими концепціями Rust.

Швидкість, безпека, єдиний двійковий файл як результат компіляції та підтримка багатоплатформеності роблять Rust ідеальною мовою для створення інструментів командного рядка, отже, для нашого проекту ми створимо власну версію класичного інструменту для пошуку через командний рядок `grep` (globally search a regular expression and print, глобальний пошук регулярного виразу і виведення). У найпростішому випадку `grep` шукає вказану стрічку у вказаному файлі. Для цього `grep` приймає параметрами шлях до файлу і стрічку, а далі читає файл, знаходить рядки що містять параметр-стрічку і друкує ці рядки.

Дорогою ми покажемо як залучити до нашого інструмента командного рядка поширені функції термінала, які використовуються в багатьох інших інструментах командного рядка. Ми прочитаємо значення змінної середовища, щоб дозволити користувачеві сконфігурувати поведінку нашого інструменту. Також ми введемо повідомлення про помилки до стандартного потоку виведення помилок (`stderr`), а замість стандартного потоку виведення (`stdout`), щоб, скажімо, користувач міг перенаправити вдалий результат до файлу і все ж побачив повідомлення про помилки на екрані.

Один з членів спільноти Rust, Andrew Gallant, вже створив повнофункціональну, дуже швидко версію `grep`, що зветься `ripgrep`. Наша версія, як порівняти, буде доволі простою, але цей розділ дасть вам певні початкові знання, що знадобляться для розуміння реальних проектів як-от `ripgrep`.

Наш проект `grep` об'єднає низку концепцій, що ви вже вивчили:

- Організація коду (застосування того, що ви вже вивчили про модулі у [Розділі 7](#))
- Використання векторів та стрічок (колекцій, [Розділ 8](#))
- Обробка помилок ([Розділ 9](#))
- Використання трейтів і часів життя, де це потрібно ([Розділ 10](#))
- Написання тестів ([Розділ 11](#))

Ми також коротко представимо замикання, ітератори і трейтові об'єкти, про які детальніше йтиметься в розділах [13](#) і [17](#).

Приймання Аргументів Командного Рядка

Створімо новий проєкт за допомогою, як завжди, `cargo new`. Ми назвемо наш проєкт `minigrep`, щоб вирізнити його від інструменту `grep`, що вже може бути встановлено у вашій системі.

```
$ cargo new minigrep
    Created binary (application) `minigrep` project
$ cd minigrep
```

Перше завдання - зробити, щоб `minigrep` приймав два аргументи командного рядка: шлях до файлу і стрічку для пошуку. Тобто ми хочемо, щоб нашу програму можна було запускати за допомогою `cargo run`, двох рисок на позначення що подальші аргументи стосуються нашої програми, а не `cargo`, стрічки для пошуку і шляху до файлу, в якому треба шукати, ось так:

```
$ cargo run -- searchstring example-filename.txt
```

Наразі програма, створена `cargo new`, не може обробляти аргументи, передані їй. Певні бібліотеки з crates.io можуть допомогти писати програму, що приймає аргументи командного рядка, але оскільки ви лише вивчаєте цю концепцію, запровадимо цю можливість самостійно.

Читання Значень Аргументів

Щоб дозволити `minigrep` читати значення аргументів командного рядка, переданих йому, нам знадобиться функція `std::env::args` зі стандартної бібліотеки Rust. Ця функція поверне ітератор аргументів командного рядка, переданих `minigrep`. Повніше про ітератори піде у [Розділі 13](#). Поки що вам лише треба знати про ітератори дві речі: ітератори створюють послідовність значень, і ми можемо викликати метод

`collect` для ітератора, щоб перетворити його на колекцію, таку як вектор, що міститиме всі елементи, створені ітератором.

`collect` method on an iterator to turn it into a collection, such as a vector, that contains all the elements the iterator produces.

Файл: `src/main.rs`

```
use std::env;

fn main() {
    let args: Vec<String> = env::args().collect();
    dbg!(args);
}
```

Listing 12-1: Collecting the command line arguments into a vector and printing them

Спершу ми вводим модуль `std::env` до області видимості за допомогою інструкції `use`, щоб можна було скористатися функцією `args` з цього модуля. Зверніть увагу, що функція `std::env::args` вложена у два рівні модулів. Як ми вже говорили у [Розділі 7](#), у випадках, коли потрібна функція, вложена глибше одного модуля, краще ввести в область видимості її батьківський модуль, аніж саму функцію. Таким чином, ми зможемо легко використовувати інші функції з `std::env`. Також це дещо менш двозначне, ніж додавання `use std::env::args` і виклик функції як просто `args`, бо просто `args` можна легко переплутати з функцією, визначеною в поточному модулі.

Функція `args` та Неправильний Юнікод

Зверніть увагу, що `std::env::args` запанікує, якщо якийсь із аргументів містить некоректний юнікод. Якщо вашій програмі треба приймати аргументи з некоректним юнікодом, скористайтеся натомість функцією `std::env::args_os`. Вона повертає ітератор, що створює значення `OsString` замість `String`. Ми вирішили скористатися `std::env::args` для простоти, бо значення `OsString` різняться між платформами і з ними складніше працювати, ніж зі `String`.

У першому рядку `main` ми викликаємо `env::args` і одразу ж використовуємо `collect`, щоб перетворити ітератор на вектор, що містить усі значення, вироблені ітератором. Ми можемо використати функцію `collect`, щоб створити багато видів колекцій, тому явно позначаємо тип `args`, щоб вказати, що нам потрібен вектор стрічок. Хоча в Rust дуже нечасто треба позначати типи, `collect` є однією з функцій, яка часто потребує анотацій, бо Rust неспроможний вивести потрібний тип колекції.

В кінці ми виводимо вектор за допомогою макросу для зневадження. Спробуймо тепер запустити код спершу без аргументів, а тоді з двома аргументами:

```
$ cargo run
  Compiling minigrep v0.1.0 (file:///projects/minigrep)
  Finished dev [unoptimized + debuginfo] target(s) in 0.61s
  Running `target/debug/minigrep`
[src/main.rs:5] args = [
  "target/debug/minigrep",
]
```

```
$ cargo run -- needle haystack
  Compiling minigrep v0.1.0 (file:///projects/minigrep)
  Finished dev [unoptimized + debuginfo] target(s) in 1.57s
  Running `target/debug/minigrep needle haystack`
[src/main.rs:5] args = [
  "target/debug/minigrep",
  "needle",
  "haystack",
]
```

Зверніть увагу, що перше значення у векторі - `"target/debug/minigrep"`, тобто назва нашого двійкового файлу. Це відповідає поведінці списку параметрів у C, що дозволяє програмам використовувати ім'я, за яким їх викликано, під час виконання. Часто буває зручно мати доступ до імені програми, якщо ви хочете вивести його у повідомленнях чи змінити поведінку програми залежно від того, який псевдонім був використаний у командному рядку для запуску програми. Але задля потреб нашого розділу ми пропустимо його і збережемо лише два потрібні параметри.

Збереження Значень Аргументів в Змінних

Програма вже може отримати значення, задані аргументами командного рядка. Тепер нам треба зберегти значення двох аргументів у змінних, щоб можна було використати ці значення далі в програмі. Це ми робимо у Блоці коду 12-2.

Файл: src/main.rs

```
use std::env;

fn main() {
    let args: Vec<String> = env::args().collect();

    let query = &args[1];
    let file_path = &args[2];

    println!("Searching for {}", query);
    println!("In file {}", file_path);
}
```

Listing 12-2: Creating variables to hold the query argument and file path argument

Як ми бачили, коли виводили вектор, ім'я програми займає перше значення у векторі за індексом `args[0]`, тому ми починаємо аргументи з індексу `1`. Перший аргумент, що приймає `minigrep` - це шукана стрічка, тож ми розміщуємо посилання на перший аргумент у змінній `query`. Другий аргумент буде шляхом до файлу, тож ми розміщуємо посилання на другий аргумент у змінній `file_path`.

Ми тимчасово виводимо значення цих змінних, щоб підтвердити, що код працює, як ми очікували. Запустимо цю програму знову з аргументами `test` і `sample.txt`:

```
$ cargo run -- test sample.txt
  Compiling minigrep v0.1.0 (file:///projects/minigrep)
  Finished dev [unoptimized + debuginfo] target(s) in 0.0s
  Running `target/debug/minigrep test sample.txt`
Searching for test
In file sample.txt
```

Чудово, програма працює! Значення потрібних нам аргументів зберігаються у правильних змінних. Пізніше ми додамо трохи обробки помилок, щоб розібратися з певними потенційними помилковими ситуаціями, на кшталт коли користувач не надає жодних параметрів; а поки що ігноруватимемо цю ситуацію і натомість займемося додаванням можливостей для читання файлів.

Читання Файлу

Тепер додамо функціональність для читання файлу, заданого параметром `file_path`. Для початку нам знадобиться файл для тестування, і ми скористаємося файлом із невеликим текстом у кілька рядків із повторенням слів. Блок коду 12-3 містить вірш Емілі Дікінсон, що добре підійде для цього! Створіть файл *poem.txt* у кореневому рівні вашого проєкту, і введіть вірш "Я ніхто! А ти хто?"

Файл: poem.txt

```
I'm nobody! Who are you?  
Are you nobody, too?  
Then there's a pair of us - don't tell!  
They'd banish us, you know.  
  
How dreary to be somebody!  
How public, like a frog  
To tell your name the livelong day  
To an admiring bog!
```

Роздрук 12-3: Вірш Емілі Дікінсон вдало підходить для тестування

Підготувавши текст, відредагуйте *src/main.rs* і додайте код для читання файлу, як вказано у Блоці коду 12-4.

Файл: src/main.rs

```
use std::env;  
use std::fs;  
  
fn main() {  
    // --snip--  
    println!("In file {}", file_path);  
  
    let contents = fs::read_to_string(file_path)  
        .expect("Should have been able to read the file");  
  
    println!("With text:\n{contents}");  
}
```

Блок коду 12-4: Читання вмісту файлу, вказаного другим параметром

Спершу, ми вводимо в область видимості відповідну частину стандартної бібліотеки за допомогою інструкції `use`; для обробки файлів потрібен `std::fs`.

У `main` нова інструкція `fs::read_to_string` бере `file_path`, відкриває цей файл і повертає `std::io::Result<String>` з його вмістом.

По тому знову додаємо тимчасову інструкцію `println!`, що видрукує значення `contents` після читання файлу, щоб ми змогли перевірити, як програма працює.

Запустимо цей код з будь-якою стрічкою першим параметром командного рядка (бо ми ще не додали частину для пошуку) і *poem.txt* другим параметром:

```
$ cargo run -- the poem.txt
Compiling minigrep v0.1.0 (file:///projects/minigrep)
Finished dev [unoptimized + debuginfo] target(s) in 0.0s
Running `target/debug/minigrep the poem.txt`
Searching for the
In file poem.txt
With text:
I'm nobody! Who are you?
Are you nobody, too?
Then there's a pair of us - don't tell!
They'd banish us, you know.

How dreary to be somebody!
How public, like a frog
To tell your name the livelong day
To an admiring bog!
```

Чудово! Код прочитав і надрукував вміст файлу. Але код має кілька недоліків. Наразі функція `main` відповідає за багато різних речей. В цілому, функції стають зрозумілішими і їх легше підтримувати, якщо кожна функція відповідає за лише одну ідею. Інша проблема полягає в тому, що ми не обробляємо помилки так добре, як могли б. Програма все ще невелика, тому ці недоліки не становлять великої проблеми, але зі зростанням програми стане важчим їх акуратно виправити. Є гарна порада - починати рефакторити код на ранній стадії розробки програми, бо значно легше рефакторити невеликі фрагменти коду. Цим ми й займемося.

Рефакторинг для Покращення Модульності та Обробки Помилки

Для покращення програми ми розв'яжемо чотири проблеми, пов'язані зі структурою програми та тим, як вона обробляє потенційні помилки. По-перше, наша функція `main` тепер виконує два завдання: розбирає параметри та читає файли. Зі зростанням нашої програми кількість окремих завдань, які обробляє функція `main`, збільшуватиметься. Зі збільшенням відповідальності функції її стає складніше розуміти, важче тестувати, важче змінювати, не порушуючи інших її частин. Найкраще розділити функціональність, щоб кожна функція відповідала за одне завдання.

Це питання також пов'язане з другою проблемою: у той час, як змінні `query` та `file_path` є конфігураційними змінними нашої програми, змінні на кшталт `contents` використовуються для реалізації логіки програм. Що довшим ставатиме `main`, то більше змінних треба буде додати в область видимості; що більше змінних в області видимості, тим складніше буде відстежувати призначення кожної з них. Найкраще згрупувати конфігураційні змінні в одну структуру, щоб унаочнити їхнє призначення.

Третя проблема полягає в тому, що ми використали `expect`, щоб вивести повідомлення про помилку, коли не вдається прочитати файл, але саме повідомлення лише каже `Should have been able to read the file`. Читання файлу може бути невдалим через багато причин: скажімо, такого файлу може не існувати, або у нас може не бути прав відкривати його. Поки що, незалежно від ситуації, ми виводимо те саме повідомлення про помилку для будь-якої причини, що не дає користувачеві жодної інформації!

По-четверте, ми використовуємо `expect` знову і знову для обробки різних помилок, і якщо користувач запустить програму, не вказавши потрібні параметри, то побачить лише повідомлення Rust про помилку `index out of bounds`, що не дуже чітко описує проблему. Найкраще буде, якщо код обробки помилок буде в одному місці, щоб той, хто підтримуватиме код у майбутньому, мав зазирнути лише в одне місце в коді, якщо треба буде змінити логіку обробки помилок. Те, що код обробки помилок знаходиться в одному місці, також гарантує, що ми друкуємо повідомлення, зрозумілі для наших кінцевих користувачів.

Щоб виправити ці чотири проблеми, зробимо рефакторинг нашого проєкту.

Розмежування Відповідальностей для Двійкових Проєктів

Організаційна проблема поділу відповідальності за різні завдання у функції `main` є спільною для багатьох двійкових проєктів. У результаті спільнота Rust розробила рекомендації для поділу окремих інтересів у двійковій програмі, коли функція `main` починає ставати великою. Процес складається з наступних кроків:

- Поділіть свою програму на *main.rs* та *lib.rs* і перенесіть логіку програми до *lib.rs*.
- Поки логіка для аналізу командного рядка невелика, вона може залишатися в *main.rs*.
- Коли обробка логіки командного рядка починає ускладнюватись, витягніть її з *main.rs* і перемістіть до *lib.rs*.

Відповідальність коду, що залишиться в функції `main` після цього, має бути обмеженою до такого:

- Виклик логіки аналізу командного рядка і передача їй значень аргументів
- Налаштування решти конфігурації
- Виклик функції `run` із *lib.rs*
- Обробка помилок, якщо `run` поверне помилку

Цей шаблон стосується поділу інтересів: *main.rs* обробляє запуск програми, а *lib.rs* обробляє всю логіку основного завдання. Оскільки функцію `main` неможливо тестувати напряму, ця структура дозволяє вам тестувати усю логіку вашої програми, перенісши її до функцій у *lib.rs*. Код, що залишився в *main.rs* буде досить маленьким, щоб перевірити його правильність, прочитавши його. Переробімо нашу програму відповідно до цього процесу.

Відокремлення Парсера Аргументів

Ми перенесемо функціональність для аналізу аргументів у функцію, котру буде викликати `main`, щоб підготувати переміщення логіки розбору командного рядка до *src/lib.rs*. Блок коду 12-5 показує початок нової функції `main`, яка викликає нову функцію `parse_config`, котру ми скоро визначимо в *src/main.rs*.

Файл: *src/main.rs*

```
fn main() {
    let args: Vec<String> = env::args().collect();

    let (query, file_path) = parse_config(&args);

    // --snip--
}

fn parse_config(args: &[String]) -> (&str, &str) {
    let query = &args[1];
    let file_path = &args[2];

    (query, file_path)
}
```

Блок коду 12-5: Вилучення функції `parse_config` з `main`

Ми все ще збираємо аргументи командного рядка до вектора, але замість присвоювати значення аргументу з індексом 1 змінній `query`, а значення аргументу з індексом 2 змінній `file_path` у функції `main`, ми передаємо весь вектор до функції `parse_config`. Функція `parse_config` містить логіку, що визначає, який аргумент потрапляє до якої змінної і передає значення на назад до `main`. Ми все ще створюємо змінні `query` та `file_path` у `main`, але `main` більше не відповідає за визначення, як співвідносяться аргументи командного рядка та змінні.

Це перероблення може виглядати надмірним для нашої програми, але ми робимо рефакторизацію невеликими поступовими кроками. Після внесення цієї зміни знову запустіть програму, щоб перевірити, що аналіз аргументів все ще працює. Дуже добра ідея - часто перевіряти ваш прогрес, щоб легше було визначити причину проблем, коли вони з'являться.

Групування Значень Конфігурації

Ми можемо зробити ще один невеликий крок, щоб поліпшити функцію `parse_config`. На цей момент вона повертає кортеж, а потім ми відразу ж розбираємо цей кортеж на окремі частини. Це ознака того, що, можливо, ми ще не досягли правильної абстракції.

Інший показник, що вказує на місце для покращення - це частина `config` функції `parse_config`, яка має на увазі, що два значення, що ми повертаємо, пов'язані і є частинами одного конфігураційного значення. Наразі ми передаємо це в структурі даних простим групуванням двох значень у кортеж, що не дуже виразно; натомість покладімо два значення в одну структуру і дамо кожному з полів змістовну назву. Таким чином ми полегшимо тим, хто підтримуватиме цей код у майбутньому, розуміння, як різні значення стосуються одне одного і яке їхнє призначення.

Блок коду 12-6 показує покращення до функції `parse_config`.

Файл: `src/main.rs`

```
fn main() {
    let args: Vec<String> = env::args().collect();

    let config = parse_config(&args);

    println!("Searching for {}", config.query);
    println!("In file {}", config.file_path);

    let contents = fs::read_to_string(config.file_path)
        .expect("Should have been able to read the file");

    // --snip--
}

struct Config {
    query: String,
    file_path: String,
}

fn parse_config(args: &[String]) -> Config {
    let query = args[1].clone();
    let file_path = args[2].clone();

    Config { query, file_path }
}
```

Блок коду 12-6: Рефакторизація функції `parse_config`, що тепер повертає екземпляр структури `Config`

Ми додали структуру, що зветься `Config`, у якій визначили поля, що зветься `query` та `file_path`. Сигнатура `parse_config` тепер показує, що вона повертає значення типу `Config`. У тілі `parse_config`, де раніше ми повертали стрічкові слайси, які посилалися на значення `String` у `args`, тепер ми задаємо значення `String`, якими володіє `Config`. Змінна `args` у `main` є власником значень аргументів і лише дозволяє функції `parse_config` позичити їх, тобто ми б порушили правила позичання Rust якби `Config` пробував взяти володіння значеннями з `args`.

Є багато способів, як ми могли б керувати даними `String`; найпростіший, хоча і дещо неефективний спосіб - викликати метод `clone` для значень. Це зробить повну копію даних для надання володіння екземпляра `Config`, що потребує більше часу і пам'яті, ніж зберігання посилання на дані стрічки. Однак клонування даних також робить наш код вкрай прямолінійним, бо нам не треба керувати часами існування посилань; за цих обставин, віддати трохи продуктивності задля спрощення є гідним компромісом.

Компроміси Використання `clone`

Існує тенденція, якої дотримується багато растацеанців, уникати використання `clone` для виправлення проблем із володінням через його ціну часу виконання. У [Розділі 13](#) ви дізнаєтеся, як застосовувати ефективніші методи для ситуацій на кшталт цієї. Та поки що цілком прийнятно скопіювати кілька стрічок для продовження розробки, бо ці копії робляться лише один раз і шлях до файлу та стрічка запиту дуже маленькі. Краще мати дещо неефективну робочу програму, ніж намагатися з першого разу переоптимізувати код. Як ви ставатимете досвідченішими з Rust, ставатиме легше починати з найефективнішого рішення, та поки що цілком прийнятно викликати `clone`.

Ми змінили `main`, і тепер він розміщує екземпляр `Config`, повернутий `parse_config`, у змінну з назвою `config`, і змінили код, що раніше розділяв змінні `query` та `file_path`, щоб він натомість використовував поля у структурі `Config`.

Тепер наш код ясніше передає, що `query` та `file_path` пов'язані, і що їхнє призначення - конфігурувати роботу програми. Будь-який код, що використовує ці значення, знає, що їх треба шукати у екземплярі `config` у полях з назвами, що відповідають їхньому призначенню.

Створення Конструктора `Config`

Ми вже перенесли логіку, що відповідає за обробку аргументів командного рядка, з `main` і помістили її у функції `parse_config`. Це допомогло нам побачити, що змінні `query` та `file_path` пов'язані і цей зв'язок має бути показаним у коді. Потім ми додали структуру `Config`, щоб назвати об'єднані за призначенням змінні `query` та `file_path` і щоб можна було повертати імена значень як поля структури з функції `parse_config`.

Тож тепер, оскільки призначення функції `parse_config` - створити екземпляр `Config`, ми можемо змінити `parse_config` зі звичайної функції на функцію, що зветься `new`, асоційовану зі структурою `Config`. Ця зміна зробить код більш ідіоматичним. Ми можемо створювати екземпляри типів зі стандартної бібліотеки, такі як `String`, викликом `String::new`. Подібним чином, змінивши `parse_config` на функцію `new`, асоційовану з `Config`, ми зможемо створювати екземпляри `Config` викликом `Config::new`. Блок коду 12-7 показує, які зміни треба зробити.

Файл: `src/main.rs`

```
fn main() {
    let args: Vec<String> = env::args().collect();

    let config = Config::new(&args);

    // --snip--
}

// --snip--

impl Config {
    fn new(args: &[String]) -> Config {
        let query = args[1].clone();
        let file_path = args[2].clone();

        Config { query, file_path }
    }
}
```

Listing 12-7: Зміна `parse_config` на `Config::new`

Ми замінили у `main` виклик `parse_config` на виклик `Config::new`. Ми змінили назву `parse_config` на `new` і перенесли її в блок `impl`, асоціювавши функцію `new` з `Config`. Спробуйте скомпілювати цей код ще раз, щоб переконатися, що він працює.

Виправлення Обробки Помилки

Тепер ми попрацюємо над виправленням обробки помилок. Згадайте, що спроби отримати доступ до значень у векторі `args` за індексами 1 чи 2 призведе до паніки програми, якщо у векторі менш ніж три елементи. Спробуйте запустити програму без будь-яких аргументів; це виглядатиме так:

```
$ cargo run
Compiling minigrep v0.1.0 (file:///projects/minigrep)
Finished dev [unoptimized + debuginfo] target(s) in 0.0s
Running `target/debug/minigrep`
thread 'main' panicked at 'index out of bounds: the len is 1 but the index is 1',
src/main.rs:27:21
note: run with `RUST_BACKTRACE=1` environment variable to display a backtrace
```

Рядок `index out of bounds: the len is 1 but the index is 1` - це повідомлення про помилку, призначене для програмістів. Воно не допоможе кінцевим користувачам зрозуміти, що вони мають робити. Полагодьмо це.

Поліпшення Повідомлення про Помилку

У Блоці коду 12-8 ми додаємо перевірку у функцію `new`, що підтверджує, що слайс достатньо довгий, перед тим як звертатися до індексів 1 та 2. Якщо слайс недостатньо довгий, програма панікує і показує краще повідомлення про помилку.

Файл: `src/main.rs`

```
// --snip--
fn new(args: &[String]) -> Config {
    if args.len() < 3 {
        panic!("not enough arguments");
    }
    // --snip--
```

Блок коду 12-8: Додавання перевірки на число аргументів

Цей код подібний до функції `Guess::new`, яку ми написали у Блоці коду 9-13, де ми викликали `panic!`, коли аргумент `value` був поза діапазоном припустимих значень. Тут, замість перевірки діапазону значень, ми перевіряємо, що довжина `args` є принаймні 3, і решта функції може працювати з припущенням, що ця умова виконується. Якщо `args` має менш ніж три елементи, ця умова буде істинною, і ми викличемо макрос `panic!`, щоб негайно завершити програму.

Після додавання цих кількох рядків коду до `new` знову запусимо програму без аргументів, щоб побачити, як помилка виглядатиме тепер:

```
$ cargo run
Compiling minigrep v0.1.0 (file:///projects/minigrep)
Finished dev [unoptimized + debuginfo] target(s) in 0.0s
Running `target/debug/minigrep`
thread 'main' panicked at 'not enough arguments', src/main.rs:26:13
note: run with `RUST_BACKTRACE=1` environment variable to display a backtrace
```

Це вже краще: тепер ми маємо зрозуміле повідомлення про помилку. Однак, ми також маємо побічну інформацію, яку не хочемо надавати нашим користувачам. Мабуть, техніка, яку ми використовували в Блоці коду 9-13, не найліпше підходить сюди: виклик `panic!` більш доречний для проблеми з програмуванням, ніж до проблеми з використанням, [що обговорювалося в Розділі 9](#). Натомість ми використаємо іншу техніку, про яку ви дізналися з Розділу 9 - [повернення `Result`](#), що позначає успіх чи помилку.

Повернення `Result` Замість Виклику `panic!`

Ми можемо натомість повернути значення `Result`, що міститиме екземпляр `Config` при успіху і описуватиме проблему у випадку помилки. Ми також збираємося змінити назву функції з `new` на `build`, бо багато програмістів очікують, що функції `new` ніколи не зазнають невдачі. Коли `Config::build` передає повідомлення до `main`, ми можемо використати тип `Result`, щоб сигналізувати про проблему. Потім ми можемо змінити `main`, щоб перетворити варіант `Err` на більш практичне повідомлення для наших користувачів без зайвого тексту про `thread 'main'` і `RUST_BACKTRACE`, як робить виклик `panic!`.

Блок коду 12-9 показує зміни до функції, що тепер зветься `Config::build`, які ми маємо зробити, щоб значення, що повертається з неї, було типу `Result`, і відповідне тіло функції. Зверніть увагу, що цей код не скомпілюється, доки ми не змінимо також і `main`, що ми робимо в наступному блоці коду.

Файл: `src/main.rs`

```
impl Config {
    fn build(args: &[String]) -> Result<Config, &'static str> {
        if args.len() < 3 {
            return Err("not enough arguments");
        }

        let query = args[1].clone();
        let file_path = args[2].clone();

        Ok(Config { query, file_path })
    }
}
```

Блок коду 12-9: Повертання `Result` з `Config::build`

Наша функція `build` повертає `Result` з екземпляром `Config` у разі успіху і `&'static str` у разі помилки. Значення наших помилок завжди будуть стрічковими літералами з часом існування `'static`.

Ми зробили дві зміни у тілі функції: замість виклику `panic!`, коли користувач не надав достатньо аргументів, ми тепер повертаємо значення `Err`, і ми обгорнули значення `Config`, що ми повертаємо, у `Ok`. Ці зміни узгоджують функцію з новою сигнатурою типу.

Повертання значення `Err` з `Config::build` дозволяє функції `main` обробити значення `Result`, повернуте з функції `build`, і вийти з процесу чистіше у випадку помилки.

Виклик `Config::build` та Обробка Помилки

Щоб обробити випадок з помилкою і вивести дружнє для користувача повідомлення, нам треба змінити `main`, щоб обробити `Result`, повернений `Config::build`, як показано у Блоці коду 12-10. Ми також візьмемо відповідальність за вихід з інструменту командного рядка з ненульовим кодом помилки з `panic!` і реалізуємо його самостійно. Ненульовий статус на виході - це угода, щоб повідомити процесу, який викликав нашу програму, що програма завершилася з помилкою.

Файл: `src/main.rs`

```
use std::process;

fn main() {
    let args: Vec<String> = env::args().collect();

    let config = Config::build(&args).unwrap_or_else(|err| {
        println!("Problem parsing arguments: {err}");
        process::exit(1);
    });

    // --snip--
```

Блок коду 12-10: Вихід з кодом помилки, якщо збірка `Config` була невдалою

У цьому блоці коду ми скористалися методом, про який ще детально не розповідали - `unwrap_or_else`, що визначено на `Result<T, E>` у стандартній бібліотеці. `unwrap_or_else` дозволяє визначати власну обробку помилок, без `panic!`. Якщо `Result` є значенням `Ok`, цей метод робить те саме, що й `unwrap`: повертає внутрішнє значення, загорнуте в `Ok`. Але якщо значення є `Err`, цей метод викликає код у *замиканні*, тобто анонімній функції, що ми визначаємо і передаємо аргументом до `unwrap_or_else`. Про замикання детальніше піде у [Розділі 13](#). Поки що вам лише слід знати, що `unwrap_or_else` передасть внутрішнє значення `Err`, тобто у нашому випадку статичну стрічку `"not enough arguments"`, що ми додали в Блоці коду 12-9, нашому замиканню, як аргумент `err`, що визначається між вертикальними лініями. Код у замиканні зможе під час виконання використати значення `err`.

Ми додали новий рядок `use`, щоб ввести `process` зі стандартної бібліотеки до області видимості. Код у замиканні, що буде виконано у випадку помилки, складається лише з двох рядків: ми виводимо значення `err` і потім викликаємо `process::exit`. Функція `process::exit` негайно зупиняє програму і повертає передане число як код статусу виходу. Це схоже на обробку помилок за допомогою `panic!`, як ми робили в Блоці коду 12-8, але ми більше не отримуємо зайвий вивід. Спробуймо:

```
$ cargo run
Compiling minigrep v0.1.0 (file:///projects/minigrep)
Finished dev [unoptimized + debuginfo] target(s) in 0.48s
Running `target/debug/minigrep`
Problem parsing arguments: not enough arguments
```

Чудово! Це повідомлення набагато дружніше до наших користувачів.

Відокремлення Логіки від `main`

Тепер, коли ми закінчили рефакторизацію аналізу конфігурації, повернімося до логіки програми. Як ми казали в "Розділення зон інтересів у двійкових проєктах", ми виділимо функцію, що зветься `run`, що міститиме всю логіку, наразі розміщену у функції `main`, яка не бере участі у встановленні конфігурації чи обробці помилок. Коли ми закінчимо, `main` стане виразним і легким для перевірки на помилки простим переглядом, і ми зможемо написати тести для решти логіки програми.

Блок коду 12-11 показує виокремлену функцію `run`. Поки що, ми робимо маленькі, поступові покращення при виділенні функції. Ми все ще визначаємо цю функцію у `src/main.rs`.

Файл: `src/main.rs`

```
fn main() {
    // --snip--

    println!("Searching for {}", config.query);
    println!("In file {}", config.file_path);

    run(config);
}

fn run(config: Config) {
    let contents = fs::read_to_string(config.file_path)
        .expect("Should have been able to read the file");

    println!("With text:\n{contents}");
}

// --snip--
```

Блок коду 12-11: Виділення функції `run`, що містить решту логіки програми

Функція `run` тепер містить решту логіки з `main`, починаючи з читання файлу. Функція `run` приймає аргументом екземпляр `Config`.

Повернення помилок з Функції `run`

Для решти логіки програми, виділеної в функцію `run`, ми можемо покращити обробку помилок, як ми зробили з `Config::build` у Блоці коду 12-9. Замість дозволяти програмі панікувати викликом `expect`, функція `run` повертатиме `Result<T, E>`, коли щось піде не так. Це дозволить нам об'єднати логіку обробки помилок у `main` у дружній для користувача спосіб. Блок коду 12-12 показує зміни, які нам треба зробити в сигнатурі і тілі `run`.

Файл: `src/main.rs`

```
use std::error::Error;

// --snip--

fn run(config: Config) -> Result<(), Box<dyn Error>> {
    let contents = fs::read_to_string(config.file_path)?;

    println!("With text:\n{contents}");

    Ok(())
}
```

Блок коду 12-12: Зміна функції `run`, що повертає `Result`

Ми зробили тут три суттєві зміни. По-перше, ми змінили тип, що повертає функція `run`, на `Result<(), Box<dyn Error>>`. Ця функція раніше повертала одиничний тип, `()`, і ми залишаємо це значення у випадку `Ok`.

Для типу помилок, ми використовуємо *трейтовий об'єкт* `Box<dyn Error>` (і ми внесли `std::error::Error` до області видимості за допомогою інструкції `use` на початку). Ми розкажемо про трейтові об'єкти у [Розділі 17](#). Поки що, вам достатньо знати, що `Box<dyn Error>` означає, що функція поверне тип, що реалізує трейт `Error`, але ми не маємо зазначати який це буде конкретний тип значення. Це надає нам гнучкості, щоб повертати значення, які можуть бути різних типів у випадках різних помилок. Ключове слово `dyn` - це скорочення для "динамічний" ("dynamic").

По-друге, ми прибрали виклик `expect`, замінивши його натомість оператором `?`, як ми й говорили у [Розділі 9](#). Замість виклику `panic!` при помилці, `?` поверне значення помилки з поточної функції тому, хто її викликав, для обробки.

По-третє, функція `run` тепер повертає значення `Ok` у випадку успіху. Ми проголосили у сигнатурі, що тип успіху функції `run` - `()`, що означає, що нам потрібно обгорнути значення одиничного типу у значення `Ok`. Цей запис `Ok(())` може спершу видаватися трохи дивним, але використання `()` подібним чином є ідіоматичним способом

позначити, що ми викликаємо `run` лише задля його побічних ефектів; він не повертає потрібного значення.

Коли ви запускаєте цей код, він скомпілюється, але покаже попередження:

```
$ cargo run the poem.txt
Compiling minigrep v0.1.0 (file:///projects/minigrep)
warning: unused `Result` that must be used
--> src/main.rs:19:5
19 |         run(config);
   |         ^^^^^^^^^^^
= note: `[warn(unused_must_use)]` on by default
= note: this `Result` may be an `Err` variant, which should be handled

warning: `minigrep` (bin "minigrep") generated 1 warning
Finished dev [unoptimized + debuginfo] target(s) in 0.71s
Running `target/debug/minigrep the poem.txt`
Searching for the
In file poem.txt
With text:
I'm nobody! Who are you?
Are you nobody, too?
Then there's a pair of us - don't tell!
They'd banish us, you know.

How dreary to be somebody!
How public, like a frog
To tell your name the livelong day
To an admiring bog!
```

Rust каже нам, що наш код проігнорував значення `Result` і що це значення `Result` може показувати, що сталася помилка. Але ми не перевіряємо, чи не було помилки, і компілятор нагадує нам, що ми, мабуть, хотіли б додати сюди код для обробки помилок! Виправмо одразу цю проблему.

Обробка Помилки, Повернутих з `run` в `main`

Ми перевірятимемо на помилки і оброблятимемо їх за допомогою техніки, подібної до тої, якою ми скористалися з `Config::build`, з невеликою відмінністю:

Файл: `src/main.rs`

```
fn main() {
    // --snip--

    println!("Searching for {}", config.query);
    println!("In file {}", config.file_path);

    if let Err(e) = run(config) {
        println!("Application error: {e}");
        process::exit(1);
    }
}
```

Ми використовуємо `if let` замість `unwrap_or_else` для перевірки, чи `run` повертає значення `Err` і викликаємо в цьому випадку `process::exit(1)`. Функція `run` не повертає значення, яке б ми хотіли отримати за допомогою `unwrap`, на відміну від `Config::build`, що повертає екземпляр `Config`. Оскільки `run` у випадку успіху повертає `()`, нас турбує лише виявлення помилки, тож нам не потрібен `unwrap_or_else` для отримання відобутого значення, яке може бути лише `()`.

Тіла `if let` та функції `unwrap_or_else` однакові в обох випадках: ми виводимо помилку і виходимо.

Виділення Коду у Бібліотечний Крейт

Наш проєкт `minigrep` поки що має непоганий вигляд! Тепер ми поділимо файл `src/main.rs` і перенесемо частину коду у файл `src/lib.rs`. Таким чином, ми зможемо тестувати код, залишивши файлу `src/main.rs` менше відповідальності.

Перенесімо весь код, крім функції `main`, з `src/main.rs` до `src/lib.rs`:

- Визначення функції `run`
- Відповідні інструкції `use`
- Визначення `Config`
- Визначення функції `Config::build`

Вміст `src/lib.rs` має містити сигнатури, показані в Блоці коду 12-13 (ми опустили тіла функцій для стислості). Зверніть увагу, що цей код не скомпілюється, поки ми не змінімо `src/main.rs` у Блоці коду 12-14.

Файл: `src/lib.rs`

```

use std::error::Error;
use std::fs;

pub struct Config {
    pub query: String,
    pub file_path: String,
}

impl Config {
    pub fn build(args: &[String]) -> Result<Config, &'static str> {
        // --snip--
    }
}

pub fn run(config: Config) -> Result<(), Box<dyn Error>> {
    // --snip--
}

```

Блок коду 12-13: Перенесення `Config` і `run` до `src/lib.rs`

Ми дещо вільно використали ключове слово `pub` для `Config`, його полів і його методу `build`, а також для функції `run`. Тепер ми маємо бібліотечний крейт, що має публічний API, який ми можемо тестувати!

Now we need to bring the code we moved to `src/lib.rs` into the scope of the binary crate in `src/main.rs`, as shown in Listing 12-14.

Файл: `src/main.rs`

```

use std::env;
use std::process;

use minigrep::Config;

fn main() {
    // --snip--
    if let Err(e) = minigrep::run(config) {
        // --snip--
    }
}

```

Блок коду 12-14: Використання бібліотечного крейту `minigrep` у `src/main.rs`

Ми додали рядок `use minigrep::Config`, щоб внести тип `Config` з бібліотечного крейту до області видимості двійкового крейту, і додали перед функцією `run` назву нашого крейту. Тепер уся функціональність має бути з'єднана і мусить працювати. Запустіть програму за допомогою `cargo run` і переконайтеся, що все працює правильно.

Хух! Добряче попрацювали, але налаштували себе на успіх у майбутньому. Тепер буде значно легше обробляти помилки, і ми зробили код більш модульним. Майже вся наша робота з цього моменту буде виконуватися в *src/lib.rs*.

Скористаймося з цієї новоствореної модульності, зробивши дещо, що було б складним зі старим кодом, але легко з новим: напишемо кілька тестів!

Розробка Функціонала Бібліотеки із Test-Driven Development

Тепер, коли ми перенесли логіку в *src/lib.rs* та залишили збір аргументів та обробку помилок в *src/main.rs*, стало набагато простіше писати тести для основного функціонала нашого коду. Ми можемо викликати функції напряду із різноманітними аргументами та перевіряти повернуті значення без потреби виклику нашого двійкового файлу із командного рядка.

У цій секції ми додамо пошукову логіку до програми `minigrep`, використовуючи стиль розробки через тестування (TDD) із наступними кроками:

1. Напишіть тест, який дає збій і запустить його, щоб переконатися, що він це робить через очікувану причину.
2. Напишіть або змініть мінімум коду, щоб новий тест пройшов.
3. Відрефакторіть щойно доданий або змінений код та впевніться, що тести продовжують проходити.
4. Повторіть з першого кроку!

Хоча це лише один з багатьох способів написання програмного забезпечення, TDD може допомагати надавати потрібного напрямку оформленню коду. Створення тесту перед тим, як написати код, який забезпечить проходження тесту, допомагає підтримувати високий рівень покриття тестуванням протягом всього процесу розробки.

Ми протестуємо імплементацію функціоналу який буде робити пошуковий запит стрічки у вмісті файлу та створювати список рядків, які відповідають запиту. Ми додамо цей функціонал у функцію під назвою `search`.

Написання Провального Тесту

Видалімо інструкції `println!` які ми використовували для перевірки поведінки програми з *src/lib.rs* та *src/main.rs*, бо нам вони більше не потрібні. Потім додамо в *src/lib.rs* модуль `tests` із тестовою функцією, як ми зробили в [Розділі 11](#). Тестова функція визначає бажану поведінку функції `search`: вона отримає запит та текст для пошуку, і вона буде повертати лише рядки з тексту, які містять запит. Блок коду 12-15 показує цей тест, який ще не компілюється.

Файл: *src/lib.rs*

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

    #[test]
    fn one_result() {
        let query = "duct";
        let contents = "\
Rust:
safe, fast, productive.
Pick three.";

        assert_eq!(vec!["safe, fast, productive."], search(query, contents));
    }
}
```

Блок коду 12-15: Створення невдалого тесту для функції `search`, яку ми хотіли б мати

Цей тест шукає рядок `"duct"`. Текст, в якому ми робимо пошук, це три рядки, лише один з яких містить `"duct"` (Зауважте, що зворотний слеш після першої подвійної лапки каже Rust не розміщувати символ нового рядку на початку цієї стрічки). Ми стверджуємо, що значення, повернене з функції `search` містить тільки рядки, які ми очікуємо.

Ми ще не готові запустити цей тест та подивитися, як він дає збій, бо тест навіть не компілюється: функція `search` ще не існує! Згідно з принципами TDD, ми додамо лише мінімум коду, щоб тест почав компілюватися та виконуватися, додав визначення функції `search`, яке завжди повертає порожній вектор, як показано в Блоці коду 12-16. Тоді тест повинен скомпілюватися та провалитися, бо порожній вектор не зіставляється з вектором, який містить рядок `"safe, fast, productive."`

Файл: `src/lib.rs`

```
pub fn search<'a>(query: &str, contents: &'a str) -> Vec<&'a str> {
    vec![]
}
```

Блок коду 12-16: Визначення функції `search`, якого досить для проходження тесту

Зауважте, що нам потрібно явно визначити час існування `'a` в сигнатурі `search` та використати цей час існування з аргументом `contents` та поверненим значенням. Згадаємо [Розділ 10](#) де час існування параметрів вказував, який час існування аргументу пов'язаний з поверненим значенням. У цьому випадку, ми вказуємо, що повернутий вектор має містити слайси стрічки, які посилаються на слайси аргументу `contents` (замість аргументу `query`).

Інакше кажучи, ми повідомляємо Rust, що дані, отримані `search` функцією будуть існувати допоки вони передаються в `search` функцію аргументом `contents`. Це важливо! Дані, на які посилається *слайс* мають бути валідними, щоб посилання було валідним; якщо компілятор вважає, що ми робимо строкові слайси `query` замість `contents`, він зробить перевірку безпеки некоректно.

Якщо ми забудемо анотації часу існування і спробуємо скомпілювати цю функцію, ми отримаємо цю помилку:

```
$ cargo build
  Compiling minigrep v0.1.0 (file:///projects/minigrep)
error[E0106]: missing lifetime specifier
  --> src/lib.rs:28:51
   |
28 | pub fn search(query: &str, contents: &str) -> Vec<&str> {
   |               ----          ----                ^ expected named lifetime
parameter
   |
   = help: this function's return type contains a borrowed value, but the
signature does not say whether it is borrowed from `query` or `contents`
help: consider introducing a named lifetime parameter
   |
28 | pub fn search<'a>(query: &'a str, contents: &'a str) -> Vec<&'a str> {
   |               ++++      ++                ++                ++

For more information about this error, try `rustc --explain E0106`.
error: could not compile `minigrep` due to previous error
```

Rust не має можливості дізнатися, який з двох аргументів нам потрібен, тому ми маємо явно це вказати. Оскільки `contents` це аргумент, який містить увесь наш текст і ми хочемо повертати відповідні частини цього тексту, ми розуміємо, що `contents` це аргумент який має бути пов'язаний з поверненням значенням використовуючи синтаксис часу існування.

Інші мови програмування не вимагають від вас пов'язувати аргументи із поверненням значенням в сигнатурі функції, але ця практика з часом стане легшою. Ви можете захотіти порівняти цей приклад із “Validating References with Lifetimes” Розділу 10.

Тепер запусимо тест:

```

$ cargo test
  Compiling minigrep v0.1.0 (file:///projects/minigrep)
  Finished test [unoptimized + debuginfo] target(s) in 0.97s
  Running unittests src/lib.rs (target/debug/deps/minigrep-9cd200e5fac0fc94)

running 1 test
test tests::one_result ... FAILED

failures:

---- tests::one_result stdout ----
thread 'main' panicked at 'assertion failed: `(left == right)`
  left: `["safe, fast, productive."`,
 right: `[]`', src/lib.rs:44:9
note: run with `RUST_BACKTRACE=1` environment variable to display a backtrace

failures:
  tests::one_result

test result: FAILED. 0 passed; 1 failed; 0 ignored; 0 measured; 0 filtered out;
finished in 0.00s

error: test failed, to rerun pass `--lib`

```

Чудово, тест провалюється, як ми й очікували. Нумо зробимо тест, який пройде!

Написання Коду для Проходження Тесту

Наразі наш тест провалюється, бо він завжди повертає порожній вектор. Щоб виправити це та імплементувати `search`, наша програма має виконати такі дії:

- Ітерувати через кожний рядок вмісту.
- Перевірити, чи містить цей рядок нашу стрічку запиту.
- Якщо так, то додати його до списку значень який ми повертаємо.
- Якщо ні, то нічого не робити.
- Повернути отриманий список рядків, які збігаються.

Пройдімо кожен крок, починаючи з ітерації по рядках.

Ітерація над Рядками із Методом `lines`

Rust має корисний метод для керування ітерацією по стрічці рядок за рядком який зручно названий `lines`, який працює як показано в Блоці Коду 12-17. Зверніть увагу, це ще не буде компілюватися.

Файл: src/lib.rs

```
pub fn search<'a>(query: &str, contents: &'a str) -> Vec<&'a str> {  
    for line in contents.lines() {  
        // do something with line  
    }  
}
```

Блок коду 12-17: Ітерація по кожному рядку в `contents`

Метод `lines` повертає ітератор. Ми поговоримо про ітератори більш детально в [Розділі 13](#), але пригадайте, що ви бачили цей спосіб використання ітератора в [Блоці Коду 3-5](#), де ми використовували цикл `for` з ітератором для виконання деякого коду на кожному елементі колекції.

Пошук Запиту в Кожному Рядку

Далі, ми перевіримо, чи містить поточний рядок стрічку запиту. На щастя, стрічки мають корисний метод названий `contains`, який робить це для нас! Додайте виклик методу `contains` в функцію `search`, як показано в Блоці Коду 12-18. Зауважте, що це ще не буде компілюватися.

Файл: src/lib.rs

```
pub fn search<'a>(query: &str, contents: &'a str) -> Vec<&'a str> {  
    for line in contents.lines() {  
        if line.contains(query) {  
            // do something with line  
        }  
    }  
}
```

Блок коду 12-18: Додавання перевірки наявності `query` в рядку

Ми зараз створюємо функціонал. Щоб він компілювався, нам потрібно повертати значення з вмісту функції, як ми вказали в її сигнатурі.

Зберігання Відповідних Рядків

Щоб завершити цю функцію, нам потрібен спосіб зберігання зіставлених рядків, які ми хочемо повертати. Для цього, ми можемо створити мутабельний вектор перед циклом `for` та викликати метод `push`, щоб зберегти `line` в векторі. Після циклу `for`, ми повертаємо вектор, як показано в Блоці Коду 12-19.

Файл: src/lib.rs

```
pub fn search<'a>(query: &'a str, contents: &'a str) -> Vec<&'a str> {
    let mut results = Vec::new();

    for line in contents.lines() {
        if line.contains(query) {
            results.push(line);
        }
    }

    results
}
```

Listing 12-19: Storing the lines that match so we can return them

Тепер функція `search` повинна повертати тільки рядки, що містять `query`, і наш тест повинен пройти. Запустимо тест:

```
$ cargo test
Compiling minigrep v0.1.0 (file:///projects/minigrep)
Finished test [unoptimized + debuginfo] target(s) in 1.22s
Running unittests src/lib.rs (target/debug/deps/minigrep-9cd200e5fac0fc94)

running 1 test
test tests::one_result ... ok

test result: ok. 1 passed; 0 failed; 0 ignored; 0 measured; 0 filtered out;
finished in 0.00s

Running unittests src/main.rs (target/debug/deps/minigrep-9cd200e5fac0fc94)

running 0 tests

test result: ok. 0 passed; 0 failed; 0 ignored; 0 measured; 0 filtered out;
finished in 0.00s

Doc-tests minigrep

running 0 tests

test result: ok. 0 passed; 0 failed; 0 ignored; 0 measured; 0 filtered out;
finished in 0.00s
```

Наш тест пройшов, тому ми знаємо, що він працює!

На цьому етапі ми могли б розглянути можливості рефакторингу імплементації функції пошуку, зберігаючи проходження тестів та зберігаючи той самий функціонал. Код функції пошуку не настільки й поганий, але він не використовує переваги деяких корисних

особливостей ітераторів. Ми повернемося до цього прикладу в [Розділі 13](#), де ми дослідимо ітератори детальніше та розглянемо, як ми можемо їх вдосконалити.

Використання Функції `search` в Функції `run`

Тепер, коли функція `search` працює та протестована, нам потрібно викликати `search` з нашої функції `run`. Нам потрібно передати значення `config.query` та `contents` яке `run` читає з файлу в функцію `search`. Потім `run` виведе в консолі кожен рядок повернений з `search`:

Файл: `src/lib.rs`

```
pub fn run(config: Config) -> Result<(), Box<dyn Error>> {
    let contents = fs::read_to_string(config.file_path)?;

    for line in search(&config.query, &contents) {
        println!("{}", line);
    }

    Ok(())
}
```

Ми досі використовуємо цикл `for` для повернення кожного рядка із `search` та його виводу в консолі.

Тепер вся програма має працювати! Нумо спробуємо, спочатку зі словом, яке має повертати річно один рядок із поеми Емілі Дікінсон, "frog":

```
$ cargo run -- frog poem.txt
Compiling minigrep v0.1.0 (file:///projects/minigrep)
Finished dev [unoptimized + debuginfo] target(s) in 0.38s
Running `target/debug/minigrep frog poem.txt`
How public, like a frog
```

Круто! Спробуємо слово, яке зіставлятиметься з кількома рядками, наприклад "body":

```
$ cargo run -- body poem.txt
Compiling minigrep v0.1.0 (file:///projects/minigrep)
Finished dev [unoptimized + debuginfo] target(s) in 0.0s
Running `target/debug/minigrep body poem.txt`
I'm nobody! Who are you?
Are you nobody, too?
How dreary to be somebody!
```

І, нарешті, переконаймося, що ми не отримуємо жодних рядків, коли ми шукаємо слово, якого немає ніде в поемі, наприклад, "monomorphization":

```
$ cargo run -- monomorphization poem.txt
Compiling minigrep v0.1.0 (file:///projects/minigrep)
Finished dev [unoptimized + debuginfo] target(s) in 0.0s
Running `target/debug/minigrep monomorphization poem.txt`
```

Блискуче! Ми побудували нашу власну мініверсію класичного інструменту та багато дізналися про структурування застосунків. Ми також дізналися дещо про ввід у файл, вивід файлу, часи існування, тестування та парсинг командного рядка.

To round out this project, we'll briefly demonstrate how to work with environment variables and how to print to standard error, both of which are useful when you're writing command line programs. [ch10-03-lifetime-syntax.html#validating-references-with-lifetimes](#) [ch10-03-lifetime-syntax.html#validating-references-with-lifetimes](#)

Робота зі Змінними Середовища

Ми покращимо `minigrep`, додавши екстра функціонал: можливість нечутливого до регістру пошуку, який користувач може увімкнути використавши змінну середовища. Ми могли б зробити цю особливість опцією командного рядку та вимагати, щоб користувачі вводили її кожного разу, коли вони хотіли б її застосувати, але, замість цього, зробивши її змінною середовища, ми дозволяємо нашим користувачам встановлювати змінну середовища одноразово і мати будь-який пошук в цій сесії терміналу нечутливим до регістру.

Написання Провального Тесту для Нечутливої до Регістру Функції `search`

Спочатку ми додаємо нову функцію `search_case_insensitive`, яка буде викликатися, коли змінна середовища має якесь значення. Ми продовжимо дотримуватися процесу TDD, так що, знову, перший крок це написати провальний тест. Ми додамо новий тест новій функції `search_case_insensitive` та перейменуємо наш старий тест із `one_result` в `case_sensitive` для уточнення відмінностей між двома тестами, як показано в Блоці Коду 12-20.

Файл: `src/lib.rs`

```

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

    #[test]
    fn case_sensitive() {
        let query = "duct";
        let contents = "\
Rust:
safe, fast, productive.
Pick three.
Duct tape.";

        assert_eq!(vec!["safe, fast, productive."], search(query, contents));
    }

    #[test]
    fn case_insensitive() {
        let query = "rUsT";
        let contents = "\
Rust:
safe, fast, productive.
Pick three.
Trust me.";

        assert_eq!(
            vec!["Rust:", "Trust me."],
            search_case_insensitive(query, contents)
        );
    }
}

```

Listing 12-20: Adding a new failing test for the case-insensitive function we're about to add

Зверніть увагу, що ми також відредагували **зміст** старого тесту. Ми додали новий рядок із текстом **"Duct tape."** використавши велику літеру D, яка має не зіставлятися з запитом **"duct"** коли ми шукаємо в чутливому до регістра режимі. Зміна старого тексту таким чином допомагає нам гарантувати, що ми не зламаємо функціонал чутливого до регістру пошуку, який ми вже імплементували. Цей тест зараз має пройти та має продовжувати проходити допоки ми працюємо над нечутливим до регістру пошуком.

Новий тест для *нечутливого до регістру* пошуку використовує **"rUsT"** як запит. В функції **search_case_insensitive**, яку ми незабаром додамо, запит **"rUsT"** має зіставлятися з рядком який містить **"Rust:"** із великою літерою R та рядком **"Trust me."**, попри те, що обидва мають різний від запиту регістр. Це наш провальний тест і він не зможе вдало компілюватися, бо ми ще не визначили функцію **search_case_insensitive**. Не соромтесь додати каркас імплементції, яка завжди повертає порожній вектор, подібно до

використаного способу в функції `search` із Блока Коду 12-16, щоб побачити, що тест компілюється та провалюється.

Реалізація Функції `search_case_insensitive`

Функція `search_case_insensitive`, показана в Блоці Коду 12-21, буде майже така сама, як функція `search`. Різниця лише в тому, що ми зробимо текст в `query` і в кожній `line` малими літерами, тому незважаючи на регістр вхідних аргументів, вони будуть однакового регістру коли ми будемо перевіряти, чи містить рядок запит.

Файл: `src/lib.rs`

```
pub fn search_case_insensitive<'a>(
    query: &str,
    contents: &'a str,
) -> Vec<&'a str> {
    let query = query.to_lowercase();
    let mut results = Vec::new();

    for line in contents.lines() {
        if line.to_lowercase().contains(&query) {
            results.push(line);
        }
    }

    results
}
```

Listing 12-21: Defining the `search_case_insensitive` function to lowercase the query and the line before comparing them

Спочатку ми зменшуємо регістр стрічки `query` та зберігаємо в затіненій змінній з такою ж назвою. Виклик `to_lowercase` на запиті необхідне, щоб незалежно від того, чи запит користувача `"rust"`, `"RUST"`, `"Rust"`, чи `"rUsT"`, ми обробляли запит, ніби він `"rust"` і були не чутливі до регістру. Хоча `to_lowercase` буде обробляти базовий Unicode, він не буде 100% чітким. Якщо ми писали б справжній застосунок, ми б хотіли додатково попрацювати тут, але ця секція про змінні середовища, а не Unicode, тому ми зупинимось на цьому.

Зауважте, що `query` тепер є `String`, а не строковим слайсом, бо виклик до `to_lowercase` створює нові дані, а не посилається на ті, що існують. Скажімо запит це, наприклад, `"rUsT"`: слайс стрічки не містить малі літери `u` або `t`, щоб ми це використали, тому ми зробимо алокацію нової `String` яка буде містити `"rust"`. Ми зараз передамо `query`, як

аргумент методу `contains` і нам потрібно додати амперсанд, бо сигнатура `contains` призначена отримувати слайс стрічки.

Далі, ми додамо виклик `to_lowercase` кожній `line`, щоб зробити всі символи малими. Тепер, коли ми перетворили `line` та `query` в нижній регістр, ми знайдемо збіги, незважаючи на регістр запиту.

Подивимось, чи ця імплементація пройде тести:

```
$ cargo test
Compiling minigrep v0.1.0 (file:///projects/minigrep)
Finished test [unoptimized + debuginfo] target(s) in 1.33s
Running unittests src/lib.rs (target/debug/deps/minigrep-9cd200e5fac0fc94)

running 2 tests
test tests::case_insensitive ... ok
test tests::case_sensitive ... ok

test result: ok. 2 passed; 0 failed; 0 ignored; 0 measured; 0 filtered out;
finished in 0.00s

Running unittests src/main.rs (target/debug/deps/minigrep-9cd200e5fac0fc94)

running 0 tests

test result: ok. 0 passed; 0 failed; 0 ignored; 0 measured; 0 filtered out;
finished in 0.00s

Doc-tests minigrep

running 0 tests

test result: ok. 0 passed; 0 failed; 0 ignored; 0 measured; 0 filtered out;
finished in 0.00s
```

Чудово! Вони пройшли. Тепер викличемо нову функцію `search_case_insensitive` з функції `run`. Спочатку ми додамо опцію конфігурації в структуру `Config` для перемикання між чутливим та не чутливим до регістру пошуком. Додавання цього поля призведе до помилки компілятора, оскільки ми ще ніде не ініціювали це поле:

Файл: `src/lib.rs`

```
pub struct Config {
    pub query: String,
    pub file_path: String,
    pub ignore_case: bool,
}
```

Ми додали поле `ignore_case`, яке містить Boolean. Далі, нам потрібно, щоб функція `run` перевіряла значення поля `ignore_case` та використовувала це, щоб вирішити, чи викликати функцію `search` чи функцію `search_case_insensitive`, як показано в Блоці Коду 12-22. Проте, це ще не буде компілюватися.

Файл: `src/lib.rs`

```
pub fn run(config: Config) -> Result<(), Box<dyn Error>> {
    let contents = fs::read_to_string(config.file_path)?;

    let results = if config.ignore_case {
        search_case_insensitive(&config.query, &contents)
    } else {
        search(&config.query, &contents)
    };

    for line in results {
        println!("{line}");
    }

    Ok(())
}
```

Блок коду 12-22: Виклик `search` або `search_case_insensitive` на основі значення в `config.ignore_case`

Наостанок, нам потрібно перевірити змінну середовища. Функції для роботи зі змінними середовища є в модулі `env` стандартної бібліотеки, тому ми внесемо цей модуль в область видимості зверху файлу `src/lib.rs`. Потім ми використаємо функцію `var` з модуля `env` для перевірки наявності значення в змінній середовища з ім'ям `IGNORE_CASE`, як показано в Блоці коду 12-23.

Файл: `src/lib.rs`

```

use std::env;
// --snip--

impl Config {
    pub fn build(args: &[String]) -> Result<Config, &'static str> {
        if args.len() < 3 {
            return Err("not enough arguments");
        }

        let query = args[1].clone();
        let file_path = args[2].clone();

        let ignore_case = env::var("IGNORE_CASE").is_ok();

        Ok(Config {
            query,
            file_path,
            ignore_case,
        })
    }
}

```

Блок коду 12-23: Перевірка, чи є якесь значення в змінній середовища з назвою `IGNORE_CASE`

Тут ми створюємо нову змінну `ignore_case`. Щоб встановити її значення, нам потрібно викликати функцію `env::var` та передати їй ім'я змінної середовища `IGNORE_CASE`. Функція `env::var` повертає `Result`, який буде вдалим варіантом `Ok`, що містить значення змінної середовища, якщо їй встановлено будь-яке значення. Він поверне варіант `Err` якщо змінна середовища не встановлена.

Ми використовуємо метод `is_ok` на `Result`, щоб перевірити чи встановлена змінна середовища, яка буде означати, що програма буде здійснювати чутливий до регістру пошук. Якщо змінній середовища `IGNORE_CASE` нічого не встановлено, `is_ok` поверне `false` та програма виконуватиме чутливий до регістру пошук. Нас не хвилює значення змінної середовища, лише чи воно встановлене чи ні, тому ми перевіряємо з `is_ok` замість використання `unwrap`, `expect`, або будь-якого іншого метода, який ми бачили в `Result`.

Ми передаємо значення змінної `ignore_case` екземпляру `Config`, щоб функція `run` могла прочитати це значення і вирішити, чи викликати `search_case_insensitive` або `search`, як ми реалізували у Блоці коду 12-22.

Спробуймо! Спочатку ми запустимо нашу програму без встановленої змінної середовища та з запитом `to`, яке буде зіставлятися з будь-яким рядком, який містить слово "to" малими літерами:

```
$ cargo run -- to poem.txt
  Compiling minigrep v0.1.0 (file:///projects/minigrep)
  Finished dev [unoptimized + debuginfo] target(s) in 0.0s
  Running `target/debug/minigrep to poem.txt`
Are you nobody, too?
How dreary to be somebody!
```

Схоже, що це все ще працює! Тепер запусимо програму з `IGNORE_CASE` встановленим на `1`, але із тим самим запитом `to`.

```
$ IGNORE_CASE=1 cargo run -- to poem.txt
```

Якщо ви використовуєте PowerShell, вам потрібно встановити змінну оточення і запустити програму як окремі команди:

```
PS> $Env:IGNORE_CASE=1; cargo run -- to poem.txt
```

Це зробить `IGNORE_CASE` збереженим до кінця вашої сесії в консолі. Це налаштування можна вимкнути з командлетом `Remove-Item`:

```
PS> Remove-Item Env:IGNORE_CASE
```

Ми повинні отримати рядки, які містять "to" та які мають бути великими літерами:

```
Are you nobody, too?
How dreary to be somebody!
To tell your name the livelong day
To an admiring bog!
```

Чудово, ми також отримуємо рядки, що містять "To"! Наша програма `minigrep` тепер може робити нечутливий до регістру пошук контрольований змінною середовища. Тепер ви знаєте як керувати опціями встановленими із використанням як аргументів командного рядка, так і змінних середовища.

Деякі програми дозволяють аргументи *та* змінні середовища для однієї й тієї ж конфігурації. У цих випадках програма вирішує, що з цього має перевагу. Для іншої самостійної вправи, спробуйте контролювати чутливість до регістру через або аргумент командного рядка або змінну середовища. Вирішуйте, чи аргумент командного рядка має бути пріоритетніше чи змінна середовища, якщо програма запускається як чутлива до регістра, а інша як нечутлива.

Модуль `std::env` містить ще багато корисного в роботі зі змінними середовища: перегляньте її документацію, щоб побачити що є можливим.

Написання Повідомлень про Помилки у Помилковий Вивід замість Стандартного Виводу

Наразі ми записуємо увесь наш вивід в термінал використовуючи макрос `println!`. В більшості терміналів є два типи виводу: *standard output* (`stdout`) для загальної інформації та *standard error* (`stderr`) для повідомлень про помилки. Це розділення дозволяє користувачам направити вдалий вивід програми в файл, але все ще виводити повідомлення про помилки на екрані.

The `println!` macro is only capable of printing to standard output, so we have to use something else to print to standard error.

Перевірка Місця Написання Помилки

Спочатку, нумо побачимо, як контент який `minigrep` виводить в консолі наразі записується в стандартний вивід, включаючи будь-які повідомлення про помилки, які замість цього ми хочемо записувати в Standard Error. Ми зробимо це перенаправивши потік стандартного виводу в файл водночас із навмисним спричиненням помилки. Ми не будемо перенаправляти стандартний помилковий потік, тому будь-який контент відправлений в Standard Error буде продовжувати показуватися на екрані.

Очікується, що програми командного рядка надсилатимуть помилкові повідомлення в потік Standard Error, щоб ми все ще могли бачити помилкові повідомлення на екрані, навіть якщо ми перенаправляємо потік стандартного виводу в файл. Наразі наша програма не добре налаштована: ми побачимо, як вона збереже вивід помилкового повідомлення в файл натомість!

Щоб продемонструвати цю поведінку, ми запустимо програму з `>` і шляхом до файлу, *output.txt*, якому ми хочемо перенаправити потік стандартного виводу. Ми не будемо передавати аргументи, що спричинить помилку:

```
$ cargo run > output.txt
```

Синтаксис `>` вказує shell писати вміст стандартного виводу в *output.txt* замість екрана. Ми не бачимо помилкове повідомлення, яке ми очікували виведеним на екран, тому це означає, що воно опинилося в файлі. Це те, що містить *output.txt*:

```
Problem parsing arguments: not enough arguments
```


Так, наше помилкове повідомлення виводиться в консолі стандартного виводу. Це набагато корисніше для таких помилкових повідомлень, щоб вони виводилися в консолі Standard Error та щоб тільки дані від вдалих запусків опинялися в файлі. Ми змінимо це.

Виведення Помилки у Помилковий Вивід

Ми використаємо код в Блоці Коду 12-24 для зміни виводу помилкових повідомлень в консолі. Через рефакторинг, який ми робили раніше в цьому розділі, увесь код який виводить помилкові повідомлення перебуває в одній функції, `main`. Стандартна бібліотека надає макрос `eprintln!` який виводить в консолі потоку Standard Error, тому змінимо два місця, де ми викликаємо `println!`, використовуючи замість цього `eprintln!` для виводу в консолі.

Файл: src/main.rs

```
fn main() {
    let args: Vec<String> = env::args().collect();

    let config = Config::build(&args).unwrap_or_else(|err| {
        eprintln!("Problem parsing arguments: {err}");
        process::exit(1);
    });

    if let Err(e) = minigrep::run(config) {
        eprintln!("Application error: {e}");
        process::exit(1);
    }
}
```

Блок коду 12-24: Запис повідомлень про помилки до стандартного помилкового виводу, замість стандартного, за допомогою `eprintln!`

Тепер запусимо програму ще раз таким же чином, без будь-яких аргументів і перенаправивши стандартний вивід за допомогою `>`:

```
$ cargo run > output.txt
Problem parsing arguments: not enough arguments
```

Тепер ми бачимо помилку на екрані і *output.txt* нічого не містить, і саме такої поведінки ми очікуємо від програм командного рядка.

Запусимо програму ще раз з аргументами, які не викликають помилки, але все ж таки перенаправивши стандартний вивід у файл, ось так:

```
$ cargo run -- to poem.txt > output.txt
```

Ми не побачимо жодного виводу в терміналі та *output.txt* буде містити наші результати:

Файл: output.txt

```
Are you nobody, too?  
How dreary to be somebody!
```

Це демонструє, що тепер ми використовуємо стандартний вивід для успішного виводу і стандартну помилку для виводу помилок у відповідних випадках.

Підсумок

В цьому розділі ми пригадали деякі основні концепти, які ви вивчили раніше та розглянули як виконувати базові I/O операції в Rust. Використовуючи аргументи командного рядка, файли, змінні середовища та макрос `eprintln!` для виведення помилок в консолі, тепер ви готові написати застосунок для командного рядка.

Поєднавши це з концептами з попередніх розділів, ваш код буде добре організованим, ефективно збирати дані в відповідні структури даних, вдало обробляти помилки та буде добре перевірено.

Далі ми детальніше розглянемо деякі функції Rust, на які вплинули функціональні мови: замикання та ітератори.

Функціональні Можливості Мови: Ітератори та Замикання

Розробка мови Rust була натхненна багатьма мовами та техніками, і суттєвий вплив на неї мало *функціональне програмування*. Програмування у функціональному стилі зазвичай вбачає у собі використання функцій як значень, передаючи їх в аргументи, повертаючи як результат інших функцій, присвоюючи до змінних для подальшого застосування і так далі.

У цьому розділі, ми не обговорюватимемо питання того, що є функціональним програмуванням, а що ні. Натомість, ми обговоримо деякі можливості Rust, що схожі на можливості багатьох мов, які часто називають функціональними.

Зокрема, ми розглянемо:

- *Замикання*, функціональну конструкцію, яку можна зберігати у змінній
- *Ітератори*, спосіб обробки послідовності елементів
- Як використовувати замикання та ітератори для покращення операцій вводу/виводу для проекту з 12 розділу
- Швидкість замикань та ітераторів (Спойлер: вони швидші ніж ви думаєте!)

Ми вже розглянули деякі можливості Rust, такі як зіставлення зі шаблоном та енуми, які теж були створені з оглядом на ідеї функціонального стилю. Оскільки опановування замикань та ітераторів це важлива частина написання ідіоматичного та швидкого коду на Rust, ми виділимо для них увесь цей розділ.

Замикання: Анонімні Функції, що Захоплюють Своє Середовище

У Rust замикання - це анонімні функції, які можна зберігати в змінній або передавати як аргументи до інших функцій. Ви можете створити замикання в одному місці, а потім викликати деінде для обчислення в іншому контексті. На відміну від функції, замикання здатні використовувати значення з області видимості в якій вони були визначені. Ми продемонструємо, як наявність замикань дозволяє повторно використовувати код та змінювати поведінку програми.

Захоплення Середовища з Замиканнями

Спочатку ми розглянемо, як можна використовувати замикання для фіксації значень середовища, в якому вони визначені, для подальшого використання. Ось сценарій: Час від часу, наша компанія по виробництву футболок роздає ексклюзивну футболку, випущену ексклюзивним тиражем, комусь із нашого списку розсилки як рекламу. Люди зі списку розсилки можуть за бажанням додати свій улюблений колір до свого профілю. Якщо людина, якій надіслали безплатну футболку, обрала свій улюблений колір, вона отримає футболку такого ж кольору. Якщо людина не зазначила свій улюблений колір, то вона отримає футболку такого кольору, якого в компанії найбільше всього.

Існує багато способів це реалізувати. Для цього прикладу, ми використаємо еnum `ShirtColor`, який складається з варіантів `Red` та `Blue` (обмежимо кількість доступних кольорів для простоти). Ми представлятимемо товарні запаси компанії за допомогою структури `Inventory`, яка має поле, що зветься `shirts`, яке містить `Vec<ShirtColor>`, що представляє кольори наявних на складі футболок. Метод `giveaway`, визначений для `Inventory`, отримує опціональний бажаний колір футболки для вручення переможцю та повертає колір футболки, яку цей переможець отримає. Ця ситуація показана в Блоці коду 13-1:

Файл: `src/main.rs`

```
#[derive(Debug, PartialEq, Copy, Clone)]
enum ShirtColor {
    Red,
    Blue,
}

struct Inventory {
    shirts: Vec<ShirtColor>,
}

impl Inventory {
    fn giveaway(&self, user_preference: Option<ShirtColor>) -> ShirtColor {
        user_preference.unwrap_or_else(|| self.most_stocked())
    }

    fn most_stocked(&self) -> ShirtColor {
        let mut num_red = 0;
        let mut num_blue = 0;

        for color in &self.shirts {
            match color {
                ShirtColor::Red => num_red += 1,
                ShirtColor::Blue => num_blue += 1,
            }
        }
        if num_red > num_blue {
            ShirtColor::Red
        } else {
            ShirtColor::Blue
        }
    }
}

fn main() {
    let store = Inventory {
        shirts: vec![ShirtColor::Blue, ShirtColor::Red, ShirtColor::Blue],
    };

    let user_pref1 = Some(ShirtColor::Red);
    let giveaway1 = store.giveaway(user_pref1);
    println!(
        "The user with preference {:?} gets {:?}",
        user_pref1, giveaway1
    );

    let user_pref2 = None;
    let giveaway2 = store.giveaway(user_pref2);
    println!(
        "The user with preference {:?} gets {:?}",
        user_pref2, giveaway2
    );
}
```

Блок коду 13-1: роздача подарунків у компанії по виробництву футболки

Змінна `store`, визначена в `main`, містить дві сині футболки і одну червону футболку, які лишилися для роздачі у рекламній акції. Ми викликаємо метод `giveaway` для користувача, що віддає перевагу червоній футолці, і для користувача, що не має особливих побажань.

Знову ж таки, цей код може бути реалізований багатьма способами, і тут, щоб сфокусуватися на замиканнях, ми дотримуватимемося концепцій, які ви вже вивчили, окрім тіла методу `giveaway`, який використовує замикання. У методі `giveaway` ми отримуємо параметром побажання типу `Option<ShirtColor>` і викликаємо на `user_preference` метод `unwrap_or_else`. Метод `unwrap_or_else` для `Option<T>` визначений у стандартній бібліотеці. Він приймає один аргумент: замикання без аргументів, що повертає значення типу `T` (того ж типу, що міститься у варіанті `Some` `Option<T>`, у цьому випадку `ShirtColor`). Якщо `Option<T>` є варіантом `Some`, `unwrap_or_else` поверне значення, що міситься у `Some`. Якщо ж `Option<T>` є варіантом `None`, `unwrap_or_else` викликає замикання і повертає значення, повернене з замикання.

Ми зазначаємо вираз замикання `|| self.most_stocked()` аргументом `unwrap_or_else`. Це замикання не приймає параметрів (якби замикання мало параметри, вони б з'явилися між вертикальними лініями). Тіло замикання викликає `self.most_stocked()`. Тут ми визначаємо замикання, і реалізація `unwrap_or_else` обчислить це замикання пізніше, якщо знадобиться його результат.

Виконавши цей код, в консолі виведеться:

```
$ cargo run
  Compiling shirt-company v0.1.0 (file:///projects/shirt-company)
  Finished dev [unoptimized + debuginfo] target(s) in 0.27s
  Running `target/debug/shirt-company`
The user with preference Some(Red) gets Red
The user with preference None gets Blue
```

Тут один цікавий момент полягає в тому, що ми вже передали замикання, яке викликає `self.most_stocked()` для поточного екземпляра `Inventory`. Стандартній бібліотеці непотрібно нічого знати про типи `Inventory` або `ShirtColor`, які ми визначили, або про логіку, яку ми бажаємо використати у даному сценарії. Замикання захоплює немутабельне посилання на екземпляр `Inventory` `self` і передає його з написаним нами кодом у метод `unwrap_or_else`. Функції, з іншого боку, не можуть захоплювати своє середовище у такий спосіб.

Виведення та Анотація Типу Замикання

Між функціями та замиканнями існує більше відмінностей. Замикання зазвичай не потребують анотації типів параметрів чи типу, який вони повертають, на відміну від функцій `fn`. Анотації типів потрібні функціям, бо типи є частиною явного інтерфейсу, відкритого вашим користувачем. Жорстке визначення інтерфейсу важливе для забезпечення того, щоб всі погоджувались з тим, які значення функція приймає та повертає. Замикання, з іншого боку, не використовуються у подібному відкритому інтерфейсі: вони зберігаються у змінних і використовуються без назв і відкривання їх користувачем ваших бібліотек.

Замикання зазвичай короткі та актуальні тільки у конкретному контексті, а не в будь-якому довільному сценарії. У цих обмежених контекстах компілятор може вивести типи параметрів і типу, що повертається, так само як може вивести типи більшості змінних (трапляються рідкісні випадки, коли компілятор потребує також анотації типів замикань).

Як і зі змінними, ми можемо за бажання додати анотації типів, коли хочемо збільшити виразність і ясність ціною більшої багатослівності, ніж потрібно. Анотування типів для замикання виглядатиме як визначення, наведене у Блоці коду 13-2. У цьому прикладі ми визначаємо замикання і зберігаємо його у змінній замість визначення замикання у місці, де ми передаємо його як аргумент, як ми робили у Блоці коду 13-1.

Файл: `src/main.rs`

```
let expensive_closure = |num: u32| -> u32 {
    println!("calculating slowly...");
    thread::sleep(Duration::from_secs(2));
    num
};
```

Блок коду 13-2: Додавання необов'язкових анотацій типу параметра і значення, яке повертає замикання

Із анотаціями типів синтаксис замикань виглядає більш схожим на синтаксис функцій. Тут ми визначаємо функцію, що додає 1 до свого параметра і замикання, що має таку саму поведінку, для порівняння. Ми додали кілька пробілів для вирівнювання відповідних частин. Це ілюструє, чим синтаксис замикань подібний до синтаксису функцій, за виключенням використання вертикальних ліній і обсягу необов'язкового синтаксису:

```
fn add_one_v1 (x: u32) -> u32 { x + 1 }
let add_one_v2 = |x: u32| -> u32 { x + 1 };
let add_one_v3 = |x|           { x + 1 };
let add_one_v4 = |x|           x + 1 ;
```

У першому рядку визначення функції, а в другому анотоване визначення замикання. На третьому рядку ми прибираємо анотацію типу з визначення замикання. На четвертому рядку ми прибираємо дужки, які є опціональними через те, що замикання містить в собі тільки один вираз. Усе це є коректними визначеннями, які будуть демонструвати під час їх виклику одну й ту саму поведінку. Рядки `add_one_v3` та `add_one_v4` вимагають, щоб замикання викликали для компіляції, бо типи будуть виведені з того, як їх використовують. Це схоже на те, як `let v = Vec::new();` потребує або анотацію типів, або додати значення певного типу у `Vec`, щоб Rust міг вивести тип.

Для визначень замикань компілятор виведе один конкретний тип для кожного параметра і для значення, що повертається. Наприклад, у Блоці коду 13-3 показано визначення замикання, що повертає значення, переданого йому як параметр. Це замикання не дуже корисне, окрім як для цього прикладу. Зауважте, що ми не додавали анотації типів до визначення. Оскільки тут немає анотації типів, ми можемо викликати замикання для будь-якого типу, що ми тут вперше і зробили з `String`. Якщо ми потім спробуємо викликати `example_closure` з цілим параметром, то дістанемо помилку.

Файл: `src/main.rs`

```
let example_closure = |x| x;  
  
let s = example_closure(String::from("hello"));  
let n = example_closure(5);
```

Блок коду 13-3: спроба викликати замикання, чий тип вже виведені, із двома різними типами

Компілятор повідомляє про таку помилку:


```

$ cargo run
   Compiling closure-example v0.1.0 (file:///projects/closure-example)
error[E0308]: mismatched types
  --> src/main.rs:5:29
   |
5  |         let n = example_closure(5);
   |         ^^^^^^^^^^^^^^^^^^^^^^^^^ help: try using a conversion method:
   |         `.to_string()`
   |         |               |
   |         |               expected struct `String`, found integer
   |         arguments to this function are incorrect
   |
note: closure parameter defined here
  --> src/main.rs:2:28
   |
2  |         let example_closure = |x| x;
   |                                ^

```

For more information about this error, try `rustc --explain E0308`.

error: could not compile `closure-example` due to previous error

Коли ми уперше викликали `example_closure` зі значенням `String`, компілятор вивів, що тип `x` і тип, що повертається із замикання, як `String`. Ці типи були зафіксовані для замикання `example_closure`, і ми отримаємо помилку типу, коли ще раз намагаємося використати інший тип для цього ж замикання.

Захоплення Посилань чи Передання Володіння

Замикання можуть захоплювати значення зі свого середовища у три способи, що прямо відповідають трьом способам передачі параметра у функцію: немутабельне позичання, мутабельне позичання і взяття володіння. Замикання вирішує, яким способом скористатися, виходячи з того, що тіло функції робить із захопленими значеннями.

У Блоці коду 13-4 ми визначаємо замикання, яке захоплює немутабельне посилання на вектор з назвою `list`, тому що йому потрібно лише немутабельне посилання для виведення значення:

Файл: `src/main.rs`

```
fn main() {
    let list = vec![1, 2, 3];
    println!("Before defining closure: {:?}", list);

    let only_borrows = || println!("From closure: {:?}", list);

    println!("Before calling closure: {:?}", list);
    only_borrows();
    println!("After calling closure: {:?}", list);
}
```

Блок коду 13-4: визначення і виклик замикання, що захоплює немутабельне посилання

Цей приклад також ілюструє, що змінна може бути зв'язана з визначенням замикання, і ми можемо пізніше викликати замикання, використовуючи назву змінної та дужки та, якби назва змінної була назвою функції.

Оскільки ми можемо мати одночасно декілька немутабельних посилань на `list`, до нього можливий доступ до визначення замикання, після визначення, але до виклику замикання і після виклику замикання. Цей код компілюється, виконується і виводить:

```
$ cargo run
  Compiling closure-example v0.1.0 (file:///projects/closure-example)
  Finished dev [unoptimized + debuginfo] target(s) in 0.43s
  Running `target/debug/closure-example`
Before defining closure: [1, 2, 3]
Before calling closure: [1, 2, 3]
From closure: [1, 2, 3]
After calling closure: [1, 2, 3]
```

Далі в Блоці коду 13-5 ми змінюємо тіло замикання, щоб воно додавало елемент до вектора `list`. Це замикання тепер захоплює мутабельне посилання:

Файл: src/main.rs

```
fn main() {
    let mut list = vec![1, 2, 3];
    println!("Before defining closure: {:?}", list);

    let mut borrows_mutably = || list.push(7);

    borrows_mutably();
    println!("After calling closure: {:?}", list);
}
```

Блок коду 13-5: визначення і виклик замикання, що захоплює мутабельне посилання

Цей код компілюється, виконується і виводить:

```
$ cargo run
Compiling closure-example v0.1.0 (file:///projects/closure-example)
Finished dev [unoptimized + debuginfo] target(s) in 0.43s
Running `target/debug/closure-example`
Before defining closure: [1, 2, 3]
After calling closure: [1, 2, 3, 7]
```

Зверніть увагу, що тепер немає `println!` між визначенням і викликом замикавання `borrow_mutably`: коли визначається `borrow_mutably`, воно захоплює мутабельне посилання на `list`. Ми не використовуємо замикавання знову після його виклику, тож мутабельне позичання закінчується. Між визначенням замикавання і його викликом не дозволене немутабельне позичання, потрібне для виведення, оскільки ніякі інші позичання не дозволені, коли є немутабельне позичання. Спробуйте додати туди `println!`, щоб побачити, яке повідомлення про помилку ви дістанете!

Якщо ви хочете змусити замикавання прийняти володіння значеннями, яке воно використовує у середовищі навіть якщо тіло замикавання не обов'язково потребує володіння, ви можете використати ключове слово `move` перед списком параметрів.

Ця техніка особливо корисна при передачі замикавання новому потоку, щоб переміщеними даними володів цей новий потік. Ми обговоримо потоки і нащо вам хотілося б користуватися ними у Розділі 16, коли ми говоримо про одночасне виконання, але поки що давайте коротко дослідимо створення нового потоку за допомогою замикавання, що вимагає ключове слово `move`. Блок коду 13-6 показує змінений Блок коду 13-4, що виводить вектор у новому потоці, а не у головному:

Файл: `src/main.rs`

```
use std::thread;

fn main() {
    let list = vec![1, 2, 3];
    println!("Before defining closure: {:?}", list);

    thread::spawn(move || println!("From thread: {:?}", list))
        .join()
        .unwrap();
}
```

Блок коду 13-6: Використання `move` для того, щоб змусити замикавання для потоку взяти володіння `list`

Ми створюємо новий потік, надаючи йому замикавання для виконання як аргумент. Тіло замикавання виводить список. У Блоці коду 13-4 це замикавання захоплює лише `list` за допомогою немутабельного посилання, бо це найменша кількість доступу до `list`,

потрібна для його виведення. У цьому прикладі, попри те, що тіло замикання все ще потребує лише немутабельного посилання, нам потрібно вказати, що `list` слід перемістити у замикання, додавши ключове слово `move` на початку визначення замикання. Новий потік може завершитися до завершення решти головного потоку, чи основний потік може завершитися першим. Якщо основний потік утримував володіння `list`, але завершився до завершення нового потоку і скинув `list`, немутабельне посилання у тому потоці стає некоректним. Відповідно, компілятор вимагає, щоб `list` буде переміщений у замикання, що передається у новий потік, щоб посилання буде коректним. Спробуйте видалити ключове слово `move` або використати `list` в основному потоці після закриття, щоб побачити помилки компілятора, які ви отримуєте!

Переміщення Захоплених Значень із Замикань і Трейти `Fn`

Коли замикання захопило посилання чи володіння значенням у місці, де це замикання визначене (таким чином впливаючи на те, що було переміщено в замикання), код у тілі замикання визначає, що відбувається з посиланнями або значеннями, коли пізніше замикання обчислюється (тим самим впливаючи на те, що буде переміщено із замикання). Тіло замикання може робити одне з: перемістити захоплене значення із замикання, змінити захоплене значення, не перемішати ані змінювати значення, чи взагалі нічого не захоплювати з середовища.

Те, як замикання захоплює і обробляє значення з середовища, впливає на те, які трейти реалізовує замикання, а трейти - спосіб функціям і структурам зазначити, які види замикань вони можуть використовувати. Замикання автоматично реалізують один, два чи всі три ці трейти `Fn` із накопиченням, залежно від того, як тіло замикання поводить себе зі значеннями:

1. `FnOnce` застосовується до замикань, які можна викликати один раз. Усі замикання реалізують щонайменше цей трейт, бо всі замикання можна викликати. Замикання, що переміщує захоплені значення зі свого тіла можуть реалізовувати лише `FnOnce` і жодного іншого з трейтів `Fn`, бо їх можна викликати лише один раз.
2. `FnMut` застосовується до замикань, які не переміщують захоплені значення зі свого тіла, але можуть їх змінювати. Ці замикання можуть бути викликані більше ніж один раз.
3. `Fn` застосовується до замикань, що не переміщують захоплені значення зі свого тіла і їх не змінюють, а також до замикань, що нічого не захоплюють із середовища. Ці замикання можуть бути викликані більше одного разу без змін середовища, що важливо у таких випадках, як одночасний виклик замикання багато разів.

Погляньмо на визначення методу `unwrap_or_else` для `Option<T>`, який ми використовували в Блоці Коду 13-1:

```
impl<T> Option<T> {
    pub fn unwrap_or_else<F>(self, f: F) -> T
    where
        F: FnOnce() -> T
    {
        match self {
            Some(x) => x,
            None => f(),
        }
    }
}
```

Згадайте, що `T` - це узагальнений тип, що представляє тип значення з варіанта `Some` із `Option`. Цей тип `T` також є типом, який повертає поверненням функція `unwrap_or_else`: код, що викликає `unwrap_or_else`, наприклад, для `Option<String>` отримає `String`.

Далі, зверніть увагу, що функція `unwrap_or_else` має додатковий параметр узагальненого типу `F`. Тип `F` є типом параметра `f`, який є замиканням, яке ми надаємо під час виклику `unwrap_or_else`.

Трейтове обмеження, вказане для узагальненого типу `F`, `FnOnce() -> T`, що означає, що `F` має бути можливо викликати один раз, вона не приймає аргументи, і повертає `T`. Використання `FnOnce` у трейтовому обмеженні виражає обмеження, що `unwrap_or_else` збирається викликати `f` не більше одного разу. У тілі `unwrap_or_else`, як ми можемо бачити, якщо `Option` є `Some`, `f` не буде викликано. Якщо `Option` є `None`, `f` буде викликана один раз. Оскільки всі замикання реалізують `FnOnce`, `unwrap_or_else` приймає найрізноманітніші типи замикань і гнучка настільки, наскільки це можливо.

Примітка: функції також можуть реалізовувати усі три трейти `Fn`. Якщо те, що ми хочемо зробити, не потребує захоплення значення з середовища, ми можемо використовувати ім'я функції замість замикання там, де нам потрібне щось, що реалізує один з трейтів `Fn`. Скажімо, для значення `Option<Vec<T>>` ми можемо викликати `unwrap_or_else(Vec::new)`, щоб отримати новий порожній вектор, якщо значення буде `None`.

Тепер подивімося на метод зі стандартної бібліотеки `sort_by_key`, визначений для слайсів, щоб побачити, як це відрізняється від `unwrap_or_else`, і чому `sort_by_key` використовує `FnMut` замість `FnOnce` як трейтове обмеження. Замикання приймає один аргумент у формі посилання на поточний елемент у слайсі, і повертає значення типу `K`, яке можна впорядкувати. Ця функція корисна, коли вам треба відсортувати слайс за певним атрибутом кожного елемента. У Блоці коду 13-7 ми маємо список екземплярів

`Rectangle` і використовуємо `sort_by_key`, щоб впорядкувати їх за атрибутом `width` за зростанням:

Файл: `src/main.rs`

```
#[derive(Debug)]
struct Rectangle {
    width: u32,
    height: u32,
}

fn main() {
    let mut list = [
        Rectangle { width: 10, height: 1 },
        Rectangle { width: 3, height: 5 },
        Rectangle { width: 7, height: 12 },
    ];

    list.sort_by_key(|r| r.width);
    println!("{:#?}", list);
}
```

Блок коду 13-7: Використання `sort_by_key` для впорядкування прямокутників за шириною

Цей код виведе:

```
$ cargo run
Compiling rectangles v0.1.0 (file:///projects/rectangles)
Finished dev [unoptimized + debuginfo] target(s) in 0.41s
Running `target/debug/rectangles`
[
  Rectangle {
    width: 3,
    height: 5,
  },
  Rectangle {
    width: 7,
    height: 12,
  },
  Rectangle {
    width: 10,
    height: 1,
  },
]
```

`sort_by_key` визначено для замикання `FnMut` тому, що вона викликає замикання кілька разів: один раз для кожного елемента у слайсі. Замикання `|r| r.width` не захоплює, не

змінює і не переміщує нічого з його середовища, тож це відповідає вимогам трейтового обмеження.

На противагу цьому, у Блоці коду 13-8 наведено приклад замикання, яке реалізує тільки трейт `FnOnce`, тому що воно переміщує значення з середовища. Компілятор не дозволить нам використовувати це замикання у `sort_by_key`:

Файл: src/main.rs

```
#[derive(Debug)]
struct Rectangle {
    width: u32,
    height: u32,
}

fn main() {
    let mut list = [
        Rectangle { width: 10, height: 1 },
        Rectangle { width: 3, height: 5 },
        Rectangle { width: 7, height: 12 },
    ];

    let mut sort_operations = vec![];
    let value = String::from("by key called");

    list.sort_by_key(|r| {
        sort_operations.push(value);
        r.width
    });
    println!("{:?}", list);
}
```

Блок коду 13-8: Спроба використати замикання `FnOnce` у `sort_by_key`

Це надуманий, заплутаний спосіб (який не працює) спробувати підрахувати кількість викликів `sort_by_key` при сортуванні `list`. Цей код намагається виконати підрахунок, виштовхуючи `value` - `String` з середовища замикання у вектор `sort_operations`. Замикання захоплює `value`, потім переміщує `value` із замикання, передаючи володіння `value` до вектора `sort_operations`. Це замикання може бути викликане один раз; спроба викликати вдруге не спрацює, оскільки `value` більше не буде в середовищі, щоб занести його до `sort_operations` знову! Таким чином це замикання реалізує лише `FnOnce`. Коли ми намагаємося скомпілювати цей код, то отримуємо помилку про те, що `value` не можна перемістити із замикання, оскільки замикання має реалізовувати `FnMut`:

```
$ cargo run
Compiling rectangles v0.1.0 (file:///projects/rectangles)
error[E0507]: cannot move out of `value`, a captured variable in an `FnMut` closure
--> src/main.rs:18:30
15 |         let value = String::from("by key called");
16 |         ----- captured outer variable
17 |         list.sort_by_key(|r| {
18 |             sort_operations.push(value);
           ^^^^^^ move occurs because `value` has type `String`, which does not implement the `Copy` trait

For more information about this error, try `rustc --explain E0507`.
error: could not compile `rectangles` due to previous error
```

Помилка вказує на рядок у тілі замикавання, що переміщує `value` з середовища. Щоб виправити це, нам потрібно змінити тіло замикавання так, щоб воно не переміщувало значення з середовища. Полічити кількість викликів `sort_by_key`, утримуючи лічильник у середовищі та збільшуючи його значення у тілі замикавання є прямішим шляхом для цього обчислення. Замикавання у Блоці коду 13-9 працює з `sort_by_key`, оскільки воно містить лише мутабельне посилання на лічильник `num_sort_operations` і тому може бути викликане більше ніж один раз:

Файл: src/main.rs

```
#[derive(Debug)]
struct Rectangle {
    width: u32,
    height: u32,
}

fn main() {
    let mut list = [
        Rectangle { width: 10, height: 1 },
        Rectangle { width: 3, height: 5 },
        Rectangle { width: 7, height: 12 },
    ];

    let mut num_sort_operations = 0;
    list.sort_by_key(|r| {
        num_sort_operations += 1;
        r.width
    });
    println!("{:#?}, sorted in {num_sort_operations} operations", list);
}
```


Блок коду 13-9: Використання замикання `FnMut` у `sort_by_key` дозволене

Трейти `Fn` мають важливе значення при визначенні або використанні функцій або типів, які використовують замикання. У наступному підрозділі ми обговоримо ітератори. Багато методів ітератора приймають аргументи-замикання, тому не забувайте, що дізналися про замикання, коли ми продовжимо!

Обробка Послідовностей Елементів з Ітераторами

Шаблон ітератора дозволяє вам виконати певну задачу з послідовністю елементів по черзі. Ітератор відповідає за логіку ітерації по елементах і визначення, коли послідовність закінчується. Коли ви користуєтесь ітераторами, вам не треба самостійно реалізовувати цю логіку.

У Rust ітератори є *лінивими*, тобто вони нічого не роблять до того моменту, коли ви викличете метод, що поглине ітератор і використає його. Наприклад, код у Блоці коду 13-10 створює ітератор по елементах вектора `v1`, викликавши метод `iter`, визначений для `Vec<T>`. Цей код як такий не робить нічого корисного.

```
let v1 = vec![1, 2, 3];  
  
let v1_iter = v1.iter();
```

Блок коду 13-10: створення ітератора

Ітератор зберігається у змінній `v1_iter`. Після того, як ми створили ітератор, ми можемо використовувати його у різні способи. У Блоці коду 3-5 з Розділу 3 ми ітерували по масиву за допомогою циклу `for`, щоб виконати певний код на кожному елементі. Під капотом тут неявно був створений і поглинутий ітератор, але до цього часу ми не звертали уваги на те, як саме це працює.

У прикладі з Блоку коду 13-11 ми відокремлюємо створення ітератора від його використання в циклі `for`. Коли цикл `for` викликають з ітератором у `v1_iter`, кожен елемент у ітераторі використовується одній ітерації циклу, який виводить кожне значення.

```
let v1 = vec![1, 2, 3];  
  
let v1_iter = v1.iter();  
  
for val in v1_iter {  
    println!("Got: {}", val);  
}
```

Блок коду 13-11: використання ітератора у циклі `for`

У мовах, які не мають у своїх стандартних бібліотеках ітераторів, ви, ймовірно, опишете такий функціонал, почавши зі змінної для індексу 0, використовуючи цю змінну для

індексування вектора, щоб отримати значення, та збільшуючи значення змінної в циклі, поки вона не досягне загальної кількості елементів у векторі.

Ітератори обробляють всю цю логіку за вас, скорочуючи код, який ви потенційно можете зіпсувати. Ітератори дають вам більше гнучкості для використання тієї ж логіки з різними типами послідовностей, а не лише структурами даних, які ви можете індексувати, такими як вектори. Розгляньмо, як ітератори це роблять.

Трейт `Iterator` і Метод `next`

Усі ітератори реалізують трейт, що зветься `Iterator`, визначений у стандартній бібліотеці. Визначення цього трейту виглядає ось так:

```
pub trait Iterator {  
    type Item;  
  
    fn next(&mut self) -> Option<Self::Item>;  
  
    // методи зі стандартною реалізацією пропущені  
}
```

Зверніть увагу, що це визначення використовує новий синтаксис: `type Item` і `Self::Item`, які визначають *асоційований тип* цього трейта. Ми глибше поговоримо про асоційовані типи у Розділі 19. Поки що, все, що вам слід знати - це те, що цей код каже, що реалізація трейту `Iterator` також вимагає, щоб ви визначили тип `Item`, і цей тип `Item` використовується як тип, що повертається методом `next`. Іншими словами, тип `Item` буде типом, повернутим з ітератора.

Трейт `Iterator` потребує від того, хто його реалізовує, визначення лише одного методу: методу `next`, який повертає за раз один елемент ітератора, обгорнутий у `Some` і, коли ітерація закінчиться, повертає `None`.

Ми можемо викликати метод `next` для ітераторів безпосередньо; Блок коду 13-12 демонструє, які значення повертаються повторюваними викликами `next` для ітератора, створеного з вектора.

Файл: `src/lib.rs`

```
#[test]
fn iterator_demonstration() {
    let v1 = vec![1, 2, 3];

    let mut v1_iter = v1.iter();

    assert_eq!(v1_iter.next(), Some(&1));
    assert_eq!(v1_iter.next(), Some(&2));
    assert_eq!(v1_iter.next(), Some(&3));
    assert_eq!(v1_iter.next(), None);
}
```

Блок коду 13-12: виклик методу `next` для ітератора

Зверніть увагу, що нам потрібно зробити `v1_iter` мутабельним: виклик методу `next` для ітератора змінює його внутрішній стан, який використовується для відстеження, де він знаходиться в послідовності. Іншими словами, цей код *поглинає*, чи використовує, ітератор. Кожен виклик `next` з'їдає елемент з ітератора. Нам не треба було робити `v1_iter` мутабельним, коли ми використали його в циклі `for`, бо цикл взяв володіння `v1_iter` і зробив його мутабельним за лаштунками.

Також зверніть увагу, що значення, які ми отримуємо від викликів `next`, є немутабельними посиланнями на значення у векторі. Метод `iter` створює ітератор по незмінних посиланнях. Якщо ми хочемо створити ітератор, який приймає володіння `v1` і повертає значення, що належать нам, ми можемо викликати `into_iter` замість `iter`. Аналогічно, якщо ми хочемо ітерувати по мутабельних посиланнях, ми можемо викликати `iter_mut` замість `iter`.

Методи, що Поглинають Ітератор

Трейт `Iterator` має ряд різних методів з реалізаціями по замовчуванню що надаються стандартною бібліотекою; ви можете дізнатися про ці методи в стандартній документації API для трейта `Iterator`. Деякі з цих методів викликають у своєму визначенні метод `next`, чому і необхідно визначити метод `next` при реалізації трейта `Iterator`.

Методи, що викликають `next`, звуться *поглинаючими адапторами*, бо їх виклик використовує ітератор. Один із прикладів - це метод `sum`, який бере володіння ітератором і ітерує по елементах, раз за разом викликаючи `next`, таким чином поглинаючи ітератор. Під час ітерації він додає кожен елемент до поточної загальної суми і повертає загальну суму, коли ітерація завершена. Блок коду 13-13 має тест, що ілюструє використання методу `sum`:

Файл: `src/lib.rs`

```
#[test]
fn iterator_sum() {
    let v1 = vec![1, 2, 3];

    let v1_iter = v1.iter();

    let total: i32 = v1_iter.sum();

    assert_eq!(total, 6);
}
```

Блок коду 13-13: виклик методу `sum` для отримання загальної суми усіх елементів ітератора

Нам не дозволено використовувати `v1_iter` після виклику `sum`, оскільки `sum` перебирає володіння ітератором, на якому його викликано.

Методи, що Створюють Інші Ітератори

Адаптери ітераторів - це методи, визначені для трейта `Iterator`, які не поглинають ітератор. Натомість вони створюють інші ітератори, змінюючи певний аспект оригінального ітератора.

Блок коду 13-14 показує приклад виклику метода-адаптора ітератора `map`, який приймає замикання, яке викличе для кожного елемента під час ітерації. Метод `map` повертає новий ітератор, який виробляє модифіковані елементи. Замикання створює новий ітератор, у якому кожен елемент вектора буде збільшено на 1:

Файл: src/main.rs

```
let v1: Vec<i32> = vec![1, 2, 3];

v1.iter().map(|x| x + 1);
```

Блок коду 13-14: Виклик адаптора `map` для створення нового ітератора

Однак, цей код видає попередження:

```
$ cargo run
  Compiling iterators v0.1.0 (file:///projects/iterators)
warning: unused `Map` that must be used
--> src/main.rs:4:5
   |
4 |     v1.iter().map(|x| x + 1);
   |     ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
   |
= note: `[warn(unused_must_use)]` on by default
= note: iterators are lazy and do nothing unless consumed

warning: `iterators` (bin "iterators") generated 1 warning
Finished dev [unoptimized + debuginfo] target(s) in 0.47s
Running `target/debug/iterators`
```

Код у Блоці коду 13-14 нічого не робить; замикання, яке ми вказали, ніколи не було викликано. Попередження нагадує нам, чому: адаптори ітераторів ліниві, і нам потрібно поглинути ітератор.

Щоб виправити це попередження і поглинути ітератор, ми використаємо метод `collect`, який ми використовували у Розділі 12 із `env::args` у Блоці коду 12-1. Цей метод поглинає ітератор і збирає отримані в результаті значення в колекцію.

У Блоці коду 13-15 ми зібрали результати ітерування по ітератору, повернутому викликом `map`, у вектор. Цей вектор в результаті міститиме всі елементи оригінального вектора, збільшені на 1.

Файл: `src/lib.rs`

```
let v1: Vec<i32> = vec![1, 2, 3];

let v2: Vec<_> = v1.iter().map(|x| x + 1).collect();

assert_eq!(v2, vec![2, 3, 4]);
```

Блок коду 13-15: виклик методу `map` для створення нового ітератора і виклик методу `collect` для поглинання цього нового ітератора і створення вектора

Оскільки `map` приймає замикання, ми можемо вказати будь-яку операцію, яку хочемо виконати з кожним елементом. Це чудовий приклад того, як замикання дозволяють вам встановити певну поведінку, використовуючи поведінку ітерацій, надану трейтом `Iterator`.

Ви можете з'єднати багато викликів адаптерів ітераторів для виконання складних дій, щоб це було читабельно. Але оскільки всі ітератори є ледачими, ви маєте викликати один з

методів, що поглинають адаптер, щоб отримати результати викликів адаптерів ітераторів.

Використання Замикань, що Захоплюють Своє Середовище

Багато адаптерів ітераторів приймають аргументами замикання, і зазвичай замикання, які ми вказуємо аргументами до адаптерів ітераторів будуть замиканнями, що захоплюють своє середовище.

Для цього прикладу ми скористаємося методом `filter`, що приймає замикання. Замикання отримає елемент з ітератора і повертає `bool`. Якщо замикання повертає `true`, значення буде включено в ітерації, вироблені `filter`. Якщо замикання повертає `false`, значення не буде включено.

У Блоці коду 13-16 ми використовуємо `filter` із замиканням, яке захоплює змінну `shoe_size` зі свого середовища для ітерування по колекції екземплярів структур `Shoe`. Воно поверне лише взуття зазначеного розміру.

Файл: `src/lib.rs`

```
#[derive(PartialEq, Debug)]
struct Shoe {
    size: u32,
    style: String,
}

fn shoes_in_size(shoes: Vec<Shoe>, shoe_size: u32) -> Vec<Shoe> {
    shoes.into_iter().filter(|s| s.size == shoe_size).collect()
}

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

    #[test]
    fn filters_by_size() {
        let shoes = vec![
            Shoe {
                size: 10,
                style: String::from("sneaker"),
            },
            Shoe {
                size: 13,
                style: String::from("sandal"),
            },
            Shoe {
                size: 10,
                style: String::from("boot"),
            },
        ];

        let in_my_size = shoes_in_size(shoes, 10);

        assert_eq!(
            in_my_size,
            vec![
                Shoe {
                    size: 10,
                    style: String::from("sneaker")
                },
                Shoe {
                    size: 10,
                    style: String::from("boot")
                },
            ]
        );
    }
}
```

Блок коду 13-16: використання методу `filter` із замиканням, що захоплює `shoe_size`

Функція `shoes_in_size` приймає володіння вектором взуття і розмір взуття. Вона повертає вектор, що містить лише взуття зазначеного розміру.

У тілі `shoes_in_size` ми викликаємо `into_iter` для створення ітератора, що приймає володіння вектором. Тоді ми викликаємо `filter`, щоб адаптувати ітератор у новий ітератор, що містить лише елементи, для яких замикання повертає `true`.

Замикання захоплює параметр `shoe_size` із середовища і порівнює значення із розміром кожної пари взуття, лишаючи тільки взуття зазначеного розміру. Нарешті, виклик `collect` збирає значення, повернуті адаптованим ітератором, у вектор, який функція повертає.

Тест показує, що коли ми викликаємо `shoes_in_size`, ми отримуємо назад лише взуття, яке має розмір, що дорівнює вказаному значенню.

Покращення Нашого Проєкту з Вводом/Виводом

Використовуючи нові знання про ітератори, ми можемо покращити проєкт введення/виведення у Розділі 12, використовуючи ітератори, щоб зробити деякі місця в коді яснішими та виразнішими. Погляньмо, як ітератори можуть поліпшити нашу реалізацію функцій `Config::build` і `search`.

Видалення Викликів `clone` з Ітератором

У Блоці коду 12-6 ми додали код, що бере слайс зі значень `String` і створили екземпляр структури `Config` індексуванням слайса і клонуванням значень, дозволивши структурі `Config` володіти цими значеннями. У Блоці коду 13-17 ми відтворили реалізацію функції `Config::build` такою, як у Блоці коду 12-23:

Файл: `src/lib.rs`

```
impl Config {
    pub fn build(args: &[String]) -> Result<Config, &'static str> {
        if args.len() < 3 {
            return Err("not enough arguments");
        }

        let query = args[1].clone();
        let file_path = args[2].clone();

        let ignore_case = env::var("IGNORE_CASE").is_ok();

        Ok(Config {
            query,
            file_path,
            ignore_case,
        })
    }
}
```

Блок коду 13-17: відтворення функції `Config::build` з Блоку коду 12-23

Тоді ми казали не хвилюватися через неефективні виклики `clone`, оскільки ми видалимо їх е майбутньому. Що ж, цей час настав!

Нам тут потрібен `clone`, тому що ми маємо слайс з елементами `String` у параметрі `args`, але функція `build` не володіє `args`. Щоб повернути володіння екземпляром `Config`, нам

довелося клонувати значення з полів `query` та `file_path` з `Config`, щоб екземпляр `Config` міг володіти своїми значеннями.

За допомогою нових знань про ітератори ми можемо змінити функцію `build`, щоб вона брала володіння ітератором як аргумент замість позичати слайс. Ми використовуємо функціональність ітератора замість коду, що перевіряє довжину слайса і індексує конкретні місця. Це прояснить, що саме робить функція `Config::build`, бо доступ до значень забезпечуватиме ітератор.

Коли `Config::build` прийме володіння ітератором та припинить використовувати операції індексації, що позичають, ми зможемо перемістити значення `String` з ітератора в `Config` замість викликати `clone` і робити новий розподіл пам'яті.

Безпосереднє Використання Повернутого Ітератора

Відкрийте файл `src/main.rs` з нашого проєкту введення/виведення, що виглядає ось так:

Файл: `src/main.rs`

```
fn main() {
    let args: Vec<String> = env::args().collect();

    let config = Config::build(&args).unwrap_or_else(|err| {
        eprintln!("Problem parsing arguments: {err}");
        process::exit(1);
    });

    // --snip--
}
```

Ми спочатку змінимо початок функції `main`, яка була в нас у Блоці коду 12-24, на код з Блоку коду 13-18, який на цей раз використовує ітератор. Це не буде компілюватися, доки ми не оновимо також `Config::build`.

Файл: `src/main.rs`

```
fn main() {
    let config = Config::build(env::args()).unwrap_or_else(|err| {
        eprintln!("Problem parsing arguments: {err}");
        process::exit(1);
    });

    // --snip--
}
```

Блок коду 13-18: передавання значення, повернутого `env::args`, до `Config::build`

Функція `env::args` повертає ітератор! Замість того, щоб збирати значення ітератора до вектора і передавання слайс до `Config::build`, тепер ми передаємо володіння ітератором, повернутим з `env::args`, напямую до `Config::build`.

Далі, нам потрібно оновити визначення `Config::build`. У файлі `src/lib.rs` вашого проекту введення/виведення змінимо сигнатуру `Config::build`, щоб вона виглядала як у Блоці коду 13-19. Це все ще не компілюється, оскільки нам потрібно оновити тіло функції.

Файл: `src/lib.rs`

```
impl Config {  
    pub fn build(  
        mut args: impl Iterator<Item = String>,  
    ) -> Result<Config, &'static str> {  
        // --snip--  
    }  
}
```

Блок коду 13-19: оновлення сигнатури `Config::build`, щоб приймала ітератор

The standard library documentation for the `env::args` function shows that the type of the iterator it returns is `std::env::Args`, and that type implements the `Iterator` trait and returns `String` values.

Ми оновили сигнатуру функції `Config::build`, зробивши параметр `args` узагальненого типу з обмеженням трейту `impl Iterator<Item = String>` замість `&[String]`. Цей синтаксис `impl Trait`, який ми обговорили у підрозділі “Трейти як параметри” Розділу 10, означає, що `args` може бути будь-якого типу, що реалізує тип `Iterator` і повертає елементи типу `String`.

Оскільки ми беремо володіння `args` і ми будемо змінювати `args`, ітеруючи крізь нього, ми можемо додати ключове слово `mut` в специфікацію параметра `args`, щоб зробити його мутабельним.

Використання Методів Трейту `Iterator` Замість Індексуювання

Далі ми виправимо тіло `Config::build`. Оскільки `args` реалізує трейт `Iterator`, ми знаємо, що можемо викликати для нього метод `next`! Блок коду 13-20 оновлює код зі Блоку коду 12-23, використовуючи метод `next`:

Файл: `src/lib.rs`

```
impl Config {
    pub fn build(
        mut args: impl Iterator<Item = String>,
    ) -> Result<Config, &'static str> {
        args.next();

        let query = match args.next() {
            Some(arg) => arg,
            None => return Err("Didn't get a query string"),
        };

        let file_path = match args.next() {
            Some(arg) => arg,
            None => return Err("Didn't get a file path"),
        };

        let ignore_case = env::var("IGNORE_CASE").is_ok();

        Ok(Config {
            query,
            file_path,
            ignore_case,
        })
    }
}
```

Блок коду 13-20: зміна тіла `Config::build` з використанням методів ітератора

Згадайте, що перша стрічка в значенні, яке повертає `env::args`, є назвою програми. Ми хочемо проігнорувати його і дістатися до наступного значення, тож спершу викличемо `next` і нічого не зробимо з поверненим значенням. По-друге, ми викликаємо `next`, щоб отримати значення, ми хочемо вставити в поле `Config query`. Якщо `next` повертає `Some`, ми використовуємо `match`, щоб витягти значення. Якщо вона повертає `None`, це означає, що було недостатньо аргументів, і ми достроково виходимо, повертаючи значення `Err`. Те саме ми робимо і зі значенням `file_path`.

Поліпшення Зрозумілості Коду з Адаптерами Ітераторів

Ми також можемо скористатися ітераторами у функції `search` у нашому проєкті введення/виведення, який відтворений тут у Блоці коду 13-21 таким, як він був 12-19:

Файл: `src/lib.rs`

```
pub fn search<'a>(query: &str, contents: &'a str) -> Vec<&'a str> {
    let mut results = Vec::new();

    for line in contents.lines() {
        if line.contains(query) {
            results.push(line);
        }
    }

    results
}
```

Блок коду 13-21: реалізація функції `search` з Блоку коду 12-19

Ми можемо зробити цей код чіткішим за допомогою методів-адаптерів ітераторів. Це також дозволить нам уникнути проміжного мутабельного вектору `results`. Функціональний стиль програмування надає перевагу мінімізації кількості мутабельних станів, щоб зробити код чистішим. Видалення мутабельного стану може уможливити подальше покращення для здійснення паралельного пошуку, оскільки ми не змогли б керувати одночасним доступом до вектора `results`. Блок коду 13-22 показує ці зміни:

Файл: `src/lib.rs`

```
pub fn search<'a>(query: &str, contents: &'a str) -> Vec<&'a str> {
    contents
        .lines()
        .filter(|line| line.contains(query))
        .collect()
}
```

Блок коду 13-22: використання методів-адаптерів ітераторів у реалізації функції `search`

Згадайте, що призначення функції `search` - повернути всі рядки в `contents`, що містять `query`. Так само як у прикладі `filter` з Блоку коду 13-16, цей код використовує адаптер `filter` для збереження тільки тих рядків, для яких `line.contains(query)` повертає `true`. Потім ми збираємо відповідні рядки у інший вектор за допомогою `collect`. Набагато простіше! Можете самі спробувати внести аналогічні зміни з використанням методів ітератора у функцію `search_case_insensitive`.

Вибір між Циклами та Ітераторами

Наступне логічне питання - який стиль вам слід обрати у вашому власному коді й чому: оригінальна реалізація з Блоку коду 13-21 чи версія з ітераторами з Блоку коду 13-22. Більшість програмістів Rust вважають за краще використовувати ітераторний стиль. До

нього дещо складніше призвичаїтися в перший час, але відколи ви набудете відчуття різноманітних ітераторів і що вони роблять, ітератори стають простішими для розуміння. Замість того, щоб займатися дрібними уточненнями в циклі й збирати нові вектори, код зосереджується на високорівневій меті циклу. Це дозволяє абстрагуватися від деякого банального коду, щоб легше було побачити концепції, унікальні для цього коду, такі як умови фільтрації, яку має пройти кожен елемент ітератора.

Але чи ці дві реалізації дійсно еквівалентні? Інтуїтивне припущення може казати, що більш низькорівневий цикл буде швидшим. Поговорімо про швидкодію.

Порівняння Швидкодії: Цикли Проти Ітераторів

Щоб визначити, використовувати цикли чи ітератори, вам треба знати, яка реалізація швидша: версія функції `search` з явним циклом `for` чи версія з ітераторами.

Ми запустили бенчмарк, завантаживши повний текст "*Пригод Шерлока Голмса*" сера Артура Конана Дойла в `String` і шукаючи там слово *the*. Ось результати бенчмарка на версії `search`, що використовує цикл `for`, і версії, що використовує ітератори:

```
test bench_search_for ... bench: 19,620,300 ns/iter (+/- 915,700)
test bench_search_iter ... bench: 19,234,900 ns/iter (+/- 657,200)
```

Версія з ітератором виявилася трохи швидшою! Ми не будемо тут пояснювати код бенчмарка, бо мета не довести, що дві версії еквівалентні, а отримати загальне уявлення про те, як ці дві реалізації порівнюються з точки зору продуктивності.

Для повнішого бенчмарку слід перевіряти, використовуючи як `contents` різні тексти різного розміру, різні слова і слова різної довжини як `query`, і всілякі інші варіації. Річ у тому, що ітератори, хоча і є високорівневою абстракцією, компілюються приблизно до приблизно такого ж коду, як ніби ви самі писали низькорівневий код. Ітератори - це одна з *абстракцій нульової вартості* Rust, що означає, що абстракція не накладає додаткових витрат часу виконання. Це аналогічно тому, як Б'ярне Строуструп, оригінальний дизайнер і реалізатор C++, визначає *нульові витрати* в "Основах C++" (2012):

Загалом, реалізації C++ підкоряються принципу нульових витрат: якщо ви чогось не використовуєте, то не платите за це. І більше: те, що ви використовуєте, ви не змогли запрограмувати вручну краще.

Як інший приклад, наступний код взятий з аудіо декодера. Алгоритм декодування використовує математичну операцію лінійного прогнозування для оцінки майбутніх значень на основі лінійної функції попередніх зразків. Цей код використовує ланцюг ітераторів для виконання математичних операцій на трьох змінних в області видимості: слайсі даних `buffer`, масиві з 12 `coefficients`, і значенні, на яке треба зсунути дані `qlp_shift`. Ми оголосили змінні, що знаходяться в цьому прикладі, але не дали їм жодних значень; хоча цей код не має особливого сенсу поза своїм контекстом, він є реальним лаконічним прикладом того, як Rust переносить високорівневі ідеї в низькорівневий код.


```
let buffer: &mut [i32];
let coefficients: [i64; 12];
let qlp_shift: i16;

for i in 12..buffer.len() {
    let prediction = coefficients.iter()
        .zip(&buffer[i - 12..i])
        .map(|(&c, &s)| c * s as i64)
        .sum::<i64>() >> qlp_shift;

    let delta = buffer[i];
    buffer[i] = prediction as i32 + delta;
}
```

Щоб обчислити значення `prediction`, цей код ітерує через кожне з 12 значень у `coefficients` і використовує метод `zip` для з'єднання значень коефіцієнтів з попередніми 12 значеннями з `buffer`. Потім для кожної пари ми множимо значення, додаємо усі результати, і зсуваємо біти в сумі на `qlp_shift` бітів праворуч.

Обчислення в застосунках на кшталт аудіо декодерів часто найвище цінують швидкодію. Тут ми створюємо ітератор, використовуючи два адаптери, а потім поглинаємо значення. У який асемблерний код скомпілюється цей код Rust? Ну, якщо щодо цього коду, то він компілюється у такий же асемблерний код, який ви б написали самі. Циклу, що відповідає ітераціям по значеннях у `coefficients`, не буде взагалі: Rust знає, що буде всього 12 ітерацій, тож він "розгортає" цикл. *Розгортання* - це оптимізація, яка видаляє накладні витрати на код, що керує циклом, а замість цього генерує код із повтореннями для кожної ітерації циклу.

Всі коефіцієнти зберігаються в регістрах, тобто доступ до значень є дуже швидким. Немає перевірок виходу за межі при доступі до масиву під час виконання. Всі ці оптимізації, які Rust може застосувати, роблять згенерований код надзвичайно ефективним. Тепер, коли ви це знаєте, ви можете використовувати ітератори та замикання без страху! Вони роблять код схожим на високорівневий, але не накладають за це штрафів на швидкодію часу виконання.

Підсумок

Замикання та ітератори - це особливості Rust, натхнені ідеями мов функціонального програмування. Вони сприяють здатності Rust чітко виражати високорівневі ідеї при низькорівневій швидкодії. Реалізація замикань та ітераторів така, що швидкодія часу виконання не страждає. Це - частина мети Rust прагнути забезпечити абстракції нульової вартості.

Тепер, коли ми покращили виразність нашого проєкту введення/виведення, подивімося на деякі додаткові можливості `cargo`, які допоможуть нам поділитися проєктом зі світом.

Більше про Cargo та Crates.io

Досі ми використовували тільки основний функціонал Cargo для збірки, запуску та тестування, але він може робити набагато більше. В цьому розділі ми обговоримо дещо з решти його більш просунутого функціонала, щоб показати вам, як робити наступне:

- Налаштування вашої збірки із release профілями
- Публікація бібліотек на crates.io
- Організація великих проєктів з робочими областями
- Встановлення з crates.io
- Розширення Cargo, використовуючи користувацькі команди

Cargo може робити набагато більше ніж функціонал, який ми охопили в цьому розділі. Для отримання повного пояснення його функціоналу дивіться [документацію](#).

Налаштування Збірок з Release Профілями

В Rust, *release профілі* є попередньо визначені та настроюваними профілями з різними конфігураціями які дозволяють програмісту мати більше контролю над різними опціями для компіляції коду. Кожен профіль налаштований незалежно від інших.

Cargo має два основні профілі: профіль `dev` який використовується під час запуску `cargo build` і профіль `release`, який використовується під час запуску `cargo build --release`. Профіль `dev` визначено з хорошими параметрами за замовчуванням для розробки і профіль `release` має хороші параметри за замовчуванням для збірки для випуску.

Ці імена профілів можуть бути знайомі з виводу ваших збірок:

```
$ cargo build
  Finished dev [unoptimized + debuginfo] target(s) in 0.0s
$ cargo build --release
  Finished release [optimized] target(s) in 0.0s
```

`dev` і `release` це різні профілі, які використовуються компілятором.

Cargo має налаштування за замовчуванням для кожного профілю, який застосовується, навіть, якщо ви ще явно не додали жодної секції `[profile.*]` в файл *Cargo.toml* проєкту. Додаючи секції `[profile.*]` будь-якому профілю, який ви бажаєте налаштувати, ви перевизначаєте будь-яку підмножину налаштувань за замовчуванням. Наприклад, ось значення за замовчуванням налаштування `opt-level` для профілів `dev` і `release`:

Файл: Cargo.toml

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

[profile.release]
opt-level = 3
```

Налаштування `opt-level` контролює кількість оптимізації, яку Rust буде застосовувати до вашого коду з діапазоном від 0 до 3. Чим більше оптимізацій застосовується, тим довше стає час компіляції, тому якщо ви часто компілюєте код під час розробки, вам знадобиться менше оптимізацій, щоб компілювати швидше, навіть якщо отриманий код повільніше. Тому `opt-level` для `dev` за замовчуванням `0`. Коли ви готові до випуску вашого коду, краще витратити більше часу на компіляцію. Ви будете компілювати в режимі випуску тільки раз, але ви запускатимете скомпільовану програму багато разів, тому режим

випуску обмінює довший час компіляції на швидший код. Саме тому `opt-level` для профілю `release` за замовчуванням `3`.

Ви можете перевизначити налаштування за замовчуванням, додав інше значення в *Cargo.toml*. Наприклад, якщо ми хочемо використовувати оптимізацію 1-го рівня в профілі розробки, ми можемо додати ці два рядки в файл *Cargo.toml* нашого проєкту:

Файл: Cargo.toml

```
[profile.dev]
opt-level = 1
```

Цей код перевизначає налаштування за замовчуванням `0`. Тепер, коли ми запустимо `cargo build`, Cargo буде використовувати за замовчуванням профіль `dev` плюс наше налаштування `opt-level`. Оскільки ви встановили `opt-level` на `1`, Cargo буде застосовувати більше оптимізацій за замовчуванням, але не настільки багато, як в режимі випуску.

Для повного списку опцій конфігурації та параметрів кожного профілю за замовчуванням дивіться [Документацію Cargo](#).

Публікація Крейта на Crates.io

Ми використовували пакети з crates.io як залежності проекту, але ви також можете поділитися своїм кодом з іншими людьми, опублікувавши ваші власні пакети. Реєстр крейтів на crates.io поширює початковий код ваших пакетів, тому він в першу чергу розміщує open-source код.

Rust і Cargo мають функціонал, який полегшує пошук та використання вашого опублікованого пакета. Далі ми поговоримо про деякі з цих можливостей і потім пояснимо, як опублікувати пакет.

Створення Корисних Документаційних Коментарів

Якісне документування ваших пакетів допоможе іншим користувачам знати, як і коли їх використовувати, тому варто вкладати час на написання документації. У Розділі 3, ми обговорювали як коментарі Rust коду використовують подвійний слеш, `//`. Rust також має особливий вид коментарів для документації, зручно відомий як *документаційні коментарі*, які також будуть створювати HTML документацію. HTML покаже зміст документаційних коментарів для елементів публічного API, розрахованих на програмістів, які зацікавлені в *використанні* вашого крейту на відміну від того, як ваш крейт *імplementовано*.

Документаційні коментарі використовують три слеші, `///`, замість двох та підтримують Markdown для форматування тексту. Розміщуйте документаційні коментарі безпосередньо перед елементом, який вони документують. Блок коду 14-1 показує документаційні коментарі для функції `add_one` крейту з назвою `my_crate`.

Файл: `src/lib.rs`

```

/// Adds one to the number given.
///
/// # Examples
///
/// ```
/// let arg = 5;
/// let answer = my_crate::add_one(arg);
///
/// assert_eq!(6, answer);
/// ```
pub fn add_one(x: i32) -> i32 {
    x + 1
}

```

Блок Коду 14-1: Коментар документації для функції

Тут ми дамо опис того, що робить функція `add_one`, почнемо розділ з заголовком `Приклади`, і надамо код, який продемонструє, як використовувати функцію `add_one`. Ми можемо створити HTML документацію з документаційних коментарів, запустивши `cargo doc`. Ця команда запускає інструмент `rustdoc`, який поширюється з Rust і кладе згенеровану HTML документацію в директорії `target/doc`.

Для зручності, запуск `cargo doc --open` збере HTML для Вашої поточної документації (а також документації для всіх залежностей вашого крейту) і відкриє результат у браузері. Перейдіть до функції `add_one` та ви побачите, як текст коментарів документації відтворюється, як показано на Малюнку 14-1:

my_crate

Functions

add_one

Crates

my_crate

Click or press 'S' to search, '?' for more options...

Function my_crate::**add_one** [–][src]

```
pub fn add_one(x: i32) -> i32
```

[–] Adds one to the number given.

Examples

```
let arg = 5;
let answer = my_crate::add_one(arg);

assert_eq!(6, answer);
```

Малюнок 14-1: HTML документація для функції `add_one`

Найпоширеніші Розділи

Ми використовували Markdown заголовок `# Examples` в Блоці Коду 14-1 для створення секції в HTML з назвою “Examples.” Ось ще кілька секцій, які автори крейтів зазвичай використовують у своїх документаціях:

- **Паніки:** Сценарії, в яких документована функція може запанікувати. Користувачі, які будуть використовувати ці функції і які не хочуть, щоб їх програма панікувала, повинні бути впевнені, що вони не викликають функції в цих ситуаціях.
- **Помилки:** Якщо функція повертає `Result`, який описує різновиди можливих помилок та які умови можуть призвести до повернення цих помилок, користувачам функції може бути корисно, щоб вони могли написати код для обробки різних помилок різними способами.
- **Безпека:** Якщо функція `unsafe` (ми обговоримо небезпечність в Розділі 19), то має бути секція, в якій пояснюється, чому функція небезпечна та її інваріанти, які мають дотримуватися користувачі функції.

Більшості документаційних коментарів не потрібні всі ці секції, але це хороший контрольний список, щоб нагадувати вам аспекти вашого коду, про які вашим користувачам буде цікаво дізнатися.

Документаційні Коментарі як Тести

Додавання прикладу блоків коду в ваші коментарі документації може допомогти продемонструвати, як ви використовуєте вашу бібліотеку, і в цьому є додатковий бонус: запуск `cargo test` запускатиме приклади коду в вашій документації як тести! Немає нічого приємнішого ніж документація з прикладами. Але немає нічого гіршого ніж приклади, які не працюють, бо код змінився з моменту написання документації. Якщо ми запустимо `cargo test` із документацією для функції `add_one` з Блока Коду 14-1, ми побачимо секцію результатів тестів наступним чином:

```
Doc-tests my_crate
```

```
running 1 test
```

```
test src/lib.rs - add_one (line 5) ... ok
```

```
test result: ok. 1 passed; 0 failed; 0 ignored; 0 measured; 0 filtered out;  
finished in 0.27s
```

А тепер, якщо ми змінимо або функцію або приклад, щоб `assert_eq!` в прикладі запанікував та знову виконаємо `cargo test`, ми побачимо, що документаційні тести

зловили, що приклад та код не синхронізовані між собою!

Коментування Вміщених Елементів

Стиль документаційних коментарів `///!` додає документацію до елемента, який містить коментарі, аніж до елементів, що слідує за коментарями. Як правило, ми використовуємо документаційні коментарі всередині кореневого файлу крейта (`src/lib.rs` за домовленістю) або всередині модуля для документування крейта або модуля в цілому.

Наприклад, для додавання документації, яка описує мету крейту `my_crate` який містить функцію `add_one`, ми додамо документаційні коментарі, які починаються з `///!` на початку файлу `src/lib.rs`, як показано в Блоці Коду 14-2:

Файл: `src/lib.rs`

```
///! # My Crate
///!
///! `my_crate` is a collection of utilities to make performing certain
///! calculations more convenient.

///! Adds one to the number given.
///! --snip--
```

Блок Коду 14-2: Повноцінна документація крейту `my_crate`

Зауважте, що тут немає коду після останнього рядку, який починається з `///!`. Оскільки ми почали коментар з `///!` замість `///`, ми документуємо предмет який міститься в коментарі, замість предмета, який слідує за коментарем. У цьому випадку, цей елемент це файл `src/lib.rs`, який містить кореневий каталог. Ці коментарі описують увесь крейт.

Коли ви запускаєте `cargo doc --open`, ці коментарі будуть показані на головній сторінці документації `my_crate` вище списку публічних елементів крейту, як показано на Рисунку 14-2:

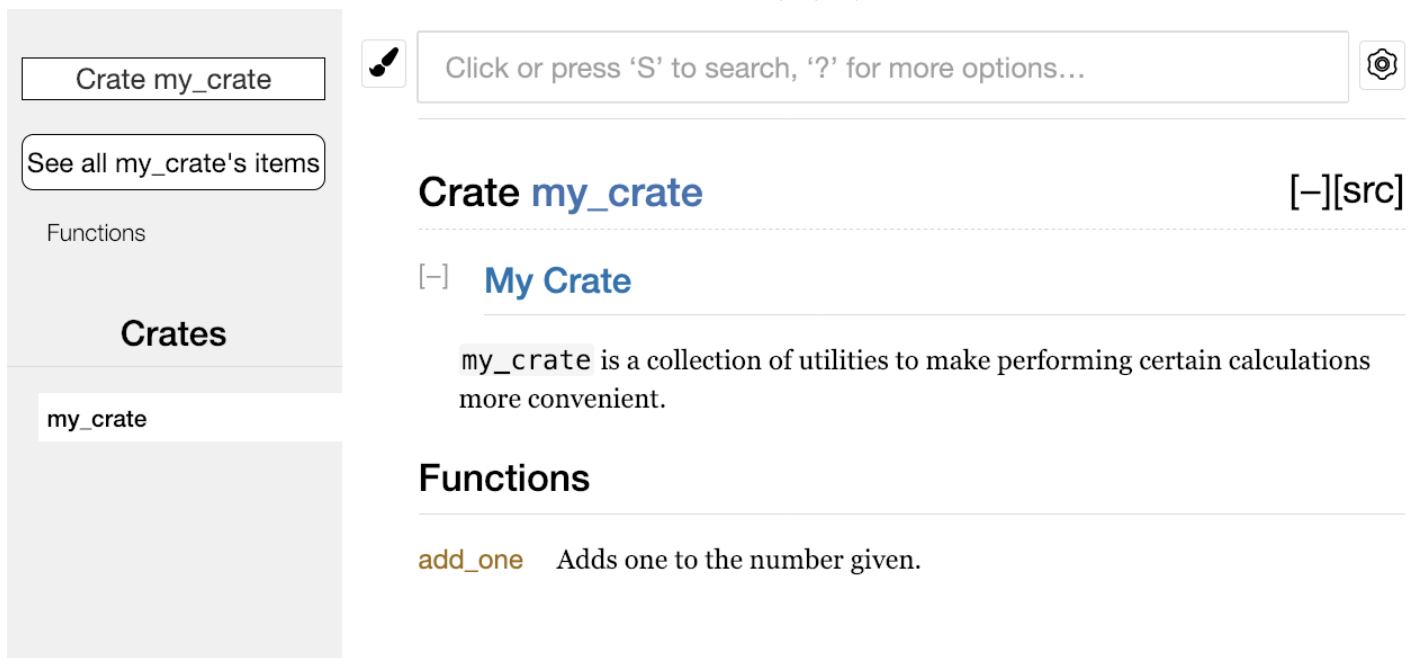


Рисунок 14-2: Оброблена документація `my_crate`, включно з коментарем, який описує крейт в цілому

Документаційні коментарі всередині елементів корисні для опису крейтів і особливо модулів. Використовуйте їх, щоб пояснювати загальну мету контейнера, щоб допомогти вашим користувачам зрозуміти організацію крейту.

Експорт Зручного Публічного API з `pub use`

Структура вашого публічного API має вирішальне значення, коли ви публікуєте крейт. Користувачі вашого крейту менш знайомі зі структурою ніж ви та можуть мати труднощі у пошуку бажаних частин, якщо у вашому крейті велика ієрархія модулів.

У Розділі 7 ми розглянули, як робити елементи публічними за допомогою ключового слова `pub` та вносити елементи в область видимості з ключовим словом `use`. Однак, структура, яка має сенс під час розробки крейту, може бути не дуже зручна для ваших користувачів. Ви можливо захочете організувати ваші структури в багаторівневій ієрархії, але потім люди які захочуть використати визначений глибоко в ієрархії тип можуть мати проблеми з з'ясуванням, що цей тип існує. Вони також можуть бути роздратованими через необхідність писати `use my_crate::some_module::another_module::UsefulType;` замість `use my_crate::UsefulType;`.

Хороші новини полягають в тому, що якщо іншим користувачам *не* зручно використовувати її з іншої бібліотеки, вам не потрібно переробляти вашу внутрішню організацію: натомість, ви можете повторно експортувати елементи, щоб зробити публічну структуру, яка відрізняється від вашої приватної структури використанням `pub`

`use`. Повторне експортування бере публічний елемент з одного місця і робить його публічним в іншому місці, ніби це було визначено в іншій локації.

Скажімо, ми зробили бібліотеку з назвою `art` для моделювання художніх концепцій. Всередині цієї бібліотеки є два модулі: модуль `kinds`, який містить два енуми із назвами `PrimaryColor` та `SecondaryColor` і модуль `utils`, який містить функцію `mix`, як показано в Блоці Коду 14-3:

Файл: `src/lib.rs`

```
#![ # Art
#![
#![ A library for modeling artistic concepts.

pub mod kinds {
    /// The primary colors according to the RYB color model.
    pub enum PrimaryColor {
        Red,
        Yellow,
        Blue,
    }

    /// The secondary colors according to the RYB color model.
    pub enum SecondaryColor {
        Orange,
        Green,
        Purple,
    }
}

pub mod utils {
    use crate::kinds::*;

    /// Combines two primary colors in equal amounts to create
    /// a secondary color.
    pub fn mix(c1: PrimaryColor, c2: PrimaryColor) -> SecondaryColor {
        // --snip--
    }
}
```

Блок Коду 14-3: Бібліотека `art` з елементами організованими в модулі `kinds` та `utils`

Рисунок 14-3 показує, як буде виглядати головна сторінка документації цього крейта згенерована з `cargo doc`:

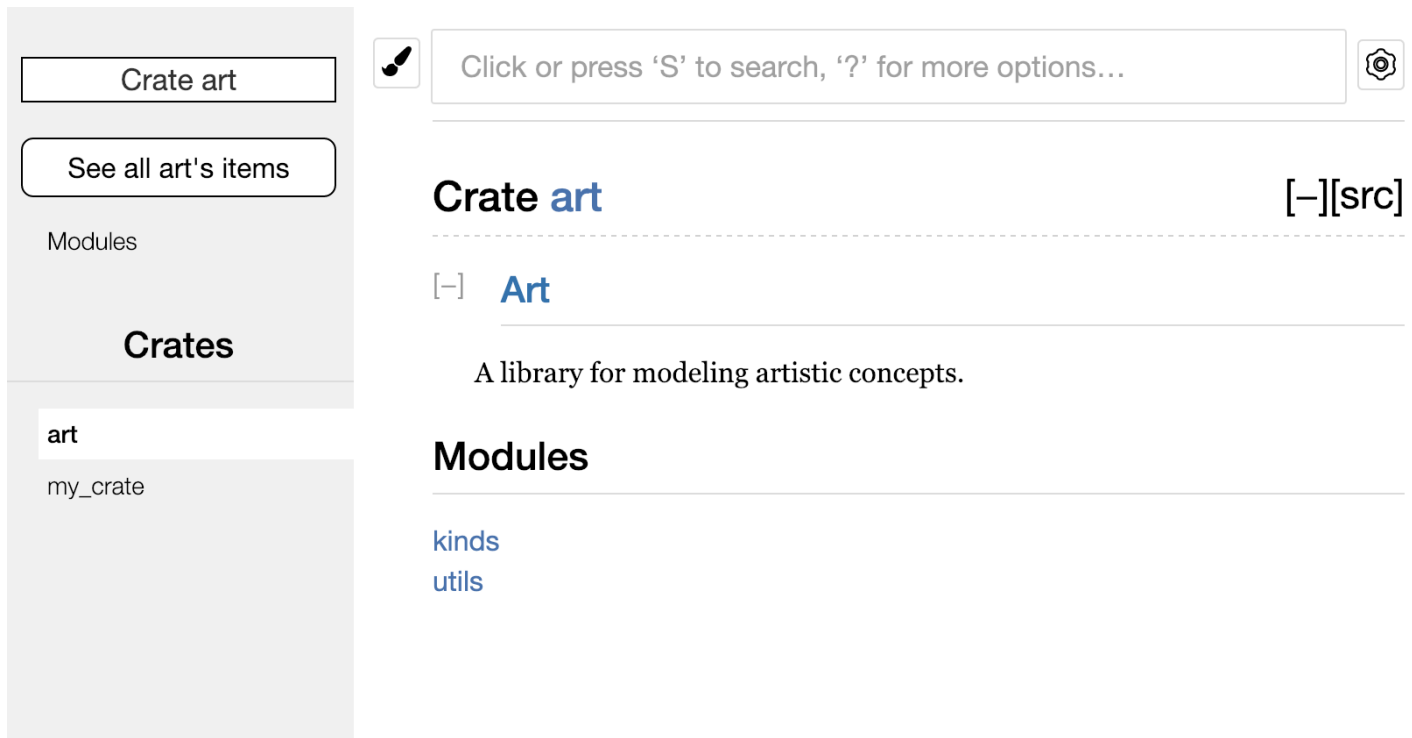


Рисунок 14-3: Головна сторінка документації `art` зі списком модулів `kinds` and `utils`

Зауважте, що типи `PrimaryColor` та `SecondaryColor` не вказані на головній сторінці, так само як і функція `mix`. Нам потрібно натиснути на `kinds` та `utils`, щоб побачити їх.

Іншому залежному від цієї бібліотеки крейту знадобиться інструкція `use`, яка приносить елементи з `art` в область видимості, вказавши визначену зараз структуру модуля. Блок коду 14-4 показує приклад крейта, який використовує елементи `PrimaryColor` та `mix` з крейту `art`:

Файл: `src/main.rs`

```
use art::kinds::PrimaryColor;
use art::utils::mix;

fn main() {
    let red = PrimaryColor::Red;
    let yellow = PrimaryColor::Yellow;
    mix(red, yellow);
}
```

Блок Коду 14-4: Крейт використовує елементи крейту `art` із його експортованою внутрішньою структурою

Автор коду в Блоці Коду 14-4, який використовує `art` крейт, має з'ясувати, що `PrimaryColor` в модулі `kinds`, а `mix` в модулі `utils`. Структура модуля `art` крейту є більш актуальною для розробників `art` крейту, ніж для його користувачів. Внутрішня

структура не містить жодної корисної інформації для когось, хто намагається зрозуміти, як використовувати `art` крейт, радше викликає плутанину, бо розробники, які використовують цей крейт мають з'ясовувати, куди дивитися та мають вказувати назву модуля в інструкції `use`.

Щоб видалити внутрішню організацію з публічного API, ми можемо змінити код крейту `art`, як показано в Блоці Коду 14-3, щоб додати інструкції `pub use` для повторного експорту елементів на вищий рівень, як показано в Блоці Коду 14-5:

Файл: `src/lib.rs`

```
//! # Art
//!
//! A library for modeling artistic concepts.

pub use self::kinds::PrimaryColor;
pub use self::kinds::SecondaryColor;
pub use self::utils::mix;

pub mod kinds {
    // --snip--
}

pub mod utils {
    // --snip--
}
```

Блок Коду 14-5: Додавання інструкції `pub use` для повторного експорту елементів

Документація API, яку `cargo doc` створює для цього крейту, тепер буде мати повторно експортований список та посилання на головній сторінці, як показано на Малюнку 14-4, полегшуючи пошук типів `PrimaryColor` та `SecondaryColor` і функції `mix`.

Рисунок 14-4: Головна сторінка документації `art`, яка має список повторно експортованих елементів

Користувачі крейту `art` все ще можуть бачити і використовувати внутрішню структуру з Блоку Коду 14-3, як продемонстровано в Блоці Коду 14-4, або вони можуть використовувати більш зручну структуру з Блоку Коду 14-5, як показано в Блоці Коду 14-6:

Файл: `src/main.rs`

```
use art::mix;
use art::PrimaryColor;

fn main() {
    // --snip--
}
```

Блок Коду 14-06: Програма, яка використовує експортовані елементи з крейту `art`

У випадках, коли є багато вкладених модулів, повторне експортування типів на вищій рівень із `pub use` може зробити суттєву різницю в досвіді використання цього крейта. Ще одним поширеним використанням `pub use` є повторне експортування визначень

залежностей в поточному крейті, щоб зробити визначення цього крейту частиною публічного API.

Створення корисної публічної структури API є скоріше мистецтвом ніж наукою, і ви можете ітерувати, щоб знайти API яке найкраще підходить для ваших користувачів. Вибір `pub use` дає вам гнучкість у тому, як ви внутрішньо структуруєте свій крейт та відділяє внутрішню структуру від того, що ви представляєте своїм користувачам. Подивімося на деякий код з встановлених вами крейтів, щоб побачити, чи їх структура відрізняється від їх публічного API.

Налаштування Облікового Запису Crates.io

Перш ніж ви зможете опублікувати якісь крейти, вам потрібно створити обліковий запис на crates.io і отримати API токен. Для цього, відвідайте домашню сторінку на crates.io і увійдіть за допомогою вашого облікового запису Github. (Обліковий запис на GitHub наразі є вимогою, але сайт може підтримувати інші способи створення облікового запису в майбутньому.) Після входу перейдіть до налаштувань облікового запису на <https://crates.io/me/> і отримайте ваш API ключ. Потім запустить команду `cargo login` із вашим API токеном наступним чином:

```
$ cargo login abcdefghijklmnopqrstuvwxyz012345
```

Ця команда повідомить Cargo про ваш API токен та збереже його локально в `~/.cargo/credentials`. Зауважте, що токен це *секрет*: не діліться ним з будь-ким іншим. Якщо ви поділитесь ним з будь-ким задля будь-якої причини, ви можете відкликати його та створити новий токен на crates.io.

Додавання Метаданих до Нового Крейту

Скажімо ви маєте крейт який ви хочете опублікувати. Перед публікацією, вам буде потрібно додати деякі метадані в секції `[package]` файлу *Cargo.toml* вашого крейту.

Вашому крейту знадобиться унікальна назва. Поки ви працюєте над крейтом локально, ви можете називати його як завгодно. Однак, назва крейту на crates.io виділяється в порядку живої черги(перший прийшов - перший отримав). Як тільки назва крейту обрана, ніхто інший не може опублікувати крейт із цією назвою. Перед спробою опублікувати крейт, пошукайте назву, яку ви бажаєте використовувати. Якщо назва зайнята, вам знадобиться обрати іншу назву та редагувати поле `name` в файлі *Cargo.toml* в секції `[package]`, щоб використати нову назву для публікації наступним чином:

Файл: Cargo.toml

```
[package]
name = "guessing_game"
```

Навіть якщо ви обрали унікальну назву, коли ви запустите `cargo publish`, щоб опублікувати крейт, ви, наразі, отримаєте попередження та, потім, помилку:

```
$ cargo publish
    Updating crates.io index
warning: manifest has no description, license, license-file, documentation,
homepage or repository.
Перегляньте https://doc.rust-lang.org/cargo/reference/manifest.html#package-metadata для додаткових інформації.
--snip--
error: failed to publish to registry at https://crates.io

Caused by:
  the remote server responded with an error: missing or empty metadata fields:
  description, license. Будь ласка перегляньте https://doc.rust-lang.org/cargo/reference/manifest.html щодо того, як завантажити метадані
```

Це помилка, оскільки у вас відсутня деяка вирішальна інформація: опис та ліцензія які необхідні для того, щоб люди знали, що ваш крейт робить та на яких умовах вони можуть його використовувати. У *Cargo.toml*, додайте опис розміром з речення або два, оскільки воно з'явиться з вашим крейтом в результаті пошуку. Для поля `license` вам потрібно надати значення *ідентифікатора ліцензії*. [Linux Foundation's Software Package Data Exchange \(SPDX\)](#) перелічує ідентифікатори які ви можете використати як це значення. Наприклад, щоб вказати, що ваш крейт використовує ліцензію MIT, додайте ідентифікатор `MIT`:

Файл: Cargo.toml

```
[package]
name = "guessing_game"
license = "MIT"
```

Якщо ви хочете використовувати ліцензію, якої немає в SPDX, то вам потрібно місце де можна розмістити текст цієї ліцензії в файлі, включно із файлом вашого проєкту, а потім використайте поле `license-file`, щоб зазначити назву цього файлу замість ключа `license`.

Супровід щодо того, яка ліцензія підійде вашому проєкту, поза рамками цієї книги. Багато людей у спільноті Rust ліцензує їх проєкти так само як і Rust, використовуючи подвійну ліцензію `MIT OR Apache-2.0`. Ця практика демонструє, що ви також можете вказати декілька ідентифікаторів ліцензій, які відокремлені `OR`, щоб використовувати декілька ліцензій в вашому проєкті.

З унікальною назвою, версією, вашим описом і доданою ліцензією файл *Cargo.toml* для проєкту, готового до публікації, може виглядати так:

Файл: Cargo.toml

```
[package]
name = "guessing_game"
version = "0.1.0"
edition = "2021"
description = "A fun game where you guess what number the computer has chosen."
license = "MIT OR Apache-2.0"

[dependencies]
```

[Cargo's documentation](#) describes other metadata you can specify to ensure others can discover and use your crate more easily.

Публікація на Crates.io

Тепер, коли ви створили обліковий запис, зберегли ваш API токен, обрали назву вашого крейту та вказали необхідні метадані, ви готові до публікації! Публікація крейту завантажує конкретну версію на [crates.io](#) для використання іншими.

Будьте обережні, оскільки публікація *перманентна*. Версія ніколи не може бути перезаписана, а код ніколи не може бути видалений. Одна з основних цілей [crates.io](#) це діяти як перманентний архів коду, щоб збірка кожного проєкту залежного від крейту на [crates.io](#) продовжувала працювати. Дозволяючи видалення версій ми робимо виконання цієї цілі неможливим. Однак, немає обмежень на кількість версій крейтів, які ви можете опублікувати.

Запустимо знову команду `cargo publish`. Тепер воно має бути вдалим:

```
$ cargo publish
Updating crates.io index
Packaging guessing_game v0.1.0 (file:///projects/guessing_game)
Verifying guessing_game v0.1.0 (file:///projects/guessing_game)
Compiling guessing_game v0.1.0
(file:///projects/guessing_game/target/package/guessing_game-0.1.0)
Finished dev [unoptimized + debuginfo] target(s) in 0.19s
Uploading guessing_game v0.1.0 (file:///projects/guessing_game)
```

Вітаємо! Ви зараз поділилися вашим кодом із Rust спільнотою та кожен може легко додати ваш крейт як залежність до його проєкту.

Публікація Нової Версії Існуючого Крейта

Коли ви внесли зміни в ваш крейт та готові випустити нову версію ви змінюєте значення `version`, яке вказане в вашому файлі *Cargo.toml* та знову публікуйте. Використовуйте [правила Семантичного Версіонування](#) для вирішення відповідного наступного номера версії, на основі зроблених вами змін. Потім запускайте `cargo publish` для завантаження нової версії.

Застарівання Старих Версій на Crates.io з `cargo yank`

Хоча ми не можемо видалити попередні версії крейта, ми можемо запобігти будь-яким майбутнім проєктам додавати цих версій як нову залежність. Це корисно, коли версія крейту зламана по тій чи іншій причині. У таких ситуаціях Cargo підтримує *висмикування* або *yanking* версії крейту.

Висмикування версії перешкоджає залежність від цієї версії новими проєктами, дозволяючи усім чинним проєктам, які залежать від неї, продовжувати її використовувати. По суті, смик означає, що всі проєкти з *Cargo.lock* не зламуються та будь-який наступний створений файл *Cargo.lock* не буде використовувати висмикану версію.

Щоб висмикнути версію крейту в директорії, яку ви раніше публікували запустіть команду `cargo yank` та зазначте яку версію ви хочете висмикнути. Наприклад, якщо ви опублікуєте крейт з назвою `guessing_game` версії 1.0.1 та захочете висмикнути її, в теці вашого проєкту для `guessing_game` ви б виконали:

```
$ cargo yank --vers 1.0.1
Updating crates.io index
Yank guessing_game@1.0.1
```

Додаючи `--undo` до команди, ви також можете скасувати висмикування і дозволити проєктам знову почати залежати від версії:

```
$ cargo yank --vers 1.0.1 --undo
Updating crates.io index
Unyank guessing_game@1.0.1
```

Висмикування *не* видаляє жодного коду. Воно не може, наприклад, випадково видалити завантажені секрети. Якщо таке станеться, вам буде потрібно негайно замінити ці секрети.

Робочі Області Cargo

В розділі 12, ми зібрали пакет, який включав двійковий крейт та бібліотечний крейт. В міру розвитку вашого проекту, ви можете виявити, що бібліотечний крейт продовжує становитися більшим і вам хочеться розділити ваш пакет на декілька бібліотечних крейтів. Cargo пропонує функціонал названий *робочими областями*, який може допомогти в керуванні кількома пов'язаними пакетами, які розробляються в тандемі.

Створення Робочої Області

Робочий область це набір пакетів, які мають спільний *Cargo.lock* та каталог для виводу. Створимо проект з використанням робочої області — ми будемо використовувати тривіальний код, щоб було легше сконцентруватися на структурі робочого простору. Існує безліч способів упорядкування робочої області, тож ми просто покажемо один з найпоширеніших способів. У нас буде робоча область, що містить двійковий файл і дві бібліотеки. Двійковий файл, який надасть основний функціонал, буде залежати від двох бібліотек. Одна бібліотека надаватиме функцію `add_one`, а друга функцію `add_two`. Ці три крейти будуть частиною одної робочої області. Ми почнемо зі створення нового каталогу для робочої області:

```
$ mkdir add  
$ cd add
```

Далі, в каталозі *add*, ми створимо файл *Cargo.toml* який налаштує всю робочу область. Цей файл не матиме секції `[package]`. Натомість він розпочнеться з секції `[workspace]`, яка дозволить нам додавати учасників до робочої області, вказавши шлях до пакета із нашим двійковим крейтом; у цьому випадку, цей шлях *adder*:

Файл: Cargo.toml

```
[workspace]  
  
members = [  
    "adder",  
]
```

Далі, ми створимо двійковий крейт `adder` запустивши `cargo new` всередині каталогу *add*:

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

Наразі ми можемо зібрати робочу область запустивши `cargo build`. Файли в вашому каталозі `add` мають виглядати наступним чином:

```
├── Cargo.lock
├── Cargo.toml
├── adder
│   ├── Cargo.toml
│   └── src
│       └── main.rs
└── target
```

Робоча область має один каталог `target` на верхньому рівні, де будуть розміщені скомпільовані артефакти; пакет `adder` не має власного каталогу `target`. Навіть якщо ми запустимо `cargo build` зсередини каталогу `adder`, всі скомпільовані артефакти все одно з'являться в `add/target`, а не в `add/adder/target`. Cargo структурує каталог `target` в робочій області наступним чином, бо крейти в робочому просторі призначені для того, щоб залежати одне від одного. Якщо кожен крейт мав би власний каталог `target`, то кожен крейт мав би повторно компілювати кожен інший крейт в робочій області, щоб розмістити артефакти в власному каталозі `target`. При спільному використанні каталогу `target`, крейти можуть уникнути непотрібних повторних збірок.

Створення Другого Пакета в Робочій Області

Далі створимо ще один (member) пакет в робочій області і назвемо його `add_one`. Змініть `Cargo.toml` верхнього рівня, вказав шлях до `add_one` в списку `members`:

Файл: `Cargo.toml`

```
[workspace]

members = [
    "adder",
    "add_one",
]
```

Потім згенеруйте новий бібліотечний крейт, названий `add_one`:

```
$ cargo new add_one --lib
Created library `add_one` package
```

Ваш каталог *add* тепер повинен мати ці каталоги та файли:

```
├── Cargo.lock
├── Cargo.toml
├── add_one
│   ├── Cargo.toml
│   └── src
│       └── lib.rs
├── adder
│   ├── Cargo.toml
│   └── src
│       └── main.rs
└── target
```

У файлі *add_one/src/lib.rs*, додамо функцію `add_one`:

Файл: *add_one/src/lib.rs*

```
pub fn add_one(x: i32) -> i32 {
    x + 1
}
```

Тепер ми можемо мати пакет `adder` із нашим двійковим файлом, залежним від пакета `add_one`, який є в нашій бібліотеці. Спочатку нам потрібно додати шлях залежності `add_one` в *adder/Cargo.toml*.

Файл: *adder/Cargo.toml*

```
[dependencies]
add_one = { path = "../add_one" }
```

Cargo не припускає, що крейти в робочій області будуть залежати одне від одного, тому нам потрібно чітко визначити відносини залежностей.

Далі, використаємо функцію `add_one` (з крейту `add_one`) в крейті `adder`. Відкрийте файл *adder/src/main.rs* і додайте рядок `use` зверху, щоб внести новий бібліотечний крейт `add_one` в область видимості. Потім змініть функцію `main` та викличте функцію `add_one`, як показано в Блоці Коду 14-7.

Файл: *adder/src/main.rs*

```
use add_one;

fn main() {
    let num = 10;
    println!("Hello, world! {num} plus one is {}!", add_one::add_one(num));
}
```

Блок Коду 14-7: Використання бібліотечного крейту `add_one` з крейту `adder`

Зберемо робочу область викликавши `cargo build` на верхньому рівні каталогу `add`!

```
$ cargo build
Compiling add_one v0.1.0 (file:///projects/add/add_one)
Compiling adder v0.1.0 (file:///projects/add/adder)
Finished dev [unoptimized + debuginfo] target(s) in 0.68s
```

Щоб запустити двійковий крейт із каталогу `add`, нам потрібно вказати, який пакет в робочій області ми хочемо запускати використовуючи аргумент `-p` та назвою пакета в `cargo run`:

```
$ cargo run -p adder
Finished dev [unoptimized + debuginfo] target(s) in 0.0s
Running `target/debug/adder`
Hello, world! 10 plus one is 11!
```

Цей код в `adder/src/main.rs`, що залежить від крейту `add_one`.

Залежність від Зовнішнього Пакета в Робочій Області

Зауважте, що робоча область має лише один файл `Cargo.lock` на верхньому рівні, замість того, щоб мати `Cargo.lock` в кожному каталозі крейту. Завдяки цьому всі крейти використовують однакову версію всіх залежностей. Якщо ми додамо пакет `rand` в файли `adder/Cargo.toml` та `add_one/Cargo.toml`, Cargo вирішить використовувати одну версію `rand` для обох та запише це в одному `Cargo.lock`. Використання одних і тих самих залежностей для всіх крейтів в одній робочій області означає, що крейти завжди будуть сумісні один з одним. Додамо крейт `rand` в секцію `[dependencies]` в файл `add_one/Cargo.toml`, щоб ми могли використовувати крейт `rand` в крейті `add_one`:

Файл: `add_one/Cargo.toml`

```
[dependencies]
rand = "0.8.5"
```

Тепер ми можемо додати `use rand;` в файл `add_one/src/lib.rs` і збірка цілої робочої області, запустивши `cargo build` в каталозі `add`, принесе та скомпілює крейт `rand`. Ми отримаємо одне попередження, бо ми не посилаємось на принесений `rand` в нашій області видимості:

```
$ cargo build
  Updating crates.io index
  Downloaded rand v0.8.5
  --snip--
  Compiling rand v0.8.5
  Compiling add_one v0.1.0 (file:///projects/add/add_one)
warning: unused import: `rand`
--> add_one/src/lib.rs:1:5
   |
1  | use rand;
   |     ^^^^
   |
   = note: `#[warn(unused_imports)]` on by default

warning: `add_one` (lib) generated 1 warning
  Compiling adder v0.1.0 (file:///projects/add/adder)
  Finished dev [unoptimized + debuginfo] target(s) in 10.18s
```

`Cargo.lock` на верхньому рівні тепер містить інформацію про залежність `add_one` від `rand`. Однак, навіть якщо `rand` використовується десь в робочій області, ми не можемо використовувати його в інших крейтах в робочій області, якщо також не додамо `rand` до їхніх файлів `Cargo.toml`. Наприклад, якщо ми додамо `use rand;` в файл `adder/src/main.rs` пакету `adder`, ми отримаємо помилку:

```
$ cargo build
  --snip--
  Compiling adder v0.1.0 (file:///projects/add/adder)
error[E0432]: unresolved import `rand`
--> adder/src/main.rs:2:5
   |
2  | use rand;
   |     ^^^^ no external crate `rand`
```

Щоб це виправити, відредагуйте файл `Cargo.toml` пакету `adder` і вкажіть, що `rand` і для нього є залежністю. Збірка пакету `adder` додасть `rand` в список залежностей `adder` в `Cargo.lock`, але жодних додаткових копій `rand` не буде завантажено. Cargo має гарантувати, що кожен крейт у кожному пакеті в робочій області використовує пакет `rand` однакової версії, що збереже нам простір та запевнить, що крейти в робочій області будуть сумісними один з одним.

Додавання Тесту до Робочої Області

Для ще одного покращення, додамо тест функції `add_one::add_one` всередині крейту `add_one`:

Файл: `add_one/src/lib.rs`

```
pub fn add_one(x: i32) -> i32 {  
    x + 1  
}  
  
#[cfg(test)]  
mod tests {  
    use super::*;  
  
    #[test]  
    fn it_works() {  
        assert_eq!(3, add_one(2));  
    }  
}
```

Тепер запустіть `cargo test` в найвищому рівні каталогу `add`. Запуск `cargo test` в робочій області із подібною до цієї структурою буде запускати тести усіх крейтів цієї робочої області:


```
$ cargo test
Compiling add_one v0.1.0 (file:///projects/add/add_one)
Compiling adder v0.1.0 (file:///projects/add/adder)
Finished test [unoptimized + debuginfo] target(s) in 0.27s
Running unittests src/lib.rs (target/debug/deps/add_one-f0253159197f7841)

running 1 test
test tests::it_works ... ok

test result: ok. 1 passed; 0 failed; 0 ignored; 0 measured; 0 filtered out;
finished in 0.00s

Running unittests src/main.rs (target/debug/deps/adder-49979ff40686fa8e)

running 0 tests

test result: ok. 0 passed; 0 failed; 0 ignored; 0 measured; 0 filtered out;
finished in 0.00s

Doc-tests add_one

running 0 tests

test result: ok. 0 passed; 0 failed; 0 ignored; 0 measured; 0 filtered out;
finished in 0.00s
```

Перша секція виводу показує, що тест `it_works` крейту `add_one` проходить. Наступна секція показує, що нуль тестів було знайдено в крейті `adder`, і потім, остання секція показує нуль документаційних тестів в крейті `add_one`.

We can also run tests for one particular crate in a workspace from the top-level directory by using the `-p` flag and specifying the name of the crate we want to test:

```
$ cargo test -p add_one
Finished test [unoptimized + debuginfo] target(s) in 0.00s
Running unittests src/lib.rs (target/debug/deps/add_one-b3235fea9a156f74)

running 1 test
test tests::it_works ... ok

test result: ok. 1 passed; 0 failed; 0 ignored; 0 measured; 0 filtered out;
finished in 0.00s

Doc-tests add_one

running 0 tests

test result: ok. 0 passed; 0 failed; 0 ignored; 0 measured; 0 filtered out;
finished in 0.00s
```

Цей вивід показує, що `cargo test` запустив лише тести за крейту `add_one` та не запускав тести крейту `adder`.

Якщо ви опублікували крейти в робочій області до crates.io, то кожен крейт в робочій області потрібно буде публікувати окремо. Як із `cargo test`, ми можемо публікувати певний крейт із нашої робочої області використовуючи позначку `-p` та вказуючи назву крейту, який ми хочемо опублікувати.

Для додаткової практики, додайте крейт `add_two` до цієї робочої області схожим чином до крейту `add_one`!

У міру зростання вашого проєкту, розгляньте можливість використання робочої області: легше зрозуміти менші, окремі компоненти ніж один великий блок коду. Щобільше, зберігаючи крейти в робочій області можна зробити координацію між крейтами легшою, якщо вони часто та одночасно змінюються.

Встановлення Двійкових Файлів з `cargo install`

Команда `cargo install` дозволяє встановлювати і використовувати бінарні крейти локально. Це не має на меті замінити системні пакети; Це має бути зручним способом для Rust розробників встановити інструменти, якими інші поділилися на crates.io. Зауважте, що ви можете встановлювати лише пакети, які мають цільовий двійковий файл. *Цільовий двійковий файл* це запускаєма програма, яка створюється, якщо крейт має файл `src/main.rs` або інший файл вказаний, як двійковий, на відміну від цільового бібліотечного файлу, який не можна запускати сам по собі, але який є придатним для додавання всередину інших програм. Зазвичай крейти мають інформацію в файлі `README` про те, чи крейт це бібліотека, має двійкову ціль, або й те й інше.

Всі встановлені з `cargo install` двійкові файли зберігаються в теці `bin` кореневого каталогу встановлення. Якщо ви встановили Rust із `rustup.rs` і не маєте жодних користувацьких конфігурацій, то цей каталог буде `$HOME/.cargo/bin`. Переконайтеся, що каталог є в вашому `$PATH`, щоб мати можливість запускати встановленні з `cargo install` програми.

Наприклад, у Розділі 12 ми згадували, що існує Rust імплементація інструменту `grep` під назвою `ripgrep` для пошуку файлів. Щоб встановити `ripgrep`, ми запустимо наступне:

```
$ cargo install ripgrep
  Updating crates.io index
  Downloaded ripgrep v13.0.0
  Downloaded 1 crate (243.3 KB) in 0.88s
  Installing ripgrep v13.0.0
--snip--
  Compiling ripgrep v13.0.0
  Finished release [optimized + debuginfo] target(s) in 3m 10s
  Installing ~/.cargo/bin/rg
  Installed package `ripgrep v13.0.0` (executable `rg`)
```

Передостанній рядок виводу показує розташування і назву встановленого двійкового файлу, який у випадку `ripgrep` має назву `rg`. Допоки у вашому `$PATH` є каталог встановлення, як говорилося раніше, ви зможете запускати `rg --help` та починати використовувати швидший, іржавіший інструмент для пошуку файлів!

Розширення Cargo із Користувацькими Командами

Cargo розроблений таким чином, щоб ви могли розширювати його за допомогою нових підкоманд без необхідності модифікації Cargo. Якщо двійковий файл у вашому `$PATH` названий `cargo-something`, то ви можете запустити його, ніби це підкоманда Cargo, викликавши `cargo something`. Користувальницькі команди, такі як ця, також зазначені під час запуску `cargo --list`. Можливість використати `cargo install` для встановлення розширень, а потім запускати їх, так само як і вбудовані інструменти Cargo є дуже зручною перевагою дизайну Cargo!

Підсумок

Обмін кодом використовуючи Cargo і crates.io є частиною того, що робить екосистему Rust корисною для багатьох різноманітних задач. Стандартна бібліотека Rust маленька та стабільна, але крейтами легко ділитися, використовувати та покращувати за графіком, відмінним від графіка розвитку мови. Не соромтеся ділитися корисними для вас кодом на crates.io; цілком ймовірно, що це буде корисно і комусь іншому!

Розумні Вказівники

Вказівник - загальна концепція для змінної, що містить адресу в пам'яті. Ця адреса посилається, або "вказує", на певну інформацію. Найбільш розповсюджений вид вказівника в Rust - це посилання, яке ви вивчили в Розділі 4. Посилання позначені символом `&` та позичають значення, на яке вказують. Вони не мають інших можливостей, крім посилання на дані, та не мають накладних витрат.

Розумні вказівники, з іншої сторони, є структурами даних що поведуть себе як вказівник, але також мають додаткові метадані та можливості. Концепція розумних вказівників не унікальна для Rust: розумні вказівники виникли в C++ та також існують в інших мовах програмування. Rust має різні розумні вказівники, визначені стандартною бібліотекою, які надають додатковий функціонал, крім наданого посиланнями. Для роз'яснення основної концепції ми подивимось на різні приклади розумних вказівників, включно з типом розумних вказівників, що *підраховує посилання*. Цей вказівник дозволяє даним мати декілька володільців завдяки відстежуванню кількості володільців. Коли володільців не залишається, інформація буде видалена.

Оскільки Rust має власну концепцію володіння та позичання, є додаткова різниця між посиланнями та розумними вказівниками: посилання лише позичають значення, але в багатьох випадках розумні вказівники *володіють* значенням, на яке вказують.

Хоча ми й не називати їх так на той момент, ми вже зустрілись з деякими розумними вказівниками в цій книзі, включно зі `String` та `Vec<T>` у Розділі 8. Обидва типи вважаються розумними вказівниками, оскільки вони володіють деякою пам'яттю і дозволяють вам маніпулювати нею. Вони також мають метадані та додаткові можливості чи гарантії. `String`, наприклад, зберігає свою місткість як метадані та має додаткову спроможність гарантувати, що його дані будуть валідним UTF-8.

Розумні вказівники зазвичай реалізуються за допомогою структур. На відміну від звичайних структур, розумні вказівники реалізують трейти `Deref` та `Drop`. Трейт `Deref` дозволяє екземпляру структури розумного вказівника поводитись, як посилання, тож ви можете писати свій код, що працюватиме як із посиланнями, так і з розумними вказівниками. Трейт `Drop` дозволяє змінити код, що виконується при виході екземпляра розумного вказівника з області видимості. У цьому розділі буде розглянуто обидва трейта та продемонстровано, чому вони важливі для розумних вказівників.

Оскільки паттерн розумного вказівника є загальним паттерном проектування, що часто використовується в Rust, цей розділ не розглядає усі можливі розумні вказівники. Багато бібліотек мають власні розумні вказівки, і ви навіть можете написати свої власні. Ми розглянемо найпоширеніші розумні вказівники стандартної бібліотеки:

- `Box<T>` для розміщення значень в heap
- `Rc<T>`, тип з підрахунком посилань, який уможливорює множинне володіння
- `Ref<T>` та `RefMut<T>`, accessed through `RefCell<T>`, тип що забезпечує виконання правил запозичення під час виконання, а не компіляції

Додатково ми розглянемо паттерн *внутрішньої мутабельності*, де немутабельний тип надає API для зміни внутрішнього значення. Також будуть розглянуті *зациклювання посилань*: як вони можуть розтратити пам'ять та як цьому запобігти.

Отже, вперед!

Використання `Box<T>` для Вказування на Дані в Купі

Найбільш простий розумний вказівник це `box`, тип якого записано в `Box<T>`. `Box` дозволяє зберігати дані в `heap`, а не на стеку. На стеку залишається вказівник на значення в `Heap`. Перегляньте Розділ 4, щоб побачити різницю між стеком та купою.

Коробки не мають накладних витрат, окрім як зберігання даних в `heap` замість того, щоб розміщувати дані на стеку. Але у них також немає багато додаткових можливостей. Найчастіше ви будете використовувати їх у таких ситуаціях:

- Якщо у вас є тип, розмір якого не може бути відомий при компілюванні і ви хочете використати значення цього типу в контексті, який вимагає точного розміру
- Якщо у вас є велика кількість даних і ви хочете передати володіння, але хочете гарантії, що дані не будуть скопійовані після виконання
- Коли ви бажаєте володіти значенням та вам важливо лише, що тип реалізує певний трейт, а не є певним типом

Ми продемонструємо першу ситуацію в "Рекурсивні типи з `Box`" секції. У другому випадку передача володіння великої кількості даних може зайняти багато часу тому, що дані копіюються зі стеку. Щоб підвищити продуктивність в такій ситуації, ми можемо зберігати велику кількість даних в `Heap` в `box`. Копіюється тільки невелика кількість даних вказівника у стеку, в той час як дані, на яких він посилається, залишається в одному місці в `Heap`. Третій випадок - відомий як трейт об'єкт `**`, займає весь розділ 17, "Використання трейт об'єктів, які допускають значення різних типів", Отже, що ви знаєте тут, ви будете використовувати ще раз в Розділі 17!

Використання `Box<T>` для Зберігання Даних в Купі

Перед тим як обговорити випадок зберігання даних в `Heap` в `Box<T>` ми розглянемо синтаксис і як взаємодіяти зі значеннями, що зберігаються в `Box<T>`.

Блок коду 15-1 показує, як використовувати `Box` для збереження значення типу `i32` у купі:

Файл: `src/main.rs`

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

Блок коду 15-1: Збереження `i32` значення в `Heap` з використанням `Box`

Ми визначили змінну `b` що має значення `Box`, яке вказує на значення `5`, яке виділяється в `heap`. Ця програма виведе на екран `b = 5`; в цьому випадку, ми можемо отримати доступ до даних у `Box`, аналогічно до того, як ми могли б зробити так, якби дані були на стеку. Будь-яке значення, наприклад коли `Box` виходить за `scope`, як це робить `b` в кінці `main`, буде звільнено. Звільнення відбувається як для коробки (на стеці), так і для значення на яке вказує (зберігаються в `heap`).

Розміщення одного значення в `heap` не дуже ефективно, тому таким чином ви будете робити не часто. Мати значення як один `i32` на стеку, де вони зберігаються за замовчуванням, більш підходить для більшості ситуацій. Розглянемо випадок, де `Box` дозволяють нам використати типи які ми б не могли застосувати без `Box`.

Рекурсивні Типи з `Box`

Складовою *рекурсивного типу* може бути значення того цього ж типу. Реалізація рекурсивного типу може бути проблемою, бо при компіляції Rust потрібно знати скільки місця займає тип. Однак вкладеність значень рекурсивних типів теоретично може тривати нескінченно, тому Rust не може знати, скільки пам'яті потребує значення. Оскільки `Box` має відомий розмір, ми можемо реалізувати рекурсивні типи вставленням `Box` у визначення рекурсивного типу.

Як приклад рекурсивного типу, давайте дослідимо *cons list*. Це тип даних, який зазвичай зустрічається у функціональних мовах програмування. Тип `cons list` ми визначимо його напям, за винятком рекурсії. Концепції у прикладі, з якими ми працюватимемо, будуть корисними при потраплянні у складніші ситуації, що стосуються рекурсивних типів.

Більше Інформації про `Cons List`

Cons list — це структура даних, що прийшла із мови програмування Lisp та його діалектів. Структура складається з вкладених пар, і є різновидом зв'язаного списку в Lisp. Ця назва походить з `cons` функції (коротко для функції "construct function") в Lisp, яка формує нову пару з двох аргументів. Викликанням `cons` до пари зі значенням та іншою парою, ми можемо створювати зв'язані списки з рекурсивних пар.

Наприклад, ось набір псевдокоду, що представляє зв'язаний список з 1, 2, 3 з кожною парою в дужках:

```
(1, (2, (3, Nil)))
```


Кожен елемент у cons list містить два елементи: значення поточного елемента і наступного елемента. Останній елемент списку містить значення `Nil`, що означає відсутність наступного елемента. Cons list створюється рекурсивним викликанням функції `cons`. Канонічне ім'я для позначення загального випадку рекурсії `Nil`. Зверніть увагу, що це не те саме, що "null" або "nil" концепція у Розділі 6, яке є недійсним або відсутнім значенням.

Cons list не є загально вживаною структурою даних в Rust. В більшості випадків, коли у вас є список елементів в Rust, `Vec<T>` є кращим варіантом для використання. Більш складні типи рекурсивних даних *корисні в різних ситуаціях*, але починаючи з cons list у цьому розділі, ми можемо ясніше дослідити, як `box` дає змогу визначити тип рекурсивних даних.

Блок коду 15-2 містить визначення енума для cons list. Зверніть увагу, що цей код не скомпілюється, тому що тип `List` не має відомого розміру, що ми продемонструємо.

Файл: src/main.rs

```
enum List {  
    Cons(i32, List),  
    Nil,  
}
```

Блок коду 15-2: Перша спроба визначення енума що представляє cons list значень типу `i32`

Примітка: Ми реалізуємо cons list, який містить лише значення `i32` як приклад. Ми могли б реалізувати її за допомогою generic, які ми розглянули у розділі 10, щоб визначити cons list для збереження значення будь-якого типу.

Використовування типу `List`, щоб зберегти список з `1, 2, 3` буде виглядати як код в Блоці коду 15-3:

Файл: src/main.rs

```
use crate::List::{Cons, Nil};  
  
fn main() {  
    let list = Cons(1, Cons(2, Cons(3, Nil)));  
}
```

Роздрук 15-3: Використання енуму `List` для збереження списку `1, 2, 3`

Перший `Cons` містить значення `1` та значення `List`. Цей `List` - інший `Cons`, який містить `2` і ще одне значення `List`. Значення `List` є ще одним `Cons`, яке містить `3` і `Cons` який нарешті `Nil`, нерекурсивний варіант, який сигналізує про кінець списку.

Якщо ми спробуємо скомпілювати код у Роздруку 15-3, ми отримаємо помилку, показану в Роздруку 15-4:

```
$ cargo run
  Compiling cons-list v0.1.0 (file:///projects/cons-list)
error[E0072]: recursive type `List` has infinite size
--> src/main.rs:1:1
   |
1  | enum List {
   | ^^^^^^^^^ recursive type has infinite size
2  |     Cons(i32, List),
   |               ---- recursive without indirection
   |
help: insert some indirection (e.g., a `Box`, `Rc`, or `&`) to make `List`
representable
2  |     Cons(i32, Box<List>),
   |               ++++++
   |               +

For more information about this error, try `rustc --explain E0072`.
error: could not compile `cons-list` due to previous error
```

Блок коду 15-4: Помилка, яку ми отримуємо при спробі визначити рекурсивний еnum

Помилка показує, що цей тип "має нескінченний розмір." Причина в тому, що ми визначили `List` з варіантом, який рекурсивний: він складається зі значення свого ж типу. Як результат, Rust не може визначити скільки місця йому потрібно для `List`. Розберімося, чому ми отримуємо цю помилку. Спочатку ми подивимось на те, як Rust вирішує, скільки місця їй потрібно зберегти значення не рекурсивного типу.

Обчислення Розміру Нерекурсивного Типу

Розглянемо повторно `Message` еnum з Розділу 6-2 коли ми дізнались про еnum в Розділі 6:

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

Щоб визначити скільки займати місця для `Message` Rust проходить через кожен з варіантів, щоб побачити, який варіант потребує найбільше місця. Rust бачить що

`Message::Quit` не потрібно місця. `Message::Move` достатньо місця як для зберігання 2 `i32` значень, і так далі. Тому що використовуватиметься лише один варіант, `Message` буде займати як найбільший з можливих своїх варіантів.

Порівняйте це з тим, що відбувається, коли Rust намагається визначити скільки місця займає рекурсивний тип `Cons` в Роздруку 15-2. Компілятор дивиться на варіант `Cons` який містить значення типу `i32` та значення типу `Cons`. Відповідно, `Cons` потребує пам'яті, що дорівнює розміру `i32` плюс розмір `Cons`. Щоб дізнатись скільки пам'яті потребує `List`, компілятор дивиться на варіанти, починаючи з `Cons`. `Cons` є значенням типу `i32` і значення типу `Cons`, і цей процес нескінченно продовжується як показано на Рисунок 15-1.

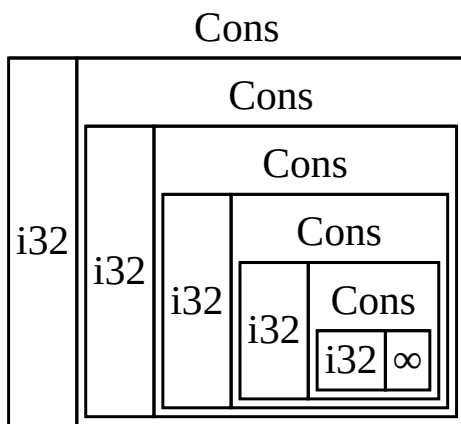


Рисунок 15-1: Нескінченний `List` що складається з безлічі `Cons` варіантів

Використання `Box<T>` для Отримання Рекурсивного Типу з Відомим Розміром

Оскільки Rust не може визначити скільки пам'яті для рекурсивно визначених типів, компілятор надає помилку з корисною пропозицією:

```
help: insert some indirection (e.g., a `Box`, `Rc`, or `&`) to make `List`
representable
2 |     Cons(i32, Box<List>),
  |               ++++++  +
```

У цій пропозиції "indirection" (опосередкованість) означає, що замість того, щоб зберігати значення безпосередньо, ми повинні змінити структуру даних, щоб зберігати значення опосередковано, зберігаючи замість нього вказівник на значення.

Тому що `Box<T>` є вказівником, Rust завжди знає, скільки потрібно пам'яті для `Box<T>`: розмір вказівника не залежить від розміру типу даних, на які вказує. Це означає, що ми

можемо розмістити `Cons` в `Box<T>` замість напряму `Cons`. `Box<T>` вказує на наступний `List`, яке буде в `Heap`, а не в `Cons`. Таким чином, у нас все ще є список, створений з іншими списками, що тримає інші списки, але ця реалізація тепер більше схожа на розміщення елементів один біля одного, а не всередині один одного.

Ми можемо змінити визначення енуму `List` з Роздруку 15-2 і використання `List` в Роздруку 15-3 до коду в Роздруку 15-5 що буде компілюватись в:

Файл: `src/main.rs`

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

use crate::List::{Cons, Nil};

fn main() {
    let list = Cons(1, Box::new(Cons(2, Box::new(Cons(3, Box::new(Nil))))));
}
```

Блок коду 15-5: визначення `List`, який використовує `Box<T>`, щоб мати відомий розмір

`Cons` потрібно мати розмір `i32` плюс пам'ять для зберігання даних вказівника `box`. Варіант `Nil` не зберігає значення, тому йому потрібно менше місця, ніж `Cons`. Ми тепер знаємо, що будь-яке значення `Cons` займе розмір `i32` плюс розмір вказівника `box`. Використовуючи `box`, ми зламали нескінченний, рекурсивний ланцюжок, таким чином, компілятор може визначити розмір, який йому потрібно щоб зберегти `List`. Рисунок 15-2 показує як зараз виглядає варіант `Cons`.

Cons

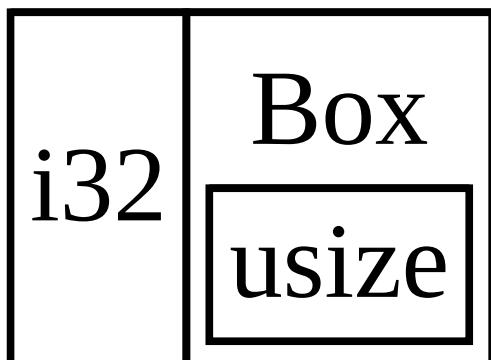


Рисунок 15-2: `List` який не є нескінченним розміром, тому що `List` містить `Box`

`Box` забезпечує лише розміщення в `Heap`; у них немає жодних інших спеціальних можливостей, які ми побачимо в інших розумних вказівниках. Також вони не мають накладних витрат на ці спеціальні можливості, тож вони можуть бути корисні у випадках як `cons list`, де розміщення в іншому місці для того, щоб мати вказівник відомого розміру - все що нам потрібно. Ми також розглянемо застосування `box` в Розділі 17.

`Box<T>` є розумним вказівником, оскільки реалізує трейт `Deref` що дозволяє `Box<T>` застосовувати як посилання. Коли значення `Box<T>` виходить з області видимості, дані в купі, на які вказує `box`, видаляться через реалізацію трейту `Drop`. Ці трейти будуть ще важливіші для функціональності, які надають інші розумні вказівники, які ми обговоримо в інших главах цього розділу. Розгляньмо ці трейти детальніше.

Ставлення до Розумних Вказівників як до Звичайних Посилань з Трейтом `Deref`

Реалізація трейта `Deref` дозволяє вам налаштувати поведінку *оператора розіменування* `*` (не плутати з оператором множення чи глобальним оператором). Релізувавши `Deref` таким чином, щоб розумний вказівник міг використовуватися як звичайне посилання, ви зможете писати код, що працює з посиланнями і використовувати цей код також із розумними вказівниками.

Спочатку подивімося, як оператор розіменування працює зі звичайними посиланнями. Потім ми спробуємо визначити власний тип, що поводить ся як `Box<T>`, і побачимо, чому оператор розіменування не працює, як посилання, для нашого щойно визначеного типу. Ми дослідимо, як реалізація трейта `Deref` дозволяє розумним вказівникам працювати у спосіб, схожий на посилання. Тоді ми розглянемо таку особливість Rust, як *приведення при розіменуванні* і як вона дозволяє нам працювати як із посиланнями, так і з розумними вказівниками.

Примітка: існує суттєва різниця між типом `MyBox<T>`, який ми збираємося описати, і справжнім `Box<T>`: наша версія не зберігатиме дані в купі. Ми зосередимося у цьому прикладі на `Deref`, тож нам не так важливо, де насправді зберігаються дані, ніж поведінка, подібна до вказівника.

Перехід за Вказівником до Значення

Звичайне посилання — це тип вказівника. Вказівник можна уявити як стрілку, що вказує на значення, розміщене деінде. У Блоці коду 15-6 ми створюємо посилання на значення `i32`, а потім використовуємо оператор розіменування, щоб перейти за посиланням до значення:

Файл: `src/main.rs`

```
fn main() {
    let x = 5;
    let y = &x;

    assert_eq!(5, x);
    assert_eq!(5, *y);
}
```

Блок коду 15-6: використання оператор розіменування, щоб перейти за посиланням до значення `i32`

Змінна `x` має значення `5` типу `i32`. Ми встановили значення `y` рівним посиланню на `x`. Ми можемо стверджувати, що `x` дорівнює `5`. Проте, якщо ми хочемо зробити твердження про значення в `y`, ми повинні виконати `*y`, щоб перейти за посиланням до значення, на яке воно вказує (тобто *розіменувати*), щоб компілятор міг порівняти фактичне значення. Розіменувавши `y`, ми отримуємо доступ до цілого значення, на яку `y` вказує, яке ми можемо порівняти з `5`.

Якби ми спробували написати натомість `assert_eq!(5, y);`, то отримали б помилку компіляції:

```
$ cargo run
   Compiling deref-example v0.1.0 (file:///projects/deref-example)
error[E0277]: can't compare `{integer}` with `&{integer}`
  --> src/main.rs:6:5
   |
6  |     assert_eq!(5, y);
   |     ^^^^^^^^^^^^^^^^^ no implementation for `{integer} == &{integer}`
   |
= help: the trait `PartialEq<&{integer}>` is not implemented for `{integer}`
= help: the following other types implement trait `PartialEq<Rhs>`:
         f32
         f64
         i128
         i16
         i32
         i64
         i8
         isize
         and 6 others
= note: this error originates in the macro `assert_eq` (in Nightly builds, run
with -Z macro-backtrace for more info)

For more information about this error, try `rustc --explain E0277`.
error: could not compile `deref-example` due to previous error
```

Порівняння числа і посилання на число не дозволене, оскільки це різні типи. Ми маємо використовувати оператор розіменування, щоб перейти за посиланням до значення, на яке воно вказує.

Використання `Box<T>` як Посилання

Ми можемо переписати код у Блоці коду 15-6, щоб використовувати `Box<T>` замість посилання; оператор розіменування, застосований до `Box<T>` у Блоці коду 15-7 працює

так само як і оператор розіменування, застосований до посилання у Блоці коду 15-6:

Файл: src/main.rs

```
fn main() {  
    let x = 5;  
    let y = Box::new(x);  
  
    assert_eq!(5, x);  
    assert_eq!(5, *y);  
}
```

Блок коду 15-7: використання оператора розіменування на `Box<i32>`

Основна відмінність між Блоком коду 15-7 і Блоком коду 15-6 полягає в тому, що в першому ми робимо `y` екземпляром `Box<T>`, що вказує на скопійоване значення `x`, а не посиланням, що вказує на значення `x`. В останньому твердженні ми можемо використати оператор розіменування, щоб перейти за вказівником у `Box<T>` так само як ми робили, коли `y` був посиланням. Далі ми дослідимо, що ж такого в `Box<T>` дає нам змогу використовувати оператор розіменування, визначивши власний тип `MyBox`.

Визначення Власного Розумного Вказівника

Створімо розумний вказівник, схожий на тип `Box<T>`, що надається стандартною бібліотекою, щоб побачити, у чому розумні вказівники поведуться інакше, ніж вказівники за замовчанням. Тоді ми розглянемо, як додати можливість використовувати оператор розіменування.

Тип `Box<T>` кінець-кінцем визначається як структура-кортеж з одним елементом, тож Блок коду 15-8 визначає тип `MyBox<T>` у той же спосіб. Ми також визначаємо функцію `new`, що відповідає функції `new`, визначеній для `Box<T>`.

Файл: src/main.rs

```
struct MyBox<T>(T);  
  
impl<T> MyBox<T> {  
    fn new(x: T) -> MyBox<T> {  
        MyBox(x)  
    }  
}
```

Блок коду 15-8: визначення типу `MyBox<T>`

Ми визначаємо структуру з назвою `MyBox` і оголошуємо узагальнений параметр `T`, оскільки ми хочемо, щоб наш тип працював зі значеннями будь-якого типу. Тип `MyBox` є структурою-кортежем з одним елементом типу `T`. Функція `MyBox::new` приймає один параметр типу `T` і повертає екземпляр `MyBox`, який містить передане значення.

Спробуймо додати функцію `main` з Блока коду 15-7 до Блоку коду 15-8 та змінити її, щоб використовувати визначений нами тип `MyBox<T>` замість `Box<T>`. Код у Блоці коду 15-9 не компілюється, оскільки Rust не знає, як розіменувати `MyBox`.

Файл: `src/main.rs`

```
fn main() {  
    let x = 5;  
    let y = MyBox::new(x);  
  
    assert_eq!(5, x);  
    assert_eq!(5, *y);  
}
```

Блок коду 15-9: спроба використовувати `MyBox<T>` тим самим способом, яким ми використовували посилання та `Box<T>`

Виходить ось така помилка компіляції:

```
$ cargo run  
Compiling deref-example v0.1.0 (file:///projects/deref-example)  
error[E0614]: type `MyBox<{integer}>` cannot be dereferenced  
--> src/main.rs:14:19  
14 |         assert_eq!(5, *y);  
   |                        ^^  
  
For more information about this error, try `rustc --explain E0614`.  
error: could not compile `deref-example` due to previous error
```

Наш тип `MyBox<T>` не можна розіменувати, оскільки ми не реалізували цю здатність для нашого типу. Щоб дозволити розіменування за допомогою оператора `*`, ми реалізуємо трейт `Deref`.

Ставлення до Типу як до Посилання з Трейтом `Deref`

Як обговорено в підрозділі "Реалізація трейту для типів" Розділу 10, щоб реалізувати трейт, ми маємо реалізувати методи, необхідні цьому трейту. Трейт

`Deref`, наданий стандартною бібліотекою, вимагає, щоб ми реалізувати один метод, що зветься `deref`, який позичає `self` і повертає посилання на внутрішні дані. Блок коду 15-10 містить реалізацію `Deref`, яку треба додати до визначення `MyBox`:

Файл: `src/main.rs`

```
use std::ops::Deref;

impl<T> Deref for MyBox<T> {
    type Target = T;

    fn deref(&self) -> &Self::Target {
        &self.0
    }
}
```

Блок коду 15-10: реалізація `Deref` для `MyBox<T>`

Запис `type Target = T;` визначає асоційований тип для використання трейтом `Deref`. Асоційовані типи дещо відрізняються від оголошення узагальненого параметра, але вам поки що не потрібно турбуватися про них; ми розглянемо її детальніше у Розділі 19.

В тіло методу `deref` ми додаємо `&self.0`, тож `Deref` повертає посилання на значення, до якого ми хочемо отримати доступ за допомогою оператора `*`; згадайте з підрозділу "Структури-кортежі без іменованих полів для створення нових типів" Розділу 5, що `.0` є способом доступу до першого значення у структурі-кортежі. Функція `main` у Блоці коду 15-9, яка викликає `*` для значення `MyBox<T>` тепер компілюється, і твердження виконуються!

Без трейту `Deref` компілятор може розіменовувати лише посилання `&`. Метод `deref` надає компілятору можливість взяти значення будь-якого типу, який реалізує `Deref`, і викликати метод `deref`, щоб отримати посилання `&`, яке він вміє розіменовувати.

Коли ми ввели `*y` у Блоці коду 15-9, за лаштунками Rust насправді запустився цей код:

```
*(&y.deref())
```

Rust замінює оператор `*` викликом методу `deref`, а потім звичайне розіменування, тож нам не треба думати, треба чи не треба викликати метод `deref`. Це особливість Rust дозволяє нам писати код, що працює так само, використовуємо ми звичайні посилання чи тип, що реалізує `Deref`.

Причина, з якої метод `deref` повертає посилання на значення, і що за дужками `(y.deref())` все ще потрібне звичайне розіменування, стосується системи володіння. Якби метод `deref` повертав значення безпосередньо замість посилання на значення,

значення було б переміщене з `self`. Ми не хочемо у цьому випадку брати володіння внутрішнім значенням у `MyBox<T>`, як і в більшості випадків, де ми використовуємо оператор розіменування.

Зверніть увагу, що оператор `*` замінюється на виклик метод `deref`, а потім виклик оператора `*` лише один раз, щоразу, коли ми використовуємо `*` у нашому коді. Оскільки підставлення оператора `*` не виконується рекурсивно до нескінченості, ми прийдемо до даних типу `i32`, що відповідають `5` у `assert_eq!` у Блоці коду 15-9.

Неявне Приведення Розіменування у Функціях та Методах

Приведення розіменування перетворює посилання на тип, що реалізує трейт `Deref`, до посилання на інший тип. Наприклад, приведення розіменування може перетворити `&String` на `&str`, тому що `String` реалізує трейт `Deref`, так, що він повертає `&str`. Приведення розіменування - це покращення для зручності, яке Rust застосовує до аргументів функцій та методів, і працює лише з типами, що реалізують трейт `Deref`. Воно застосовується автоматично, коли ми передаємо посилання на значення певного типу як аргумент функції чи метода, що не відповідає типу параметра у визначенні функції чи метода. Послідовність викликів методу `deref` перетворює тип, наданий нами, на тип, потрібний параметру.

Приведення розіменування було додане в Rust, щоб програмістам, що пишуть виклики функцій та методів, не було потрібно додавати стільки явних посилань і розіменувань за допомогою `&` і `*`. Приведення розіменування також дозволяє легше писати код, що працює як з посиланнями, так і з розумними вказівниками.

Щоб побачити, як працює приведення розіменування, застосуємо тип `MyBox<T>`, який ми визначили у Блоці коду 15-8, разом із реалізацією `Deref`, яку ми додали в Блоці коду 15-10. Блок коду 15-11 показує визначення функції, що має параметром стрічковий слайс:

Файл: `src/main.rs`

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

Блок коду 15-11: Функція `hello`, що має параметр `name` типу `&str`

Ми можемо викликати функцію `hello` аргументом - стрічковим слайсом, наприклад `hello("Rust");`. Приведення розіменування уможливорює виклик `hello` з посиланням на значення типу `MyBox<String>`, як показано в Блоці коду 15-12:

Файл: src/main.rs

```
fn main() {  
    let m = MyBox::new(String::from("Rust"));  
    hello(&m);  
}
```

Блок коду 15-12: Виклик `hello` з посиланням на значення `MyBox<String>`, яке працює завдяки приведенню розіменування

Тут ми викликаємо функцію `hello` з аргументом `&m`, який є посиланням на значення типу `MyBox<String>`. Оскільки ми реалізували трейт `Deref` для `MyBox<T>` у Блоці коду 15-10, Rust може перетворити `&MyBox<String>` на `&String` викликавши `deref`. Стандартна бібліотека надає реалізацію `Deref` для `String`, що повертає стрічковий слайс, і про це сказано в документації API для `Deref`. Rust викликає `deref` знову, щоб перетворити `&String` на `&str`, який відповідає визначенню функції `hello`.

Якби Rust не мав приведення розіменування, нам довелося б писати код, як у Блоці коду 15-13 замість коду з Блоку коду 15-12, щоб викликати `hello` для значення типу `&MyBox<String>`.

Файл: src/main.rs

```
fn main() {  
    let m = MyBox::new(String::from("Rust"));  
    hello(&(*m)[..]);  
}
```

Блок коду 15-13: код, який довелося б писати, якби Rust не мав приведення розіменування

`(*m)` розіменовує `MyBox<String>` у `String`. Потім `&` і `[..]` беруть стрічковий слайс зі `String`, що дорівнює всій стрічці, щоб відповідати сигнатурі `hello`. Цей код без приведення розіменування складніше читати, писати і розуміти з усіма цими символами. Приведення розіменування дозволяє Rust обробляти для нас такі перетворення автоматично.

Коли трейт `Deref` визначений для залучених типів, Rust аналізує ці типи і використовує `Deref::deref` стільки разів, скільки треба, щоб отримати посилання, що відповідає типу параметра. Скільки разів треба додати `Deref::deref` визначається під час компіляції, тож немає ніяких втрат часу виконання за переваги приведення розіменування!

Як Приведення Розіменування Взаємодіє з Мутабельністю

Подібно до того, як ви використовуєте трейт `Deref`, щоб перевизначити оператор `*` для імутабельних посилань, ви можете скористатися трейтом `DerefMut`, щоб перевизначити оператор `*` для мутабельних посилань.

Rust виконує приведення розіменування, коли виявляє типи і реалізації трейтів у трьох випадках:

- З `&T` в `&U`, якщо `T: Deref<Target=U>`
- З `&mut T` в `&mut U`, якщо `T: DerefMut<Target=U>`
- З `&mut T` в `&U`, якщо `T: Deref<Target=U>`

Перші два випадки однакові, окрім того, що другий реалізує мутабельність. Перший випадок застосовується, що якщо є `&T`, і `T` реалізує `Deref` у якийсь тип `U`, то ми можете прозоро отримати `&U`. Другий випадок застосовується що таке саме приведення розіменування виконується для мутабельних посилань.

Третій випадок хитріший: Rust також приведе мутабельне посилання до немутабельного. Але зворотне *неможливе*: немутабельні посилання ніколи не приводяться до мутабельних посилань. Через правила позичання, якщо ви маєте мутабельне посилання, це мутабельне посилання має бути єдиним посиланням на ці дані (інакше програма не скомпілюється). Перетворення мутабельного посилання на немутабельне ніколи не порушить правила позичання. Перетворення немутабельного посилання на мутабельне посилання вимагало б, щоб початкове немутабельне посилання було єдиним немутабельним посиланням на ці дані, але правила позичання не гарантують цього. Тому Rust не може зробити припущення про те, чи перетворення немутабельного посилання на мутабельне посилання є можливим.

Виконання Коду при Очищенні з Трейтом `Drop`

Другий важливий трейт для шаблону розумного вказівника - `Drop`, який дозволяє вам налаштувати, що відбувається, коли значення збирається вийти з області видимості. Ви можете створити реалізацію трейту `Drop` для будь-якого типу, і цей код може використовуватися для звільнення ресурсів на кшталт файлів чи мережевих з'єднань.

Ми презентуємо `Drop` у контексті розумних вказівників, оскільки функціональність трейту `Drop` практично завжди використовується при реалізації розумних вказівників. Наприклад, коли `Box<T>` скидається, то звільняє простір у купі, виділений при створенні.

У деяких мовах для деяких типів програміст повинен викликати код для звільнення пам'яті або ресурсів кожного разу, коли він завершує використовувати екземпляр такого типу. Прикладами можуть бути файли, сокети чи блокування доступу до даних. Якщо програміст забуде це зробити, система може перенавантажитися і впасти. У Rust ви можете вказати, що спеціальний шматок коду має бути виконано, коли значення виходить з зони видимості, і компілятор додасть цей код автоматично. В результаті вам не треба стежити за ретельним розміщенням коду очищення всюди в програмі для завершення роботи з екземпляром певного типу - і все одно ви не допустите витоку ресурсів!

Ви вказуєте код, що потрібно виконати, коли значення виходить з області видимості, реалізуючи трейт `Drop`. Трейт `Drop` потребує реалізації одного методу, що зветься `drop`, який приймає мутабельне посилання на `self`. Щоб побачити, коли Rust викликає `drop`, тимчасово реалізуймо `drop` з інструкціями `println!`.

Блок коду 15-14 показує структуру `CustomSmartPointer`, чия єдина особлива функціональність полягає в тому, що вона виводить `Dropping CustomSmartPointer!`, коли екземпляр виходить із зони видимості, щоб показати, коли Rust запускає функцію `drop`.

Файл: `src/main.rs`

```

struct CustomSmartPointer {
    data: String,
}

impl Drop for CustomSmartPointer {
    fn drop(&mut self) {
        println!("Dropping CustomSmartPointer with data `{}`!", self.data);
    }
}

fn main() {
    let c = CustomSmartPointer {
        data: String::from("my stuff"),
    };
    let d = CustomSmartPointer {
        data: String::from("other stuff"),
    };
    println!("CustomSmartPointers created.");
}

```

Блок коду 15-14: структура `CustomSmartPointer`, що реалізує трейт `Drop`, в якому ми розміщуємо наш код для очищення

Трейт `Drop` включено до прелюдії, тож нам не треба вводити її в область видимості. Ми реалізуємо трейт `Drop` для `CustomSmartPointer` і надаємо реалізацію методу `drop`, що викликає `println!`. Саме у тілі функції `drop` треба розмістити логіку, яку ви хочете виконати, коли екземпляр типу виходить з області видимості. Тут ми виводимо деякий текст для наочної демонстрації, коли саме Rust викличе `drop`.

У `main` ми створимо два екземпляри `CustomSmartPointer` і виведемо `CustomSmartPointers created`. Наприкінці `main` наші екземпляри `CustomSmartPointer` вийдуть з області видимості, і Rust викличе код, який ми розмістили у методі `drop`, вивівши наше останнє повідомлення. Зверніть увагу що нам не треба явно викликати метод `drop`.

Коли ми запустимо цю програму, то побачимо таке:

```

$ cargo run
Compiling drop-example v0.1.0 (file:///projects/drop-example)
Finished dev [unoptimized + debuginfo] target(s) in 0.60s
Running `target/debug/drop-example`
CustomSmartPointers created.
Dropping CustomSmartPointer with data `other stuff`!
Dropping CustomSmartPointer with data `my stuff`!

```

Rust автоматично викликав для нас `drop`, коли наші екземпляри вийшли з області видимості, виконавши зазначений код. Змінні очищуються у зворотному порядку від

створення, тож `d` буде очищено перед `c`. Мета цього прикладу - надати Вам візуальний посібник того, як працює метод `drop`; зазвичай ви вказуєте код для очищення, який треба запустити вашому типу, а не друкуєте повідомлення.

Завчасне Очищення Значення з `std::mem::drop`

На жаль, зовсім не очевидно, як вимкнути автоматичну функціональність `drop`. Відключати `drop` зазвичай не потрібно; весь сенс трейту `Drop` полягає в тому, що про нього компілятор дбає автоматично. Однак, іноді ви можете захотіти очистити значення завчасно. Візьмемо такий приклад: розумні вказівники, що керують блокуванням; ви можете захотіти примусово запустити метод `drop`, що відпускає блокування, щоб інший код у цій області видимості міг захопити це блокування. Rust не дозволяє вам викликати метод `drop` трейту `Drop` вручну; натомість ви маєте викликати функцію `std::mem::drop`, надану стандартною бібліотекою, якщо ви хочете примусово очистити значення перед до закінчення її області видимості.

Якщо ми спробуємо викликати метод `drop` трейту `Drop` вручну, змінивши функцію `main` з Блоку коду 15-14, як показано у Блоці коду 15-15, то отримаємо помилку компілятора:

Файл: src/main.rs

```
fn main() {
    let c = CustomSmartPointer {
        data: String::from("some data"),
    };
    println!("CustomSmartPointer created.");
    c.drop();
    println!("CustomSmartPointer dropped before the end of main.");
}
```

Блок коду 15-15: спроба викликати метод `drop` з риси `Drop` вручну для раннього очищення

Якщо ми спробуємо скомпілювати цей код, то отримаємо таку помилку:


```
$ cargo run
  Compiling drop-example v0.1.0 (file:///projects/drop-example)
error[E0040]: explicit use of destructor method
  --> src/main.rs:16:7
   |
16 |     c.drop();
   |     ^^^^^^
   |
   | explicit destructor calls not allowed
   | help: consider using `drop` function: `drop(c)`

For more information about this error, try `rustc --explain E0040`.
error: could not compile `drop-example` due to previous error
```

Ця помилка каже, що ми не можемо явно викликати `drop`. Повідомлення використовує загальнопрограмістський термін *деструктор*, що позначає функцію, яка очищує екземпляр. *Деструктор* є аналогом до *конструктора*, який створює екземпляр. Функція `drop` в Rust - це деструктор.

Rust не дозволяє нам викликати `drop` явно, бо Rust все одно автоматично викличе `drop` для цього значення наприкінці `main`. Це спричинить помилку *подвійного звільнення*, бо Rust спробує очистити те саме значення двічі.

Ми не можемо вимкнути автоматичне додавання `drop` там, де значення виходить за межі області видимості, і ми не можемо викликати метод `drop` явно. Тож якщо нам треба змусити значення очиститися раніше, ми використовуємо функцію `std::mem::drop`.

Функція `std::mem::drop` відрізняється від методу `drop` у трейті `Drop`. Ми викликаємо її, передаючи аргументом значення, яке ми хочемо примусово очистити. Ця функція є в прелюдії, таким чином, ми можемо змінити `main` у Блоці коду 15-15, щоб викликати функцію `drop`, як показано в Блоці коду 15-16:

Файл: src/main.rs

```
fn main() {
    let c = CustomSmartPointer {
        data: String::from("some data"),
    };
    println!("CustomSmartPointer created.");
    drop(c);
    println!("CustomSmartPointer dropped before the end of main.");
}
```

Блок коду 15-16: виклик `std::mem::drop`, щоб явно очистити значення до того, як вони вийде з області видимості

Виконання цього коду виведе наступне:

```
$ cargo run
Compiling drop-example v0.1.0 (file:///projects/drop-example)
Finished dev [unoptimized + debuginfo] target(s) in 0.73s
Running `target/debug/drop-example`
CustomSmartPointer created.
Dropping CustomSmartPointer with data `some data`!
CustomSmartPointer dropped before the end of main.
```

Текст `Видалення CustomSmartPointer з даними `деякі дані`!` виводиться між текстами `CustomSmartPointer created.` і `CustomSmartPointer dropped before the end of main.`, показуючи, що код методу `drop` викликається для очищення `c` в цьому місці.

Ви можете використати код, вказаний у реалізації трейту `Drop`, у різні способи, щоб зробити очищення зручним і безпечним: зокрема, ви могли б використати його для створення власного розподільвача пам'яті! Завдяки трейту `Drop` і системі володіння Rust, вам не треба пам'ятати про очищення, бо Rust робить це автоматично.

Також вам не доведеться турбуватися про проблеми, що можуть виникнути через випадкове очищення значень, які все ще використовуються: система власності, яка гарантує, що посилання завжди дійсні, також гарантує, що метод `drop` буде викликаний лише один раз, коли значення більше не використовуватиметься.

Тепер, коли ми дослідили `Box<T>` і деякі характеристики розумних вказівників, погляньмо на деякі інші розумні вказівники, визначені у стандартній бібліотеці.

Rc<T> - Розумний Вказівник з Лічильником Посилань

У більшості випадків володіння є прозорим: ви точно знаєте, яка змінна володіє певним значенням. Проте є випадки, коли одне значення може мати декілька власників.

Наприклад, в структурах даних - графах багато ребер можуть вказувати на один вузол, і цей вузол концептуально є володінням усіх ребер, які ведуть у нього. Вузол не повинен бути очищений, поки існують ребра, які вказують на нього, і тому він має власників.

Вам треба явно дозволити множинне володіння, використовуючи тип Rust **Rc<T>**, що означає *лічильник посилань*. Тип **Rc<T>** відстежує кількість посилань на значення, щоб визначити, чи значення все ще використовується. Якщо лишилося нуль посилань на значення, то значення можна очистити, не зробивши жодне посилання некоректним.

Можна уявити собі **Rc<T>** як телевізор у сімейній кімнаті. Коли одна людина входить, щоб подивитися телевізор, його вмикають. Інші теж можуть зайти до кімнати і подивитися телевізор. Коли остання людина залишає кімнату, телевізор вимикають, бо його більше не використовують. Якщо хтось вимкне телевізор, поки інші все ще його дивитимуться, решта глядачів телевізора обуриться!

Ми використовуємо тип **Rc<T>**, коли ми хочемо виділити деякі дані в купі, щоб різні частини програми могли їх читати, і не можемо визначити під час компіляції, яка частина останньою завершить використовувати ці дані. Якби ми знали, яка частина завершить останньою, то могли б просто надати цій частині володіння, і були б застосовані звичайні правила володіння часу компіляції.

Зверніть увагу, що **Rc<T>** використовується тільки в однопотоківих сценаріях. Коли ми обговорюватимемо конкурентність у Розділі 16, то розглянемо, як облічувати посилання в багатопотокових програмах.

Використання Rc<T> для Спільного Використання Даних

Повернімося до прикладу зі списком Cons з Блоку коду 15-5. Пам'ятайте, що ми визначили його за допомогою **Box<T>**. Цього разу створимо два списки, що спільно володіють третім. Концептуально це схоже на Рисунок 15-3:

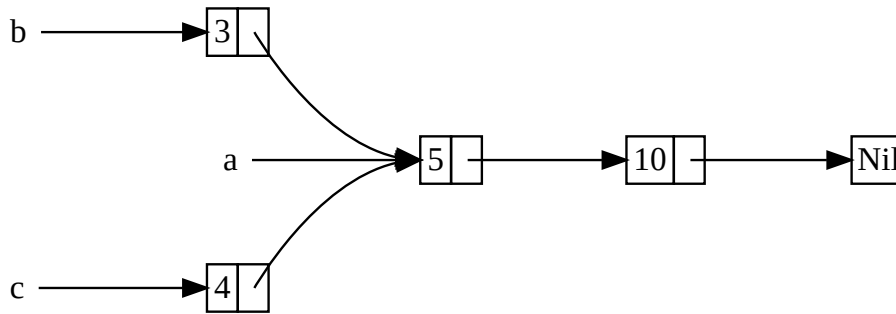


Рисунок 15-3: Два списки, **b** and **c**, спільно володіють третім списком, **a**

Ми створимо список **a**, що містить 5, а потім 10. Тоді ми зробимо ще два списки: **b**, що починається з 3 і **c**, що починається з 4. Обидва списки **b** та **c** далі продовжуються першим списком **a**, що містить 5 і 10. Іншими словами, обидва списки спільно використовують перший, що містить 5 і 10.

Спроба реалізувати цией сценарій за допомогою нашого визначення `List` із `Box<T>` не спрацює, як показано в Блоці коду 15-17:

Файл: `src/main.rs`

```

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

use crate::List::{Cons, Nil};

fn main() {
    let a = Cons(5, Box::new(Cons(10, Box::new(Nil))));
    let b = Cons(3, Box::new(a));
    let c = Cons(4, Box::new(a));
}
  
```

Блок коду 15-17: демонстрація, що ми не можемо мати два списки, створені з `Box<T>`, які намагаються спільно володіти третім списком

Якщо ми скопілюємо цей код, ми отримаємо цю помилку:

```

$ cargo run
  Compiling cons-list v0.1.0 (file:///projects/cons-list)
error[E0382]: use of moved value: `a`
  --> src/main.rs:11:30
   |
9  |         let a = Cons(5, Box::new(Cons(10, Box::new(Nil))));
   |         - move occurs because `a` has type `List`, which does not implement
the `Copy` trait
10 |         let b = Cons(3, Box::new(a));
   |                                - value moved here
11 |         let c = Cons(4, Box::new(a));
   |                                ^ value used here after move

```

For more information about this error, try `rustc --explain E0382`.
 error: could not compile `cons-list` due to previous error

Варіанти `Cons` володіють даними, що в них знаходяться, тож коли ми створюємо список `b`, `a` переміщується в `b` і `b` володіє `a`. Тоді ж, коли ми намагаємося знову використати `a` при створенні `c`, нам це заборонено через те, що `a` було переміщено.

Ми могли б змінити визначення `Cons`, щоб він зберігав посилання, але тоді нам потрібно було б вказати параметри часу існування. Вказуючи параметри часу існування, ми вказуємо, що кожен елемент у списку буде існувати принаймні стільки ж, скільки весь список. Це так для елементів і списків у Блоці коду 15-17, але не для кожного сценарію.

Натомість ми змінимо наше визначення `List`, застосувавши `Rc<T>` замість `Box<T>`, як показано в Блоці коду 15-18. Варіант `Cons` тепер буде складатися зі значення і `Rc<T>`, що вказує на `List`. Коли ми створюємо `b`, замість того, перебрати володіння `a`, ми клонуватимемо `Rc<List>`, що `a` містить, збільшуючи таким чином кількість посилань з одного до двох і дозволяючи `a` та `b` розділити володіння даними в цьому `Rc<List>`. Ми також скопонуємо `a`, коли створюватимемо `c`, збільшуючи кількість посилань з двох до трьох. Кожного разу, як ми викликаємо `Rc::clone`, кількість посилань на дані в `Rc<List>` буде збільшуватися, і дані не будуть очищені, поки кількість посилань на них не сягне нуля.

Файл: src/main.rs

```
enum List {
    Cons(i32, Rc<List>),
    Nil,
}

use crate::List::{Cons, Nil};
use std::rc::Rc;

fn main() {
    let a = Rc::new(Cons(5, Rc::new(Cons(10, Rc::new(Nil)))));
    let b = Cons(3, Rc::clone(&a));
    let c = Cons(4, Rc::clone(&a));
}
```

Блок коду 15-18: визначення `List` за допомогою `Rc<T>`

Нам потрібно додати інструкцію `use`, щоб винести `Rc<T>` в область видимості, оскільки його немає у прелюдії. У `main` ми створюємо список, що складається з 5 і 10 і зберігаємо його у новому `Rc<List>` `a`. Потім ми створюємо `b` і `c`, викликаємо функцію `Rc::clone` і передаємо посилання на `Rc<List>` в `a` як аргумент.

Ми могли б викликати `a.clone()`, а не `Rc::clone(&a)`, але в Rust діє домовленість використовувати в цьому випадку `Rc::clone`. Реалізація `Rc::clone` не робить глибокої копії всіх даних, на відміну від реалізацій реалізації `clone` для більшості типів. Виклик `Rc::clone` тільки збільшує кількість посилань, що не забирає багато часу. Глибокі копії даних можуть зайняти багато часу. Використовуючи `Rc::clone` для обліку посилань, ми можемо візуально розрізнити `clone`, що роблять глибоку копію, і ті, які збільшують кількість посилань. При пошуку проблем з продуктивністю в коді нам потрібно лише розглянути `clone` глибокого копіювання і може не враховувати виклики `Rc::clone`.

Клонування `Rc<T>` Збільшує Значення Лічильника Посилань

Змінімо наш робочий приклад у Блоці коду 15-18 так, щоб ми могли переглянути зміни лічильника посилань, коли ми створюємо і очищуємо посилання на `Rc<List>` в `a`.

У Блоці коду 15-19 ми змінімо `main` так, щоб вона мала внутрішню область видимості навколо списку `c`; тоді ми можемо побачити, як змінюється кількість посилань, коли `c` виходить за межі видимості.

Файл: `src/main.rs`

обговорювалися в Розділі 4: кілька мутабельних позичань до одного місця можуть спричинити гонитву даних і неузгодженість. Але ж мати можливість змінювати дані так зручно! У наступному підрозділі ми обговоримо шаблон внутрішньої мутабельності та тип `RefCell<T>`, який ви можете використовувати разом з `Rc<T>` для роботи з цим обмеженням немутабельності.

`RefCell<T>` та Шаблон Внутрішньої Мутабельності

Внутрішня мутабельність - це шаблон проектування в Rust, що дозволяє вам змінювати дані, навіть якщо є немутабельні посилання на ці дані; зазвичай, ця дія заборонена правилами позичання. Щоб змінювати дані, цей шаблон використовує **небезпечний** код у структурі даних, щоб обійти звичайні правила Rust, що керують мутабельністю та позичанням. Небезпечний код повідомляє компілятор, що ми перевіряємо правила самостійно, а не покладаємося на компілятор, щоб перевіряти їх для нас; ми детальніше поговоримо про небезпечний код у Розділі 19.

Ми можемо використовувати типи, які використовують шаблон внутрішньої мутабельності тільки тоді, коли ми можемо гарантувати дотримання правил позичання під час виконання, тоді як компілятор не може гарантувати цього. **Небезпечний** код застосовується загорнутим у безпечне API, і зовнішній тип лишається немутабельним.

Дослідимо цю концепцію, розглянувши тип `RefCell<T>`, який слідує шаблону внутрішньої мутабельності.

Забезпечення Дотримання Правил Позичання Впродовж Часу Виконання з `RefCell<T>`

На відміну від `Rc<T>`, тип `RefCell<T>` представляє єдине володіння даними, що він містить. То що ж відрізняє `RefCell<T>` від типу на кшталт `Box<T>`? Згадайте правила позичання, які ви вивчили у Розділі 4:

- У будь-який час можна мати *або* одне мутабельне посилання, `<0>`або`/0>` будь-яку кількість немутабельних посилань.
- Посилання завжди мають бути коректними.

За допомогою посилань і `Box<T>` інваріанти правил позичання забезпечуються під час компіляції. За допомогою `RefCell<T>`, ці інваріанти забезпечуються *під час виконання*. При застосуванні посилань, якщо ви порушите ці правила, то отримаєте помилку компілятора. При застосуванні `RefCell<T>`, якщо ви порушите ці правила, ваша програма запанікує і завершиться.

Перевага перевірки правил позичання під час компіляції полягає в тому, що помилки будуть виявлені раніше під час розробки і немає впливу на продуктивність часу виконання, бо весь аналіз проведений заздалегідь. З цих причин перевірка правил

позичання під час компіляції є найкращим вибором у більшості випадків, чому це і є замовчуванням Rust.

Перевагою перевірки правил позичання під час виконання є те, що уможливорюються певні безпечні для пам'яті сценарії, які були б заборонені перевітками часу компіляції. Статичний аналіз, як і компілятор Rust, за своєю природою консервативний. Певні властивості коду неможливо виявити лише аналізом коду: найвідоміший приклад - Проблема зупинки, про яку в цій книзі не йдеться, але це цікава тема для дослідження.

Оскільки певний аналіз неможливий, то якщо компілятор Rust не може бути впевненим, що код відповідає правилам володіння, він може відхилити коректну програму; таким чином, він консервативний. Якби Rust прийняв некоректну програму, користувачі не змогли б довіряти гарантіям, забезпеченим Rust. Однак, якщо Rust відхиляє правильну програму, програмісту буде незручно, але нічого катастрофічного не може статися. Тип `RefCell<T>` є корисним, коли ви впевнені, що ваш код слідує правилам запозичень, але компілятор не в змозі зрозуміти і гарантувати це.

Подібно до `Rc<T>`, `RefCell<T>` призначено лише для використання в однопоточковому сценарії видасть вам помилку часу компіляції, якщо ви спробуєте використати його в багатопотоковому контексті. Ми поговоримо про те, як отримати функціональність `RefCell<T>` в багатопотоковій програмі у Розділі 16.

Ось коротке зведення, коли обирати `Box<T>`, `Rc<T>`, або `RefCell<T>`:

- `Rc<T>` дозволяє декілька власників одних даних; `Box<T>` і `RefCell<T>` мають одного власника.
- `Box<T>` дозволяє немутабельні чи мутабельні позичання, перевірені під час компіляції; `Rc<T>` дозволяє лише немутабельні позичання, перевірені під час компіляції; `RefCell<T>` дозволяє немутабельні чи мутабельні позичання, перевірені під час виконання.
- Оскільки `RefCell<T>` дозволяє мутабельні позичання, перевірені під час виконання, ви можете змінити значення всередині `RefCell<T>`, навіть коли `RefCell<T>` є немутабельним.

Зміна значення всередині немутабельного значення - це шаблон *внутрішньої мутабельності*. Подивімося на ситуацію, в якій внутрішня мутабельність корисна і дослідимо, як її використовувати.

Внутрішня Мутабельність: Мутабельне Позичання Немутабельного Значення

З правил запозичення випливає, що якщо ви маєте немутабельне значення, то ви не можете його мутабельно позичити. Наприклад, цей код не компілюється:

```
fn main() {  
    let x = 5;  
    let y = &mut x;  
}
```

Якби ви спробували скомпілювати цей код, то отримали б таку помилку:

```
$ cargo run  
    Compiling borrowing v0.1.0 (file:///projects/borrowing)  
error[E0596]: cannot borrow `x` as mutable, as it is not declared as mutable  
--> src/main.rs:3:13  
   |  
2  |     let x = 5;  
   |         - help: consider changing this to be mutable: `mut x`  
3  |     let y = &mut x;  
   |             ^^^^^^ cannot borrow as mutable  
  
For more information about this error, try `rustc --explain E0596`.  
error: could not compile `borrowing` due to previous error
```

Проте, існують ситуації, в яких було б зручним для значення змінювати себе у своїх методах, але виглядати немутабельним для іншого коду. Код за межами методів цього значення не має можливості змінювати значення. Використання `RefCell<T>` є одним зі способів отримати можливість внутрішньої мутабельності, але `RefCell<T>` не повністю оминає правила позичання: borrow checker у компіляторі дозволяє цю внутрішню мутабельність, і правила позичання перевіряються під час виконання програми. Якщо ви порушите ці правила, ви отримаєте `panic!` замість помилки компілятора.

Пропрацюємо практичний приклад, де ми можемо використати `RefCell<T>` для зміни немутабельного значення і побачити, чому це корисно.

Сценарій Використання Внутрішньої Мутабельності: Імітаційні Об'єкти

Іноді під час тестування програміст може використовувати тип замість іншого типу, для того, щоб отримати певну поведінку та перевірити коректність його реалізації. Такий тип-замінник зветься *тест-дублером*. Можна розглядати його як "дублера-каскадера" у фільмі, де інша людина замінює актора для виконання певної ризикованої сцени. Тест-дублери замінюють інші типи при виконанні тестів. *Імітаційні об'єкти* - це спеціальні типи тест-

дублерів, що записують, що відбувається під час тесту, щоб ви могли перевірити, що мали місце правильні дії.

У Rust немає об'єктів у тому ж сенсі, в якому вони є в інших мовах, і в Rust немає вбудованої функціональності імітаційних об'єктів у стандартній бібліотеці, як в деяких інших мовах. Однак, ви точно можете створити структуру, що буде служити тій же меті, що й імітаційний об'єкт.

Ось цей сценарій ми будемо тестувати: ми створимо бібліотеку, яка буде відстежувати значення до максимального значення і надсилатиме повідомлення залежно від того, наскільки близьке поточне значення до максимального значення. Цю бібліотеку можна використовувати, наприклад, для відстеження квоти користувача на кількість викликів API, які їм дозволено зробити.

Наша бібліотека забезпечить тільки функціональність відстеження, наскільки близьким до максимального є значення і коли та якими мають бути повідомлення. Очікується, що застосунки, які використовують нашу бібліотеку, забезпечать механізм відправлення повідомлень: застосунок може відправити повідомлення у застосунок, надіслати електронного листа, текстове повідомлення або щось інше. Бібліотека не має знати про такі подробиці. Все, що їй потрібно - щось, що реалізує наданий нами трейт, що зветься **Messenger**. Блок коду 15-20 показує код бібліотеки:

Файл: `src/lib.rs`

```

pub trait Messenger {
    fn send(&self, msg: &str);
}

pub struct LimitTracker<'a, T: Messenger> {
    messenger: &'a T,
    value: usize,
    max: usize,
}

impl<'a, T> LimitTracker<'a, T>
where
    T: Messenger,
{
    pub fn new(messenger: &'a T, max: usize) -> LimitTracker<'a, T> {
        LimitTracker {
            messenger,
            value: 0,
            max,
        }
    }

    pub fn set_value(&mut self, value: usize) {
        self.value = value;

        let percentage_of_max = self.value as f64 / self.max as f64;

        if percentage_of_max >= 1.0 {
            self.messenger.send("Error: You are over your quota!");
        } else if percentage_of_max >= 0.9 {
            self.messenger
                .send("Urgent warning: You've used up over 90% of your quota!");
        } else if percentage_of_max >= 0.75 {
            self.messenger
                .send("Warning: You've used up over 75% of your quota!");
        }
    }
}

```

Блок коду 15-20: бібліотека для відстеження наближення значення до максимального значення і попередження, коли значення сягає певних рівнів

Важливою частиною цього коду є те, що трейт `Messenger` має один метод, що зветься `send`, який приймає немутабельне посилання на `self` і текст повідомлення. Цей трейт є інтерфейсом нашого імітаційного об'єкта, який треба реалізувати, щоб імітаційний об'єкт можна було використовувати так само, як і реальний об'єкт. Іншою важливою частиною є те, що ми хочемо перевірити поведінку методу `set_value` у `LimitTracker`. Ми можемо змінити значення, що передається як параметр `value`, але `set_value` не поверне нам нічого, на чому можна робити тестові твердження. Ми хочемо мати змогу сказати, що якщо ми створюємо `LimitTracker` з чимось, що реалізує трейт `Messenger` і конкретним

значенням `max`, то коли ми передаємо різні числа для `value`, месенджеру накажуть відправляти відповідні повідомлення.

Нам потрібен імітаційний об'єкт, який, замість того, щоб надіслати електронне або текстове повідомлення коли ми викликаємо `send`, лише стежитиме за повідомленнями, які йому сказано надіслати. Ми можемо створити новий екземпляр імітаційного об'єкта, створити `LimitTracker`, який використовує цей імітаційний об'єкт, викликати метод `set_value` для `LimitTracker` і перевірити, чи цей імітаційний об'єкт містить повідомлення, на які ми очікуємо. Блок коду 15-21 показує спробу реалізувати імітаційний об'єкт, що робить саме це, але borrow checker не дозволяє так робити:

Файл: `src/lib.rs`

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

    struct MockMessenger {
        sent_messages: Vec<String>,
    }

    impl MockMessenger {
        fn new() -> MockMessenger {
            MockMessenger {
                sent_messages: vec![],
            }
        }
    }

    impl Messenger for MockMessenger {
        fn send(&self, message: &str) {
            self.sent_messages.push(String::from(message));
        }
    }

    #[test]
    fn it_sends_an_over_75_percent_warning_message() {
        let mock_messenger = MockMessenger::new();
        let mut limit_tracker = LimitTracker::new(&mock_messenger, 100);

        limit_tracker.set_value(80);

        assert_eq!(mock_messenger.sent_messages.len(), 1);
    }
}
```

Блок коду 15-21: спроба реалізувати `MockMessenger`, не дозволена borrow checker

Файл: src/lib.rs

```

#[cfg(test)]
mod tests {
    use super::*;
    use std::cell::RefCell;

    struct MockMessenger {
        sent_messages: RefCell<Vec<String>>,
    }

    impl MockMessenger {
        fn new() -> MockMessenger {
            MockMessenger {
                sent_messages: RefCell::new(vec![]),
            }
        }
    }

    impl Messenger for MockMessenger {
        fn send(&self, message: &str) {
            self.sent_messages.borrow_mut().push(String::from(message));
        }
    }

    #[test]
    fn it_sends_an_over_75_percent_warning_message() {
        // --snip--

        assert_eq!(mock_messenger.sent_messages.borrow().len(), 1);
    }
}

```

Блок коду 15-22: Використання `RefCell<T>`, щоб змінити внутрішнє значення, поки зовнішнє значення вважається немутабельним

Поле `sent_messages` тепер має тип `RefCell<Vec<String>>`, а не `Vec<String>`. У функції `new` ми створюємо новий екземпляр `RefCell<Vec<String>>` навколо порожнього вектора.

Для реалізації метода `send` перший параметр все ще є немутабельним позичанням `self`, що відповідає за визначенню трейта. Ми викликаємо `borrow_mut` для `RefCell<Vec<String>>` у `self.sent_messages`, щоб отримати мутабельне посилання на значення всередині `RefCell<Vec<String>>`, тобто вектор. Тоді ми можемо викликати `push` для мутабельного посилання на вектор, щоб зберігати повідомлення, надіслані протягом тесту.

Остання зміна, яку ми повинні зробити - в твердженні тесту: щоб подивитись, скільки елементів є у внутрішньому векторі, ми викликаємо `borrow` для `RefCell<Vec<String>>`,

щоб отримати немутабельне посилання на вектор.

Тепер, коли ви побачили, як використовувати `RefCell<T>`, зануримось у те, як воно працює!

Відстеження Позичань Впродовж Часу Виконання з `RefCell<T>`

Створюючи немутабельні і мутабельні посилання, ви використовуємо, відповідно, записи `&` та `&mut`. Для `RefCell<T>` ми використовуємо методи `borrow` і `borrow_mut`, які є частиною безпечного API `RefCell<T>`. Метод `borrow` повертає розумний вказівник типу `Ref<T>`, а `borrow_mut` повертає розумний вказівник типу `RefMut<T>`. Обидва типи реалізують `Deref`, тому ми можемо працювати з ними як зі звичайними посиланнями.

`RefCell<T>` відстежує, скільки є активних розумних вказівників `Ref<T>` і `RefMut<T>` у кожен момент. Кожного разу коли ми викликаємо `borrow`, `RefCell<T>` збільшує кількість активних немутабельних позичань. Коли значення `Ref<T>` виходить з області видимості, кількість немутабельних позичань зменшується на один. Так само як правила позичання часу компіляції, `RefCell<T>` дозволяє мати багато немутабельних позичань або одне мутабельне позичання в будь-який момент часу.

Якщо ми спробуємо порушити ці правила, то замість помилки компілятора, як це стається з посиланнями, реалізація `RefCell<T>` запанікує під час виконання. Блок коду 15-23 показує зміну реалізації `send` з Блоку коду 15-22. Ми навмисно намагаємось створити два немутабельні позичання активними в одній області видимості, щоб продемонструвати, що `RefCell<T>` запобігає цьому під час виконання.

Файл: `src/lib.rs`

```
impl Messenger for MockMessenger {
    fn send(&self, message: &str) {
        let mut one_borrow = self.sent_messages.borrow_mut();
        let mut two_borrow = self.sent_messages.borrow_mut();

        one_borrow.push(String::from(message));
        two_borrow.push(String::from(message));
    }
}
```

Блок коду 15-23: створення двох мутабельних посилань в одній області видимості, щоб побачити, що `RefCell<T>` запанікує

Ми створюємо змінну `one_borrow` для розумного вказівника `RefMut<T>`, повернутого з `borrow_mut`. Потім так само створюємо ще одне мутабельне позичання в змінній `two_borrow`. Це створює два мутабельні позичання в одній області видимості, що є

забороненим. Коли ми запускаємо тести для нашої бібліотеки, код з Блоку коду 15-23 скомпілюється без будь-яких помилок, але тест не провалиться:

```
$ cargo test
Compiling limit-tracker v0.1.0 (file:///projects/limit-tracker)
Finished test [unoptimized + debuginfo] target(s) in 0.91s
Running unittests src/lib.rs (target/debug/deps/limit_tracker-
e599811fa246dbde)

running 1 test
test tests::it_sends_an_over_75_percent_warning_message ... FAILED

failures:

---- tests::it_sends_an_over_75_percent_warning_message stdout ----
thread 'main' panicked at 'already borrowed: BorrowMutError', src/lib.rs:60:53
note: run with `RUST_BACKTRACE=1` environment variable to display a backtrace

failures:
    tests::it_sends_an_over_75_percent_warning_message

test result: FAILED. 0 passed; 1 failed; 0 ignored; 0 measured; 0 filtered out;
finished in 0.00s

error: test failed, to rerun pass `--lib`
```

Зверніть увагу, що код панікував з повідомленням `already borrowed: BorrowMutError`. Ось так `RefCell<T>` обробляє порушення правил позичання під час виконання.

Перехоплення помилок під час виконання, а не під час компіляції, як ми зробили це тут, означає що ви потенційно знаходитимете помилки у вашому коді пізніше під час розробки. Можливо, лише тоді, коли ваш код уже буде розгорнуто у кінцевого користувача. Крім того, ваш код матиме незначне зниження продуктивності у піл час виконання у результаті відстеження позичань під час виконання замість часу компіляції. Проте використання `RefCell<T>` уможливорює запис імітаційних об'єктів, які можуть змінювати себе, щоб відстежувати повідомлення, які вони отримували під час використання в контексті, де допускаються лише немутабельні значення. Ви можете використовувати `RefCell<T>` не зважаючи на його недоліки, щоб отримати більше функціональності, ніж надають звичайні посилання.

Декілька Власників Мутабельних Даних з Поєднанням `Rc<T>` та `RefCell<T>`

Звичайний спосіб використання є `RefCell<T>` - комбінація з `Rc<T>`. Згадайте, що `Rc<T>` дозволяє мати кілька володільців одних даних, але надає лише немутабельний доступ до

цих даних. Якщо ви маєте `Rc<T>`, що містить `RefCell<T>`, ви можете отримати значення, що може мати кількох власників і яке ви можете змінювати!

Наприклад, згадайте приклад зі списком `cons` у Блоці коду 15-18, де ми використовували `Rc<T>`, щоб дозволити декільком спискам ділитися володінням іншим списком. Оскільки `Rc<T>` має лише немутабельні значення, ми не можемо змінити жодне зі значень у списку після їх створення. Додамо `RefCell<T>`, щоб отримати можливість змінити значення у списках. Блок коду 15-24 показує, що використовуючи `RefCell<T>` у визначенні `Cons` ми можемо змінити значення, збережене у всіх списках:

Файл: `src/main.rs`

```
#[derive(Debug)]
enum List {
    Cons(Rc<RefCell<i32>>, Rc<List>),
    Nil,
}

use crate::List::{Cons, Nil};
use std::cell::RefCell;
use std::rc::Rc;

fn main() {
    let value = Rc::new(RefCell::new(5));

    let a = Rc::new(Cons(Rc::clone(&value), Rc::new(Nil)));

    let b = Cons(Rc::new(RefCell::new(3)), Rc::clone(&a));
    let c = Cons(Rc::new(RefCell::new(4)), Rc::clone(&a));

    *value.borrow_mut() += 10;

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

Блок коду 15-24: використання `Rc<RefCell<i32>>` для створення `List`, який ми можемо змінювати

Ми створюємо значення, що є екземпляром `Rc<RefCell<i32>>`, і зберігаємо його у змінній з назвою `value`, щоб пізніше мати можливість доступу до нього. Потім створили `List` в `a` з варіантом `Cons`, що містить `value`. Ми маємо клонувати `value`, щоб обидва `a` і `value` мали володіння над внутрішнім значенням `5` замість передачі володіння з `value` до `a` чи щоб `a` позичало `value`.

Ми обгорнули список `a` у `Rc<T>`, тож коли ми створюємо списки `b` та `c`, вони обидва можуть посилатися на `a`, як ми робили у Блоці коду 15-18.

Після створення списків у `a`, `b` і `c`, ми хочемо додати 10 до значення в `value`. Ми зробимо це, викликавши `borrow_mut` для `value`, що використовує автоматичне розіменування, обговорене в Розділі 5 (див. підрозділ "А де ж оператор `->?`"), для розіменування `Rc<T>` до внутрішнього значення `RefCell<T>`. Метод `borrow_mut` повертає розумний вказівник `RefMut<T>`, і ми використовуємо на ньому оператор розіменування та змінюємо внутрішнє значення.

Коли ми виводимо `a`, `b` та `c`, ми бачимо, що всі вони мають змінене значення 10 замість 5:

```
$ cargo run
Compiling cons-list v0.1.0 (file:///projects/cons-list)
Finished dev [unoptimized + debuginfo] target(s) in 0.63s
Running `target/debug/cons-list`
a after = Cons(RefCell { value: 15 }, Nil)
b after = Cons(RefCell { value: 3 }, Cons(RefCell { value: 15 }, Nil))
c after = Cons(RefCell { value: 4 }, Cons(RefCell { value: 15 }, Nil))
```

Ця техніка дуже акуратна! Використовуючи `RefCell<T>`, ми маємо зовнішньо немутабельне значення `List`. Але ми можемо використати методи `RefCell<T>`, які надають доступ до його внутрішньої мутабельності, тож ми можемо змінювати наші дані в разі потреби. Перевірка правил запозичення часу виконання захищає нас від гонитви даних, і це іноді варто виміняти на крихту швидкості швидкістю заради цієї гнучкості в наших структурах даних. Зверніть увагу, що `RefCell<T>` не працює в багатопотоковому коді! `Mutex<T>` є потоково-безпечною версією `RefCell<T>`, і ми обговоримо `Mutex<T>` в Розділі 16.

Цикли Посилань Можуть Спричинити Витік Пам'яті

Гарантії безпеки пам'яті Rust, ускладнюють, але не унеможливлюють, випадкове створення пам'яті, що ніколи не було очищеною (це зветься *витік пам'яті*). Повне запобігання витокам пам'яті не є однією з гарантій Rust, тобто витоки пам'яті вважаються безпечними в Rust. Як ми можемо бачити, Rust дозволяє витоки пам'яті за допомогою `Rc<T>` і `RefCell<T>`: можна створити посилання, де елементи посилаються один на одного в циклі. Це створює витік пам'яті, бо лічильник посилань кожного елементу в циклі ніколи не сягне 0, і значення ніколи не будуть очищені.

Створення Циклу Посилань

Подивімося, як може виникнути цикл посилань і як цьому запобігти, почавши з визначення енуму `List` і методу `tail` у Блоці коду 15-25:

Файл: src/main.rs

```
use crate::List::{Cons, Nil};
use std::cell::RefCell;
use std::rc::Rc;

#[derive(Debug)]
enum List {
    Cons(i32, RefCell<Rc<List>>),
    Nil,
}

impl List {
    fn tail(&self) -> Option<&RefCell<Rc<List>>> {
        match self {
            Cons(_, item) => Some(item),
            Nil => None,
        }
    }
}

fn main() {}
```

Блок коду 15-25: визначення списку `cons`, що містить `RefCell<T>`, щоб ми могли змінювати, на що посилається варіант `Cons`

Ми використовуємо інший різновид визначення `List`, ніж у Блоці коду 15-5. Другий елемент у варіанті `Cons` тепер є `RefCell<Rc<List>>`, що означає, що замість можливості

змінювати значення `i32`, як це було в Блоці коду 15-24, ми хочемо змінити значення `List`, на яке вказує варіант `Cons`. Ми також додаємо метод `tail`, щоб зробити зручним доступ до другого елементу в варіанті `Cons`.

У Блоці коду 15-26 ми додаємо функцію `main`, що використовує визначення з Блока коду 15-25. Цей код створює список `a` і список `b`, що вказує на список `a`. Тоді він змінює список `a` так, що той вказує на `b`, утворивши цикл посилань. Вддовж усього шляху розставлені інструкції `println!`, щоб показати значення лічильників посилань в різних точках цього процесу.

Файл: `src/main.rs`

```
fn main() {
    let a = Rc::new(Cons(5, RefCell::new(Rc::new(Nil))));

    println!("a initial rc count = {}", Rc::strong_count(&a));
    println!("a next item = {:?}", a.tail());

    let b = Rc::new(Cons(10, RefCell::new(Rc::clone(&a))));

    println!("a rc count after b creation = {}", Rc::strong_count(&a));
    println!("b initial rc count = {}", Rc::strong_count(&b));
    println!("b next item = {:?}", b.tail());

    if let Some(link) = a.tail() {
        *link.borrow_mut() = Rc::clone(&b);
    }

    println!("b rc count after changing a = {}", Rc::strong_count(&b));
    println!("a rc count after changing a = {}", Rc::strong_count(&a));

    // Uncomment the next line to see that we have a cycle;
    // it will overflow the stack
    // println!("a next item = {:?}", a.tail());
}
```

Блок коду 15-26: створення циклу посилань з двох значень `List`, що вказують одне на іншого

Ми створили екземпляр `Rc<List>`, що містить значення `List`, у змінній `a`, з початковим списком `5, nil`. Далі ми створюємо екземпляр `Rc<List>`, що містить інше значення `List`, у змінній `b`, зі значенням 10, що вказує на список `a`.

Ми змінюємо `a` так, що воно вказує на `b` замість `nil`, утворивши цикл. Ми робимо це за допомогою методу `tail`, щоб отримати посилання на `RefCell<Rc<List>>` у `a`, який ми кладемо у змінну `link`. Потім ми використовуємо метод `borrow_mut` для

`RefCell<Rc<List>>`, щоб змінити значення всередині з `Rc<List>`, що містить значення `Nil`, на `Rc<List>` в `b`.

Коли ми запустимо цей код із закоментованим поки що останнім `println!`, то отримаємо таке:

```
$ cargo run
  Compiling cons-list v0.1.0 (file:///projects/cons-list)
  Finished dev [unoptimized + debuginfo] target(s) in 0.53s
  Running `target/debug/cons-list`
a initial rc count = 1
a next item = Some(RefCell { value: Nil })
a rc count after b creation = 2
b initial rc count = 1
b next item = Some(RefCell { value: Cons(5, RefCell { value: Nil }) })
b rc count after changing a = 2
a rc count after changing a = 2
```

Лічильник посилань екземплярів `Rc<List>` в обох `a` і `b` має значення 2 після того, як ми змінюємо список в `a`, щоб він вказував на `b`. У кінці `main` Rust очищує змінну `b`, яка зменшує лічильник посилань екземпляра `Rc<List>` `b` з 2 до 1. Пам'ять, яку `Rc<List>` тримає в купі, не буде скинуто у в цей момент, тому що лічильник посилань дорівнює 1, а не 0. Потім Rust очищує `a`, так само зменшуючи кількість посилань екземпляра `Rc<List>` `a` з 2 до 1. Пам'ять цього екземпляра також не може бути очищена, тому що інший екземпляр `Rc<List>` досі посилається на неї. Пам'ять, виділена для списку, залишиться незвільненою назавжди. Щоб візуалізувати цей цикл посилань, ми створили діаграму на Рисунку 15-4.

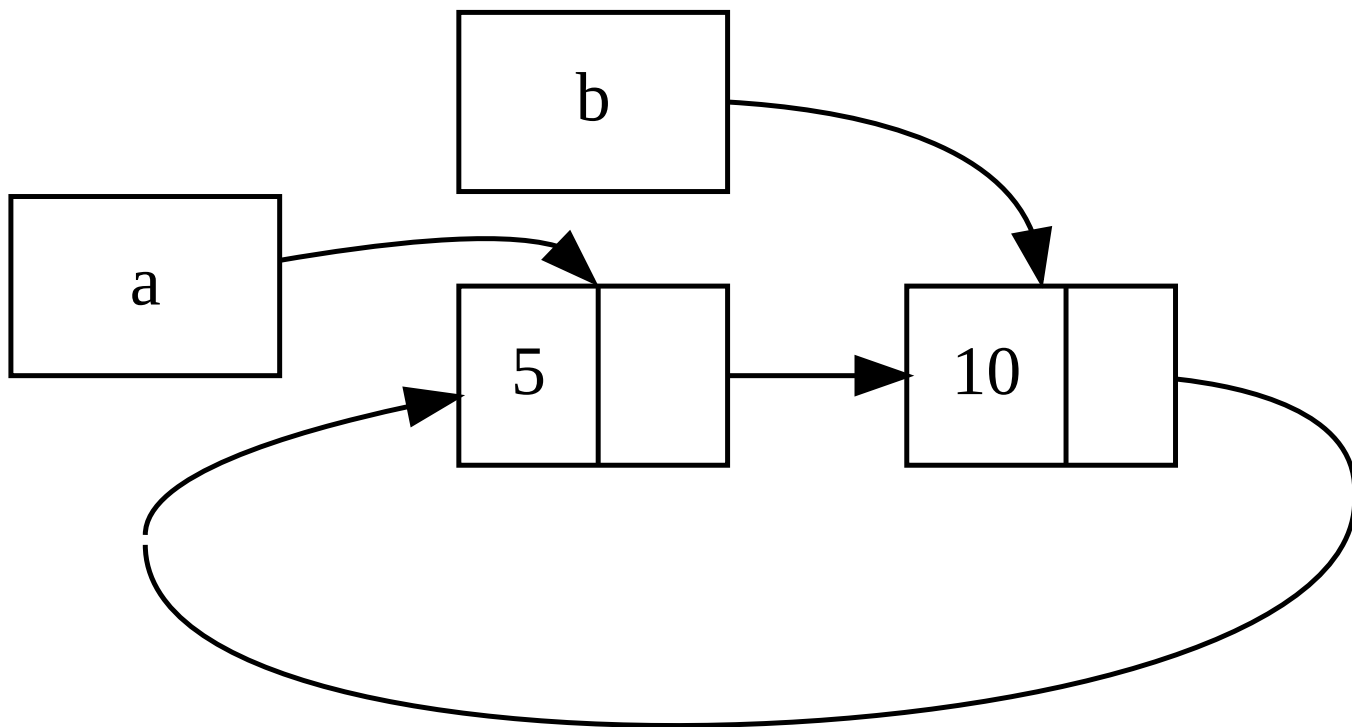


Рисунок 15-4: цикл посилань у списках `a` і `b`, що вказують один на іншого

Якщо ви розкоментуєте останній `println!` і запустите програму, Rust спробує надрукувати цей цикл з `a`, що вказує на `b` що вказує на `a` і так далі, поки стек не переповниться.

Порівняно з реальними програмами, наслідки створення циклу посилань в цьому прикладі не дуже страшні: одразу після того, як ми створили цикл посилань, програма завершується. Однак, якщо складніша програма виділяє багато пам'яті в циклі працює досить тривалий час, програма буде використовувати більше пам'яті, ніж їй потрібно і може перенавантажити систему, призвівши до вичерпання доступної пам'яті.

Створити цикл посилань - не проста задача, але й не неможлива. Якщо у вас є значення `RefCell<T>`, що містять `Rc<T>` або аналогічну вкладену комбінацію типів з внутрішньою мутабельністю і лічильником посилань, ви повинні переконатися, що не створюєте циклів; ви не можете розраховувати на те, що Rust їх виявить. Створення циклу посилань буде логічною помилкою у вашій програмі, і ви маєте використовувати автоматизовані тести, надавати код для огляду іншим програмістам та інші практики розробки програм, щоб мінімізувати їхню можливість.

Іншим рішенням для уникнення циклів посилань є реорганізація ваших структур даних структур так, щоб деякі посилання виражали володіння, а деякі ні. У результаті можуть виникати цикли, утворені кількома відносинами володіння і кількома без володіння, і тільки відносини володіння працюють впливають на те, чи можна очистити значення. У Блоці коду 15-25 ми завжди хочемо, щоб варіант `Cons` володів списком, тому

реорганізація структури даних неможлива. Подивімося на приклад графу, зробленого з батьківських і дочірніх вузлів, щоб побачити, коли відносини без володіння є адекватним способом запобігти циклу посилань.

Запобігання Циклам Посилань: Перетворення `Rc<T>` на `Weak<T>`

Поки що ми продемонстрували, що виклик `Rc::clone` збільшує `strong_count` у екземплярі `Rc<T>`, і екземпляр `Rc<T>` очищується лише якщо його `strong_count` дорівнює 0. Ви також можете створити *слабке посилання* на значення в екземплярі `Rc<T>`, викликавши `Rc::downgrade` і передавши посилання до `Rc<T>`. Сильні посилання - це спосіб поділитися володінням екземпляром `Rc<T>`. Слабкі посилання не виражають відношення володіння і їхня кількість не впливає на те, коли екземпляр `Rc<T>` буде очищено. Вони не викликають циклу посилань, оскільки будь-який цикл, який передбачає деякі слабкі посилання, буде зламано, коли лічильник сильних посилань набуде значення 0.

Коли ви викликаєте `Rc::downgrade`, ви отримуєте розумний вказівник типу `Weak<T>`. Замість збільшувати `strong_count` у екземплярі `Rc<T>` на 1, виклик `Rc::downgrade` збільшує `weak_count` на 1. Тип `Rc<T>` тип використовує `weak_count`, щоб відстежувати, скільки існує посилань `Weak<T>`, подібно до `strong_count`. Різниця в тому, що `weak_count` не має бути 0, щоб екземпляр `Rc<T>` був очищений.

Оскільки значення, на яке посилається `Weak<T>`, може бути очищене, щоб зробити будь-що зі значенням, на яке вказує `Weak<T>`, ви маєте переконаватися, що воно ще існує. Це можна зробити, викликавши метод `upgrade` екземпляру `Weak<T>`, який поверне `Option<Rc<T>>`. Ви отримаєте результат `Some`, якщо значення `Rc<T>` ще не було очищене, і результат `None`, якщо значення `Rc<T>` було очищене. Оскільки `upgrade` повертає `Option<Rc<T>>` Rust гарантує, що варіанти `Some` і `None` будуть оброблені, і не буде некоректного вказівника.

Як приклад, замість того, щоб використовувати список, елементи якого знають лише про наступний елемент, ми створимо дерево, чиї елементи будуть знати про дочірні елементи і про свої батьківські елементи.

Створення Деревоподібної Структури Даних: `Node` з Дочірніми Вузлами

Для початку ми побудуємо дерево з вузлами, що знають про свої дочірні вузли. Ми створимо структуру `Node`, що міститиме значення `i32`, а також посилання на свої дочірні значення `Node`:

Файл: `src/main.rs`

```
use std::cell::RefCell;
use std::rc::Rc;

#[derive(Debug)]
struct Node {
    value: i32,
    children: RefCell<Vec<Rc<Node>>>,
}
```

Ми хочемо, щоб `Node` володів своїми дочірніми вузлами, і хочемо ділитися цим володінням зі змінними, щоб ми могли отримати доступ до кожного `Node` в дереві безпосередньо. Для цього ми визначаємо `Vec<T>` елементів, значення яких мають тип `Rc<Node>`. Ми також хочемо змінити, які вузли є дочірніми для іншого вузла, тож ми маємо в `children` `RefCell<T>` навколо `Vec<Rc<Node>>`.

Далі ми використаємо визначення нашої структури і створимо один екземпляр `Node` з назвою `leaf`, значенням 3 і без дочірніх вузлів, і інший екземпляр з назвою `branch`, значенням 5 і `leaf` як один з дочірніх вузлів, як показано у Блоці коду 15-27:

Файл: src/main.rs

```
fn main() {
    let leaf = Rc::new(Node {
        value: 3,
        children: RefCell::new(vec![]),
    });

    let branch = Rc::new(Node {
        value: 5,
        children: RefCell::new(vec![Rc::clone(&leaf)]),
    });
}
```

Блок коду 15-27: створення вузла `leaf` без дочірніх вузлів і вузла `branch` з одним дочірнім вузлом `leaf`

Ми клонуємо `Rc<Node>` в `leaf` і зберігаємо його в `branch`, що означає, що `Node` в `leaf` має два володільці: `leaf` і `branch`. Ми можемо дістатися з `branch` до `leaf` через `branch.children`, але немає жодного способу дістатися з `leaf` до `branch`. Причина в тому, що `leaf` не має посилання на `branch` і не знає, що вони пов'язані. Нам потрібно, щоб `leaf` знав, що `branch` - це його батьківський елемент. Цим ми й займемося.

Додавання Посилання з Дочірнього Вузла на Батьківський Вузол

Щоб надати дочірньому вузлу інформацію про батьківський, ми повинні додати поле `parent` до визначення структури `Node`. Проблема в тому, щоб вирішити, якого типу має бути `parent`. Ми знаємо, що він не може містити `Rc<T>`, бо це створить цикл посилань, адже `leaf.parent` вказуватиме на `branch`, а `branch.children` вказуватиме на `leaf`, що призведе до того, що їхні значення `strong_count` ніколи не стануть 0.

Подумаємо про відносини іншим чином: батьківський вузол повинен володіти дочірніми, і якщо батьківський вузол очищується, його дочірні вузли також мають бути очищені. Однак дочірній вузол не має володіти батьківським: якщо ми очищуємо дочірній вузол, батьківський має лишитися. Це якраз випадок для слабких посилань!

Отже, замість `Rc<T>`, для типу `parent` скористаємося `Weak<T>`, а точніше, `RefCell<Weak<Node>>`. Тепер наш вузол `Node` виглядає наступним чином:

Файл: `src/main.rs`

```
use std::cell::RefCell;
use std::rc::{Rc, Weak};

#[derive(Debug)]
struct Node {
    value: i32,
    parent: RefCell<Weak<Node>>,
    children: RefCell<Vec<Rc<Node>>>,
}
```

Вузол тепер може посилатися на батьківський вузол, але не володіє ним. У Блоці коду 15-28 ми оновлюємо `main`, щоб використати нове визначення, так щоб вузол `leaf` мав спосіб послатися на батьківський вузол `branch`:

Файл: `src/main.rs`

```
fn main() {
    let leaf = Rc::new(Node {
        value: 3,
        parent: RefCell::new(Weak::new()),
        children: RefCell::new(vec![]),
    });

    println!("leaf parent = {:?}", leaf.parent.borrow().upgrade());

    let branch = Rc::new(Node {
        value: 5,
        parent: RefCell::new(Weak::new()),
        children: RefCell::new(vec![Rc::clone(&leaf)]),
    });

    *leaf.parent.borrow_mut() = Rc::downgrade(&branch);

    println!("leaf parent = {:?}", leaf.parent.borrow().upgrade());
}
```

Блок коду 15-28: вузол `leaf` зі слабким посиланням на свій батьківський вузол `branch`

Створення вузла `leaf` виглядає схожим на Блок коду 15-27, за винятком поля `parent`: `leaf` спершу не має батьківського вузла, тож ми створюємо новий, порожній екземпляр посилання `Weak<Node>`.

На цей момент, коли ми намагаємося отримати посилання на батьківський вузол вузла `leaf` за допомогою методу `upgrade`, то отримуємо значення `None`. Ми бачимо це з того, що виводить перша інструкція `println!`:

```
leaf parent = None
```

Коли ми створюємо вузол `branch`, він також матиме нове посилання `Weak<Node>` в полі `parent`, оскільки `branch` не має батьківського вузла. Але, як і раніше, `leaf` є одним з дочірніх посилань у `branch`. Але коли ми вже маємо екземпляр `Node` у `branch`, то ми можемо змінити `leaf`, щоб дати йому посилання `Weak<Node>` на його батька. Ми використовуємо метод `borrow_mut` для `RefCell<Weak<Node>>` в полі `parent` змінної `leaf`, а потім ми використовуємо функцію `Rc::downgrade`, щоб створити посилання `Weak<Node>` на `branch` з `Rc<Node>` у `branch`.

Коли ми ще раз виводимо батька `leaf` ще раз, цього разу ми отримуємо варіант `Some`, що містить `branch`: тепер `leaf` може отримати доступ свого батька! Коли ми виводимо `leaf`, то також уникаємо циклу, який врешті-решт переповнив би стеку, як було у Блоці коду 15-26; посилання `Weak<Node>` виводяться як `(Weak)`:

```
leaf parent = Some(Node { value: 5, parent: RefCell { value: (Weak) },
children: RefCell { value: [Node { value: 3, parent: RefCell { value: (Weak) },
children: RefCell { value: [] } } ] } })
```

Відсутність нескінченного виведення показує, що цей код не створив циклу посилань. Ми також можемо сказати це, переглянувши значення, які ми отримуємо від виклику

`Rc::strong_count` та `Rc::weak_count`.

Візуалізація Змін `strong_count` та `weak_count`

Подивімося, як змінюються значення `strong_count` і `weak_count` екземплярів `Rc<Node>`, створивши нову внутрішню область видимості і перемістивши туди створення `branch`. Зробивши це, ми зможемо побачити, що відбувається, коли `branch` створюється, а потім очищується, коли виходить за межі області видимості. Зміни показані у Блоці коду 15-29:

Файл: `src/main.rs`

```

fn main() {
    let leaf = Rc::new(Node {
        value: 3,
        parent: RefCell::new(Weak::new()),
        children: RefCell::new(vec![]),
    });

    println!(
        "leaf strong = {}, weak = {}",
        Rc::strong_count(&leaf),
        Rc::weak_count(&leaf),
    );

    {
        let branch = Rc::new(Node {
            value: 5,
            parent: RefCell::new(Weak::new()),
            children: RefCell::new(vec![Rc::clone(&leaf)]),
        });

        *leaf.parent.borrow_mut() = Rc::downgrade(&branch);

        println!(
            "branch strong = {}, weak = {}",
            Rc::strong_count(&branch),
            Rc::weak_count(&branch),
        );

        println!(
            "leaf strong = {}, weak = {}",
            Rc::strong_count(&leaf),
            Rc::weak_count(&leaf),
        );
    }

    println!("leaf parent = {:?}", leaf.parent.borrow().upgrade());
    println!(
        "leaf strong = {}, weak = {}",
        Rc::strong_count(&leaf),
        Rc::weak_count(&leaf),
    );
}

```

Блок коду 15-29: створення `branch` у внутрішній області видимості і перевірка лічильників сильних та слабких посилань

Після створення `leaf` його `Rc<Node>` налічує 1 сильне посилання і 1 слабе. У внутрішній області видимості ми створюємо `branch` і пов'язуємо її з `leaf`. У цей момент, коли ми виводимо лічильники, `Rc<Node>` у `branch` матиме лічильник сильних 1 і слабких 1 (у `leaf.parent`, що вказує на `branch` за допомогою `Weak<Node>`). Коли ми виводимо

лічильники `leaf`, то бачимо, що сильних посилань 2, бо `branch` тепер зберігає клон `Rc<Node>` з `leaf` у `branch.children`, але все ще має 0 слабких посилань.

Коли внутрішня область видимості закінчується, `branch` виходить з видимості і лічильних сильних посилань `Rc<Node>` зменшується до 0, тож `Node` очищується. Лічильник слабких посилань 1 у `leaf.parent` не має стосунку до того, чи очиститься `Node`, тож ми більше не маємо витоків пам'яті!

Якщо ми спробуємо дістатися до батька змінної `leaf` після виходу з області видимості, ми знову отримаємо `None`. Наприкінці програми `Rc<Node>` у `leaf` має лічильник сильних посилань 1 і слабких 0, бо змінна `leaf` тепер знову є єдиним посиланням на `Rc<Node>`.

Вся логіка, яка керує лічильниками та очищенням значень, вбудована в `Rc<T>` і `Weak<T>` і їхні реалізації трейту `Drop`. Вказавши у визначенні `Node`, що стосунки дочірнього вузла до батьківського мають бути посиланням `Weak<T>`, ви змогли отримати взаємні посилання з батьківських вузлів до дочірніх і назад, не створивши циклу посилань і витoku пам'яті.

Підсумок

Цей розділ висвітлив, як використовувати розумні вказівники для отримання гарантій та недоліків, що відрізняються від тих, які Rust робить за замовчуванням для звичайних посилань. Тип `Box<T>` має відомий розмір і вказує на дані, розташовані в купі. Тип `Rc<T>` відстежує кількість посилань на дані в купі, так що ці дані можуть мати кілька власників. Тип `RefCell<T>` завдяки внутрішній мутабельності надає нам тип, який ми можемо використовувати, коли потребуємо незмінного типу, але має можливість змінювати внутрішнє значення цього типу; він також застосовує правила позичання під час виконання замість часу компіляції.

Також ми обговорили трейти `Deref` і `Drop`, які дозволяють застосувати функціональність розумних вказівників. Ми дослідили цикли посилань, які можуть викликати витoki пам'яті, і як запобігти їм за допомогою `Weak<T>`.

Якщо ця глава зацікавила вас і ви хочете реалізувати свої власні розумні вказівники, перегляньте "[The Rustonomicon](#)" для отримання додаткової корисної інформації.

Далі ми поговоримо про конкурентне виконання в Rust. Ви дізнаєтеся про ще кілька нових розумних вказівників.

Безстрашна Конкурентність

Ще одна головна мета Rust є безпечна та ефективна обробка конкурентності.

Конкурентне програмування, коли різні частини програми виконуються незалежно, та *паралельне програмування*, коли різні частини програми виконуються одночасно, стає все більш важливим, оскільки більше комп'ютерів використовують декілька процесорів. Історично, програмування в цих контекстах було дуже складне і схильне до помилок: Rust сподівається змінити це.

Спочатку, команда Rust думала що забезпечення безпеки пам'яті та запобігання проблем конкурентності були двома різними проблемами, які слід вирішувати різними методами. З часом, команда виявила що системи властності та типів є потужним набором інструментів що може допомогти з проблемами захисту пам'яті *і* конкурентності! Використовуючи перевірку власника та типів, багато помилок конкурентної призводять до помилок під час компіляції в Rust, а не під час виконання. Тому замість, того щоб витратити багато часу у спробах відтворити конкретні обставини, за якої відбуватиметься помилка конкурентності під час виконання, неправильний код відмовиться компілюватись та виведе помилку що пояснює проблему. Як результат, ви можете виправити ваш код під час роботи над ним, а не потенційно після того, як він був відправлений у виробництво. Ми назвемо цей аспект Rust *безстрашною конкурентністю*. Безстрашна конкурентність дозволяє вам писати код що вільний від складних для пошуку помилок, та легкий для рефакторингу без запровадження нових помилок.

Примітка: Заради спрощення, ми будемо більшість проблем називати як *конкурентність*, а не більш точним визначенням *конкурентність та/або паралелізм*. Якби ця книга була про конкурентність та/або паралелізм, ми були б більш конкретними. Для цього розділу, будь ласка, коли ми використовуємо *конкурентність* в себе в голові замінійте на *конкурентність та/або паралелізм*.

Багато мов дуже догматичні в вирішенні проблеми обробки конкурентності. Наприклад, Erlang має дуже елегантний функціонал для конкурентної передачі повідомлень, але також лише незрозумілі методи спільного використання стану між потоками. Підтримка лише частини можливих вирішень є розумною стратегією для мов високого рівня, тому що мови високого рівня обіцяють переваги від передачі частини контролю задля отримання абстракцій. Проте, мови низького рівня очікувано надають вирішення кращі за швидкодією в будь-яких випадках, але мають менше абстракцій від заліза. Тому, Rust пропонує різноманітні інструменти для моделювання проблем в будь-який спосіб що відповідає вашій ситуації та вимогам.

В даному розділі ми розглянемо наступні теми:

- Як створити потоки для запуску кілька частин коду одночасно
- Конкурентна *передача повідомлень*, де канали відсилають повідомлення між потоками
- Конкурентний *спільний стан*, де декілька потоків мають доступ до одної частини даних
- `Sync` та `Send` трейти, які поширюють гарантії паралельності Rust на типи, визначені користувачем, а також типи, надані стандартною бібліотекою

Використання Потоків для Одночасного Виконання Коду

В більшості сучасних операційних систем, код виконуємої програми запускається в середині *процесу*, а операційна система керує багатьма процесами одночасно. Всередині програми ви можете мати незалежні частини, які виконуються одночасно. Функціонал, який виконує такі незалежні частини називають *потоками* або *тредами*. Наприклад, вебсервер може використовувати декілька потоків і таким чином мати змогу обробляти більше одного запиту одночасно.

Розділення обчислень між декількома потоками для одночасного виконання кількох завдань може покращити швидкість виконання програми, але також підвищує складність програми. Оскільки потоки можуть виконуватись одночасно, немає жодної гарантії стосовно порядку виконання частин коду в різних потоках. Це може призвести до наступних проблем:

- стан гонитви, за якого потоки отримують доступ до даних або ресурсів в довільному порядку
- дедлоки, коли два потоки чекають один одного, перешкоджаючи продовженню виконання обох потоків
- помилки, що виникають виключно за певних обставин і які важко відворити та надійно виправити

Rust намагається пом'якшити негативні наслідки використання потоків, але програмування в багатопоточному контексті все ще вимагає ретельного обдумування та потребує структуру коду, відмінну від програм, що виконуються в одному потоці.

Мови програмування імплементують потоки декількома різними способами, а багато операційних систем надають API, який мова може використовувати для створення нових потоків. Стандартна бібліотека Rust використовує модель імплементації потоку *1:1*, за якої програма використовує один потік операційної системи на один потік, що використовує мова. Існують крейти, котрі імплементують інші моделі потоків, що йдуть на інші компроміси порівняно з моделлю 1:1.

Створення Нового Потoku зі `spawn`

Для того, щоб створити новий потік, ми викликаємо функцію `thread::spawn` і передаємо їй замикання (ми вже говорили про них в Розділі 13), що містить в собі код, який ми

хочемо запустити всередині нового потоку. Приклад у Лістингу 16-1 виводить на екран деякий текст з основного потоку, а також інший текст з нового потоку:

Файл: src/main.rs

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

fn main() {
    thread::spawn(|| {
        for i in 1..10 {
            println!("hi number {} from the spawned thread!", i);
            thread::sleep(Duration::from_millis(1));
        }
    });

    for i in 1..5 {
        println!("hi number {} from the main thread!", i);
        thread::sleep(Duration::from_millis(1));
    }
}
```

Блок коду 16-1: Створення нового потоку для виводу на екран деяку стрічку, поки основний потік виводить іншу стрічку

Зверніть увагу, що коли основний потік Rust програми завершується, всі створені потоки припиняють існувати, незалежно від того завершили вони своє виконання чи ні. Вивід цієї програми може дещо відрізнятись від запуску до запуску, але виглядатиме приблизно так:

```
hi number 1 from the main thread!
hi number 1 from the spawned thread!
hi number 2 from the main thread!
hi number 2 from the spawned thread!
hi number 3 from the spawned thread!
hi number 3 from the spawned thread!
hi number 4 from the main thread!
hi number 4 from the spawned thread!
hi number 5 from the spawned thread!
```

Виклики `thread::sleep` змушують потік припинити виконання на короткий період часу, дозволяючи виконувати інший потік. Ймовірно, потоки будуть виконуватись по черзі, проте це не гарантовано, а залежить від того, як ваша операційна система планує виконання потоків. Цього разу, основний потік вивів на екран стрічку першим, незважаючи на те, що print statement в новому потоці з'являється у коді раніше. Незважаючи на те, що ми запрограмували новостворений потік виводити екран стрічки

доти, поки `i` не дорівнюватиме 9, він вивів лише 5 стрічок до завершення основного потоку.

Якщо ви запускаєте цей код і бачите лише вивід з основного потоку або ж не бачите жодного оверлепу, спробуйте збільшити діапазон чисел, щоб створити для операційної системи більше умов для переключання між потоками.

Очікування Завершення Виконання Всіх Потоків з Використанням Дескрипторів `join`

Код в Блоці коду 16-1 не лише передчасно зупиняє створений потік в більшості випадків через завершення основного потоку, але оскільки гарантії щодо порядку виконання потоків відсутні, ми не можемо гарантувати, що створені потоки взагалі будуть виконуватись!

Ми можемо вирішити проблему, коли створений потік не виконується або ж передчасно завершує виконання, зберігаючи значення, що повертає `thread::spawn` в змінну. `thread::spawn` повертає значення, що має тип `JoinHandle`. `JoinHandle` - це owned value (значення, яким володіють), котре при виклику на ньому методу `join`, чекатиме завершення виконання потоку. Блок коду 16-2 демонструє як використовувати `JoinHandle` потоку, котрий ми створили в блоці коду 16-1, а також викликати `join` щоб пересвідчитись, що новостворений потік закінчує виконання раніше, ніж `main`:

Файл: `src/main.rs`

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

fn main() {
    let handle = thread::spawn(|| {
        for i in 1..10 {
            println!("hi number {} from the spawned thread!", i);
            thread::sleep(Duration::from_millis(1));
        }
    });

    for i in 1..5 {
        println!("hi number {} from the main thread!", i);
        thread::sleep(Duration::from_millis(1));
    }

    handle.join().unwrap();
}
```

Блок коду 16-2: Збереження `JoinHandle` з `thread::spawn` щоб гарантувати виконання потоку до завершення

Виклик `join` на обробнику (хендлері) блокує потік, що наразі виконується аж до моменту, коли потік, представлений обробником, не завершиться. *Блокування* потоку означає, що такий потік не може виконуватися або ж завершитись. Оскільки ми помістили виклик `join` після циклу `for` в основному потоці, виконання Блоку коду 16-2 має вивести на екран щось схоже на наступне:

```
hi number 1 from the main thread!
hi number 2 from the main thread!
hi number 1 from the spawned thread!
hi number 3 from the main thread!
hi number 2 from the spawned thread!
hi number 4 from the main thread!
hi number 3 from the spawned thread!
hi number 4 from the spawned thread!
hi number 5 from the spawned thread!
hi number 6 from the spawned thread!
hi number 7 from the spawned thread!
hi number 8 from the spawned thread!
hi number 9 from the spawned thread!
```

Два потоки продовжують виконуватися по черзі, але основний потік чекає через виклик `handle.join()` і не завершується, доки не завершиться створений потік.

Однак давайте поглянемо, що трапиться, якщо замість цього розмістити `handle.join()` перед циклом `for` всередині `main`, як тут:

Файл: `src/main.rs`

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

fn main() {
    let handle = thread::spawn(|| {
        for i in 1..10 {
            println!("hi number {} from the spawned thread!", i);
            thread::sleep(Duration::from_millis(1));
        }
    });

    handle.join().unwrap();

    for i in 1..5 {
        println!("hi number {} from the main thread!", i);
        thread::sleep(Duration::from_millis(1));
    }
}
```

Основний потік чекатиме на завершення виконання створеного треду і лише після цього виконуватиме цикл `for`, тому вивід більше не буде чергуватись, як показано тут:

```
hi number 1 from the spawned thread!  
hi number 2 from the spawned thread!  
hi number 3 from the spawned thread!  
hi number 4 from the spawned thread!  
hi number 5 from the spawned thread!  
hi number 6 from the spawned thread!  
hi number 7 from the spawned thread!  
hi number 8 from the spawned thread!  
hi number 9 from the spawned thread!  
hi number 1 from the main thread!  
hi number 2 from the main thread!  
hi number 3 from the main thread!  
hi number 4 from the main thread!
```

Дрібні деталі, такі як-от місце виклику `join`, можуть впливати на те чи будуть ваші потоки виконуватись одночасно.

Використання `move` Замикань із Потоками

Ми будемо часто використовувати ключове слово `move` разом з замиканнями, переданими всередину `thread::spawn`, оскільки замикання тоді почне володіти значеннями, які вона використовує з оточуючого контексту (середовища), таким чином передаючи володіння значеннями з одного потоку в інший. У підрозділі "[Захоплення посилань або переміщення володіння](#)" Розділу 13, ми розглядали

`move` в контексті замикань. Зараз же, ми сконцентруємось більше на взаємодії між `move` та `thread::spawn`.

Зверніть увагу, в Блоці коду 16-1 замикання, яке ми передаємо в `thread::spawn` не приймає жодних аргументів: ми не використовуємо жодних даних з основного потоку в коді створеного потоку. Для того, щоб використовувати дані з основного потоку в створеному потоці, замикання створеного потоку має захопити (capture) потрібні значення. Блок коду 16-3 показує спробу створити вектор в основному потоці, а потім використати його в створеному. Однак, це не запрацює одразу, як ви зможете незабаром пересвідчитись.

Файл: `src/main.rs`

```
use std::thread;

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

    let handle = thread::spawn(|| {
        println!("Here's a vector: {:?}", v);
    });

    handle.join().unwrap();
}
```

Блок коду 16-3: Спроба використати вектор, створений основним потоком, в іншому потоці

Замикання використовує `v`, отже, воно захопить `v` і зробить його частиною контексту замикання. Оскільки `thread::spawn` виконує замикання в новому потоці, ми повинні мати доступ до `v` всередині нового потоку. Однак, коли ми скомпілюємо цей приклад, ми отримаємо наступну помилку:

```
$ cargo run
Compiling threads v0.1.0 (file:///projects/threads)
error[E0373]: closure may outlive the current function, but it borrows `v`, which
is owned by the current function
--> src/main.rs:6:32
6 |     let handle = thread::spawn(|| {
  |                               ^^ may outlive borrowed value `v`
7 |         println!("Here's a vector: {:?}", v);
  |                                     - `v` is borrowed here
  |
note: function requires argument type to outlive `'static`
--> src/main.rs:6:18
6 |     let handle = thread::spawn(|| {
  |         ^
7 |         println!("Here's a vector: {:?}", v);
8 |     });
  |     ^
help: to force the closure to take ownership of `v` (and any other referenced
variables), use the `move` keyword
6 |     let handle = thread::spawn(move || {
  |                               +++++

For more information about this error, try `rustc --explain E0373`.
error: could not compile `threads` due to previous error
```

Rust здогадується як захопити `v`, і саме тому `println!` потребує тільки посилання на `v`, а замикання намагається запозичити `v`. Однак є проблема: Rust не може здогадатись як

довго потік буде виконуватись, отже, Rust не знає чи буде посилання на `v` завжди валідним (valid).

Блок коду 16-4 показує випадок, який, швидше за все, матиме невалідне посилання на `v`:

Файл: src/main.rs

```
use std::thread;

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

    let handle = thread::spawn(|| {
        println!("Here's a vector: {:?}", v);
    });

    drop(v); // oh no!

    handle.join().unwrap();
}
```

Блок коду 16-4: Потік з замиканням, що намагається захопити (capture) посилання на `v` з основного потоку, який видаляє `v`

Якщо Rust дозволить нам виконати цей код, існує ймовірність, що створений потік буде негайно переведений в бекграунд (background) і не буде виконуватись взагалі. Створений потік має посилання на `v` всередині себе, але основний потік негайно викидає (drops) `v`, використовуючи функцію `drop`, котру ми розглядали у Розділі 15. Потім, коли створений потік починає виконуватись, `v` більше невалідний, тому посилання на нього також невалідне. О ні!

Щоб виправити помилку компілятора в блоці коду 16-3, ми можемо скористатись порадою, яку надає повідомлення про помилку:

```
help: to force the closure to take ownership of `v` (and any other referenced
variables), use the `move` keyword
6 |     let handle = thread::spawn(move || {
  |                               +++++
```

Додаючи ключове слово `move` перед замиканням, ми змушуємо замикання взяти володіння над значеннями, котрі воно використовує, не дозволяючи Rust робити припущення стосовно позичання (borrowing) значень. Модифікація до Блоку коду 16-3, показана в Блоці коду 16-5 скомпільюється і виконуватиметься так, як ми задумали:

Файл: src/main.rs


```
use std::thread;

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

    let handle = thread::spawn(move || {
        println!("Here's a vector: {:?}", v);
    });

    handle.join().unwrap();
}
```

Блок коду 16-5: Використання ключового слова `move`, щоб примусити замикання взяти володіння над значеннями, які воно використовує

В нас може виникнути спокуса спробувати той самий підхід, щоб виправити код у Блоці коду 16-4, де основний потік викликав `drop`, використовуючи замикання з `move`. Однак, це не спрацює, оскільки те, що Блок коду 16-4 намагається зробити, заборонено з іншої причини. Якщо ми додамо `move` до замикання, ми перенемістили `v` в контекст замикання, тому ми більше не можемо викликати `drop` на ньому в основному потоці. Замість цього ми отримаємо помилку компіляції:

```
$ cargo run
Compiling threads v0.1.0 (file:///projects/threads)
error[E0382]: use of moved value: `v`
--> src/main.rs:10:10
4 |     let v = vec![1, 2, 3];
  |     - move occurs because `v` has type `Vec<i32>`, which does not
  |     implement the `Copy` trait
5 |
6 |     let handle = thread::spawn(move || {
  |                                ----- value moved into closure here
7 |         println!("Here's a vector: {:?}", v);
  |                                         - variable moved due to use in
  |                                         closure
...
10 |     drop(v); // oh no!
   |           ^ value used here after move

For more information about this error, try `rustc --explain E0382`.
error: could not compile `threads` due to previous error
```

Правила володіння Rust знову нас врятували! Ми отримали помилку в Блоці коду 16-3, оскільки Rust був консервативним і позичив лише `v` для потоку, а отже, основний потік міг теоретично зробити посилання, що використовувалось у створеному потоці, невалідним. Сказавши Rust передати володіння `v` створеному потоку, ми гарантуємо Rust, що основний потік більше не використовуватиме `v`. Якщо ми змінимо Блок коду 16-

4 таким же чином, то ми порушимо правила володіння, коли намагатимемось використати `v` в основному потоці. Ключове слово `move` бере гору над правилами володіння, що Rust використовує за замовчуванням; таким чином не ми не порушуємо правила володіння.

Маючи базове розуміння потоків і API потоків (прикладний програмний інтерфейс потоків), давайте поглянемо що ми можемо *робити* з потоками.

Застосування Обміну Повідомлень для Передавання Даних між Потоками

Одним із набираючих популярність підходів для забезпечення безпечної конкурентності є *обмін повідомленнями*, коли потоки або ж актори комунікують надсилаючи один одному повідомлення, що містять дані. Ось основна ідея, виражена в слогані з [документації мови програмування Go](https://go.dev/doc/effective_go#concurrency): "Не комунікуйте за допомогою спільної пам'яті; замість цього, діліться пам'яттю комунікуючи."

Для досягнення конкурентності за допомогою обміну повідомленнями, стандартна бібліотека Rust надає імплементацію *каналів*. Канал - це загальна концепція програмування, основна ідея якої полягає в тому, що дані надсилаються з одного потоку в інший.

Ви можете уявити канал в програмуванні схожим на напрямлений канал води, такий як струмок чи річка. Якщо ви помістите щось на кшталт гумової качечки в річку, вона попливе вниз за течією аж до кінця водного шляху.

Канал має дві половини: передавач (transmitter) і отримувач (receiver). Передавач - це місце, де ви пускаєте за течією гумових качечок, а отримувач - це місце куди гумова качечка потрапляє в кінці течії. Одна частина вашого коду викликає методи передавача з даними, які ви хочете відправити, а інша частина перевіряє отримувач на наявність отриманих повідомлень. Канал вважається *закритим* якщо передавач або ж отримувач були видалені.

Надалі, ми попрацюємо над програмою, яка має один потік для генерації значень і надсилання їх по каналу, а інший потік отримуватиме значення і виводитиме їх на екран. Ми відправлятимемо прості значення між потоками використовуючи канали для ілюстрації даного функціоналу. Як тільки ви познайомитесь з даним підходом, ви зможете використовувати канали для будь-яких потоків, яким потрібно комунікувати між собою, таким як чат-системи або ж системи, де багато потоків виконують частини обчислень і надсилають результати в один потік, котрий агрегує ці результати.

Спочатку, в Блоці коду 16-6, ми створимо канал, але не будемо нічого з ним робити. Зверніть увагу, що даний приклад поки що не скомпільюється, оскільки Rust не може визначити які типи значень ми хочемо надсилати через канал.

Файл: `src/main.rs`

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

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

Блок коду 16-6: Створення каналу і присвоєння двох його половин в `tx` і `rx`

Ми створюємо новий канал за допомогою функції `mpsc::channel`, `mpsc` означає *multiple producer, single consumer* (декілька виробників, один споживач). Словом, спосіб, в який стандартна бібліотека Rust імплементує канали означає, що канал може мати декілька *відправляючих* кінців, які створюють значення, але лише один *споживаючий* кінець, який споживає значення. Уявіть декілька струмків, що зливаються в одну велику річку: все що буде відправлено за течією будь-якого з струмків в кінці-кінців потрапить в річку. Наразі ми почнемо з одного виробника, але ми додамо ще декілька коли змусимо цей приклад працювати.

Функція `mpsc::channel` повертає кортеж, першим елементом якого є відправляючий кінець - передавач, а другим елементом є отримуючий кінець - отримувач. Абревіатури `tx` і `rx` традиційно використовуються в багатьох сферах для позначення *передавача* (*transmitter*) та *отримувача* (*receiver*) відповідно, тому ми називаємо наші змінні таким чином, щоб позначити відповідні кінці каналу. Ми використовуємо інструкцію `let` з шаблоном, що деструктуризує кортежі; ми обговоримо використання шаблонів в інструкціях `let` та деструктуризацію в Розділі 18. Наразі ж знайте, що використання інструкції `let` в такий спосіб є зручним підходом для витягування (extract) частин кортежу, який повертає після виконання `mpsc::channel`.

Давайте перемістимо передавач в створений потік і попросимо його надіслати одну стрічку, щоб даний потік комунікував з основним потоком, як показано в Блоці коду 16-7. Це як помістити гумову качечку в річку вище за течією або ж надіслати чат-повідомлення з одного потоку в інший.

Файл: src/main.rs

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

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

    thread::spawn(move || {
        let val = String::from("hi");
        tx.send(val).unwrap();
    });
}
```

Блок коду 16-7: Переміщення `tx` в створений потік і відправка "hi"

Знову ж таки, ми використовуємо `thread::spawn` щоб створити новий потік і потім використовуємо `move` щоб помістити `tx` всередину замикання, адже таким чином потік володітиме `tx`. Створений потік має володіти передавачем, щоб мати можливість надсилати повідомлення по каналу. Передавач має метод `send`, котрий приймає значення, яке ми хочемо надіслати. Метод `send` повертає `Result<T, E>`, отже, якщо отримувач був видалений й немає куди надіслати значення, операція поверне помилку. В даному прикладі, ми викликаємо `unwrap`, щоб наш код запанікував у випадку помилки. Однак в справжньому додатку ми б обробили помилки належним чином: поверніться до Розділу 9 щоб переглянути стратегії належної обробки помилок.

В Блоці коду 16-8 ми в основному потоці отримаємо/дістанемо значення з отримувача. Це як дістати гумову качечку з води в кінці річки або ж отримати повідомлення в чаті.

Файл: src/main.rs

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

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

    thread::spawn(move || {
        let val = String::from("hi");
        tx.send(val).unwrap();
    });

    let received = rx.recv().unwrap();
    println!("Got: {}", received);
}
```

Блок коду 16-8: Отримання значення "hi" в основному потоці та вивід його на екран

Отримувач має два корисні методи: `recv` та `try_recv`. Ми використовуємо `recv`, скорочено від *receive*, який заблокує виконання основного потоку і чекатиме доки значення буде надіслане в канал. Як тільки значення буде надіслане, `recv` поверне його, обернувши в `Result<T, E>`. Коли передавач закриється, `recv` поверне помилку, яка сигналізує про те, що значення більше не надходитимуть.

Метод `try_recv` не блокує основний потік, а натомість одразу повертає `Result<T, E>`: значення `Ok`, котре містить повідомлення, якщо воно доступне, і `Err` якщо цього разу немає жодних повідомлень. Використання `try_recv` корисне якщо потік має виконувати іншу роботу, очікуючи на повідомлення: ми можемо написати цикл, котрий періодично викликає `try_recv`, обробляє повідомлення, якщо воно доступне, а в іншому випадку деякий час виконує іншу роботу аж до наступної перевірки.

Ми використали `recv` в цьому прикладі для простоти; ми не маємо жодної іншої роботи для основного потоку, окрім очікування повідомлень, тому блокування основного потоку є доречним/виправданим.

Коли ми запустимо код з Блоку коду 16-8, ми побачимо, що значення виводиться з основного потоку:

```
Got: hi
```

Ідеально!

Канали та Передача Володіння

Правила володіння відіграють важливу роль в обміні повідомленнями, оскільки вони допомагають вам писати безпечний конкурентний код. Запобігання помилкам в конкурентних програмах - це перевага, яку надає мислення в термінах володіння в ваших Rust програмах. Давайте проведемо експеримент для демонстрації того як канали та володіння працюють разом для запобігання проблемам: ми спробуємо використати значення `val` в створеному потоці вже *після* того, як ми надіслали його далі по каналу. Спробуйте скомпілювати код з Блоку коду 16-9 щоб побачити чому він не пропускається компілятором:

Файл: src/main.rs

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

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

    thread::spawn(move || {
        let val = String::from("hi");
        tx.send(val).unwrap();
        println!("val is {}", val);
    });

    let received = rx.recv().unwrap();
    println!("Got: {}", received);
}
```

Блок коду 16-9: Спроба використати `val` після того, як воно було надіслане в канал

Тут ми намагаємось вивести `val` на екран вже після того як ми надіслали його по каналу за допомогою `tx.send`. Дозволяти таке було б поганою ідеєю: як тільки значення буде надіслане в інший потік, такий потік може модифікувати або ж навіть видалити значення,

перш ніж ми спробуємо використати його знову. Потенційно, зміни в іншому потоці можуть привести до помилок або ж неочікуваних результатів через суперечливі (inconsistent) або ж неіснуючі дані. Однак, Rust видасть помилку якщо ми спробуємо скомпілювати код з Блоку коду 16-9:

```
$ cargo run
  Compiling message-passing v0.1.0 (file:///projects/message-passing)
error[E0382]: borrow of moved value: `val`
  --> src/main.rs:10:31
   |
8  |         let val = String::from("hi");
   |         --- move occurs because `val` has type `String`, which does not
   |         implement the `Copy` trait
9  |         tx.send(val).unwrap();
   |         --- value moved here
10 |         println!("val is {}", val);
   |                                ^^^ value borrowed here after move

= note: this error originates in the macro `$crate::format_args_nl` which comes
from the expansion of the macro `println` (in Nightly builds, run with -Z macro-
backtrace for more info)

For more information about this error, try `rustc --explain E0382`.
error: could not compile `message-passing` due to previous error
```

Наша помилка в роботі з конкурентністю спричинила помилку компіляції. Функція `send` бере володіння над своїм параметром, а коли значення переміщується (moved), отримувач бере над ним володіння. Це не дає нам випадково повторно використати значення після того як ми його надіслали; правила володіння перевіряють чи все гаразд.

Відправлення Декількох Значень та Спостереження за Очікуванням Отримувача

Код в Блоці коду 16-8 скомпілювався і виконався, але він не продемонстрував, що два окремі потоки спілкуються між собою через канал. В Блоці коду 16-10 ми зробили деякі зміни, які підтверджують, що код в Блоці коду 16-8 виконується конкурентно: створений потік тепер відсилатиме декілька повідомлень і робитиме секундну паузу між кожним повідомленням.

Файл: src/main.rs

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

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

    thread::spawn(move || {
        let vals = vec![
            String::from("hi"),
            String::from("from"),
            String::from("the"),
            String::from("thread"),
        ];

        for val in vals {
            tx.send(val).unwrap();
            thread::sleep(Duration::from_secs(1));
        }
    });

    for received in rx {
        println!("Got: {}", received);
    }
}
```

Блок коду 16-10: Відправка декількох повідомлень із паузою між відправками

Цього разу створений потік має вектор стрічок, які ми хочемо надіслати в основний потік. Ми ітеруємось по ним, надсилаючи кожен рядок окремо, і робимо паузу між кожною відправкою, викликаючи функцію `thread::sleep` із значенням `Duration` в 1 секунду.

В основному потоці, ми більше не викликаємо функцію `recv` явно: замість цього ми розглядаємо `rx` як ітератор. Отримуючи значення, ми виводимо його на екран. Якщо канал закриється, ітерування припиниться.

Під час виконання коду із Блоку коду 16-10, ви маєте побачити наступний вивід із 1-секундною паузою між кожним рядком:

```
Got: hi
Got: from
Got: the
Got: thread
```

Оскільки ми не маємо жодного коду, що призупиняє або ж відкладає виконання циклу `for` в основному потоці, ми можемо сказати, що основний потік очікує отримання значень із створеного потоку.

Створення Декількох Виробників із Клонуванням Передавача

Раніше ми вже згадували, що `mpsc` - це аббревіатура для *multiple producer, single consumer* (кілька виробників, один споживач). Даваймо використаємо `mpsc` і розширимо код в Блоці коду 16-10 щоб створити кілька потоків, котрі надсилають дані одному й тому ж отримувачу. Ми можемо зробити це клонувавши передавач, як показано в Блоці коду 16-11:

Файл: `src/main.rs`

```
// --snip--

let (tx, rx) = mpsc::channel();

let tx1 = tx.clone();
thread::spawn(move || {
    let vals = vec![
        String::from("hi"),
        String::from("from"),
        String::from("the"),
        String::from("thread"),
    ];

    for val in vals {
        tx1.send(val).unwrap();
        thread::sleep(Duration::from_secs(1));
    }
});

thread::spawn(move || {
    let vals = vec![
        String::from("more"),
        String::from("messages"),
        String::from("for"),
        String::from("you"),
    ];

    for val in vals {
        tx.send(val).unwrap();
        thread::sleep(Duration::from_secs(1));
    }
});

for received in rx {
    println!("Got: {}", received);
}

// --snip--
```

Блок коду 16-11: Відправка декількох повідомлень з кількох виробників (producers)

Цього разу, перед тим як ми створимо перший потік, ми викликаємо `clone` на передавачі. Це дасть нам новий передавач, який ми зможемо потім передати в створений потік. Ми передаємо оригінальний передавач в другий створений потік. Це дає нам два потоки, кожен з яких надсилає різні повідомленнями одному отримувачу.

Коли ви виконаєте код, ваш вивід має виглядати приблизно так:

```
Got: hi  
Got: more  
Got: from  
Got: messages  
Got: for  
Got: the  
Got: thread  
Got: you
```

Ви можете бачити значення в іншому порядку, залежно від вашої системи. Саме це робить конкурентність цікавою, але й складною одночасно. Якщо ви поекспериментуєте з `thread::sleep`, підставляючи різні значення в різні потоки, кожен запуск буде ще більш недетермінованим і щоразу створюватиме різний вивід.

Тепер, коли ми поглянули на те, як працюють канали, давайте розглянемо інший метод конкурентності.

Конкурентність зі Спільним Станом

Обмін повідомленнями - чудовий, але не єдиний спосіб роботи з конкурентністю. Іншим способом може бути доступ декількох потоків до спільних даних. Розглянемо наступну частину слогану з документації мови програмування Go ще раз: "не комунікуйте за допомогою спільної пам'яті."

Як би виглядала комунікація за допомогою спільної пам'яті? Окрім того, чому ентузіасти обміну повідомленнями застерігають від використання спільної пам'яті?

У певному сенсі, канали в будь-якій мові програмування схожі на одноособове володіння, тому що як тільки ви передали значення по каналу, ви не повинні більше використовувати таке значення. Конкурентність із спільною пам'яттю нагадує множинне володіння: декілька потоків одночасно мають доступ до однієї і тієї ж області пам'яті. Як ви могли бачити в Розділі 15, де розумні вказівники робили множинне володіння можливим, таке володіння може додати програмі складності, оскільки потрібно управляти різними власниками (owners). Система типів Rust та правила володіння дуже допомагають здійснювати таке управління коректно. Наприклад, давайте розглянемо м'ютекси, один з найпоширеніших примітивів конкурентності для роботи із спільною пам'яттю.

Використання М'ютексів для Отримування Доступу до Даних Лише з Одного Потoku в Один Момент Часу

Mutex (м'ютекс) - це аббревіатура для *mutual exclusion* (взаємне виключення), оскільки м'ютекс дозволяє лише одному потоку отримувати доступ до даних в будь-який момент часу. Для того, щоб отримати доступ до даних у м'ютексі, потік має спочатку повідомити, що він бажає отримати доступ, запросивши отримати *блокування* (*lock*) м'ютексу. Блокування - це структура даних, що є частиною м'ютексу і відстежує хто саме має ексклюзивний доступ до даних. Саме тому, м'ютекс описують як *захист* даних, які він в собі зберігає, за допомогою системи блокування.

М'ютекси мають репутацію складного в використанні механізму, оскільки ви маєте пам'ятати два правила:

- Ви повинні спробувати отримати блокування перед використанням даних.
- Коли ви закінчите працювати з даними, що захищає м'ютекс, ви маєте розблокувати дані, щоб інші потоки могли отримати блокування.

Метафорою для м'ютексу можна вважати панельну дискусію на конференції лише з одним мікрофоном. Перед тим як інший учасник дискусії зможе говорити, він повинен попросити або показати, що він хоче скористатись мікрофоном. Коли він отримає мікрофон, він може говорити стільки, скільки вважає за потрібне, а потім передати мікрофон наступному учаснику дискусії, який просить слово. Якщо учасник дискусії забуває передати мікрофон після того, як він закінчив, то ніхто інший не матиме змоги говорити. Якщо управління спільним мікрофоном піде неправильно, то панельна дискусія не працюватиме так, як заплановано!

Правильне управління м'ютексами може бути неймовірно складним, ось чому так багато людей з ентузіазмом ставиться до каналів. Однак, завдяки системі типів Rust та правилам володіння, ви не можете помилитись при блокуванні та розблокуванні.

API `Mutex<T>`

Щоб продемонструвати як використовувати м'ютекс, давайте почнемо з використання м'ютексу в однопоточному контексті, як показано в Блоці коду 16-12:

Файл: `src/main.rs`

```
use std::sync::Mutex;

fn main() {
    let m = Mutex::new(5);

    {
        let mut num = m.lock().unwrap();
        *num = 6;
    }

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

Блок коду 16-12: Експерименти з API `Mutex<T>` в однопоточному контексті для простоти

Як і з багатьма типами, ми створюємо `Mutex<T>`, використовуючи функцію `new`. Для доступу до даних всередині м'ютекса, ми використовуємо метод `lock` для отримання блокування. Цей виклик заблокує поточний потік, щоб він не міг виконувати жодну роботу до моменту поки не настане наша черга отримувати блокування.

Виклик `lock` завершиться неуспішно, якщо інший потік, котрий тримав блок, запанікував (panicked). В такому випадку, ніхто ніколи не зможе отримати блок, тому ми вирішили використати `unwrap` і змусити потік запанікувати, якщо ми опинимось в такій ситуації.

Після того, як ми отримали блокування, ми можемо розглядати повернуте значення, яке в даному випадку називається `num`, як мутабельне посилання на дані всередині. Система типів гарантує, що ми отримуємо блокування перед тим як використати значення в `m`. Тип `m` - `Mutex<i32>`, а не `i32`, тому ми зобов'язані викликати `lock` щоб мати змогу використовувати значення `i32`. Ми не можемо забути про це; інакше система типів не дозволить нам отримати доступ до внутрішнього `i32`.

Як ви могли запідозрити, `Mutex<T>` є розумним вказівником. Точніше, виклик `lock` повертає розумний покажчик, котрий називається `MutexGuard`, загорнутий в `LockResult`, який ми обробили за допомогою виклика `unwrap`. `MutexGuard` - це розумний вказівник, що реалізує `Deref`, щоб вказувати на внутрішні дані; розумний вказівник такж має реалізацію `Drop`, котра вивільняє блок автоматично, коли `MutexGuard` виходить за межі області видимості, що відбувається в кінці внутрішньої області видимості. Як наслідок, ми не ризикуємо забути розблокувати блок і заблокувати використання м'ютексу іншими потоками, оскільки розблокування блоку відбувається автоматично.

Після видалення блоку, ми можемо вивести на екран значення м'ютексу і побачити, що ми змогли змінити внутрішнє `i32` на 6.

Спільне Використання `Mutex<T>` Декількома Потоками

Тепер давайте спробуємо, використати значення з декількох різних потоків за допомогою `Mutex<T>`. Ми запустимо 10 потоків і кожен з них буде збільшувати значення лічильника на 1, таким чином лічильник змінюватиме значення від 0 до 10. Наступний приклад в Блоці коду 16-3 містить помилку компіляції і ми використаємо цю помилку щоб дізнатися більше про використання `Mutex<T>` і як Rust допомагає нам правильно його використовувати.

Файл: `src/main.rs`

```

use std::sync::Mutex;
use std::thread;

fn main() {
    let counter = Mutex::new(0);
    let mut handles = vec![];

    for _ in 0..10 {
        let handle = thread::spawn(move || {
            let mut num = counter.lock().unwrap();

            *num += 1;
        });
        handles.push(handle);
    }

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

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

```

Блок коду 16-13: Десять потоків по черзі інкрементують лічильник, захищений за допомогою `Mutex<T>`

Ми створюємо змінну `counter`, що містить `i32` всередині `Mutex<T>`, так само як ми зробили в Блоці коду 16-12. Далі, ми створюємо 10 потоків, що ітеруються по діапазону (range) чисел. Ми використовуємо `thread::spawn` і передаємо кожному потоку одне й те саме замикання, котре переміщує лічильник всередину потоку, отримує блокування `Mutex<T>`, викликаючи метод `lock`, а потім додає 1 до значення всередині м'ютексу. Коли потік завершує виконання замикання, `num` виходить з області видимості, звільняє блок (lock), щоб інший потік міг його отримати.

В основному потоці, ми збираємо (collect) всі обробники (join handles). Після цього, так само як і в Блоці коду 16-2, ми викликаємо `join` на кожному обробнику, щоб впевнитись, що всі потоки завершуються. В цей момент основний потік отримає блокування і виведе на екран результат виконання цієї програми.

Ми натякнули, що цей приклад не скомпілюється. А тепер давайте дізнаємось чому!

```

$ cargo run
  Compiling shared-state v0.1.0 (file:///projects/shared-state)
error[E0382]: use of moved value: `counter`
  --> src/main.rs:9:36
   |
5  |         let counter = Mutex::new(0);
   |         ----- move occurs because `counter` has type `Mutex<i32>`, which
does not implement the `Copy` trait
...
9  |         let handle = thread::spawn(move || {
   |                                     ^^^^^^^ value moved into closure here, in
previous iteration of loop
10 |             let mut num = counter.lock().unwrap();
   |             ----- use occurs due to use in closure

For more information about this error, try `rustc --explain E0382`.
error: could not compile `shared-state` due to previous error

```

У повідомленні про помилку вказано, що значення `counter` вже було переміщено в попередній ітерації циклу. Rust говорить нам, що ми не можемо перемістити володіння блокуванням `counter` в декілька потоків. виправимо помилку компіляції за допомогою множинного володіння, про яке ми говорили в Розділі 15.

Множинне Володіння з Декількома Потоками

В Розділі 15, ми надали значення декільком власникам, використовуючи розумний вказівник `Rc<T>` щоб створити значення з підрахунком посилань. Зробімо тут те саме і подивимось, що станеться. Ми загорнемо `Mutex<T>` в `Rc<T>` в Блоці коду 16-14 і скопіюємо `Rc<T>` перед переміщенням володіння всередину потоку.

Файл: src/main.rs

```
use std::rc::Rc;
use std::sync::Mutex;
use std::thread;

fn main() {
    let counter = Rc::new(Mutex::new(0));
    let mut handles = vec![];

    for _ in 0..10 {
        let counter = Rc::clone(&counter);
        let handle = thread::spawn(move || {
            let mut num = counter.lock().unwrap();

            *num += 1;
        });
        handles.push(handle);
    }

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

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

Блок коду 16-14: Спроба використати `Rc<T>` щоб дозволити потокам володіти `Mutex<T>`

Компілюємо знов і отримуємо... інші помилки! Компілятор нас багато чому вчить.


```

$ cargo run
  Compiling shared-state v0.1.0 (file:///projects/shared-state)
error[E0277]: `Rc<Mutex<i32>>` cannot be sent between threads safely
  --> src/main.rs:11:36
   |
11 |         let handle = thread::spawn(move || {
   |                                   ^^^^^
   |                                   |
   |                                   ----- within this `[closure@src/main.rs:11:36:
11:43]`,
   |                                   |
   |                                   required by a bound introduced by this call
12 |         let mut num = counter.lock().unwrap();
13 |
14 |         *num += 1;
15 |     });
   |     ^ `Rc<Mutex<i32>>` cannot be sent between threads safely
   = help: within `[closure@src/main.rs:11:36: 11:43]`, the trait `Send` is not
implemented for `Rc<Mutex<i32>>`
note: required because it's used within this closure
  --> src/main.rs:11:36
   |
11 |         let handle = thread::spawn(move || {
   |                                   ^^^^^^^^^
note: required by a bound in `spawn`

For more information about this error, try `rustc --explain E0277`.
error: could not compile `shared-state` due to previous error

```

Ох, це повідомлення про помилку доволі багатослівне! Ось важлива частина, на яку треба звернути увагу: ``Rc<Mutex<i32>>` cannot be sent between threads safely`. Компілятор також повідомляє нам чому: `the trait `Send` is not implemented for `Rc<Mutex<i32>>``. Ми поговоримо про `Send` в наступній секції: це один з трейтів, що гарантують, що типи, котрі ми використовуємо в потоках, призначені для використання в конкурентних ситуаціях.

На жаль, `Rc<T>` небезпечно спільно використовувати в декількох потоках. Коли `Rc<T>` керує підрахунком посилань, він додає одиницю до лічильника за кожен виклик `clone` і віднімає одиницю від лічильника, кожного разу коли значення клону видаляється. Проте він не використовує жодних примітивів конкурентності, щоб переконатися, що зміни лічильника не будуть перервані іншим потоком. Це може призвести до неправильного підрахунку посилань - проблем, які дуже важко помітити й ідентифікувати, і можуть призвести до витоків пам'яті (memory leaks) або ж значення може бути видалене, до того як ми з ним закінчимо. Нам потрібен тип, ідентичний `Rc<T>`, але такий, що робить зміни до лічильника підрахунку посилань в потокобезпечний (thread-safe) спосіб.

Атомарний Підрахунок Посилань із `Arc<T>`

На щастя, `Arc<T>` є типом, схожим на `Rc<T>`, але який безпечно використовувати в конкурентних ситуаціях. Літера *a* означає *atomic*, тобто це тип з *атомарним підрахуванням посилань*. Атоміки - це додатковий вид примітивів конкурентності, які ми не будемо тут детально розглядати: див. документацію стандартної бібліотеки для `std::sync::atomic` для більш докладної інформації. На даному етапі вам лише необхідно знати, що атоміки працюють як примітивні типи, але безпечні для спільного використання декількома потоками.

Ви можете запитати, чому всі примітивні типи не є атомарними і чому типи стандартної бібліотеки не використовують `Arc<T>` за замовчуванням. Причиною є те, що безпека потоків супроводжується зниженням швидкості виконання, а це штраф, який ви хочете заплатити лише тоді, коли це дійсно необхідно. Якщо ви просто виконуєте операції над значеннями в межах одного потоку, ваш код може працювати швидше, якщо йому не потрібно застосовувати гарантії, котрі надають атоміки.

Давайте повернемося до нашого прикладу: `Arc<T>` і `Rc<T>` мають однаковий API, тому ми просто виправляємо нашу програму змінюючи рядок з `use`, виклик `new`, а також виклик `clone`. Код в Блоці коду 16-15 нарешті скопілюється й виконається:

Файл: src/main.rs

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

fn main() {
    let counter = Arc::new(Mutex::new(0));
    let mut handles = vec![];

    for _ in 0..10 {
        let counter = Arc::clone(&counter);
        let handle = thread::spawn(move || {
            let mut num = counter.lock().unwrap();

            *num += 1;
        });
        handles.push(handle);
    }

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

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

Блок коду 16-15: Використання `Arc<T>` для обгортання `Mutex<T>` щоб мати можливість поділитися володінням між кількома потоками

Цей код виводить на екран наступне:

```
Result: 10
```

Ми зробили це! Ми рахували від 0 до 10, що може здатися не дуже вражаючим, але це навчило нас багато чому про `Mutex<T>` та безпеку потоків. Ви також можете використовувати структуру цієї програми для виконання більш складних операцій, ніж просто збільшення лічильника. Використовуючи цю стратегію, ви можете розділити обчислення на незалежні частини, потім розділити ці частини між потоками, а потім використати `Mutex<T>`, щоб кожен потік оновив кінцевий результат своєю частиною.

Завважте, що якщо ви виконуєте прості числові операції, є типи простіші за `Mutex<T>`, що визначені в модулі `std::sync::atomic` стандартної бібліотеки. Згадані типи забезпечують безпечний, конкурентний, атомарний доступ до примітивних типів. Для цього прикладу ми вирішили використовувати `Mutex<T>` із примітивним типом щоб ми могли зосередитися на тому, як працює `Mutex<T>`.

Подібності між `RefCell<T>/Rc<T>` та `Mutex<T>/Arc<T>`

Ви могли помітити, що `counter` є імутабельним, але ми могли б отримати мутабельне посилання на значення в ньому; це означає, що `Mutex<T>` забезпечує внутрішню мутабельність (interior mutability), як це робить `Cell`. Таким же чином ми використовували `RefCell<T>` у Розділі 15, щоб дозволити нам змінювати контент всередині `Rc<T>`, ми використовуємо `Mutex<T>` щоб змінити вміст у `Arc<T>`.

Ще одна деталь, яку слід зазначити, полягає в тому, що Rust не може захистити вас від усіх видів логічних помилок під час використання `Mutex<T>`. Згадайте, що в Розділі 15 ми обговорювали, що використання `Rc<T>` супроводжується ризиком створення циклічних посилань, де два значення `Rc<T>` посилаються один на одного, спричиняючи витоки пам'яті (memory leaks). Подібним чином, використання `Mutex<T>` несе з собою ризик створення взаємних блокувань. Це відбувається, коли операція потребує блокування двох ресурсів і кожен з двох потоків отримав оне з блокувань, таким чином змушуючи їх вічно чекати один одного. Якщо вас цікавлять взаємні блокування, спробуйте створити Rust програму, яка має взаємне блокування; потім пошукайте стратегії вирішення проблеми взаємних блокувань для м'ютексів в будь-якій мові та спробуйте реалізувати їх на Rust. API документація стандартної бібліотеки для `Mutex<T>` і `MutexGuard` надає корисну інформацію.

Ми завершимо цей розділ розповіддю про трейти `Send` і `Sync` і те, як ми можемо їх використовувати разом з власними типами.

Розширювана Конкурентність із Трейтами `Sync` та `Send`

Цікаво, що мова Rust має дуже небагато функціоналу для конкурентності. Майже весь функціонал конкурентності, про який ми будемо говорити в цьому розділі, є частиною стандартної бібліотеки, а не мови. Ваші опції для роботи з конкурентністю не обмежуються мовою чи стандартною бібліотекою; ви можете написати власний функціонал для конкурентності або використовувати ті, що вже були написані іншими.

Однак, дві концепції конкурентності вбудовані в мову: трейти `Sync` and `Send` із `std::marker`.

Дозвіл на Передачу Володіння між Потоками з `Send`

Маркерний трейт `Send` підказує нам, що володіння значенням типу, який реалізує `Send` можна передавати між потоками. Майже кожен тип в Rust реалізує `Send`, але є деякі винятки, включаючи `Rc<T>`: він не може реалізовувати `Send`, оскільки якщо ви клонували значення `Rc<T>` і спробували передати володіння клоном в інший потік, обидва потоки могли оновити лічильник підрахунку посилань одночасно. З цієї причини `Rc<T>` реалізовано для використання в одному потоці, коли ви не хочете жертвувати ефективністю виконання коду.

Тому, система типів Rust і межі трейтів (trait bounds) гарантують, що ви ніколи не зможете випадково небезпечно надіслати значення `Rc<T>` між потоками. Коли ми спробували зробити це в Блоці коду 16-14, то отримали помилку `the trait Send is not implemented for Rc<Mutex<i32>>`. Коли ми почали використовувати `Arc<T>`, який реалізує `Send`, код скомпілювався.

Будь-який тип, який повністю складається з типів, що реалізують `Send`, також автоматично позначається як `Send`. Майже всі примітивні типи реалізують `Send`, окрім сирих вказівників (raw pointers), які ми обговоримо в Розділі 19.

Дозвіл на Доступ з Декількох Потоків з `Sync`

Маркерний трейт `Sync` підказує нам, що на тип, котрий реалізує `Sync`, безпечно посилатись із декількох потоків. Іншими словами, будь-який тип `T` реалізує `Sync`, якщо `&T` (іmutable посилання на `T`) реалізує `Send`, тобто що посилання може бути

безпечно передане в інший потік. Подібно до `Send`, примітивні типи реалізують `Sync`, а типи, що складаються з типів, котрі реалізують `Sync` також позначаються як `Sync`.

Розумний вказівник `Rc<T>` також не реалізує `Sync` з тих самих причин з яких не реалізує `Send`. Тип `RefCell<T>` (про який ми говорили в Розділі 15) і сімейство пов'язаних типів `Cell<T>` також не реалізують `Sync`. Реалізація перевірки запозичень (borrow checking), яку `RefCell<T>` виконує під час виконання програми, не є потокобезпечною. Розумний покажчик `Mutex<T>` реалізує `Sync` і може бути спільно використовуватись декількома потоками, як ви могли побачити в секції "Спільне використання `Mutex<T>` декількома потоками" секції

Ручна Реалізація `Send` і `Sync` є Небезпечною

Оскільки типи, які складаються з типів, що реалізують `Send` і `Sync`, автоматично також реалізують `Send` і `Sync`, нам не потрібно реалізовувати ці трейти вручну. Як маркерні трейти, вони навіть не мають жодних методів, які потрібно реалізовувати. Вони просто корисні для забезпечення виконання інваріантів, пов'язаних із конкурентністю.

Ручна реалізація цих трейтів передбачає використання `unsafe Rust` коду. Ми поговоримо про використання `unsafe Rust` коду в Розділі 19; наразі ж, важливою інформацією є те, що для створення нових конкурентних типів, які не складаються з `Send` і `Sync`, потрібно ретельно продумати гарантії безпеки. "The Rustonomicon" містить більше інформації про такі гарантії та способи їх забезпечення.

Підсумок

Це не останній раз, коли ви читаєте про конкурентність у цій книзі: проєкт у Розділі 20 використовуватиме концепції цього розділу в більш реалістичній ситуації, ніж менші приклади, які тут розглядались.

Як вже згадувалося раніше, оскільки лише дуже маленька доля функціоналу для роботи з конкурентністю є частиною мови Rust, багато рішень реалізовано як крейти. Вони розвиваються швидше, ніж стандартна бібліотека, тому обов'язково шукайте в інтернеті поточні, найсучасніші крейти для використання при роботі з конкурентністю.

Стандартна бібліотека Rust надає канали для обміну повідомленнями і типи розумних вказівників, такі як `Mutex<T>` і `Arc<T>`, які безпечно використовувати в конкурентних контекстах. Система типів і borrow checker гарантують, що код, який використовує ці рішення, не призведе до гонитви даних або недійсних (невалідних) посилань. Як тільки ви зможете досягти того, що ваш код скомпільовався, ви можете бути впевнені, що він

успішно працюватиме в декількох потоках без типових для інших мов помилок, котрі важко відстежити. Конкурентне програмування більше не є концепцією, якої варто боятися: безстрашно робіть свої програми конкурентними!

Далі ми поговоримо про ідіоматичні способи моделювання проблем і структурування рішень, по мірі того як ваші Rust програми стають більшими. Крім того, ми обговоримо, як ідіоми Rust пов'язані з ідіомами, що можуть бути вам знайомі з об'єктно-орієнтованого програмування. [ch16-03-shared-state.html#sharing-a-mutex-between-multiple-threads](https://pavloslav.github.io/RustBookUkr/print.html#ch16-03-shared-state.html#sharing-a-mutex-between-multiple-threads)

Особливості Об'єктоорієнтованого Програмування в Rust

Об'єктоорієнтоване програмування (ОП) - це спосіб моделювання програм. Об'єкти як програмна концепція були вперше представлені в мові програмування Simula в 1960-х роках. Ці об'єкти вплинули на архітектуру програмного забезпечення Алана Кея, в якій об'єкти відправляли повідомлення один одному. Щоб описати цю архітектуру, він придумав термін *об'єктоорієнтоване програмування* у 1967 році. Є багато конкуруючих визначень ОП, і згідно з деякими з цих визначень Rust є об'єктоорієнтованим, але згідно з іншими - ні. В цьому розділі ми розглянемо деякі аспекти, які заведено вважати об'єктоорієнтованими, і те, як вони перекладаються на ідіоматичний Rust. Після чого ми покажемо, як реалізувати шаблони об'єктоорієнтованого дизайну в Rust, і обговоримо компроміси між ними та впровадженням рішення з використанням деяких сильних сторін Rust.

Характеристики Об'єктоорієнтованих Мов

Спільнота програмістів не дійшла згоди в питанні того, що повинна містити мова, щоб вважатися об'єктно орієнтованою. На Rust вплинуло багато парадигм програмування, включно з ООП; наприклад, ми розглянули особливості, які прийшли з функціонального програмування, у Розділі 13. Беззаперечно, ООП мови містять у собі деякі спільні характеристики, а саме об'єкти, інкапсуляцію і наслідування. Тож розгляньмо, що кожна з цих характеристик значить і чи Rust її підтримує.

Об'єкти Містять Дані та Поведінку

Книга Еріха Гамми, Річарда Гелма, Ральфа Джонсона і Джона Вліссайдса *Design Patterns: Elements of Reusable Object-Oriented Software* (Addison-Wesley Professional, 1994), яку в розмовній мові називають книгою *Банди Чотирьох*, є каталогом шаблонів об'єктно орієнтованого дизайну. В книзі ООП визначається наступним способом:

Об'єктно орієнтовані програми складаються з об'єктів. *Об'єкт* формується як даними, так і процедурами, котрі працюють з цими даними. Цими процедурами є так звані *методи* або *операції*.

Користуючись цим визначенням, Rust є об'єктно орієнтованим: структури та енуми містять дані, а блоки `impl` дозволяє реалізовувати методи для структур і еnumів. Не зважаючи на те, що структури й енуми з методами не *називають* об'єктами, вони містять той самий функціонал, що й об'єкти згідно з визначенням Банди Чотирьох.

Інкапсуляція, яка Приховує Деталі Реалізації

Іншим аспектом, який часто асоціюють з ООП, є ідея *інкапсуляції*. Головною ідеєю цього аспекту є те, що деталі реалізації об'єкта не є доступними з коду, який цей об'єкт користує. З цього випливає, що єдиним способом взаємодії з об'єктом є його публічне API; код, який використовує об'єкт, не повинен мати можливості прямого доступу до даних об'єкту, його внутрішнього стану чи безпосередньої зміни поведінки об'єкта. Це дозволяє програмісту змінювати і рефакторити внутрішній код об'єкта без необхідності зміни коду, який використовує об'єкт.

Ми обговорили, як контролювати інкапсуляцію, у Розділі 7: ми можемо використовувати ключове слово `pub`, щоб визначити, які модулі, типи, функції і методи нашого коду повинні бути публічними. За замовчанням усе є приватним. Наприклад, ми можемо визначити структуру `AveragedCollection`, котра містить поле - вектор значень `i32`. Структура також може містити поле, яке зберігає середнє значення в векторі, що дозволяє не перераховувати середнє значення кожен раз, коли хтось його запросить. Іншими словами, `AveragedCollection` буде кешувати підраховане середнє значення для нас. В Блоці коду 17-1 міститься визначення структури `AveragedCollection`:

Файл: `src/lib.rs`

```
pub struct AveragedCollection {  
    list: Vec<i32>,  
    average: f64,  
}
```

Блок коду 17-1: Структура `AveragedCollection`, що зберігає список цілих чисел і середнє значення елементів у колекції

Структуру позначено `pub`, щоб інший код міг її використовувати, але поля структури залишаться приватними. Це важливо, оскільки ми хочемо гарантувати, що коли б ми не додали чи забрали якесь значення зі списку -- середнє значення теж оновилось. Ми досягаємо цього, реалізуючи для структури методи `add`, `remove` і `average` так, як показано в Блоці коду 17-2:

Файл: `src/lib.rs`

```
impl AveragedCollection {
    pub fn add(&mut self, value: i32) {
        self.list.push(value);
        self.update_average();
    }

    pub fn remove(&mut self) -> Option<i32> {
        let result = self.list.pop();
        match result {
            Some(value) => {
                self.update_average();
                Some(value)
            }
            None => None,
        }
    }

    pub fn average(&self) -> f64 {
        self.average
    }

    fn update_average(&mut self) {
        let total: i32 = self.list.iter().sum();
        self.average = total as f64 / self.list.len() as f64;
    }
}
```

Блок коду 17-2: Реалізації публічних методів `add`, `remove` і `average` для `AveragedCollection`

Публічні методи `add`, `remove` і `average` є єдиним способом, щоб отримати доступ чи змінити дані в екземплярі `AveragedCollection`. Коли ми додаємо елемент до `list` за допомогою методу `add` чи видаляємо методом `remove`, реалізація всіх методів викличе приватний метод `update_average`, який обробить зміну середнього значення.

Ми залишаємо поля `list` і `average` приватними, щоб не було іншого способу отримати доступ до них напряму. В іншому випадку, поле `average` може виявитися не синхронізованим з полем `list`. Метод `average` повертає значення, яке міститься в полі `average`, дозволяючи користувачам структури прочитати середнє значення, але не модифікувати його.

Оскільки ми інкапсулювали деталі реалізації структури `AveragedCollection`, ми без проблем можемо змінити аспекти реалізації структури в майбутньому. Наприклад, ми можемо використати `HashSet<i32>` замість `Vec<i32>` для поля `list`. Доки сигнатура публічних методів `add`, `remove` і `average` залишається незмінною, використання `AveragedCollection` не потрібно буде змінювати. Якби ми зробили `list` публічним, можливо б довелося змінювати спосіб взаємодії зі структурою: `HashSet<i32>` і `Vec<i32>`

мають різні способи додавання і видалення елементів, тому користувачеві структури, скоріше за все, довелося б змінювати використання структури `list`.

Якщо інкапсуляція є обов'язковим аспектом для об'єктно орієнтованої мови, то Rust можна вважати такою. Наявність чи відсутність ключового слова `pub` дозволяє інкапсулювати деталі реалізації.

Спадкування як Система Типів та як Система Спільного Використання Коду

Наслідування — це механізм, за допомогою якого об'єкт може успадковувати елементи з визначення іншого об'єкта, таким чином отримуючи дані та поведінку батьківського об'єкта без потреби визначати їх знову.

Якщо мова повинна мати наслідування, щоб бути об'єктно орієнтованою, то Rust не є нею. Тут нема способу визначити структуру, яка успадковує поля та реалізації методів батьківської структури, без використання макросу.

Однак, якщо ви звикли мати успадкування у своєму наборі інструментів програмування, ви можете використовувати інші рішення в Rust, залежно від того, чому ви спочатку звернулися до успадкування.

Є дві основні причини, щоб використовувати наслідування. Перша з них, це щоб перевикористати код: ви можете реалізувати поведінку для якогось одного типу, а наслідування дозволить вам перевикористати реалізацію для іншого типу. Ви можете використати обмежену версію цього підходу в Rust, з допомогою усталеної реалізації трейту, яку ви бачили в роздруківці 10-14, коли ми додали усталену реалізацію методу `summarize` для трейту `Summary`. Кожний тип, який реалізовує трейт `Summary` матиме доступним метод `summarize` без повторного написання коду. Це є схожим до батьківського класу, який містить реалізацію методу, і дочірнього класу, який успадковує реалізацію цього ж методу. Також у випадках реалізації трейту `Summary`, ми можемо перевизначити усталену реалізацію методу `summarize` власною, що схоже до перевизначення батьківського методу в дочірньому класі.

Інша причина використання наслідування пов'язана з системою типів: дозволяти використовувати дочірній тип в тих місцях, де очікується батьківський тип. Це також називається *поліморфізмом*, що означає, що ви можете без проблем замінити один об'єкт на інший, якщо той задовольняє певні вимоги.

Поліморфізм

Для багатьох людей поліморфізм є синонімом до наслідування. Але це більш загальний концепт, що описує код, який працює з декількома типами. При наслідуванні ці типи повинні бути підкласами.

На заміну цьому, Rust використовує абстрактний тип для абстрагування над різними можливими типами та трейтами, щоб накласти обмеження на те, що мають надавати ці типи. Це іноді називають *обмеженим параметричним поліморфізмом*.

Останнім часом програмісти почали значно рідше використовувати наслідування, оскільки наявний великий ризик перевикористання більше коду ніж необхідно. Підкласи не завжди повинні мати всі ті ж самі характеристики, що і їхні батьки, однак ми завжди робимо так з наслідуванням. Це може зробити архітектуру програм менш гнучкою. Також це створює можливість викликати методи в підкласах попри те, що це не має сенсу чи викликає помилки, бо цей метод не застосовний для цього підкласу. На додачу до цього, деякі мови забороняють множинне наслідування (тобто підклас може наслідуватися лише під одного класу), що ще сильніше зменшує гнучкість програм.

З цих причин Rust використовує інший підхід: використання трейтів замість успадкування. Подивімося, як трейти забезпечують поліморфізм у Rust.

Використання Трейт-Об'єктів, які Допускають Значення Різних Типів

У розділі 8 ми казали, що одним з обмежень векторів є те, що вони можуть зберігати елементи тільки одного типу. Ми обійшли цю проблему в Роздруку 8-9, де ми визначили еnum `SpreadsheetCell` який мав варіанти для зберігання цілих чисел, чисел з рухомою комою й тексту. Це означало, що ми мали змогу зберігати різні типи даних в кожній комірці та все одно мати вектор, який представляє рядок комірок. Це дуже гарне рішення коли наші взаємозамінні елементи є типами з фіксованим набором, відомим на етапі компіляції.

Проте іноді ми хочемо, щоб користувач нашої бібліотеки зміг розширити набір типів, які є допустимими в конкретній ситуації. Щоб показати, як ми можемо досягти цього, ми створимо приклад інструменту з графічним інтерфейсом користувача (GUI), який ітерує список елементів, викликаючи метод `draw` на кожному з них, щоб намалювати його на екрані — це поширена техніка для GUI інструментів. Ми створимо бібліотечний крейт `gui`, який містить структуру бібліотеки GUI. Цей крейт може містити деякі готові до використання типи, наприклад тип `Button` чи `TextField`. Крім того, користувачі крейту `gui` можуть захотіти створити свої власні типи, які можуть бути намальовані: наприклад, один програміст може додати тип `Image`, а інший - `SelectBox`.

Ми не будемо реалізовувати повноцінну GUI бібліотеку для цього прикладу, але покажемо як її частини будуть поєднуватися. Коли ми пишемо бібліотеку, ми не можемо знати та визначити всі типи, які можуть захотіти створити інші програмісти. Але ми знаємо що `gui` повинен відстежувати багато значень різного типу та викликати метод `draw` кожного з цих по-різному типізованому значень. Крейт не повинен знати, що станеться, коли ми викличемо метод `draw`, просто у значення буде доступний для виклику такий метод.

Для того, щоб зробити це на мові, в якій є наслідування, ми можемо визначити клас під назвою `Component`, який має метод `draw`. Інші класи, такі як `Button`, `Image`, та `SelectBox`, можуть успадкуватися від `Component` й таким чином успадкувати метод `draw`. Кожен з них може перевизначити реалізацію методу `draw`, щоб описати власну поведінку, але фреймворк може розглядати всі типи ніби вони є екземпляром `Component` та міг би викликати їх метод `draw`. Але, оскільки Rust не має механізму успадкування, нам потрібен інший спосіб структурувати `gui` бібліотеку, щоб дозволити користувачам розширювати її новими типами.

Визначення Трейту для Спільної Поведінки

Для реалізації поведінки, яку ми хочемо мати в `gui`, визначимо трейт під назвою `Draw`, який буде містити один метод `draw`. Тоді ми можемо визначити вектор, який приймає *трейт-об'єкт*. Трейт-об'єкт вказує як на екземпляр типу, що реалізує вказаний нами трейт, так і на внутрішню таблицю, що використовується для пошуку методів трейту вказаного типу під час виконання. Ми створюємо трейт-об'єкт в такому порядку: використовуємо якийсь вид вказівнику, наприклад посилання `&` або розумний вказівник `Box<T>`, потім ключове слово `dyn` й відповідний трейт. (Ми будемо говорити чому трейт-об'єкти повинні використовувати вказівник у Розділі 19 в секції "[Dynamically Sized Types and the Sized Trait](#).") Ми можемо використовувати трейт-об'єкт замість узагальненого або конкретного типу. Де б ми не використовували трейт-об'єкт, система типів Rust забезпечить, що під час компіляції будь-яке значення використане у цьому контексті буде реалізовувати трейт трейт-об'єкту. Отже, ми не повинні знати всі можливі типи під час компіляції.

Ми нагадували, що в Rust ми не називаємо структури та енуми "об'єктами", щоб розрізнити їх з об'єктами в інших мовах програмування. У структурі або енумі, дані в полях структури та поведінка в блоку `impl` розділені, тоді як в інших мовах вони об'єднанні в один концепт, який часто називають об'єкт. Однак, трейт-об'єкти є більше схожими на об'єкти в інших мовах, в тому сенсі що вони об'єднують дані та поведінку. Але трейт-об'єкти відрізняються від традиційних об'єктів у том, що ми не можемо додати дані до трейт-об'єкту. Трейт-об'єкти загалом не настільки корисні як об'єкти в інших мовах програмування: їх конкретна ціль - забезпечити абстракцію через загальну поведінку.

Блок коду 17-3 показує, як визначити трейт під назвою `Draw` з одним методом `draw`:

Файл: `src/lib.rs`

```
pub trait Draw {
    fn draw(&self);
}
```

Блок коду 17-3: Визначення трейту `Draw`

Цей синтаксис має бути знайомим після наших дискусій про те, як визначати трейти в розділі 10. Далі йде новий синтаксис: у Роздруку 17-4 визначена структура під назвою `Screen`, яка містить вектор з ім'ям `components`. Цей вектор має тип `Box<dyn Draw>`, який і є трейт-об'єктом; це позначення будь-якого типу всередині `Box`, який реалізує трейт `Draw`.

Файл: `src/lib.rs`

```
pub struct Screen {  
    pub components: Vec<Box<dyn Draw>>,  
}
```

Блок коду 17-4: Визначення структури `Screen` з полем `components`, яке є вектором трейт-об'єктів, що реалізують трейт `Draw`

У структурі `Screen` ми визначено метод під назвою `run`, який буде викликати метод `draw` кожного елементу вектора `components`, як показано у Блоці коду 17-5:

Файл: `src/lib.rs`

```
impl Screen {  
    pub fn run(&self) {  
        for component in self.components.iter() {  
            component.draw();  
        }  
    }  
}
```

Блок коду 17-5: Метод `run` в структурі `Screen`, який викликає метод `draw` кожного компоненту

Це працює інакше ніж визначення структури, яка використовує параметр узагальненого типу з обмеженнями трейтів. Узагальнений параметр типу може бути замінений тільки одним конкретним типом, тоді як трейт-об'єкти дозволяють декільком конкретним типам бути на його місці під час виконання. Наприклад, визначимо структуру `Screen` використовуючи узагальнені типи та обмеження трейту в Блоці коду 17-6:

Файл: `src/lib.rs`

```
pub struct Screen<T: Draw> {  
    pub components: Vec<T>,  
}  
  
impl<T> Screen<T>  
where  
    T: Draw,  
{  
    pub fn run(&self) {  
        for component in self.components.iter() {  
            component.draw();  
        }  
    }  
}
```


Блок коду 17-6: Альтернативна реалізація структури `Screen` та її методу `run` за допомогою узагальнених типів та обмежень трейту

Це обмежує екземпляр `Screen` до одного з двох можливих варіантів: наповнений лише компонентами типу `Button`, або лише компонентами типу `TextField`. Якщо у вас коли-небудь будуть тільки однорідні колекції, використання узагальнених типів та обмежень трейту краще, оскільки визначення будуть мономорфізованими під час компіляції для використання з конкретними типами.

З іншого боку, за допомогою методу, який використовує трейт-об'єкт, один екземпляр `Screen` може містити `Vec<T>`, який містить `Box<Button>`, так само як і `Box<TextField>`. Нумо подивімось як це працює, а потім поговоримо про вплив на швидкодію під час виконання.

Реалізація Трейту

Тепер ми додамо деякі типи, які реалізуються трейт `Draw`. Запровадимо тип `Button`. Знову ж таки, фактична реалізація бібліотеки GUI виходить за межі цієї книги, тому тіло методу `draw` не буде мати ніякої корисної реалізації. Щоб уявити, як може виглядати така реалізація, структура `Button` може мати поля для `width`, `height`, та `label`, як показано в Роздруку 17-7:

Файл: `src/lib.rs`

```
pub struct Button {
    pub width: u32,
    pub height: u32,
    pub label: String,
}

impl Draw for Button {
    fn draw(&self) {
        // code to actually draw a button
    }
}
```

Блок коду 17-7: Структура `Button`, яка реалізує трейт `Draw`

Поля `width`, `height`, та `label` структури `Button` будуть відрізнятися від полів інших компонентів; наприклад, тип `TextField` міг би мати такі самі поля плюс поле `placeholder`. Кожен тип, який ми хочемо намалювати на екрані, буде реалізовувати трейт `Draw`, але буде мати інший код методу `draw` для визначення того, як саме малювати конкретний тип, наприклад `Button` в цьому прикладі (без фактичного коду GUI, який виходить за межі цього розділу). Наприклад, тип `Button` може мати додаткові блоки

`impl`, що містять методи, які визначають що станеться, коли користувач натисне на кнопку. Такі методи не застосовуватимуться до таких типів, як `TextField`.

Якщо користувач нашої бібліотеки вирішить реалізувати структуру `SelectBox`, яка має `width`, `height`, та `options` поля, він реалізує також і трейт `Draw` для структури `SelectBox`, як показано в Роздрукі 17-8:

Файл: `src/lib.rs`

```
use gui::Draw;

struct SelectBox {
    width: u32,
    height: u32,
    options: Vec<String>,
}

impl Draw for SelectBox {
    fn draw(&self) {
        // code to actually draw a select box
    }
}
```

Блок коду 17-8: Інший крейт використовує `gui` та реалізує трейт `Draw` для структури `SelectBox`

Тепер користувач нашої бібліотеки може написати свою `main` функцію, щоб створити екземпляр `Screen`. До екземпляра `Screen`, він може додати `SelectBox` та `Button`, розмістивши кожен з них у `Box<T>`, щоб він став трейт-об'єктом. Потім він може викликати метод `run` в екземпляра `Screen`, який викличе метод `draw` для кожного компонента. Роздрук 17-9 показує цю реалізацію:

Файл: `src/lib.rs`

```

use gui::{Button, Screen};

fn main() {
    let screen = Screen {
        components: vec![
            Box::new(SelectBox {
                width: 75,
                height: 10,
                options: vec![
                    String::from("Yes"),
                    String::from("Maybe"),
                    String::from("No"),
                ],
            }),
            Box::new(Button {
                width: 50,
                height: 10,
                label: String::from("OK"),
            }),
        ],
    };

    screen.run();
}

```

Блок коду 17-9: Використання трейт-об'єктів для зберігання значень різних типів, які реалізують той самий трейт

Коли ми писали бібліотеку, ми не знали, що хтось може додати тип `SelectBox`, але наша реалізація `Screen` мала змогу працювати з новим типом та малювати його, тому що `SelectBox` реалізує трейт `Draw`, що означає, що він реалізує метод `draw`.

Ця концепція, яка стосується тільки повідомлень на які значення відповідає, на відміну від конкретного типу в значення, аналогічна концепції *duck typing* (качкової типізації) у динамічно типізованих мовах: якщо хтось ходить як качка та крякає як качка, то він - качка! У реалізації методу `run` структури `Screen` в Роздруку 17-5, `run` не повинен знати конкретний тип кожного компонента. Він не перевіряє чи є компонент екземпляром `Button` чи `SelectBox`, він просто викликає метод `draw` компоненту. Вказавши `Box<dyn Draw>` як тип значень у векторі `components`, ми визначили `Screen` для значень у яких ми можемо викликати метод `draw`.

Перевага використання трейт-об'єктів і системи типів Rust для написання коду подібного до коду з використанням качкової типізації полягає в тому, що нам ніколи не потрібно перевіряти, чи реалізує значення певний метод під час виконання або турбуватися про отримання помилок якщо значення не реалізує метод. Rust не буде компілювати наш код, якщо значення не реалізують трейт потрібного трейт-об'єкту.

Наприклад, Блок коду 17-10 показує, що станеться, якщо ми спробуємо створити `Screen` з `String` як компонент:

Файл: `src/lib.rs`

```
use gui::Screen;

fn main() {
    let screen = Screen {
        components: vec![Box::new(String::from("Hi"))],
    };

    screen.run();
}
```

Роздрук 17-10: Спроба використати тип, який не реалізує трейт трейт-об'єкту

Ми отримаємо помилку, тому що `String` не реалізує трейт `Draw`:

```
$ cargo run
   Compiling gui v0.1.0 (file:///projects/gui)
error[E0277]: the trait bound `String: Draw` is not satisfied
--> src/main.rs:5:26
   |
5  |         components: vec![Box::new(String::from("Hi"))],
   |                                ^^^^^^^^^^^^^^^^^^^^^^^^^ the trait `Draw` is not
   | implemented for `String`
   |
   = help: the trait `Draw` is implemented for `Button`
   = note: required for the cast from `String` to the object type `dyn Draw`

For more information about this error, try `rustc --explain E0277`.
error: could not compile `gui` due to previous error
```

Ця помилка дає зрозуміти, що або ми передаємо в компонент `Screen` щось, що ми не збиралися передавати, і тоді ми повинні передати інший тип, або ми повинні реалізувати трейт `Draw` у типу `String`, щоб `Screen` міг викликати `draw` у нього.

Трейт-Об'єкти Виконують Динамічну Диспетчеризацію

Нагадаємо, у секції “Швидкодія коду з узагальненими типами” розділу 10 обговорюється процес мономорфізації, який виконується компілятором, коли ми використовуємо обмеження трейтів для узагальнених типів: компілятор генерує конкретні типи, які ми використовуємо замість параметра узагальненого типу. Код, отриманий в результаті мономорфізації, виконує *статичну диспетчеризацію*, коли компілятор знає який метод ви викликаєте під час компіляції. Це протилежний підхід до *динамічної диспетчеризації*, коли

компілятор не може сказати під час компіляції, який метод ви викликаєте. У випадках динамічної диспетчеризації компілятор генерує код, який під час виконання визначає, який метод необхідно викликати.

Коли ми використовуємо трейт-об'єкти, Rust має використовувати динамічну диспетчеризацію. Компілятор не знає всі типи, які можуть бути використані з кодом, який використовує трейт-об'єкти, тому він не знає, який метод реалізований для якого типу при виклику. Замість цього, під час виконання, Rust використовує вказівники всередині трейт-об'єкту, щоб дізнатися який метод викликати. Такий пошук провокує додаткові витрати під час виконання, які не потребуються під час статичної диспетчеризації. Динамічна диспетчеризація також не дозволяє компілятору обрати вбудовування коду метода, що робить неможливим деякі оптимізації. Однак, ми отримали додаткову гнучкість у коді, який ми написали у Роздруку 17-5, і змогли підтримати у Роздруку 17-9, так що це - компроміс для розгляду. ch10-01-syntax.html#performance-of-code-using-generics

Реалізація Об'єктоорієнтованого Шаблону Проектування

Шаблон "Стан" - це об'єктоорієнтований шаблон проектування. Сенса патерну полягає в тому, що ми визначаємо набір станів, в яких може знаходитися значення. Стани представлені набором *об'єктів стану*, а поведінка значення змінюється в залежності від його стану. Розглянемо на прикладі структури допису в блозі, що має поле для збереження її стану, яке буде об'єктом стану з набору "чернетка" (draft), "очікування перевірки" (review) або "опубліковано" (published).

Об'єкти стану мають спільну функціональність: звісно в Rust, ми використовуємо структури й трейти, а не об'єкти та наслідування. Кожний об'єкт стану відповідає за свою поведінку й сам визначає, коли він повинен перейти в інший стан. Значення, яке зберігає об'єкт стану, нічого не знає про різницю в поведінці станів або про те, коли один стан повинен перейти в інший.

Перевага використання патерну "Стан" полягає в тому, що при змінненні бізнес-вимог до програми нам не потрібно буде змінювати код значення, що зберігає стан, або код, який використовує це значення. Нам потрібно буде оновити код всередині одного з об'єктів стану, щоб змінити його правила чи можливо додати більше об'єктів стану.

Спочатку, ми реалізуємо шаблон "Стан" більш традиційним об'єктоорієнтованим способом, а потім використаємо більш ідіоматичний для Rust підхід. Розглянемо поетапну реалізацію робочого процесу публікації в блозі з використанням патерну "Стан".

Остаточна функціональність буде виглядати наступним чином:

1. Створення допису в блозі починається з пустої чернетки.
2. Коли чернетка готова, робиться запит на схвалення допису.
3. Коли допис буде схвалено, він опублікується.
4. Тільки опубліковані дописи блогу повертають контент для друку, тому несхвалені дописи не можуть випадково бути опубліковані.

Будь-які інші зміни, зроблені в дописі, не повинні мати ефекту. Наприклад, якщо ми спробуємо затвердити чернетку допису в блозі перед тим, як ми подали запит на затвердження, допис має залишатися неопублікованою чернеткою.

Лістинг 17-11 показує цей процес у вигляді коду: це приклад використання API (прикладного програмного інтерфейсу), який ми будемо впроваджувати у бібліотечному крейті під назвою `blog`. Цей приклад не скопіюється, тому що ми ще не встигли реалізувати крейт `blog`.

Файл: src/main.rs

```
use blog::Post;

fn main() {
    let mut post = Post::new();

    post.add_text("I ate a salad for lunch today");
    assert_eq!("", post.content());

    post.request_review();
    assert_eq!("", post.content());

    post.approve();
    assert_eq!("I ate a salad for lunch today", post.content());
}
```

Блок коду 17-11: Код, який демонструє поведінку, яку ми хочемо, щоб мав крейт `blog`

Ми хочемо дозволити користувачеві створити новий допис у блозі за допомогою `Post::new`. Ми хочемо дозволити додавати текст у допис блогу. Якщо ми спробуємо отримати зміст допису до схвалення публікації, ми не повинні отримувати ніякого тексту, оскільки допис все ще є чернеткою. Ми додали `assert_eq!` в коді для демонстрації цілей. Ідеальним модульним (unit) тестом для цього було б твердження, що чернетка допису повертає порожній рядок з методу `content`, але ми не будемо писати тести для цього прикладу.

Далі ми хочемо дозволити запит на схвалення допису, і також щоб `content` повертав порожню стрічку під час очікування схвалення. Коли допис пройде перевірку, він повинен бути опублікований, тобто виклик методу `content` буде повертати текст допису.

Зверніть увагу, що єдиний тип з крейту, з яким ми взаємодіємо - це тип `Post`. Цей тип буде використовувати патерн "Стан" і буде містити значення, яке буде одним з трьох об'єктів станів, які представляють різні стани, в яких може знаходитися допис: "чернетка", "очікування перевірки", або "опубліковано". Керування переходом з одного стану в інший буде здійснюватися внутрішньою логікою типу `Post`. Стани будуть перемикатися в результаті реакції на виклик методів екземпляру `Post` користувачами нашої бібліотеки, але користувачі не повинні керувати зміною станів напряму. Крім того, користувачі не повинні мати можливість помилитися зі станами, наприклад, опублікувати повідомлення до його перевірки.

Визначення `Post` і Створення Нового Екземпляру в Стані Чернетки

Нумо почнімо реалізовувати бібліотеку! Ми знаємо, що нам потрібна публічна структура `Post`, яка зберігає деякий вміст, тому ми почнемо з визначення структури та пов'язаною з нею публічною функцією `new` для створення екземпляра `Post`, як показано в Блоці коду 17-12. Ми також зробимо приватний трейт `State`, який буде визначати поведінку, що повинні будуть мати всі об'єкти станів структури `Post`.

Далі `Post` буде містити трейт-об'єкт `Box<dyn State>` всередині `Option<T>` в приватному полі `state` для зберігання об'єкту стану. Трохи пізніше ви зрозумієте, навіщо потрібно використання `Option<T>`.

Файл: `src/lib.rs`

```
pub struct Post {
    state: Option<Box<dyn State>>,
    content: String,
}

impl Post {
    pub fn new() -> Post {
        Post {
            state: Some(Box::new(Draft {})),
            content: String::new(),
        }
    }
}

trait State {}

struct Draft {}

impl State for Draft {}
```

Блок коду 17-12. Визначення структури `Post` та функції `new`, яка створює новий екземпляр `Post`, трейту `State` і структури `Draft`

Трейт `State` визначає поведінку, яку спільно використовують різні стани допису. Всі об'єкти станів (`Draft` - чернетка, `PendingReview` - очікування перевірки, `Published` - опубліковано) будуть реалізовувати трейт `State`. Зараз у цього трейту немає ніяких методів, і ми почнемо з визначення `Draft`, тому що, що це перший стан, з якого, як ми хочемо, публікація буде починати свій шлях.

Коли ми створюємо новий екземпляр `Post`, ми встановлюємо його поле `state` в значення `Some`, що містить `Box`. Цей `Box` вказує на новий екземпляр структури `Draft`. Це гарантує, щоразу, коли ми створюємо новий екземпляр `Post`, він з'явиться як

чернетка. Оскільки поле `state` в структурі `Post` є приватним, нема ніякого способу створити `Post` в якомусь іншому стані! У функції `Post::new` ми ініціалізуємо поле `content` новим пустим рядком типу `String`.

Зберігання Тексту Вмісту Публікації

У Блоці коду 17-11 показано, що ми хочемо мати можливість викликати метод `add_text` і передати йому `&str`, яке додається до текстового вмісту допису блогу. Ми реалізуємо цю можливість як метод, а не робимо поле `content` публічним, використовуючи `pub`, щоб пізніше ми могли реалізувати метод, який буде керувати тим, як дані поля `content` будуть зчитуватися. Метод `add_text` досить простий, тому додаймо його реалізацію в блок `impl Post` у Блоці коду 17-13:

Файл: `src/lib.rs`

```
impl Post {  
    // --snip--  
    pub fn add_text(&mut self, text: &str) {  
        self.content.push_str(text);  
    }  
}
```

Блок коду 17-13. Реалізація методу `add_text` для додавання тексту до `content` (вмісту) допису

Метод `add_text` приймає змінюване посилання на `self`, тому що ми змінюємо екземпляр `Post`, для якого викликаємо `add_text`. Потім ми викликаємо `push_str` для `String` у поля `content` і передаємо `text` аргументом для додавання до збереженого `content`. Ця поведінка не залежить від стану, в якому знаходяться допис, таким чином він не є частиною патерну "Стан". Метод `add_text` взагалі не взаємодіє з полем `state`, але це частина поведінки, яку ми хочемо підтримувати.

Забезпечення Відсутності Вмісту Чернетки Публікації

Навіть після того, як ми викликали метод `add_text` і додали деякий контент в наш допис, ми хочемо, щоб метод `content` повертав порожній стрічковий слайс, тому, що допис все ще знаходиться в стані чернетки, як це показано в рядку 7 Блока коду 17-11. Поки що реалізуймо метод `content` найпростішим способом, який буде задовольняти цій вимозі: будемо завжди повертати порожній стрічковий слайс. Ми змінимо код пізніше, як тільки реалізуємо можливість змінити стан допису, щоб вона могла бути опублікована. Поки що

дописи можуть знаходитися тільки в стані чернетки, тому вміст допису завжди повинен бути пустим. Лістинг 17-14 показує цю реалізацію-заглушку:

Файл: src/lib.rs

```
impl Post {  
    // --snip--  
    pub fn content(&self) -> &str {  
        ""  
    }  
}
```

Блок коду 17-14: Додавання реалізації-заглушки для методу `content` в `Post`, яка завжди повертає порожній стрічковий слайс

З доданим методом `content` усе в Блоці коду 17-11 працює, як треба, аж до рядка 7.

Запит на Перевірку Публікації Змінює Її Стан

Далі нам потрібно додати функціональність для запиту на перевірку допису, який повинен змінити її стан з `Draft` на `PendingReview`. Лістинг 17-15 показує такий код:

Файл: src/lib.rs

```

impl Post {
    // --snip--
    pub fn request_review(&mut self) {
        if let Some(s) = self.state.take() {
            self.state = Some(s.request_review())
        }
    }
}

trait State {
    fn request_review(self: Box<Self>) -> Box<dyn State>;
}

struct Draft {}

impl State for Draft {
    fn request_review(self: Box<Self>) -> Box<dyn State> {
        Box::new(PendingReview {})
    }
}

struct PendingReview {}

impl State for PendingReview {
    fn request_review(self: Box<Self>) -> Box<dyn State> {
        self
    }
}

```

Блок коду 17-15: Реалізація методу `request_review` в структурі `Post` і трейті `State`

Ми додаємо в `Post` публічний метод з іменем `request_review`, який буде приймати змінюване посилання на `self`. Далі ми викликаємо внутрішній метод `request_review` для поточного стану `Post`, і цей другий метод `request_review` поглинає поточний стан та повертає новий стан.

Ми додаємо метод `request_review` в трейт `State`; всі типи, які реалізують цей трейт, тепер повинні будуть реалізувати метод `request_review`. Зверніть увагу, що замість `self`, `&self`, або `&mut self` як першого параметра метода в нас вказаний `self: Box<Self>`. Цей синтаксис означає, що метод дійсний тільки при його виклику з обгорткою `Box`, яка містить наш тип. Цей синтаксис стає власником `Box<Self>`, і робить старий стан недійсним, тому значення стану `Post` може бути перетворення в новий стан.

Щоб поглинути старий стан, метод `request_review` повинен стати власником значення стану. Це місце, де приходиться на допомогу тип `Option` поля `state` допису `Post`: ми викликаємо метод `take`, щоб забрати значення `Some` з поля `state` і залишити замість нього значення `None`, тому що Rust не дозволяє мати неініціалізовані поля в структурах.

Це дозволяє переміщувати значення `state` з `Post`, а не запозичувати його. Потім ми встановимо нове значення `state` як результат цієї операції.

Нам потрібно тимчасово встановити `state` в `None` замість того, щоб встановити його напряду за допомогою коду на кшталт `self.state = self.state.request_review();` щоб отримати власність над значенням `state`. Це гарантує, що `Post` не зможе використовувати старе значення `state` після того, як ми перетворили його в новий стан.

Метод `request_review` в `Draft` повинен повернути екземпляр нової структури `PendingReview` обгорнутої в `Box`, яка є станом, коли допис очікує на перевірку. Структура `PendingReview` також реалізує метод `request_review`, але не виконує ніяких трансформацій. Вона повертає сама себе, тому що, коли ми робимо запит на перевірку допису, який вже знаходиться в стані `PendingReview`, вона все одно повинна продовжувати залишатися в стані `PendingReview`.

Тепер ми починаємо бачити переваги патерну "Стан": метод `request_review` для `Post` однаковий, він не залежить від значення `state`. Кожен стан сам несе відповідальність за власну поведінку.

Залишимо метод `content` в `Post` без змін, тобто який повертає порожній стрічковий слайс. Тепер ми можемо мати `Post` як у стані `PendingReview`, так і в стані `Draft`, але ми хочемо отримати таку саму поведінку в стані `PendingReview`. Лістинг 17-11 тепер працює до рядка 10!

Додавання Методу `approve` для Зміни Поведінки Методу `content`

Метод `approve` ("схвалити") буде аналогічним методу `request_review`: він буде встановлювати в `state` значення, яке повинен мати допис при його схваленні, як показано в Блоці коду 17-16:

Файл: `src/lib.rs`

```

impl Post {
    // --snip--
    pub fn approve(&mut self) {
        if let Some(s) = self.state.take() {
            self.state = Some(s.approve())
        }
    }
}

trait State {
    fn request_review(self: Box<Self>) -> Box<dyn State>;
    fn approve(self: Box<Self>) -> Box<dyn State>;
}

struct Draft {}

impl State for Draft {
    // --snip--
    fn approve(self: Box<Self>) -> Box<dyn State> {
        self
    }
}

struct PendingReview {}

impl State for PendingReview {
    // --snip--
    fn approve(self: Box<Self>) -> Box<dyn State> {
        Box::new(Published {})
    }
}

struct Published {}

impl State for Published {
    fn request_review(self: Box<Self>) -> Box<dyn State> {
        self
    }

    fn approve(self: Box<Self>) -> Box<dyn State> {
        self
    }
}

```

Блок коду 17-16: Реалізація методу `approve` для типу `Post` і трейту `State`

Ми додаємо метод `approve` в трейт `State` та додаємо нову структуру, яка реалізує трейт `State` для стану `Published`.

Подібно до того, як працює метод `request_review` для `PendingReview`, якщо ми викличемо метод `approve` для `Draft`, це не буде мати ніякого ефекту, тому що `approve`

поверне `self`. Коли ми викликаємо метод `approve` для `PendingReview`, він повертає новий, обгорнутий у `Box`, екземпляр структури `Published`. Структура `Published` реалізує трейт `State`, і як для методу `request_review`, так і для методу `approve` вона повертає себе, тому що в цих випадках допис повинен залишатися в стані `Published`.

Тепер нам потрібно оновити метод `content` для `Post`. Ми хочемо, щоб значення, яке повертається з `content`, залежало від поточного стану `Post`, тому ми збираємося делегувати частину функціональності `Post` в метод `content`, визначений для `state`, як показано в Блоці коду 17-17:

Файл: `src/lib.rs`

```
impl Post {  
    // --snip--  
    pub fn content(&self) -> &str {  
        self.state.as_ref().unwrap().content(self)  
    }  
    // --snip--  
}
```

Блок коду 17-17: Оновлення методу `content` в структурі `Post` для делегування частини функціональності методу `content` структури `State`

Оскільки наша ціль полягає в тому, щоб зберегти ці дії всередині структур, які реалізують трейт `State`, ми викликаємо метод `content` у значення в полі `state` і передаємо екземпляр публікації (тобто `self`) як аргумент. Потім ми повертаємо значення, яке нам повертає виклик методу `content` поля `state`.

Ми викликаємо метод `as_ref` у `Option`, тому що нам потрібне посилання на значення всередині `Option`, а не володіння значенням. Оскільки `state` є типом `Option<Box<dyn State>>`, то під час виклику методу `as_ref` повертається `Option<&Box<dyn State>>`. Якби ми не викликали `as_ref`, отримали б помилку, тому що ми не можемо перемістити `state` з запозиченого параметра `&self` функції.

Далі ми викликаємо метод `unwrap`. Ми знаємо, що цей метод тут ніколи не призведе до аварійного завершення програми, бо всі методи `Post` влаштовані таким чином, що після їх виконання, в поле `state` завжди міститься значення `Some`. Це один з випадків, про яких ми говорили в розділі "Випадки, коли у вас більше інформації, ніж у компілятора" розділу 9 - випадок, коли ми знаємо, що значення `None` ніколи не зустрінеться, навіть якщо компілятор не може цього зрозуміти.

Тепер, коли ми викликаємо `content` у `&Box<dyn State>`, в дію вступає перетворення під час розіменування (deref coercion) для `&` та `Box`, тому в підсумку метод `content` буде викликаний для типу, який реалізує трейт `State`. Це означає, що нам потрібно додати

метод `content` у визначення трейту `State`, і саме там ми розмістимо логіку для з'ясування того, який вміст повертати, в залежності від поточного стану, як показано в Блоці коду 17-18:

Файл: `src/lib.rs`

```
trait State {  
    // --snip--  
    fn content<'a>(&self, post: &'a Post) -> &'a str {  
        ""  
    }  
}  
  
// --snip--  
struct Published {}  
  
impl State for Published {  
    // --snip--  
    fn content<'a>(&self, post: &'a Post) -> &'a str {  
        &post.content  
    }  
}
```

Блок коду 17-18: Додавання методу `content` в трейт `State`

Ми додаємо реалізацію за замовчуванням метода `content`, який повертає порожній стрічковий слайс. Це означає, що нам не прийдеться реалізовувати `content` в структурах `Draft` та `PendingReview`. Структура `Published` буде перевизначати метод `content` та поверне значення з `post.content`.

Зверніть увагу, що для цього метода нам потрібні анотації часу життя, як ми обговорювали в розділі 10. Ми беремо посилання на `post` як аргумент та повертаємо посилання на частину цього `post`, тому час життя посилання, що повертається, пов'язаний з часом життя аргументу `post`.

І ось, ми закінчили - тепер все у Блоці коду 17-11 працює! Ми реалізували патерн "Стан", який визначає правила процесу роботи з дописом у блозі. Логіка, що пов'язана з цими правилами, знаходиться в об'єктах станів, а не розпорошена по всій структурі `Post`.

Чому не Енум?

Можливо, вам було цікаво, чому ми не використовували `enum` з різними можливими станами допису як варіантів. Це, безумовно, одне з можливих рішень, спробуйте його реалізувати та порівняйте кінцеві результати, щоб обрати, який з варіантів вам подобається більше! Одним з недоліків використання перерахунку є те, що в

кожному місці, де перевіряється його значення, потрібен вираз `match` або щось подібне для обробки всіх можливих варіантів. Можливо в цьому випадку нам доведеться повторювати більше коду, ніж це було в рішенні з трейт-об'єктом.

Компроміси Шаблону "Стан"

Ми показали, що Rust здатен реалізувати об'єктоорієнтований шаблон "Стан" для інкапсуляції різної поведінки публікації в кожному стані. Методи в `Post` нічого не знають про різні види поведінки. З таким способом організації коду, нам достатньо поглянути тільки на один його фрагмент, щоб дізнатися відмінності в поведінці опублікованого допису: в реалізацію трейту `State` у структурі `Published`.

Якби ми збиралися створити альтернативну реалізацію, не використовуючи патерн "Стан", ми могли б використовувати вирази `match` в методах структури `Post` або навіть в коді `main` для перевірки стану допису та зміни його поведінки в цих місцях. Це означало б, що нам би довелося аналізувати декілька фрагментів коду, щоб зрозуміти як себе веде допис в опублікованому стані! Якби ми вирішили додати ще станів, стало б ще гірше: кожному з цих виразів `match` знадобилися б додаткові гілки.

За допомогою патерну "Стан" методи `Post` та ділянки, де ми використовуємо `Post`, не потребують використання виразів `match`, а для додавання нового стану потрібно буде тільки додати нову структуру та реалізувати методи трейту для цієї структури.

Реалізацію з використанням патерну "Стан" легко розширити для додавання нової функціональності. Щоб побачити, як легко підтримувати код, який використовує даний патерн, спробуйте виконати декілька з пропозицій нижче:

- Додайте метод `reject`, який змінює стан публікації з `PendingReview` назад на `Draft`.
- Вимагайте два виклики метода `approve`, спершу ніж переводити стан в `Published`.
- Дозвольте користувачам додавати текстовий вміст тільки тоді, коли публікація знаходиться в стані `Draft`. Порада: нехай об'єкт стану вирішує, чи можна змінювати вміст, але не відповідає за зміну `Post`.

Одним з недоліків патерну "Стан" є те, що оскільки стани самі реалізують переходи між собою, деякі з них виходять пов'язаними один з одним. Якщо ми додамо інший стан між `PendingReview` та `Published`, наприклад `Scheduled` ("заплановано"), то доведеться змінювати код в `PendingReview`, щоб воно тепер переходило в стан `Scheduled`. Якби не потрібно було змінювати `PendingReview` при додаванні нового стану, було б менше роботи, але це означало б, що ми переходимо на інший шаблон проектування.

Іншим недоліком є дублювання деякої логіки. Щоб усунути деяке дублювання, ми могли б спробувати зробити реалізацію за замовчуванням для методів `request_review` та `approve` трейту `State`, які повертають `self`; однак, це б порушило безпечність об'єкта, тому що трейт не знає, яким конкретно буде `self`. Ми хочемо мати можливість використовувати `State` як трейт-об'єкт, тому нам потрібно, щоб його методи були об'єктно-безпечними.

Інше дублювання містять подібні реалізації методів `request_review` та `approve` у `Post`. Обидва методи делегують реалізації одного й того самого методу значенню поля `state` типа `Option` і встановлює результатом нове значення поля `state`. Якби у `Post` було багато методів, що дотримувалися цього шаблону, ми могли б розглянути визначення макроса для усунення повторів (дивись секцію "[Макроси](#)" розділу 19).

Реалізуючи патерн "Стан" таким чином, як він визначений для об'єктноорієнтованих мов, ми не використовуємо переваги Rust на повну. Нумо подивімось на деякі зміни, які ми можемо зробити в крейті `blog`, щоб неприпустимі стани й переходи перетворити в помилки часу компіляції.

Кодування Станів та Поведінки у Вигляді Типів

Ми покажемо вам, як переосмислити патерн "Стан", щоб отримати інший набір компромісів. Замість того, щоб повністю інкапсулювати стани й переходи, таким чином, щоб зовнішній код не знав про них, ми будемо кодувати стани з допомогою різних типів. Отже, система перевірки типів Rust буде перешкоджати спробам використовувати чернетки там, де дозволені тільки опубліковані дописи, викликаючи помилки компіляції.

Розгляньмо першу частину `main` в Блоці коду 17-11:

Файл: `src/main.rs`

```
fn main() {  
    let mut post = Post::new();  
  
    post.add_text("I ate a salad for lunch today");  
    assert_eq!("", post.content());  
}
```

Ми все ще дозволяємо створювати нові дописи у чернетці використовуючи `Post::new` і можливість додавати текст до змісту повідомлення. Але замість метода `content` у чернетці, що повертає порожню стрічку, ми зробимо так, що у чернеток взагалі не буває методу `content`. Таким чином, якщо ми спробуємо отримати вміст чернетки, отримаємо помилку компілятора, що повідомляє про відсутність методу. Як результат ми не зможемо випадково відобразити вміст чернетки допису в програмі, що працює, тому що цей код

навіть не скомпілюється. В Блоці коду 17-19 показано визначення структур `Post` та `DraftPost`, а також методів для кожної з них:

Файл: `src/lib.rs`

```
pub struct Post {
    content: String,
}

pub struct DraftPost {
    content: String,
}

impl Post {
    pub fn new() -> DraftPost {
        DraftPost {
            content: String::new(),
        }
    }

    pub fn content(&self) -> &str {
        &self.content
    }
}

impl DraftPost {
    pub fn add_text(&mut self, text: &str) {
        self.content.push_str(text);
    }
}
```

Блок коду 17-19: Структура `Post` з методом `content` та структура `DraftPost` без методу `content`

Обидві структури `Post` та `DraftPost` мають приватне поле `content`, що зберігає текст допису. Структури більше не мають поля `state`, тому що ми перемістили логіку кодування стану в типи структур. Структура `Post` буде являти собою опублікований допис, і в неї є метод `content`, який повертає `content`.

У нас все ще є функція `Post::new`, але замість повернення екземпляра `Post` вона повертає екземпляр `DraftPost`. Оскільки поле `content` є приватним і немає ніяких функцій, які повертають `Post`, вже не вийде створити екземпляр `Post`.

Структура `DraftPost` має метод `add_text`, тому ми можемо додавати текст до `content` як і раніше, але врахуйте, що в `DraftPost` не визначений метод `content`! Тепер програма гарантує, що всі дописи починаються як чернетки, а чернетки не мають контенту для відображення. Будь-яка спроба подолати ці обмеження призведе до помилки компілятора.

Реалізація Переходів як Трансформації в Інші Типи

Як же нам опублікувати допис? Ми хочемо забезпечити дотримання правила, відповідно якому чернетка допису повинна бути перевірена та схвалена до того, як допис буде опублікований. Допис, що знаходиться в стані очікування перевірки, також не повинен вміти відображати вміст. Реалізуймо ці обмеження, додавши ще одну структуру, `PendingReviewPost`, визначивши метод `request_review` у `DraftPost`, що повертає `PendingReviewPost`, і визначивши метод `approve` у `PendingReviewPost`, що повертає `Post`, як показано в Блоці коду 17-20:

Файл: `src/lib.rs`

```
impl DraftPost {
    // --snip--
    pub fn request_review(self) -> PendingReviewPost {
        PendingReviewPost {
            content: self.content,
        }
    }
}

pub struct PendingReviewPost {
    content: String,
}

impl PendingReviewPost {
    pub fn approve(self) -> Post {
        Post {
            content: self.content,
        }
    }
}
```

Блок коду 17-20: `PendingReviewPost`, що створюється шляхом виклику методу `request_review` екземпляру `DraftPost` і метод `approve`, який перетворює `PendingReviewPost` в опублікований `Post`

Методи `request_review` та `approve` забирають у володіння `self`, таким чином поглинаючи екземпляри `DraftPost` і `PendingReviewPost`, які потім перетворюються в `PendingReviewPost` та опублікований `Post`, відповідно. Таким чином, в нас не буде ніяких довгоживучих екземплярів `DraftPost`, після того, як ми викликали в них `request_review` і так далі. У структурі `PendingReviewPost` не визначений метод `content`, тому спроба прочитати її вміст призводить до помилки компілятора, як і у випадку з `DraftPost`. Тому що єдиним способом отримати опублікований екземпляр `Post`, у якого дійсно є визначений метод `content`, є викликом метода `approve` у екземпляра `PendingReviewPost`, а єдиний спосіб отримати `PendingReviewPost` - це викликати метод

`request_review` в екземпляра `DraftPost`, тобто ми закодували процес зміни станів допису за допомогою системи типів.

Але ми також повинні зробити невеличкі зміни в `main`. Методи `request_review` та `approve` повертають нові екземпляри, а не змінюють структуру, до якої вони звертаються, тому нам потрібно додати більше виразів `let post =`, затіняючи присвоювання для збереження екземплярів, що повертаються. Ми також не можемо використовувати твердження (assertions) для чернетки та допису, який очікує на перевірку, що вміст повинен бути пустим рядком, бо вони нам більше не потрібні: тепер ми не зможемо скомпілювати код, який намагається використовувати вміст дописів, що знаходяться в цих станах. Оновлений код в `main` показано в Блоці коду 17-21:

Файл: `src/main.rs`

```
use blog::Post;

fn main() {
    let mut post = Post::new();

    post.add_text("I ate a salad for lunch today");

    let post = post.request_review();

    let post = post.approve();

    assert_eq!("I ate a salad for lunch today", post.content());
}
```

Блок коду 17-21: Зміни в `main`, які використовують нову реалізацію процесу підготовки допису блогу

Зміни, які нам треба було зробити в `main`, щоб перевизначити `post` означають, що ця реалізація тепер не зовсім відповідає об'єктноорієнтованому патерну "Стан": перетворення між станами більше не інкапсульовані всередині реалізації `Post` повністю. Проте, ми отримали велику вигоду в тому, що неприпустимі стани тепер неможливі завдяки системі типів та їх перевірці, що відбувається під час компіляції! Це гарантує, що деякі помилки, такі як відображення вмісту неопублікованого допису, будуть знайдені ще до того, як вони дійдуть до користувачів.

Спробуйте виконати завдання, які були запропоновані на початку цього розділу, у версії крейта `blog`, яким він став після Блока коду 17-20, щоб сформулювати свою думку про дизайн цієї версії коду. Зверніть увагу, що деякі інші завдання в цьому варіанті вже можуть бути виконані.

Ми побачили, що хоча Rust і здатен реалізувати об'єктноорієнтовані шаблони проектування, в ньому також доступні й інші шаблони, такі як кодування стану за

допомогою системи типів. Ці шаблони мають різні компроміси. Хоча ви, можливо, дуже добре знайомі з об'єктноорієнтованими патернами, переосмислення проблем для використання переваг і можливостей Rust може дати такі вигоди, як запобігання деяких помилок під час компіляції. Об'єктноорієнтовані патерни не завжди будуть найкращим рішенням в Rust через наявність деяких можливостей, таких як володіння, якого немає в об'єктноорієнтованих мов.

Підсумок

Незалежно від того, вважаєте ви Rust об'єктноорієнтованою мовою чи ні, прочитавши цей розділ, ви тепер знаєте, що можна використовувати трейт-об'єкти для впровадження деяких об'єктноорієнтованих можливостей у Rust. Динамічна диспетчеризація може дати коду певну гнучкість в обмін на невеличке погіршення швидкодії програми під час виконання. Ви можете використовувати цю гнучкість для реалізації об'єктноорієнтованих патернів, які можуть покращити супроводжуваність вашого коду. Rust також має особливості, такі як власність, які не мають об'єктноорієнтовані мови. Об'єктноорієнтовані патерни не завжди будуть найкращим способом скористатися перевагами Rust, але є доступною опцією.

Далі ми розглянемо патерни, які є ще однією особливістю мови Rust, що дає більше гнучкості. Ми трохи зустрічалися з ними впродовж всієї книги, але все ще не проаналізували всі їх можливості. Вперед до нових можливостей!

Шаблони та Зіставлення Шаблонів

Шаблони це спеціальний синтаксис в Rust для порівняння з певною структурою типів, складною чи простою. Використання шаблонів у поєднанні з виразами `match` та іншими конструкціями дає нам більше контролю над нашою програмою. Шаблон складається з певної комбінації наступного:

- Літералів
- Деструктуризованих масивів, еnumів, структур або кортежів
- Змінних
- Символів узагальнення
- Заповнювачів

`x`, `(a, 3)`, та `Some(Color::Red)` це декілька прикладів шаблонів. У контекстах, у яких шаблони дійсні, ці компоненти описують форму даних. Наші програми потім зіставляють значення з шаблонами для визначення коректності форми даних та продовження виконання коду.

Щоб використати шаблон, ми порівнюємо його з якимось значенням. Ми використовуємо частини значення в нашому коді, якщо значенню зіставляється зі шаблоном. Згадайте вирази `match` в Розділі 6, такі як `coin-sorting machine example`(скопювати переклад з шостого розділу), які використовували шаблони. Якщо значення підходить формі шаблону, то ми можемо найменувати та використати певні частини цього значення. Якщо не підходить, то пов'язаний з шаблоном код не виконається.

Цей розділ є довідником для всього пов'язаного з шаблонами. Ми розглянемо місця де можна використовувати шаблони, різницю між спростовними і неспростовними шаблонами та різновиди синтаксису шаблонів, які ви можете побачити. Наприкінці цього розділу ви будете знати як використовувати шаблони для вираження багатьох концептів чітким способом.

Усі Місця Можливого Використання Шаблонів

Шаблони з'являються в багатьох місцях в Rust, і ви ними багато користуєтеся навіть не підозрюючи про це! В цьому розділі ми розглянемо всі місця, де допускаються шаблони.

Рукави Виразу `match`

Як обговорювалося в Розділі 6, ми використовуємо шаблони в рукавах виразів `match`. Формально, вирази `match` визначені як ключове слово `match`, значення яке буде зіставлятися та один або більше рукавів зіставлення, що складаються з шаблону та виразу для виконання, якщо значення зіставляється зі шаблоном рукава, як тут:

```
match VALUE {  
    PATTERN => EXPRESSION,  
    PATTERN => EXPRESSION,  
    PATTERN => EXPRESSION,  
}
```

For example, here's the `match` expression from Listing 6-5 that matches on an `Option<i32>` value in the variable `x`:

```
match x {  
    None => None,  
    Some(i) => Some(i + 1),  
}
```

Шаблони в цьому виразі `match` це `None` та `Some(i)` ліворуч від кожної стрілки.

Одна з вимог виразів `match` це необхідність бути *вичерпним* у сенсі, що всі можливі значення виразу `match` повинні бути враховані. Один зі способів переконатися, що ви охопили всі можливі варіанти, - це мати загальний шаблон для останнього рукава: наприклад, назва змінної, що збігається з будь-яким значенням, не може не спрацювати і, таким чином, охоплює всі варіанти, що залишилися.

Зокрема, шаблон `_` буде відповідати будь-чому, але він ніколи не зв'язується зі змінною, тому його часто використовують в останньому рукаві виразу `match`. Шаблон `_` може бути корисним, наприклад, коли потрібно ігнорувати будь-яке не вказане значення. Ми розглянемо шаблон `_` більш детально в секції "Ігнорування Значень в Шаблоні" пізніше в цьому розділі.

Умовні Вирази `if let`

В Розділі 6 ми обговорювали використання виразів `if let` в основному як рівнозначний та коротший спосіб написання `match`, який лише зіставляється в одному випадку. При необхідності, `if let` може мати відповідний `else`, що містить код для виконання на випадок невідповідності шаблону в `if let`.

В Блоці Коду 18-1 показано, що також можливо поєднувати вирази `if let`, `else if`, та `else if let`. Це надає нам більшу гнучкість, ніж вираз `match`, в якому ми можемо представити тільки одне значення для порівняння з шаблонами. Також Rust не вимагає, щоб умови в послідовності `if let`, `else if`, `else if let` стосувалися одна одної.

Код у Блоці Коду 18-1 визначає, яким кольором зробити ваш фон, виходячи з низки перевірок за кількома умовами. Для цього прикладу ми створили змінні з жорстко заданими значеннями, які справжня програма може отримати з вхідних даних користувача.

Файл: `src/main.rs`

```
fn main() {
    let favorite_color: Option<&str> = None;
    let is_tuesday = false;
    let age: Result<u8, _> = "34".parse();

    if let Some(color) = favorite_color {
        println!("Using your favorite color, {color}, as the background");
    } else if is_tuesday {
        println!("Tuesday is green day!");
    } else if let Ok(age) = age {
        if age > 30 {
            println!("Using purple as the background color");
        } else {
            println!("Using orange as the background color");
        }
    } else {
        println!("Using blue as the background color");
    }
}
```

Блок Коду 18-1: Змішування `if let`, `else if`, `else if let`, та `else`

Якщо користувач вказує улюблений колір, цей колір використовується як фоновий. Якщо улюблений колір не вказано і сьогодні Вівторок, то фоновим кольором буде зелений. Інакше, якщо користувач вказує свій вік стрічкою і ми можемо успішно розібрати її як число, то колір буде фіолетовим або помаранчевим в залежності від значення числа. Якщо жодна з цих умов не застосовується, колір фону буде синім.

Ця умовна структура дозволяє нам підтримувати складні вимоги. З жорстко заданими значеннями як ми маємо тут, цей приклад виведе в консолі **Використовую фіолетовий колір як колір фону**.

Ви можете побачити, що **if let** також може впроваджувати затінені змінні аналогічним чином що і рукави **match**: рядок **if let Ok(age) = age** запроваджує нову затінену змінну **age**, яка містить значення всередині **Ok**. Це означає, що нам потрібно помістити умову **if age > 30** в цей блок: ми не можемо об'єднати ці дві умови в **if let Ok(age) = age && age > 30**. Значення затіненої змінної **age**, яку ми хочемо порівняти з 30, не дійсне до тих пір, поки не почнеться новий діапазон з фігурної дужки.

Недоліком використання виразів **if let** є те, що компілятор не перевіряє вичерпність, тоді як при використанні виразів **match** він це робить. Якби ми пропустили останній блок **else** і, відповідно, пропустили обробку деяких випадків, компілятор не попередив би нас про можливу логічну помилку.

Умовні Цикли **while let**

Подібно до конструкції **if let**, умовний цикл **while let** дозволяє циклу **while let** працювати допоки шаблон продовжує збігатися. У Блоці Коду наведено код циклу **while let**, який використовує вектор як стек і виводить в консолі значення вектора у зворотному порядку, в якому вони були додані.

```
let mut stack = Vec::new();

stack.push(1);
stack.push(2);
stack.push(3);

while let Some(top) = stack.pop() {
    println!("{}", top);
}
```

Блок Коду 18-2: Використання циклу **while let** для виводу значень в консоль, допоки **stack.pop()** повертає **Some**

Цей приклад виводить в консолі 3, 2, і потім 1. Метод **pop** бере останній елемент з вектора і повертає **Some(значення)**. Якщо вектор порожній, **pop** поверне **None**. Цикл **while** продовжує виконання коду в своєму блоці допоки **pop** повертає **Some**. Коли **pop** поверне **None**, цикл зупиниться. Ми можемо використовувати **while let** для вилучення кожного елементу зі стека.

Цикли `for`

В циклі `for`, значення яке безпосередньо слідує за ключовим словом `for` є шаблоном. Наприклад, `x` в `for x in y` є шаблоном. Блок Коду 18-3 демонструє як використовувати шаблон в циклі `for` для деструктуризації або розбирання на частини кортежу, як частини циклу `for`.

```
let v = vec!['a', 'b', 'c'];

for (index, value) in v.iter().enumerate() {
    println!("{}", value, index);
}
```

Блок Коду 18-3: Використання шаблону в циклі `for` для деструктуризування кортежу

Код в Блоці Коду 18-3 виведе в консоль наступне:

```
$ cargo run
Compiling patterns v0.1.0 (file:///projects/patterns)
Finished dev [unoptimized + debuginfo] target(s) in 0.52s
Running `target/debug/patterns`
a is at index 0
b is at index 1
c is at index 2
```

Ми адаптуємо ітератор використовуючи метод `enumerate` таким чином, щоб він генерував значення та індекс для цього значення, поміщені в кортеж. Перше згенероване значення - кортеж `(0, 'a')`. При зіставленні цього значення з шаблоном `(index, value)`, `index` буде `0`, а `value` буде `'a'`, виводячи перший рядок виводу в консоль.

Інструкції `let`

До цього розділу ми явно обговорювали тільки використання шаблонів з `match` та `if` `let`, але насправді ми використовували шаблони і в інших місцях, в тому числі і в операторах `let`. Наприклад, розглянемо це просте присвоювання змінної з використанням `let`:

```
let x = 5;
```

Кожного разу, коли ви використовували інструкцію `let`, ви використовували шаблони, хоча, можливо, ви цього навіть не усвідомлювали! Більш формально, оператор `let` виглядає так:

```
let PATTERN = EXPRESSION;
```

В інструкціях типу `let x = 5;` з назвою змінної в слоті `PATTERN` назва змінної є лише особливо простою формою шаблону. Rust порівнює вираз із шаблоном і призначає будь-які знайдені імена. Таким чином, у прикладі `let x = 5;` `x` - це шаблон, який означає "прив'язати до змінної `x` все, що зіставляється з цим виразом." Оскільки назва `x` - це весь шаблон, цей шаблон фактично означає "прив'язати все до змінної `x`, незалежно від її значення."

To see the pattern matching aspect of `let` more clearly, consider Listing 18-4, which uses a pattern with `let` to destructure a tuple.

```
let (x, y, z) = (1, 2, 3);
```

Блок Коду 18-4: Використання шаблону для деструктуризації кортежу та створення трьох змінних одразу

Тут ми зіставляємо кортеж з шаблоном. Rust зіставляє значення `(1, 2, 3)` із шаблоном `(x, y, z)` та бачить, що значення зіставляються, тому Rust пов'язує `1` до `x`, `2` до `y` та `3` до `z`. Ви можете думати про цей шаблон кортежу як про три окремих вкладених шаблонів змінних всередині.

Якщо кількість елементів у шаблоні не відповідає кількості елементів у кортежі, то загальний тип не буде збігатися і ми отримаємо помилку компілятора. Наприклад, у Блоці Коду 18-5 показано спробу деструктуризації кортежу з трьома елементами на дві змінні, що не спрацює.

```
let (x, y) = (1, 2, 3);
```

Блок Коду 18-5: Неправильна побудова шаблону, змінні якого не відповідають кількості елементів у кортежі

Спроба скомпілювати цей код призведе до помилки цього типу:

```
$ cargo run
  Compiling patterns v0.1.0 (file:///projects/patterns)
error[E0308]: mismatched types
  --> src/main.rs:2:9
   |
2 |     let (x, y) = (1, 2, 3);
   |           ^^^^^^  ----- this expression has type `({integer}, {integer}, {integer})`
   |           |
   |           expected a tuple with 3 elements, found one with 2 elements
   |
   = note: expected tuple `({integer}, {integer}, {integer})`
           found tuple `(_, _)`

For more information about this error, try `rustc --explain E0308`.
error: could not compile `patterns` due to previous error
```

Щоб виправити помилку, ми можемо проігнорувати один або більше значень кортежу, використовуючи `_` або `..`, як ви побачите в секції "Ігнорування Значень в Шаблоні". Якщо проблема в тому, що в шаблоні занадто багато змінних, то рішення полягає в узгодженні типів шляхом видалення змінних так, щоб кількість змінних дорівнювала кількості елементів в кортежі.

Параметри Функції

Параметри функції також можуть бути шаблонами. Код у Блоці Коду 18-6, який оголошує функцію з назвою `foo`, яка отримує один параметр з назвою `x` типу `i32`, вже повинен виглядати знайомим.

```
fn foo(x: i32) {
    // code goes here
}
```

Блок Коду 18-6: Сигнатура функції використовує шаблони в параметрах

Частина `x` - це шаблон! Ми можемо зіставляти кортеж в аргументах функції із шаблоном, як ми зробили з `let`. В Блоці Коду 18-7 значення кортежу розділяються, коли ми передаємо їх до функції.

Файл: src/main.rs

```
fn print_coordinates(&(x, y): &(i32, i32)) {  
    println!("Current location: ({} , {})", x, y);  
}  
  
fn main() {  
    let point = (3, 5);  
    print_coordinates(&point);  
}
```

Блок Коду 18-7: Функція з параметрами, які розкладають кортеж

Цей код виводить в консоль `Current location: (3, 5)`. Значення `&(3, 5)` зіставляються з шаблоном `&(x, y)`, тому `x` має значення `3` та `y` має значення `5`.

Ми також можемо використовувати шаблони в списках параметрів замикань, так само як в списках параметрів функцій, оскільки замикання схожі до функцій, як обговорювалося в Розділі 13.

Наразі ви побачили декілька способів використання шаблонів, але вони не працюють однаково у всіх місцях можливого використання. У деяких місцях шаблони мають бути неспростовні; в інших умовах вони можуть бути спростовними. Ми обговоримо ці дві концепції далі. [ch18-03-pattern-syntax.html#ignoring-values-in-a-pattern](https://pavloslav.github.io/RustBookUkr/print.html#ignoring-values-in-a-pattern)

Спростовуваність: Чи Може Шаблон Бути Невідповідним

Шаблони бувають двох видів: спростовні та неспростовні. Шаблони, які збігаються з будь-яким можливим переданим значенням, є *неспростовними*. Прикладом може бути `x` в інструкції `let x = 5;` тому що `x` збігається з будь-яким значенням і тому не може не збігатися. Шаблони, які можуть не збігатися з деякими можливими значеннями, є *спростовними*. Прикладом може бути `Some(x)` у виразі `if let Some(x) = a_value`, тому що якщо значення змінної `a_value` буде `None`, а не `Some`, то шаблон `Some(x)` не буде зіставлятися.

Параметри функцій, інструкції `let` і цикли `for` можуть приймати тільки неспростовні шаблони, тому що програма не може зробити нічого путнього, коли значення не збігаються. Вирази `if let` і `while let` допускають спростовні і неспростовні шаблони, але компілятор застерігає від неспростовних шаблонів, оскільки за визначенням вони призначені для обробки можливих збоїв: функціональність умови полягає в її здатності працювати по-різному в залежності від успіху або невдачі.

В цілому, ви не повинні турбуватися про різницю між спростовуваними і неспростовуваними шаблонами; однак, ви повинні бути знайомі з концепцією спростовуваності, щоб мати можливість реагувати, коли ви бачите її в повідомленні про помилку. У таких випадках вам потрібно буде змінити або шаблон, або конструкцію, в якій ви використовуєте шаблон, в залежності від бажаної поведінки коду.

Розгляньмо приклад того, що відбувається, коли ми намагаємося використати спростовуваний шаблон там, де Rust вимагає неспростовний шаблон і навпаки. У Блоці Коду 18-8 показано інструкцію `let`, але для шаблону ми вказали `Some(x)`, спростовуваний шаблон. Як і слід було очікувати, цей код не буде компілюватися.

```
let Some(x) = some_option_value;
```

Блок Коду 18-8: Спроба використання спростовуваного шаблону з `let`

Якщо значення `some_option_value` було б значенням `None`, то воно не відповідало б шаблону `Some(x)`, що означає, що шаблон є спростовуваним. Однак, інструкція `let` може приймати тільки неспростовний шаблон, тому що код не може зробити нічого коректного зі значенням `None`. Під час компіляції Rust поскаржиться, що ми намагалися використати спростовуваний шаблон там, де потрібен неспростовний шаблон:

```
$ cargo run
  Compiling patterns v0.1.0 (file:///projects/patterns)
error[E0005]: refutable pattern in local binding: `None` not covered
--> src/main.rs:3:9
   |
3  |     let Some(x) = some_option_value;
   |         ^^^^^^^ pattern `None` not covered
   |
= note: `let` bindings require an "irrefutable pattern", like a `struct` or an
`enum` with only one variant
= note: for more information, visit https://doc.rust-lang.org/book/ch18-02-
refutability.html
note: `Option<i32>` defined here
= note: the matched value is of type `Option<i32>`
help: you might want to use `if let` to ignore the variant that isn't matched
   |
3  |     let x = if let Some(x) = some_option_value { x } else { todo!() };
   |         ++++++                               ++++++

For more information about this error, try `rustc --explain E0005`.
error: could not compile `patterns` due to previous error
```

Оскільки ми не покрили (і не могли покрити!) кожне допустиме значення шаблоном `Some(x)`, Rust справедливо видає помилку компілятора.

Якщо у нас є спростовний шаблон там, де потрібен неспростовний, ми можемо виправити це, змінивши код, який використовує шаблон: замість використання `let`, ми можемо використати `if let`. Тоді, якщо шаблон не збігається, код просто пропустить код у фігурних дужках, даючи йому можливість продовжити правильне виконання коду. У Блоці Коду 18-9 показано, як виправити код у Блоці Коду 18-8.

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

Блок Коду 18-9: Використання `if let` і блоку зі спростовуваними шаблонами замість `let`

Ми дали коду вихід! Цей код абсолютно дійсний, хоча це означає, що ми не можемо використовувати неспростовний шаблон без отримання помилки. Якщо ми дамо `if let` шаблон, який завжди збігатиметься, наприклад, `x`, як показано у Блоці Коду 18-10, компілятор видає попередження.

```
if let x = 5 {
    println!("{}", x);
};
```

Блок Коду 18-10: Спроба використати неспростовний шаблон з `if let`

Rust скаржитися, що немає сенсу використовувати `if let` з неспростовним шаблоном:

```
$ cargo run
  Compiling patterns v0.1.0 (file:///projects/patterns)
warning: irrefutable `if let` pattern
--> src/main.rs:2:8
   |
2  |     if let x = 5 {
   |         ^^^^^^^
= note: `[warn(irrefutable_let_patterns)]` on by default
= note: this pattern will always match, so the `if let` is useless
= help: consider replacing the `if let` with a `let`

warning: `patterns` (bin "patterns") generated 1 warning
  Finished dev [unoptimized + debuginfo] target(s) in 0.39s
  Running `target/debug/patterns`
5
```

З цієї причини рукави збігів повинні використовувати спростовувані шаблони, за винятком останнього плеча, яке повинно збігатися з будь-якими значеннями, що залишилися, з неспростовним шаблоном. Rust дозволяє використовувати неспростовний шаблон у `match` лише з одним рукавом, але цей синтаксис не є особливо корисним і може бути замінений простішим оператором `let`.

Тепер, коли ви знаєте, де використовувати шаблони і в чому різниця між спростовуваними і не спростовуваними шаблонами, розгляньмо весь синтаксис, який можна використовувати для створення шаблонів.

Синтаксис Шаблонів

У цьому розділі ми збираємо всі синтаксичні конструкції, що використовуються в шаблонах, і обговорюємо, чому і коли ви можете захотіти використовувати кожен з них.

Зіставлення з Літералами

Як ви бачили у Розділі 6, можна зіставляти шаблони з літералами напряму. Наведемо декілька прикладів в наступному коді:

```
let x = 1;

match x {
    1 => println!("one"),
    2 => println!("two"),
    3 => println!("three"),
    _ => println!("anything"),
}
```

Цей код виведе в консолі `one`, оскільки значення в `x` дорівнює 1. Цей синтаксис корисний, коли ви хочете, щоб ваш код виконував дію, якщо він отримує певне значення.

Зіставлення з Іменованими Змінними

Іменовані змінні - це незаперечні шаблони, які відповідають будь-якому значенню, і ми багато разів використовували їх у книзі. Однак, існує ускладнення при використанні іменованих змінних у виразах `match`. Оскільки `match` починає нову область видимості, змінні, оголошені як частина шаблону всередині виразу `match`, будуть затінювати змінні з тією ж назвою за межами конструкції `match`, як і у випадку з усіма змінними. У Блоці Коду 18-11 оголошується змінна з назвою `x` зі значенням `Some(5)` та змінна `y` зі значенням `10`. Потім ми створюємо вираз `match` над значенням `x`. Подивіться на шаблони в рукавах `match` і `println!` наприкінці, і перед тим, як запускати цей код або читати далі, спробуйте з'ясувати, що виведе код в консолі.

Файл: `src/main.rs`

```
let x = Some(5);
let y = 10;

match x {
    Some(50) => println!("Got 50"),
    Some(y) => println!("Matched, y = {y}"),
    _ => println!("Default case, x = {:?}", x),
}

println!("at the end: x = {:?}", y = {y}", x);
```

Блок Коду 18-11: Вираз `match` з рукавом, що вводить затінену змінну `y`

Розглянемо, що відбувається при виконанні виразу `match`. Шаблон у першому рукаві порівняння не збігається із заданим значенням `x`, тому код продовжується.

Шаблон у другому рукаві порівняння вводить нову змінну з назвою `y`, яка буде відповідати будь-якому значенню всередині значення `Some`. Оскільки ми знаходимося в новій області видимості всередині виразу `match`, це нова змінна `y`, а не та `y`, яку ми оголосили на початку зі значенням 10. Ця нова прив'язка `y` буде відповідати будь-якому значенню всередині `Some`, яке ми маємо в `x`. Таким чином, ця нова `y` зв'язується з внутрішнім значенням `Some` в `x`. Це значення `5`, тому вираз для цього рукава виконується і виводить в консолі `Matched, y = 5`.

Якби значення `x` було б `None` замість `Some(5)`, шаблони в перших двох рукавах не збіглися б, тому значення збіглося б з підкресленням. Ми не створювали змінну `x` у шаблоні підкреслення, тому `x` у виразі - це все ще зовнішній `x`, який не був затінений. У цьому гіпотетичному випадку `match` виведе в консолі `Default case, x = None`.

Коли вираз `match` виконано, його область видимості закінчується, так само як і область видимості внутрішньої `y`. Останній `println!` виведе в консолі `at the end: x = Some(5), y = 10`.

Щоб створити вираз `match`, який порівнює значення зовнішніх `x` і `y`, замість того, щоб вводити затінену змінну, нам потрібно буде використовувати умовний запобіжник. Ми поговоримо про запобіжники пізніше в розділі "Додаткові умови з запобіжниками".

Декілька Шаблонів

У виразах `match` ви можете зіставляти кілька шаблонів, використовуючи синтаксис `|`, який є оператором шаблону *or*. Наприклад, у наступному коді ми порівнюємо значення `x` з рукавами `match`, перше з яких має опцію *or*, що означає, що якщо значення `x` збігається з будь-яким зі значень у цьому рукаві, код цього рукава буде виконано:

```
let x = 1;

match x {
    1 | 2 => println!("one or two"),
    3 => println!("three"),
    _ => println!("anything"),
}
```

Цей код виведе в консоль `one or two`.

Зіставлення Діапазонів Значень з `..=`

Синтаксис `..=` дозволяє робити інклюзивне зіставлення, зіставлення з діапазоном включно з останнім його значенням. В наступному коді буде виконана гілка, шаблон якої зіставляється з будь-яким значенням заданого діапазону:

```
let x = 5;

match x {
    1..=5 => println!("one through five"),
    _ => println!("something else"),
}
```

Якщо `x` дорівнює 1, 2, 3, 4, або 5, то буде обрана перша гілка виразу `match`. Цей синтаксис більш зручний для зіставлення декількох значень ніж використання оператора `|` для вираження тої самої ідеї; якщо ми використовували б `|`, нам було б потрібно вказати `1 | 2 | 3 | 4 | 5`. Вказання діапазону набагато коротше, особливо якщо ми хочемо зіставляти, скажімо, будь-яке число між 1 та 1,000!

Компілятор перевіряє, що діапазон не порожній під час компіляції, і оскільки єдиними типами, для яких Rust може сказати, чи є діапазон порожнім чи ні, є `char` та числові значення, діапазони дозволяють лише числові або `char` значення.

Ось приклад використання діапазонів значень `char`:

```
let x = 'c';

match x {
    'a'..='j' => println!("early ASCII letter"),
    'k'..='z' => println!("late ASCII letter"),
    _ => println!("something else"),
}
```

Rust може визначити, що `'c'` в першому діапазоні шаблону та виведе в консоль `early ASCII letter`.

Деструктурування для Розбору Значень на Частини

Ми також використовуємо шаблони для деструктуризації структур, еnumів та кортежів для використання різних частин їх значень. Розглянемо покроково кожне значення.

Деструктурування Структур

У Блоці Коду 18-12 показана структура `Point` з двома полями `x` і `y`, яку ми можемо розбити на частини за допомогою шаблону з інструкцією `let`.

Файл: `src/main.rs`

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

fn main() {
    let p = Point { x: 0, y: 7 };

    let Point { x: a, y: b } = p;
    assert_eq!(0, a);
    assert_eq!(7, b);
}
```

Блок Коду 18-12: Деструктуризація полів структури в окремі змінні

В цьому коді створюються змінні `a` та `b`, які відповідають значенням полів `x` та `y` структури `p`. Цей приклад показує, що назви змінних у шаблоні не обов'язково повинні збігатися з назвами полів структури. Однак, зазвичай назви змінних збігаються з назвами полів, щоб полегшити запам'ятовування того, які змінні походять з яких полів. Через таке поширене використання, а також через те, що запис `let Point { x: x, y: y } = p;` містить багато повторень, Rust має скорочення для шаблонів, які відповідають полям struct: вам потрібно лише перерахувати назву поля struct, і змінні, створені на основі шаблону, матимуть ті ж самі назви. Блок Коду 18-13 працює так само як і Блок Коду 18-12, але змінні, що створюються в шаблоні `let`, є `x` і `y` замість `a` і `b`.

Файл: `src/main.rs`

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

fn main() {
    let p = Point { x: 0, y: 7 };

    let Point { x, y } = p;
    assert_eq!(0, x);
    assert_eq!(7, y);
}
```

Блок коду 18-13: Деструктуризація полів структури за допомогою скорочення

Цей код створить змінні `x` та `y`, які відповідають полям `x` та `y` змінної `p`. В результаті змінні `x` та `y` містять значення зі структури `p`.

Ми також можемо деструктурувати за допомогою буквених значень як частини шаблону `struct` замість того, щоб створювати змінні для всіх полів. Це дозволяє нам перевіряти деякі з полів на наявність певних значень, створюючи змінні для деструктуризації інших полів.

У Блоці Коду 18-14 ми маємо вираз `match`, який розділяє значення `Point` на три випадки: точки, які лежать безпосередньо на осі `x` (що вірно, коли `y = 0`), на осі `y` (`x = 0`), або не лежать ні на одній з них.

Файл: `src/main.rs`

```
fn main() {
    let p = Point { x: 0, y: 7 };

    match p {
        Point { x, y: 0 } => println!("On the x axis at {x}"),
        Point { x: 0, y } => println!("On the y axis at {y}"),
        Point { x, y } => {
            println!("On neither axis: ({x}, {y})");
        }
    }
}
```

Блок Коду 18-14: Деструктуризація та зіставлення буквених значень в одному шаблоні

Перший рукав буде відповідати будь-якій точці, що лежить на осі `x`, вказуючи, що поле `y` збігається, якщо його значення збігається з `0`. Шаблон все ще створює змінну `x`, яку ми можемо використовувати в коді для цього рукава.

Аналогічно, другий рукав зіставляє будь-яку точку на осі `y`, вказуючи, що поле `x` збігається, якщо його значення дорівнює `0`, і створює змінну `y` для значення поля `y`. Третій рукав не визначає ніяких літералів, тому воно відповідає будь-якій іншій `Point` і створює змінні для полів `x` і `y`.

У цьому прикладі значення `p` збігається з другим рукавом, оскільки `x` містить `0`, тому цей код виведе в консолі `On the y axis at 7`.

Пам'ятайте, що вираз `match` припиняє перевірку рукавів після того, як знайде перший збіг, тому навіть якщо `Point { x: 0, y: 0}` знаходиться як на осі `x`, так і на осі `y`, цей код виведе в консолі `On the x axis at 0`.

Деструктурування Енумів

У цій книзі ми вже деструктурували енуми (наприклад, в Блоці Коду 6-5 Розділу 6), але ми ще окремо не обговорювали, що шаблон деструктурування енума повинен відповідати тому, як визначаються збережені в енумі дані. Як приклад, у Блоці Коду 18-15 ми використовуємо еnum `Message` з Блоку Коду 6-2 і пишемо `match` з шаблонами, які деструктуруватимуть кожне внутрішнє значення.

Файл: `src/main.rs`

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

fn main() {
    let msg = Message::ChangeColor(0, 160, 255);

    match msg {
        Message::Quit => {
            println!("The Quit variant has no data to destructure.");
        }
        Message::Move { x, y } => {
            println!("Move in the x direction {x} and in the y direction {y}");
        }
        Message::Write(text) => {
            println!("Text message: {text}");
        }
        Message::ChangeColor(r, g, b) => {
            println!("Change the color to red {r}, green {g}, and blue {b}",)
        }
    }
}
```

Блок Коду 18-15: Деструктурування варіантів енума, що містять різні види значень

Цей код виведе в консолі `Change the color to red 0, green 160, and blue 255`. Спробуйте змінити значення `msg`, щоб побачити виконання коду з інших рукавів.

Для варіантів енума без даних, таких як `Message::Quit`, ми не можемо деструктурувати значення далі. Ми тільки можемо зіставити буквально значення `Message::Quit`, і жодних змінних у цьому шаблоні немає.

Для структуро-подібних варіантів енума, таких як `Message::Move`, ми можемо використовувати шаблон схожий з тим, що ми вказували для зіставлення структур. Після назви варіанту ми ставимо фігурні дужки, а потім перелічуємо поля зі змінними, щоб розбити все на частини, які будуть використані в коді для цього рукава. Тут ми використовуємо скорочену форму, як ми це робили в Блоці Коду 18-13.

Шаблони кортежо-подібних варіантів енума, таких як `Message::Write`, що містить кортеж з одним елементом, і `Message::ChangeColor`, що містить кортеж з трьома елементами подібні до шаблону, який ми вказуємо для зіставлення кортежів. Кількість змінних у шаблоні повинна відповідати кількості елементів у варіанті, який ми порівнюємо.

Деструктурування Вкладених Структур та Енумів

Дотепер всі наші приклади стосувалися зіставлення структур або енумів глибиною в один рівень, але зіставлення може працювати і на вкладених елементах! Наприклад, ми можемо переробити код у Блоці Коду 18-15 для додавання підтримки RGB та HSV кольорів у повідомленні `ChangeColor`, як показано у Блоці Коду 18-16.

```

enum Color {
    Rgb(i32, i32, i32),
    Hsv(i32, i32, i32),
}

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

fn main() {
    let msg = Message::ChangeColor(Color::Hsv(0, 160, 255));

    match msg {
        Message::ChangeColor(Color::Rgb(r, g, b)) => {
            println!("Change color to red {r}, green {g}, and blue {b}");
        }
        Message::ChangeColor(Color::Hsv(h, s, v)) => {
            println!("Change color to hue {h}, saturation {s}, value {v}")
        }
        _ => (),
    }
}

```

Блок Коду 18-16: Зіставлення з вкладеними енумами

Шаблон першого рукава у виразі `match` відповідає варіанту енуму `Message::ChangeColor`, який містить варіант `Color::Rgb`; потім шаблон зв'язується з трьома внутрішніми значеннями `i32`. Шаблон другого рукава також відповідає варіанту енуму `Message::ChangeColor`, але внутрішній ену́м замість цього збігається з `Color::Hsv`. Ми можемо вказувати такі складні умови в одному виразі `match`, навіть якщо залучені два ену́ми.

Деструктурування Структур та Кортежів

Ми можемо змішувати, зіставляти та вкладати деструктуризуючі шаблони і складнішими способами. В наступному прикладі показано складна деструктуризація, де ми вкладаємо структури та кортежі в кортеж та деструктуризуємо все примітивні значення:

```
let ((feet, inches), Point { x, y }) = ((3, 10), Point { x: 3, y: -10 });
```

Цей код дозволяє нам розбивати складні типи на складові частини, щоб ми могли окремо використовувати потрібні нам значення.

Деструктуризація за допомогою шаблонів є зручним способом використання фрагментів значень, таких як значення з кожного поля в структурі, окремо один від одного.

Ігнорування Значень Шаблону

Ви бачили, що іноді корисно ігнорувати значення в шаблоні, наприклад в останньому рукаві `match`, щоб отримати загальний шаблон, який не робить деструктуризації, але враховує всі можливі значення, що залишилися. Існує декілька способів ігнорувати цілі значення або частини значень у шаблоні: використання шаблону `_` (який ви вже бачили), використання шаблону `_` всередині іншого шаблону, використання імені, яке починається з символу підкреслення, або використання `..` для ігнорування решти частини значення. Розглянемо, як і навіщо використовувати кожен з цих шаблонів.

Ігнорування Всього Значення з `_`

Ми використали символ підкреслення як загальний шаблон, який буде відповідати будь-якому значенню, але не прив'язуватиметься до нього. Це особливо корисно як останній рукав виразу `match`, але ми також можемо використовувати його в будь-якому шаблоні, включно з параметрами функцій, як показано в Блоці Коду 18-17.

Файл: `src/main.rs`

```
fn foo(_: i32, y: i32) {  
    println!("This code only uses the y parameter: {}", y);  
}  
  
fn main() {  
    foo(3, 4);  
}
```

Блок Коду 18-17: Використання `_` в сигнатурі функції

Цей код повністю проігнорує значення `3`, передане першим аргументом, і виведе `Даний код використовує тільки у параметр: 4`.

У більшості випадків, коли вам більше не потрібен певний параметр функції, ви змінили б підпис так, щоб він не включав невикористаний параметр. Ігнорування параметру функції може бути особливо корисним у випадках, коли, наприклад, ви реалізуєте трейт і вам потрібна сигнатура певного типу, але тіло функції у вашій реалізації не потребує жодного з параметрів. Ви тоді уникаєте попередження компілятора про невикористані параметри функції, яке було б у випадку використання назви.

Ігнорування Частин Значення з Вкладеним

Ми також можемо використовувати `_` всередині іншого шаблону, щоб ігнорувати тільки частину значення, наприклад, коли ми хочемо перевірити тільки частину значення, але не використовуємо інші частини у відповідному коді, який ми хочемо виконати. У Блоці Коду 18-18 наведено код що відповідає за керування значенням налаштувань. Бізнес-вимоги полягають в тому, що користувачеві не повинно бути дозволено перезаписувати існуюче задане налаштування, але користувач може скасувати налаштування та надати йому значення, якщо воно наразі не задане.

```
let mut setting_value = Some(5);
let new_setting_value = Some(10);

match (setting_value, new_setting_value) {
    (Some(_), Some(_)) => {
        println!("Can't overwrite an existing customized value");
    }
    _ => {
        setting_value = new_setting_value;
    }
}

println!("setting is {:?}", setting_value);
```

Блок Коду 18-18: Використання підкреслення всередині шаблонів, які відповідають варіантам `Some`, коли нам не потрібно використовувати значення всередині `Some` варіанту

Цей код виведе `Неможливо перезаписати існуюче користувацьке значення`, а потім `налаштування це Some(5)`. У першому рукаві `match` нам не потрібно зіставляти або використовувати значення всередині будь-якого з варіантів `Some`, але нам потрібно перевірити випадок, коли `setting_value` і `new_setting_value` є варіантом `Some`. У цьому випадку ми виводимо причину незмінності `setting_value`, і воно не зміниться.

У всіх інших випадках (якщо або `setting_value`, або `new_setting_value` є `None`), виражених шаблоном `_` у другому плечі, ми хочемо дозволити `new_setting_value` стати `setting_value`.

Ми також можемо використовувати підкреслення в декількох місцях в межах одного шаблону, щоб ігнорувати певні значення. У Блоку Коду 18-19 наведено приклад ігнорування другого та четвертого значень у кортежі з п'яти елементів.

```
let numbers = (2, 4, 8, 16, 32);

match numbers {
    (first, _, third, _, fifth) => {
        println!("Some numbers: {first}, {third}, {fifth}")
    }
}
```

Блок Коду 18-19: Ігнорування кількох частин кортежу

Цей код виведе `Деякі числа: 2, 8, 32`, а значення 4 та 16 будуть проігноровані.

Ігнорування Невикористаної Змінної, Починаючи Її Назву з `_`

Якщо ви створюєте змінну, але ніде її не використовуєте, Rust зазвичай попередить про це, оскільки невикористана змінна може бути помилкою. Однак, іноді буває корисно мати можливість створити змінну, яку ви поки що не будете використовувати, наприклад, коли ви створюєте прототип або тільки починаєте проєкт. У цій ситуації ви можете заборонити Rust попереджати вас про невикористану змінну, почавши назву змінної з символу підкреслення. У Блоку Коду 18-20 ми створюємо дві невикористовувані змінні, але при компіляції цього коду ми повинні отримати попередження лише про одну з них.

Файл: src/main.rs

```
fn main() {
    let _x = 5;
    let y = 10;
}
```

Блок Коду 18-20: Початок назви змінної з символу підкреслення, щоб уникнути попередження про невикористані змінні

Тут ми отримуємо попередження про невикористання змінної `y`, але не отримуємо попередження про невикористання `_x`.

Зверніть увагу, що існує тонка різниця між використанням тільки `_` і використанням імені яке починається з підкреслення. Синтаксис `_x` все ще прив'язує значення до змінної тоді як `_` не прив'язує взагалі. Щоб показати випадок, коли ця відмінність має значення, в Блоці Коду 18-21 ми наведемо помилку.

```
let s = Some(String::from("Hello!"));

if let Some(_s) = s {
    println!("found a string");
}

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

Блок Коду 18-21: Невикористана змінна, що починається з підкреслення все ще прив'язує значення, що може отримати над ним володіння

Ми отримаємо помилку, тому що значення `s` однаково буде переміщено в `_s`, що не дозволить нам використовувати `s` знову. Однак, використання символу підкреслення самого по собі ніколи не призведе до прив'язки до значення. Блок Коду 18-22 скопілюється без помилок тому що `s` не переміщується в `_`.

```
let s = Some(String::from("Hello!"));

if let Some(_) = s {
    println!("found a string");
}

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

Блок Коду 18-22: Використання символу підкреслення не прив'язує значення

Цей код працює, оскільки ми ніколи та ні до чого не прив'язували `s`; воно не зміщене.

Ігнорування Решти Частин Значення з `..`

Для значень, які мають багато частин, ми можемо використовувати синтаксис `..`, щоб використовувати певні частини та ігнорувати решту, уникаючи необхідності підкреслення кожного ігнорованого значення. Шаблон `..` ігнорує будь-які частини значення, які ми не зіставили явно в інших частинах шаблону. У Блоці Коду 18-23 ми маємо структуру `Point`, яка зберігає координату в тривимірному просторі. У виразі `match` ми хочемо оперувати тільки координатою `x` і ігнорувати значення в полях `y` та `z`.

```

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

let origin = Point { x: 0, y: 0, z: 0 };

match origin {
    Point { x, .. } => println!("x is {}", x),
}

```

Блок Коду 18-23: Ігнорування всіх полів `Point` крім для `x` з використанням `..`

Ми перераховуємо значення `x`, а потім просто додаємо шаблон `..`. Це швидше, ніж перераховувати `y: _` і `z: _`, особливо коли ми працюємо зі структурами, які мають багато полів, в ситуаціях, коли тільки одне або два поля є релевантними.

Синтаксис `..` буде поширюватися на стільки значень, скільки потрібно. У Блоці Коду 18-24 показано, як використовувати `..` з кортежем.

Файл: `src/main.rs`

```

fn main() {
    let numbers = (2, 4, 8, 16, 32);

    match numbers {
        (first, .., last) => {
            println!("Some numbers: {first}, {last}");
        }
    }
}

```

Блок Коду 18-24: Порівняння тільки першого та останнього значень кортежу та ігнорування усіх інших значень

У цьому коді першому та останньому значенню відповідають `first` та `last`. `..` буде зіставлятися та ігнорувати зі всім посередині.

Однак використання `..` має бути однозначним. Якщо незрозуміло, які значення призначені для зіставлення, а які слід ігнорувати, Rust видасть помилку. У Блоці Коду 18-25 наведено приклад неоднозначного використання `..`, тому він не буде компілюватися.

Файл: `src/main.rs`

```
fn main() {
    let numbers = (2, 4, 8, 16, 32);

    match numbers {
        (.., second, ..) => {
            println!("Some numbers: {}", second)
        },
    }
}
```

Блок Коду 18-25: Спроба використання `..` неоднозначним способом

Якщо ми скомпілюємо цей приклад, ми отримаємо цю помилку:

```
$ cargo run
   Compiling patterns v0.1.0 (file:///projects/patterns)
error: `..` can only be used once per tuple pattern
--> src/main.rs:5:22
5 |         (.., second, ..) => {
  |         --             ^^ can only be used once per tuple pattern
  |         |
  |         previously used here

error: could not compile `patterns` due to previous error
```

Rust не може визначити, скільки значень у кортежі слід ігнорувати, перш ніж знайти значення з `second`, а потім скільки наступних значень слід ігнорувати після цього. Цей код може означати, що ми хочемо ігнорувати `2`, пов'язати `second` з `4`, а потім ігнорувати `8`, `16` та `32`; або що ми хочемо ігнорувати `2` і `4`, зв'язати `second` з `8`, а потім ігнорувати `16` і `32`; тощо. Назва змінної `second` нічого особливого для Rust не означає, тому ми отримуємо помилку компілятора, тому що використання `..` в двох таких місцях є неоднозначним.

Додаткові Умови з Запобіжниками Зіставлення

Запобіжник `match` є додатковою умовою `if`, зазначеною після шаблону в рукаві `match`, яка також повинна збігатися для того, щоб цей рукав було обрано. Запобіжники `match` корисні для вираження складніших ідей, ніж дозволяє один тільки шаблон.

В умові можуть використовуватись змінні, створені в шаблоні. У Блоці Коду 18-26 показано `match`, де перший рукав має шаблон `Some(x)`, а також має запобіжник `match if x % 2 == 0` (що буде істинним, якщо число парне).

```
let num = Some(4);

match num {
    Some(x) if x % 2 == 0 => println!("The number {} is even", x),
    Some(x) => println!("The number {} is odd", x),
    None => (),
}
```

Блок Коду 18-26: Додавання запобіжника зіставлення до шаблону

Цей приклад виведе в консолі `The number 4 is even`. Коли `num` порівнюється з у першому рукаві, вони збігаються, оскільки `Some(4)` збігається з `Some(x)`. Потім запобіжник `match` перевіряє, чи залишок від ділення `x` на 2 дорівнює 0, і якщо це так, то вибирається перший рукав.

Якби замість `num` було `Some(5)`, то запобіжник `match` у першому рукаві був би хибним, оскільки остача від ділення 5 на 2 дорівнює 1, що не дорівнює 0. Rust потім перейде до другого рукава, який збігатиметься, тому що другий рукав не має запобіжника `match` і тому збігається з будь-яким варіантом `Some`.

Немає можливості виразити умову `if x % 2 == 0` в шаблоні, тому запобіжник дає можливість виразити цю логіку. Недоліком цієї додаткової виразності є те, що компілятор не намагатиметься перевіряти на вичерпність, коли задіяні вирази запобіжнику `match`.

У Блоці Коду 18-11 ми згадували, що могли б використовувати запобіжники `match` для вирішення нашої проблеми тінізації шаблонів. Нагадаємо, що ми створили нову змінну всередині шаблону у виразі `match` замість того, щоб використовувати змінну за межами `match`. Ця нова змінна означала, що ми не могли перевіряти значення зовнішньої змінної. У Блоці Коду 18-27 показано, як ми можемо використовувати запобіжник `match`, щоб розв'язати цю проблему.

Файл: `src/main.rs`

```
fn main() {
    let x = Some(5);
    let y = 10;

    match x {
        Some(50) => println!("Got 50"),
        Some(n) if n == y => println!("Matched, n = {n}"),
        _ => println!("Default case, x = {:?}", x),
    }

    println!("at the end: x = {:?}", y = {y}", x);
}
```

Блок Коду 18-27: Використання запобіжника match для перевірки рівності із зовнішньою змінною

Цей код виведе в консолі `Default case, x = Some(5)`. Шаблон у другому рукаві збігу не вводить нову змінну `y`, яка б затінювала зовнішню `y`, що означає, що ми можемо використовувати зовнішню `y` у запобіжнику `match`. Замість того, щоб вказати шаблон як `Some(y)`, що затінило б зовнішнє `y`, ми вказуємо `Some(n)`. Це створює нову змінну `n`, яка нічого не затінює, оскільки немає змінної `n` за межами `match`.

Запобіжник `match` `if n == y` не є шаблоном і тому не вводить нових змінних. Цей `y` дорівнює зовнішньому `y`, а не новому затіненому `y`, і ми можемо шукати значення, яке має те саме значення, що й зовнішнє `y`, порівнюючи `n` з `y`.

Ви також можете використовувати `or` оператор `|` в запобіжнику `match` для вказування декількох шаблонів; умова запобіжнику `match` буде застосовуватися до всіх шаблонів. У Блоці Коду 18-28 показано черговість при об'єднанні шаблону, який використовує `|` з запобіжником `match`. Важливою частиною цього прикладу є те, що запобіжник `match` `if y` застосовується до `4`, `5`, та `6`, хоча це може виглядати як `if y` тільки застосовується лише до `6`.

```
let x = 4;
let y = false;

match x {
  4 | 5 | 6 if y => println!("yes"),
  _ => println!("no"),
}
```

Блок Коду 18-28: Об'єднання декількох шаблонів за допомогою запобіжнику match

Умова `match` вказує, що рукав збігається, тільки якщо значення `x` дорівнює `4`, `5` або `6` та якщо `y` дорівнює `true`. При виконанні цього коду шаблон першого рукава збігається, оскільки `x` дорівнює `4`, але запобіжник збігу `if y` є хибним, тому перший рукав не обирається. Код переходить на другий рукав, який збігається, і ця програма виводить в консолі `no`. Причина в тому, що умова `if` застосовується до всього шаблону `4 | 5 | 6`, а не тільки до останнього значення `6`. Іншими словами, черговість запобіжнику `match` відносно шаблону наступна:

```
(4 | 5 | 6) if y => ...
```

замість:

```
4 | 5 | (6 if y) => ...
```


Після виконання коду, поведінка пріоритету очевидна: якби запобіжник `match` був застосований тільки до кінцевого значення в списку значень, заданих з допомогою оператора `|`, то рукав збігся б і програма вивела б `yes`.

@ Зв'язування

`@`, що вимовляється "оператор *at*", дозволяє нам створити змінну, яка містить значення, у той час, коли ми перевіряємо значення на відповідність шаблону. У Блоці коду 18-29 ми хочемо перевірити, що поле `id` у `Message::Hello` є в межах `3..=7`. Ми також хочемо зв'язати значення зі змінною `id_variable`, щоб ми могли використати її у коді рукава. Ми могли назвати цю змінну `id`, так само як і поле, але для цього прикладу ми використовуємо іншу назву.

```
enum Message {
    Hello { id: i32 },
}

let msg = Message::Hello { id: 5 };

match msg {
    Message::Hello {
        id: id_variable @ 3..=7,
    } => println!("Found an id in range: {}", id_variable),
    Message::Hello { id: 10..=12 } => {
        println!("Found an id in another range")
    }
    Message::Hello { id } => println!("Found some other id: {}", id),
}
```

Блок коду 18-29: використання `@` для зв'язування зі значенням у шаблоні і водночас перевірки

Цей приклад виведе в консолі `Found an id in range: 5`. Зазначивши `id_variable @` перед інтервалом `3..=7` ми захоплюємо будь-яке значення, що відповідає інтервалу, перевіряючи при цьому, що значення відповідає шаблону.

У другому рукаві, де є лише інтервал, зазначений у шаблоні, код, асоційований з цим рукавом, не має змінної, яка містила б фактичне значення поля `id`. Значення поля `id` могло б бути 10, 11, або 12, але код, що йде з цим шаблоном, не знає, яким воно є. Код шаблону нездатний використати значення поля `id`, оскільки ми не зберегли значення `id` у змінній.

В останньому рукаві, де ми вказали змінну без інтервалу, ми маємо значення, доступне для використання в коді рукава, в змінній з назвою `id`. Це тому, що ми скористалися

скороченим синтаксисом поля структури. Але ми не застосували жодної перевірки для поля `id` у цьому рукаві, як робили у двох перших рукавах: будь-яке значення відповідає цьому шаблону.

Використання `@` дозволяє нам перевірити значення і зберегти його в змінній в одному шаблоні.

Підсумок

Шаблони в Rust дуже корисні для розрізнення між різновидами даних. При використанні у виразах `match` Rust гарантує, що ваші шаблони покривають усі можливі значення, бо інакше ваша програма не скомпілюється. Шаблони в інструкціях `let` і параметрах функцій роблять ці конструкції кориснішими, дозволяючи деструктуризацію значень на менші частини одночасно з що присвоєнням їх змінним. Ми можемо створювати прості або складні шаблони відповідно до наших потреб.

Далі, у передостанньому розділі книжки ми подивимося на деякі розширені аспекти низки функціоналів Rust.

Просунуті Можливості

Наразі, ви вже вивчили найуживаніші частини мови програмування Rust. Перш ніж ми реалізуємо ще один проект у Розділі 20, ми розглянемо кілька аспектів мови, з якими ви будете стикатися час від часу, але не використовуватимете їх на постійній основі. Ви можете використовувати цей розділ як довідник, коли ви зіткнетеся з чимось невідомим. Описаний тут функціонал корисний в дуже специфічних ситуаціях. Хоча ви, ймовірно, не будете користуватися ним часто, ми хочемо переконатися, що ви добре розумієте всі можливості, які пропонує Rust.

В цьому розділі, ми розглянемо:

- Небезпечний Rust: як відмовитися від деяких гарантій Rust і взяти на себе відповідальність за дотримання цих гарантій
- Просунуті трейти: асоційовані типи (associated types), параметри типу за замовчуванням, повністю кваліфікований синтаксис, шаблон нового типу (newtype) відносно трейтів
- Просунуті типи: більше про шаблон нового типу (newtype), псевдонім типу, тип "never", а також типи із динамічним розміром
- Просунуті функції та замикання: вказівники на функції та повертаючі замикання
- Макроси: способи визначати код, що визначає інший код в час компіляції (compile time)

Це набір функціоналу Rust, де кожен знайде щось для себе! Давайте починати!

Небезпечний Rust

Весь код, що ми до цього моменту обговорювали, мав гарантії безпеки пам'яті, які Rust забезпечував під час компіляції. Однак, всередині Rust прихована інша мова, що не надає ці гарантії: вона називається *небезпечний Rust* і працює так само як і звичайний Rust, але надає додаткові суперсили.

Небезпечний Rust існує тому, що за своєю природою, статичний аналіз є консервативним. Коли компілятор намагається визначити, чи відповідає код гарантіям, для нього краще відхилити деякі допустимі програми, ніж дозволити скомпілювати певні недопустимі програми. Хоча код *може бути* в порядку, якщо компілятор Rust не має достатньо інформації, щоб бути в цьому впевненим, він відхилить такий код. У цьому разі ви можете використати небезпечний код, щоб сказати компілятору, "Довірся мені, я знаю що роблю". Однак зауважте, що ви використовуєте небезпечний Rust на свій страх і ризик: якщо ви неправильно використаєте небезпечний код, можуть виникнути проблеми з безпекою пам'яті, такі як розіменування нульового вказівника.

Інша причина, чому Rust має небезпечне альтер его, полягає в тому, що комп'ютерне обладнання за своєю суттю є небезпечним. Якби Rust не дозволяв вам виконувати небезпечні операції, ви не могли б виконувати певні завдання. Rust вимушений дозволити вам займатися низькорівневим системним програмуванням, таким як безпосередня взаємодія з операційною системою чи навіть написання власної операційної системи. Робота з низькорівневим системним програмуванням є однією з цілей мови. Дослідимо, що ми можемо зробити з небезпечним Rust і як це зробити.

Небезпечні Суперсили

Щоб перейти до небезпечного Rust, скористайтеся ключовим словом `unsafe` і почніть новий блок, що містить небезпечний код. У небезпечному Rust ви можете виконувати п'ять дій, які недоступні у безпечному Rust, ми називаємо їх *небезпечними суперсилами*. Ці суперсили дозволяють:

- Розіменувати сирий вказівник
- Викликати небезпечну функцію або метод
- Отримати доступ до мутабельної статичної змінної або модифікувати її
- Реалізувати небезпечний трейт
- Отримати доступ до полів `union`

Важливо розуміти, що `unsafe` не вимикає перевірку позик чи відключає будь-які перевірки безпеки Rust: якщо ви використовуєте посилання в небезпечному коді, його все одно буде перевірено. Ключове слово `unsafe` лише надає доступ до цих п'яти можливостей, які потім не перевіряються компілятором на безпечність використання пам'яті. Ви, як і раніше, маєте певний ступінь безпеки всередині небезпечного блоку.

Крім того, `unsafe` не означає, що код усередині блоку обов'язково створює небезпеку чи точно матиме проблеми з безпекою пам'яті: передбачається, що ви, як програміст, гарантуєте, що код всередині блоку `unsafe` буде працювати з пам'яттю коректно.

Люди роблять помилки, але вимога, щоб ці п'ять небезпечних операцій були в блоках, позначених як `unsafe`, дає вам знати, що помилки, пов'язані з безпекою пам'яті, мають бути якомусь із таких блоків `unsafe`. Хай блоки `unsafe` будуть якомога меншими; ви будете вдячні пізніше, коли будете досліджувати помилки в пам'яті.

Для ізоляції небезпечного коду, наскільки це можливо, найкраще розміщати небезпечний код у безпечній абстракції та надавати безпечний API, про що ми поговоримо пізніше в цьому розділі, коли розберемо небезпечні функції та методи. Частини стандартної бібліотеки реалізовані як безпечні абстракції навколо небезпечного коду, що пройшов перевірку. Обгортання небезпечного коду в безпечну абстракцію запобігає необхідності використовувати `unsafe` у всіх місцях, де ви або ваші користувачі можуть захотіти використати функціонал, реалізований за допомогою `unsafe`, тому що використання безпечної абстракції є безпечним.

Подивімося на кожну з п'яти небезпечних суперсил по черзі. Ми також подивімося на деякі абстракції, що надають безпечний інтерфейс для небезпечного коду.

Розіменування Сирого Вказівника

У Розділі 4, підрозділі "Підвішені посилання", ми згадували, що компілятор гарантує, що посилання є завжди коректними. Небезпечний Rust має два нові типи під назвою

сирі вказівники, схожі на посилання. Як і з посиланнями, сирі вказівники можуть бути немутабельними або мутабельними і записуються як `*const T` і `*mut T` відповідно. Зірочка тут не є оператором розіменування; це частина назви типу. У контексті сирих вказівників, *немутабельність* означає, що вказівнику не можна присвоїти значення після розіменування.

На відміну від посилань і розумних вказівників, сирі вказівники:

- Можуть ігнорувати правила позичання, маючи як немутабельні, так і мутабельні вказівники або декілька мутабельних вказівників на одне місце
- Не гарантують, що вказують на коректну пам'ять

- Можуть бути null
- Не реалізують жодного автоматичного очищення

Відмовляючись від цих гарантій Rust, ви поступаєтеся гарантованою безпекою в обмін на вищу продуктивність чи здатність взаємодії з іншою мовою чи обладнанням, які не забезпечують гарантій Rust.

Блок коду 19-1 показує, як створити немутабельний і мутабельний сирі вказівники з посилання.

```
let mut num = 5;  
  
let r1 = &num as *const i32;  
let r2 = &mut num as *mut i32;
```

Блок коду 19-1: Створення сирих вказівників із посилань

Зверніть увагу, що ми не включаємо в цей код ключове слово `unsafe`. Ми можемо створювати сирі вказівники у безпечному коді; ми не можемо лише розіменувувати сирі вказівники поза блоками `unsafe`, як ви зараз побачите.

Ми створили сирі вказівники за допомогою `as`, щоб перетворити немутабельне і мутабельне посилання у відповідні типи сирих вказівників. Оскільки ми створили їх безпосередньо з посилань, які є гарантовано коректними, ми знаємо, що ці конкретні сирі вказівники є коректними, але ми не можемо робити таке припущення про довільні сирі вказівники.

Щоб продемонструвати це, дали ми створимо сирий вказівник, у коректності якого ми не можемо бути певними. Блок коду 19-2 показує, як створити сирий вказівник до довільного місця у пам'яті. Спроба використання довільної пам'яті є невизначеною операцією: за вказаною адресою можуть бути дані або ні, компілятор може оптимізувати код, прибравши доступ до пам'яті, або програма може завершитися з помилкою сегментації. Зазвичай немає жодної причини писати подібний код, але це можливо.

```
let address = 0x012345usize;  
let r = address as *const i32;
```

Блок коду 19-2: Створення сирого вказівника на довільну адресу пам'яті

Пригадайте, що ми можемо створювати сирі вказівники в безпечному коді, але ми не можемо *розіменувувати* сирі вказівники і читати дані, на які вони вказують. У Блоці коду 19-3 ми використовуємо оператор розіменування `*` на сирому вказівнику, що потребує блоку `unsafe`.

```
let mut num = 5;

let r1 = &num as *const i32;
let r2 = &mut num as *mut i32;

unsafe {
    println!("r1 is: {}", *r1);
    println!("r2 is: {}", *r2);
}
```

Блок коду 19-3: Розіменування сирого вказівника в блоці `unsafe`

Створення вказівника не може нашкодити; лише тоді, коли ми намагаємося отримати доступ до значення, на яке він указує, ми можемо отримати в результаті некоректне значення.

Зауважте, що у Блоках коду 19-1 і 19-3 ми створили сирі вказівники `*const i32` і `*mut i32`, які обидва вказують на те саме місце в пам'яті, де зберігається `num`. Якби ми натомість спробували створити немутабельне і мутабельне посилання на `num`, код би не скомпілювався, бо правила володіння Rust забороняють мати мутабельне посилання одночасно з немутабельними посиланнями. З сирими вказівниками ми можемо створити мутабельний і немутабельний вказівники на одне й те саме місце і змінити дані через мутабельний вказівник, потенційно створивши гонитву даних. Будьте обережні!

З усіма цими небезпеками, нащо вам узагалі потрібні сирі вказівники? Одним з основних застосувань є взаємодія з кодом C, як ви побачите в наступному розділі, "[Виклик небезпечної функції або Методу](#)." Інший сценарій використання - побудова безпечної абстракції, яку borrow checker не розуміє. Ми представимо небезпечні функції, а потім подивимося на приклад безпечної абстракції, яка використовує небезпечний код.

Виклик Небезпечної Функції або Методу

Другий тип операцій, які ви можете виконувати в блоці `unsafe` - це виклик небезпечних функцій. Небезпечні функції та методи виглядають точно як звичайні функції та методи, але мають додаткове `unsafe` перед початком визначення. Ключове слово `unsafe` в цьому контексті позначає, що функція має вимоги, яких ми маємо дотримуватися при виклику цієї функції, бо Rust не може гарантувати виконання цих вимог. Викликаючи небезпечну функцію в межах блоку `unsafe`, ми заявляємо, що читали документацію цієї функції і беремо на себе відповідальність за дотримання контрактів функції.

Ось небезпечна функція з назвою `dangerous` яка не робить нічого в своєму тілі:

```
unsafe fn dangerous() {}

unsafe {
    dangerous();
}
```

Ми маємо викликати функцію `dangerous` з окремого блоку `unsafe`. Якщо ми спробуємо викликати `dangerous` без блоку `unsafe`, то отримаємо помилку:

```
$ cargo run
   Compiling unsafe-example v0.1.0 (file:///projects/unsafe-example)
error[E0133]: call to unsafe function is unsafe and requires unsafe function or block
  --> src/main.rs:4:5
   |
4  |     dangerous();
   |     ^^^^^^^^^^^ call to unsafe function
   |
   = note: consult the function's documentation for information on how to avoid undefined behavior

For more information about this error, try `rustc --explain E0133`.
error: could not compile `unsafe-example` due to previous error
```

Блоком `unsafe` ми запевняємо Rust, що ми прочитали документацію функції, розуміємо, як її правильно використовувати, і ми підтверджуємо, що виконуємо контракт функції.

Тіла небезпечних функцій є фактично блоками `unsafe`, таким чином, щоб виконати інші небезпечні операції в небезпечній функції, нам не потрібно додавати ще один блок `unsafe`.

Створення Безпечної Абстракції над Небезпечним Кодом

Те, що функція містить небезпечний код, не означає, що нам потрібно позначити всю функцію як небезпечну. Насправді обгортання небезпечного коду в безпечну функцію є звичайною абстракцією. Як приклад, розглянемо функцію `split_at_mut` зі стандартної бібліотеки, якій потрібен небезпечний код для роботи. Ми дослідимо, як ми можемо її реалізувати. Цей безпечний метод визначено на мутабельних слайсах: він бере слайс і робить з нього два, ділячи слайс по індексу, заданому аргументом. Блок коду 19-4 показує, як використовувати `split_at_mut`.


```
let mut v = vec![1, 2, 3, 4, 5, 6];

let r = &mut v[..];

let (a, b) = r.split_at_mut(3);

assert_eq!(a, &mut [1, 2, 3]);
assert_eq!(b, &mut [4, 5, 6]);
```

Блок коду 19-4: Використання безпечної функції `split_at_mut`

Ми не можемо реалізувати цю функцію за допомогою лише безпечного Rust. Спроба може бути дещо схожою на Блок коду 19-5, але вона не компілюється. Для простоти, ми реалізуємо `split_at_mut` як функцію, а не метод, і тільки для слайсів значень `i32` замість узагальненого типу `T`.

```
fn split_at_mut(values: &mut [i32], mid: usize) -> (&mut [i32], &mut [i32]) {
    let len = values.len();

    assert!(mid <= len);

    (&mut values[..mid], &mut values[mid..])
}
```

Блок коду 19-5: Спроба реалізації `split_at_mut` використовуючи лише безпечний Rust

Функція спочатку отримує загальну довжину слайса. Потім стверджує, що індекс, заданий параметром, знаходиться в слайсі, перевіривши, що він менший чи дорівнює довжині. Твердження означає, що якщо ми передамо індекс, більший за довжину, щоб розділити по ньому слайс, функція запанікує перед спробою використати цей індекс.

Тоді ми повертаємо два мутабельні слайси у кортежі: один від початку вихідного слайса до індексу `mid`, і другий з `mid` до кінця слайса.

Коли ми спробуємо скомпілювати код в Блоці коду 19-5, ми отримаємо помилку.

```
$ cargo run
Compiling unsafe-example v0.1.0 (file:///projects/unsafe-example)
error[E0499]: cannot borrow `*values` as mutable more than once at a time
--> src/main.rs:6:31
|
1 | fn split_at_mut(values: &mut [i32], mid: usize) -> (&mut [i32], &mut [i32]) {
|                                     - let's call the lifetime of this reference `l`
...
6 |     (&mut values[..mid], &mut values[mid..])
|     ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
|     |           |           |
|     |           |           second mutable borrow occurs here
|     |           first mutable borrow occurs here
|     returning this value requires that `*values` is borrowed for `l`

For more information about this error, try `rustc --explain E0499`.
error: could not compile `unsafe-example` due to previous error
```

Перевірка позичання Rust не розуміє, що ми позичаємо різні частини слайса; вона лише знає, що ми позичаємо з одного слайса двічі. Позичання різних частин слайса є принципово правильним, оскільки два слайси не перетинаються, але Rust недостатньо розумна, щоб знати це. Коли ми знаємо, що код не містить помилок, але Rust так не вважає, настає час долучити небезпечний код.

Блок коду 19-6 показує, як використовувати блок `unsafe`, сирий вказівник і деякі виклики небезпечних функцій, щоб реалізація `split_at_mut` запрацювала.

```
use std::slice;

fn split_at_mut(values: &mut [i32], mid: usize) -> (&mut [i32], &mut [i32]) {
    let len = values.len();
    let ptr = values.as_mut_ptr();

    assert!(mid <= len);

    unsafe {
        (
            slice::from_raw_parts_mut(ptr, mid),
            slice::from_raw_parts_mut(ptr.add(mid), len - mid),
        )
    }
}
```

Блок коду 19-6: Використання небезпечного коду у реалізації функції `split_at_mut`

Згадайте з підрозділу "Тип даних слайс" Розділу 4, що слайси є вказівником на певні дані і довжиною слайса. Ми використовуємо метод `len`, щоб отримати довжину слайса, і метод `as_mut_ptr`, щоб отримати сирий вказівник зі слайса. У цьому випадку, оскільки ми

маємо мутабельний слайс зі значень `i32`, `as_mut_ptr` повертає сирий вказівник типу `*mut i32`, який ми зберігаємо у змінній `ptr`.

Ми зберігаємо твердження, що індекс `mid` знаходиться у межах слайса. Далі ми дістаємося небезпечного коду: функція `slice::from_raw_parts_mut` приймає сирий вказівник і довжину, і створює слайс. Ми використовуємо цю функцію для створення слайса, що починається з `ptr` має довжину `mid` елементів. Тоді ми викликаємо метод `add` для `ptr` з `mid` як аргументом, щоб отримати сирий вказівник, що починається з `mid`, і створюємо слайс за допомогою цього вказівника і числа елементів, що залишилися після `mid`, як довжини.

Функція `slice::from_raw_parts_mut` є небезпечною, бо приймає сирий вказівник і має покладатися на те, що цей вказівник є коректним. Метод `add` для сирих вказівників також є небезпечним, бо має покладатися на те, що місце зсуву також є коректним вказівником. Саме тому ми маємо поставити блок `unsafe` навколо наших викликів `slice::from_raw_parts_mut` і `add`, щоб ми могли їх викликати. Поглянувши на код і додавши твердження, що `mid` має бути меншим або рівним `len`, ми можемо сказати що всі сирі вказівники, що використовуються в блоці `unsafe`, будуть коректними вказівниками на дані в межах слайса. Це є прийнятним і доречним використанням `unsafe`.

Зверніть увагу, що нам не потрібно позначати остаточну функцію `split_at_mut` як `unsafe`, і ми можемо викликати цю функцію з безпечного Rust. Ми створили безпечну абстракція для небезпечного коду з реалізацією функції, які використовує код `unsafe` у безпечний спосіб, тому що він створює тільки коректні вказівники з даних, до яких ця функція має доступ.

Натомість використання `slice::from_raw_parts_mut` у Блоці коду 19-7, схоже, призведе до падіння при використанні слайса. Цей код бере довільне місце в пам'яті і створює слайс довжиною 10 000 елементів.

```
use std::slice;

let address = 0x01234usize;
let r = address as *mut i32;

let values: &[i32] = unsafe { slice::from_raw_parts_mut(r, 10000) };
```

Блок коду 19-7: Створення слайса з довільного розташування в пам'яті

Ми не володіємо пам'яттю у цьому довільному місці, і немає гарантії, що слайс, створений цим кодом, містить коректні значення `i32`. Спроба використання `values`, ніби це коректний слайс, призводить до невизначеної поведінки.

Використання `extern` Функцій для Виклику Зовнішнього Коду

Іноді вашому коду Rust потрібно взаємодіяти з кодом, написаним іншою мовою. Для цього Rust має ключове слово `extern`, яке полегшує створення і використання *Інтерфейсу Зовнішніх Функцій (Foreign Function Interface, FFI)*. FFI - це засіб мови програмування для визначення функцій і дозволу іншій (зовнішній) мові програмування викликати ці функції.

Блок коду 19-8 демонструє, як налаштувати інтеграцію із функцією `abs` зі стандартної бібліотеки C. Функції, проголошені в блоках `extern`, завжди є небезпечними для виклику з коду Rust. Причина в тому, що інші мови не забезпечують правила і гарантії Rust, і Rust не може перевірити їх, тож відповідальність за гарантування безпеки покладається на програміста.

Файл: src/main.rs

```
extern "C" {  
    fn abs(input: i32) -> i32;  
}  
  
fn main() {  
    unsafe {  
        println!("Absolute value of -3 according to C: {}", abs(-3));  
    }  
}
```

Блок коду 19-8: проголошення і виклик зовнішньої (`extern`) функції, написаної іншою мовою

У блоці `extern "C"`, ми перелічуємо назви і сигнатури зовнішніх функцій з іншої мови, які ми хочемо викликати. Частина `"C"` визначає, який *двійковий інтерфейс застосунку (application binary interface, ABI)* використовується зовнішньою функцією: ABI визначає спосіб виклику функції на рівні асемблера. ABI `"C"` є найпоширенішим і відповідає ABI мови програмування C.

Виклик Функцій Rust з Інших Мов

Ми також можемо скористатися `extern`, щоб створити інтерфейс, що дозволяє іншим мовам викликати функції Rust. Замість створення цілого блоку `extern`, додамо ключове слово `extern` і зазначимо ABI, який треба використовувати перед ключовим словом `fn` у відповідній функції. Нам також треба додати анотацію `#[no_mangle]`, щоб сказати компілятору Rust не перетворювати назву цієї функції. *Перетворення (mangling)* - це коли компілятор змінює назву, яку ми дали функції, на

іншу назву, яка містить більше інформації для інших частин процесу компіляції, але є менш зручною для людини. Кожен компілятор мови програмування дещо по-різному перетворює назви, тому для того, щоб функцію Rust можна було назвати в інших мовах, ми маємо відключити перетворення назв компілятором Rust.

У наступному прикладі ми робимо функцію `call_from_c` доступною з C після того, як вона буде скомпільована у спільну бібліотеку та злінкована з C:

```
#[no_mangle]
pub extern "C" fn call_from_c() {
    println!("Just called a Rust function from C!");
}
```

Використання `extern` не вимагає використання `unsafe`.

Доступ або Модифікація Мутабельних Статичних Змінних

У цій книзі ми ще не говорили про *глобальні змінні*, які Rust підтримує, але які можуть створювати проблеми з правилами володіння Rust. Якщо два потоки отримують доступ до однієї мутабельної глобальної змінної, це може викликати гонитву даних.

У Rust глобальні змінні називаються *статичними* змінними. Блок коду 19-9 показує приклад визначення і використання статичної змінної зі значенням стрічкового слайсу.

Файл: `src/main.rs`

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

fn main() {
    println!("name is: {}", HELLO_WORLD);
}
```

Блок коду 19-9: визначення і використання немутабельної статичної змінної

Статичні змінні подібні до констант, які ми обговорювали в підрозділі "Константи" у Розділі 3. Назви статичних змінних за домовленістю пишуться

`ВЕРХНІМ_РЕГІСТРОМ_З_ПІДКРЕСЛЕННЯМИ`. Статичні змінні можуть зберігати лише посилання з часом існування `'static`, що означає, що компілятор Rust може знайти час існування, а ми не зобов'язані анотувати його явно. Доступ до немутабельних статичних змінних є безпечним.

Тонка різниця між константами і немутабельними статичними змінними полягає в тому, що значення в статичній змінній має фіксовану адресу в пам'яті. Коли ви використовуєте

значення, то завжди матимете доступ до тих самих даних. Константи, з іншого боку, можуть дублювати дані всюди, де їх використовують. Інша відмінність полягає в тому, що статичні змінні можуть бути мутабельними. Доступ і зміна мутабельних статичних змінних є *небезпечним*. Блок коду 19-10 показує, як проголошувати, отримувати доступ і змінювати мутабельну статичну змінну, що називається **COUNTER**.

Файл: src/main.rs

```
static mut COUNTER: u32 = 0;

fn add_to_count(inc: u32) {
    unsafe {
        COUNTER += inc;
    }
}

fn main() {
    add_to_count(3);

    unsafe {
        println!("COUNTER: {}", COUNTER);
    }
}
```

Блок коду 19-10: читання і запис мутабельної статичної змінної є небезпечним

Як і зі звичайними змінними, ми визначаємо мутабельність ключовим словом **mut**. Будь-який код, який читає чи записує **COUNTER**, має бути в блоці **unsafe**. Цей код компілюється і виводить **COUNTER: 3**, як ми й маємо очікувати, бо він однопоточний. Якщо ж багато потоків матимуть доступ до **COUNTER**, це, швидше за все, призведе до гонитви даних.

З глобально доступними мутабельними даними важко забезпечити, щоб не було гонитви даних, і саме тому Rust вважає мутабельні статичні змінні небезпечними. Де це можливо, бажано використовувати методи конкурентності та потокобезпечні розумні вказівники, які ми обговорювали в Розділі 16, щоб компілятор перевіряв, що доступ до даних з різних потоків здійснюється безпечно.

Реалізація Небезпечного Трейта

Ми можемо скористатися **unsafe** для реалізації небезпечного трейта. Трейт є небезпечним, якщо хоча б один з його методів має якийсь інваріант, який компілятор не може перевірити. Ми проголошуємо, що трейт є **небезпечним**, додаючи ключове слово **unsafe** перед **trait** та позначивши реалізацію трейта як **unsafe**, як показано у Блоці коду 19-11.

```
unsafe trait Foo {  
    // methods go here  
}  
  
unsafe impl Foo for i32 {  
    // method implementations go here  
}  
  
fn main() {}
```

Блок коду 19-11: визначення та реалізація небезпечного трейта

За допомогою `unsafe impl`, ми обіцяємо, що дотримуватимемося інваріантів, які компілятор не може перевірити.

Як приклад, згадайте маркерні трейти `Sync` і `Send`, які ми обговорювали в підрозділі "Розширювана конкурентність із трейтами `Sync` і `Send`" у Розділі 16: компілятор реалізує ці трейти автоматично, якщо наші типи повністю складаються з типів `Send` і `Sync`. Якщо ми реалізуємо тип, який містить тип, що не є `Send` або `Sync`, такий як сирі вказівники, і ми хочемо позначити цей тип як `Send` або `Sync`, ми маємо використовувати `unsafe`. Довіра не може переконатися, що наш тип дотримується гарантій, щоб його можна було безпечно передавати між потоками або мати до нього доступ з декількох потоків; таким чином, нам потрібно робити ці перевірки вручну і позначити це за допомогою `unsafe`.

Доступ до Полів Об'єднання

Остання дія, яка працює лише за допомогою `unsafe` - це доступ до полів об'єднання. Об'єднання (`union`) схоже на структуру `struct`, але лише одне проголошене поле використовується у конкретному екземплярі у кожен певний момент часу. Об'єднання передусім використовується для інтерфейсу з об'єднаннями в коді C. Доступ до полів об'єднання є небезпечним, бо Rust не може гарантувати, який саме тип даних зараз зберігається у екземплярі об'єднання. Більше про об'єднання ви можете дізнатися у [Довіднику Rust](#).

Коли Використовувати Небезпечний Код

Використання `unsafe` для отримання однієї з п'яти дій (суперсил), про які ми щойно говорили, не є неправильним чи навіть несхвальним. Але код `unsafe` складніше зробити коректним, бо компілятор не може підтримувати безпеку пам'яті. Коли ви маєте причину використовувати `unsafe`, ви можете так робити, а наявність явних анотацій `unsafe` полегшує відстеження джерела проблем, коли вони виникають. [ch04-02-references-and-borrowing.html#dangling-references](#) [ch03-01-variables-and-mutability.html#constants](#) [ch16-](#)

04-extensible-concurrency-sync-and-send.html#extensible-concurrency-with-the-sync-and-send-traits

Поглиблено про Трейти

Ми вже розповідали про трейти у підрозділі “Трейти: визначення загальної поведінки” Розділу 10, але ми не говорили про глибші деталі. Тепер, коли ви більше знаєте про Rust, ми можемо перейти до дрібніших деталей.

Зазначення Заповнювача Типу у Визначенні Трейтів за Допомогою Асоційованих Типів

Асоційовані типи зв'язують заповнювач типу з трейтом таким чином, що методи трейту можуть використовувати ці заповнювачі типу у своїх сигнатурах. Той, хто реалізовуватиме трейт, зазначить конкретний тип, що буде використовуватися, замість заповнювача типу для конкретної реалізації. Таким чином ми можемо визначити трейт, що використовує деякі типи, без потреби точно знати ці типи до моменту реалізації трейту.

Ми описували більшість поглиблених особливостей у цьому розділі як такі, що рідко потрібні. Пов'язані типи десь посередині: вони використовуються рідше за функціонал, описаний в решті книги, але частіше ніж багато іншого функціоналу, обговорюваного в цьому розділі.

Одним з прикладів трейту з асоційованим типом є трейт `Iterator`, наданий стандартною бібліотекою. Асоційований тип називається `Item` і позначає тип значень, по яких ітерує тип, що реалізує трейт `Iterator`. Визначення трейту `Iterator` показано у Блоці коду 19-12.

```
pub trait Iterator {  
    type Item;  
  
    fn next(&mut self) -> Option<Self::Item>;  
}
```

Блок коду 19-12: визначення трейту `Iterator`, що має асоційований тип `Item`

Тип `Item` є заповнювачем, і визначення методу `next` показує, що він повертає значення типу `Option<Self::Item>`. Ті, хто реалізовуватимуть трейт `Iterator`, зазначать конкретний тип для `Item`, і метод `next` повертатиме `Option`, що міститиме значення цього конкретного типу.

Асоційовані типи можуть видатися концепцією, подібною до узагальнень, у тому, що останні дозволяють визначити функцію без зазначення, які типи вона може обробляти. Для вивчення відмінностей між двома концепціями, погляньмо на реалізацію трейту `Iterator` для типу, що зветься `Counter` із зазначеним типом `Item` `u32`:

Файл: `src/lib.rs`

```
impl Iterator for Counter {  
    type Item = u32;  
  
    fn next(&mut self) -> Option<Self::Item> {  
        // --snip--  
    }  
}
```

Цей синтаксис здається схожим на узагальнені параметри. То чому ж просто не визначити трейт `Iterator` з узагальненим параметром, як показано в Блоці коду 19-13?

```
pub trait Iterator<T> {  
    fn next(&mut self) -> Option<T>;  
}
```

Блок коду 19-13: гіпотетичне визначення трейту `Iterator` за допомогою узагальненого параметра

Різниця полягає в тому, що при використанні узагальнених параметрів, як у Блоці коду 19-13, ми маємо анотувати типи у кожній реалізації; а оскільки ми також можемо реалізувати `Iterator<String>` для `Counter` чи будь-якого іншого типу, ми можемо мати багато реалізацій `Iterator` для `Counter`. Іншими словами, коли трейт має узагальнений параметр, він може бути реалізованим для типу багато разів, кожного разу для іншого конкретного типу узагальненого параметра. Коли ми використовуємо метод `next` для `Counter`, нам доведеться надавати анотації типу, щоб позначити, яку реалізацію `Iterator` ми хочемо використати.

З асоційованими типами нам не треба анотувати типи, бо ми не можемо реалізувати трейт для типу кілька разів. У Блоці коду 19-12, з визначенням, яке використовує асоційовані типи, ми можемо обрати тип `Item` лише один раз, бо може бути лише один `impl Iterator for Counter`. Нам не треба зазначати, що ми хочемо ітератор по значеннях `u32` всюди, де ми викликаємо `next` для `Counter`.

Асоційовані типи також стають частиною контракту трейту: ті, хто реалізують трейт, мають зазначити тип для заповнювача асоційованого типу. Асоційовані типи часто мають назву, що описує, як цей тип буде використано, і документування асоційованого типу в документації API є доброю практикою.

Узагальнені Параметри Типу за Замовчуванням і Перевантаження Операторів

Коли ми використовуємо узагальнені параметри типу, то можемо вказати конкретний тип за замовчуванням для узагальненого типу. Це усуває потребу для тих, хто реалізовуватиме трейт, вказувати конкретний тип, якщо тип за замовчуванням працює. Ви можете вказати тип за замовчуванням при проголошенні узагальненого типу за допомогою синтаксису `<PlaceholderType=ConcreteType>`.

Чудовий приклад ситуації, коли ця техніка корисна, це *перевантаження операторів*, де ви налаштовуєте поведінку оператора (наприклад, `+`) в певних ситуаціях.

Rust не дозволяє вам створювати власні оператори або перевантажувати довільні оператори. Але ви можете перевантажити операції і відповідні трейти, перелічені в `std::ops`, реалізуючи трейти, пов'язані з оператором. Наприклад, у Блоці коду 19-14 ми перевантажуємо оператор `+`, щоб додавати два екземпляри `Point`. Ми робимо це, реалізуючи трейт `Add` для `Point`:

Файл: src/main.rs

```
use std::ops::Add;

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

impl Add for Point {
    type Output = Point;

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

fn main() {
    assert_eq!(
        Point { x: 1, y: 0 } + Point { x: 2, y: 3 },
        Point { x: 3, y: 3 }
    );
}
```

Блок коду 19-14: реалізація трейту `Add` для перевантаження оператора `+` для екземплярів `Point`

Метод `add` додає значення `x` двох екземплярів `Point` і значення `y` двох екземплярів `Point`, щоб створити нову `Point`. Трейт `Add` має асоційований тип, що називається `Output`, який визначає тип, який повертає метод `add`.

Узагальнений параметр типу за замовчанням у цьому коді належить трейту `Add`. Ось його визначення:

```
trait Add<Rhs=Self> {  
    type Output;  
  
    fn add(self, rhs: Rhs) -> Self::Output;  
}
```

Цей код має виглядати в цілому знайомо: трейт з одним методом і асоційованим типом. Новим тут є `Rhs=Self`: цей синтаксис зветься *параметром типу за замовчанням*.

Узагальнений параметр типу `Rhs` (скорочено для "right hand side" - "правий бік") визначає тип параметра `rhs` у методі `add`. Якщо ми не зазначимо конкретний тип для `Rhs`, коли ми реалізуємо трейт `Add`, тип `Rhs` буде взято за замовчанням як `Self`, тобто тип, для якого ми реалізуємо `Add`.

Коли ми реалізували `Add` для `Point`, ми використали значення за замовчанням для `Rhs`, бо ми хотіли додавати два екземпляри `Point`. Розгляньмо приклад реалізації трейта `Add`, де ми хочемо виставити свій тип `Rhs`, а не використовувати значення за замовчуванням.

Ми маємо дві структури, `Millimeters` і `Meters`, що містять значення в різних одиницях. Ця тонка обгортка типу, що існує, у іншу структуру відома як *шаблон новий тип*, який ми описували детальніше у підрозділі ["Використання шаблону новий тип для реалізації зовнішніх трейтів на зовнішніх типах"](#). Ми хочемо додавати значення у міліметрах до значень у метрах, щоб реалізація

`Add` коректно виконувала перетворення. Ми можемо реалізувати `Add` для `Millimeters` з `Meters` як `Rhs`, як показано в Блоці коду 19-15.

Файл: `src/lib.rs`

```
use std::ops::Add;

struct Millimeters(u32);
struct Meters(u32);

impl Add<Meters> for Millimeters {
    type Output = Millimeters;

    fn add(self, other: Meters) -> Millimeters {
        Millimeters(self.0 + (other.0 * 1000))
    }
}
```

Блок коду 19-15: реалізація трейту `Add` для `Millimeters`, щоб додавати `Millimeters` до `Meters`

Щоб додати `Millimeters` і `Meters`, вказуємо `impl Add<Meters>`, щоб встановити значення параметра типу `Rhs` замість встановленого за замовчуванням `Self`.

Параметри типу за замовчанням використовуються у двох основних випадках:

- Щоб розширити тип, не порушуючи коду, що існує
- Щоб дозволити налаштування у певних випадках, не потрібних більшості користувачів

Трейт `Add` зі стандартної бібліотеки є прикладом другого призначення: зазвичай ви додаєте однакові типи, але трейт `Add` надає можливість глибшого налаштування. Використовуючи параметр типу за замовчуванням в трейті `Add` означає, що у більшості випадків вам не потрібно вказувати додатковий параметр. Іншими словами, не потрібно вказувати шматок шаблонного коду реалізації, що полегшує використання трейту.

Перше призначення схоже на друге, але навпаки: якщо ви хочете додати параметр типу до трейту, що вже існує, ви можете надати йому параметр типу за замовчуванням, щоб розширити функціональність трейту не порушивши наявного коду реалізації.

Повністю Кваліфікований Синтаксис для Уникнення Двозначностей: Виклик Методів з Однаковою Назвою

Ніщо у Rust не забороняє трейту мати метод з такою самою назвою, як і в іншому трейті, ані реалізувати обидва трейти для одного типу. Також можна реалізувати метод з такою ж назвою, як у трейтах, напрямую для типу.

При виклику методів з однаковою назвою вам треба вказати Rust, котрий саме метод ви хочете використати. Розгляньмо код у Блоці коду 19-16, де ми визначили два трейти,

`Pilot` і `Wizard`, що обидва мають метод `fly`. Тоді ми реалізуємо обидва трейти для типу `Human`, що також має реалізований на ньому метод з назвою `fly`. Кожен метод `fly` робить щось інше.

Файл: `src/main.rs`

```
trait Pilot {
    fn fly(&self);
}

trait Wizard {
    fn fly(&self);
}

struct Human;

impl Pilot for Human {
    fn fly(&self) {
        println!("This is your captain speaking.");
    }
}

impl Wizard for Human {
    fn fly(&self) {
        println!("Up!");
    }
}

impl Human {
    fn fly(&self) {
        println!("*waving arms furiously*");
    }
}
```

Блок коду 19-16: два трейти маю визначення, що містить метод `"fly"` і реалізовані для типу `Human`, а також метод `fly`, реалізований безпосередньо для `Human`

Коли ми викликаємо `fly` на екземплярі `Human`, компілятор за замовчуванням викликає метод, реалізований безпосередньо на типі, як показано у Блоці коду 19-17.

Файл: `src/main.rs`

```
fn main() {
    let person = Human;
    person.fly();
}
```

Блок коду 19-17: виклик `fly` для екземпляра `Human`

Запуск цього коду виведе `*waving arms furiously*`, показуючи, що Rust викликав метод `fly`, реалізований безпосередньо для `Human`.

Щоб викликати методи `fly` з трейту `Pilot` або трейту `Wizard`, нам треба використати більш явний синтаксис, щоб зазначити, який саме метод `fly` ми маємо на увазі. Блок коду 19-18 демонструє такий синтаксис.

Файл: `src/main.rs`

```
fn main() {
    let person = Human;
    Pilot::fly(&person);
    Wizard::fly(&person);
    person.fly();
}
```

Блок коду 19-18: уточнення, метод `fly` якого саме трейту ми хочемо викликати

Зазначення назви трейту перед назвою методу прояснює для Rust, котру реалізацію `fly` ми хочемо викликати. Ми також могли б написати `Human::fly(&person)`, що є еквівалентом `person.fly()`, який ми використали в Блоці коду 19-18, але так трохи довше писати, якщо нам не треба уникнути двозначності.

Виконання цього коду виведе наступне:

```
$ cargo run
Compiling traits-example v0.1.0 (file:///projects/traits-example)
Finished dev [unoptimized + debuginfo] target(s) in 0.46s
Running `target/debug/traits-example`
This is your captain speaking.
Up!
*waving arms furiously*
```

Оскільки метод `fly` приймає параметр `self`, то якщо ми маємо два *тупи*, що обидва реалізують один *трейт*, Rust може зрозуміти, яку реалізацію трейту використати, виходячи з типу `self`.

Однак асоційовані функції, що не є методами, не мають параметру `self`. Коли є багато типів чи трейтів, що визначають функції, що не є методами, з однаковими назвами, Rust не завжди знає, який тип ви мали на увазі, якщо ви не використаєте *повний кваліфікований синтаксис*. Наприклад, у Блоці коду 19-19 ми створюємо трейт для притулку тварин, що хоче називати всіх маленьких собак *Spot*. Ми створюємо трейт `Animal` з асоційованою функцією - не методом `baby_name`. Трейт `Animal` реалізований для структури `Dog`, для якої ми також визначаємо напряду асоційовану функцію - не методом `baby_name`.

Файл: src/main.rs

```

trait Animal {
    fn baby_name() -> String;
}

struct Dog;

impl Dog {
    fn baby_name() -> String {
        String::from("Spot")
    }
}

impl Animal for Dog {
    fn baby_name() -> String {
        String::from("puppy")
    }
}

fn main() {
    println!("A baby dog is called a {}", Dog::baby_name());
}

```

Блок коду 19-19: трейт з асоційованою функцією і тип з асоційованою функцією з такою ж назвою, що реалізує цей трейт

Ми реалізували код для називання всіх цуценят Spot у асоційованій функції `baby_name`, визначеній для `Dog`. Тип `Dog` також реалізує трейт `Animal`, що описує характеристики, спільні для всіх тварин. Дитинчата собак зветься цуценятами, і це виражено в реалізації трейту `Animal` для `Dog` у функції `baby_name`, асоційованій з трейтом `Animal`.

У `main` ми викликаємо функцію `Dog::baby_name`, яка викликає асоційовану функцію, визначену безпосередньо для `Dog`. Цей код виводить таке:

```

$ cargo run
Compiling traits-example v0.1.0 (file:///projects/traits-example)
Finished dev [unoptimized + debuginfo] target(s) in 0.54s
Running `target/debug/traits-example`
A baby dog is called a Spot

```

Ми хотіли не такого виведення. Ми хотіли викликати функцію `baby_name`, що є частиною трейту `Animal`, який ми реалізували для `Dog`, щоб код вивів `A baby dog is called a puppy`. Техніка зазначення назви трейту, яку ми використали у Блоці коду 19-18 тут не допомагає; якщо ми змінимо `main` на код, наведений у Блоці коду 19-20, ми отримаємо помилку компіляції.

Файл: src/main.rs


```
fn main() {
    println!("A baby dog is called a {}", Animal::baby_name());
}
```

Блок коду 19-20: спроба викликати функцію `baby_name` з трейту `Animal`, але Rust не знає, яку реалізацію використати

Оскільки `Animal::baby_name` не має параметру `self`, і можуть бути інші типи, що реалізують трейт `Animal`, Rust не може з'ясувати, яку реалізацію `Animal::baby_name` ми хочемо. Ми отримуємо цю помилку компілятора:

```
$ cargo run
Compiling traits-example v0.1.0 (file:///projects/traits-example)
error[E0790]: cannot call associated function on trait without specifying the
corresponding `impl` type
--> src/main.rs:20:43
  |
2 |     fn baby_name() -> String;
  |     ----- `Animal::baby_name` defined here
...
20 |     println!("A baby dog is called a {}", Animal::baby_name());
  |                                           ^^^^^^^^^^^^^^^^^^^^^ cannot call
associated function of trait
  |
help: use the fully-qualified path to the only available implementation
20 |     println!("A baby dog is called a {}", <::Dog as Animal>::baby_name());
  |                                           ++++++++ +

For more information about this error, try `rustc --explain E0790`.
error: could not compile `traits-example` due to previous error
```

Щоб розрізнити реалізації і сказати Rust, що ми хочемо використати реалізацію `Animal` для `Dog`, а не реалізацію `Animal` для якогось іншого типу, ми маємо використати повний кваліфікований синтаксис. Блок коду 19-21 демонструє, як використовувати повний кваліфікований синтаксис.

Файл: `src/main.rs`

```
fn main() {
    println!("A baby dog is called a {}", <Dog as Animal>::baby_name());
}
```

Блок коду 19-21: використання повністю кваліфікованого синтаксису, щоб вказати, що ми хочемо викликати функцію `baby_name` з трейту `Animal`, реалізованого для `Dog`

Ми надаємо Rust анотацію типу в кутових дужках, що показує, що ми хочемо викликати метод `baby_name` з трейту `Animal`, як він реалізований для `Dog`, кажучи, що ми хочемо розглядати тип `Dog` як `Animal` для цього виклику функції. Цей код тепер виведе те, що ми хотіли:

```
$ cargo run
  Compiling traits-example v0.1.0 (file:///projects/traits-example)
  Finished dev [unoptimized + debuginfo] target(s) in 0.48s
  Running `target/debug/traits-example`
A baby dog is called a puppy
```

В цілому, повний кваліфікований синтаксис визначений таким чином:

```
<Type as Trait>::function(receiver_if_method, next_arg, ...);
```

Для асоційованих функцій, які не є методами, не використовується `receiver`: там буде лише список решти аргументів. Ви можете використовувати повний кваліфікований синтаксис всюди, де викликаєте функції або методи. Однак, ви можете пропустити будь-яку частину синтаксису, яку Rust може визначити за допомогою іншої інформації у програмі. Вам потрібно використовувати цей більш багатослівний синтаксис у випадках, коли є декілька реалізацій, які використовують одну назву і Rust потребує допомоги, щоб визначити, яку реалізацію ви хочете викликати.

Використання Супертрейтів для Вимоги Функціонала Одного Трейта в Іншому Трейті

Іноді ви можете написати визначення трейта, що залежить від іншого трейта: для типу, що реалізує перший трейт, ви хочете вимагати, щоб цей тип також реалізовував другий трейт. Вам може бути таке потрібно, якщо визначення вашого трейта використовує асоційовані елементи другого трейта. Трейт, на який покладається визначення вашого трейта, зветься *супертрейтом* вашого трейта.

Наприклад, припустимо, що ми хочемо зробити трейт `OutlinePrint` з методом `outline_print`, що виводить задане значення, форматоване рамкою з зірочок. Тобто, якщо структура `Point` реалізує трейт зі стандартної бібліотеки `Display` і виводить `(x, y)`, то коли ми викликаємо `outline_print` на екземплярі `Point`, що має значення `1` для `x` і `3` для `y`, він має вивести таке:

```

*****
*           *
* (1, 3) *
*           *
*****

```

У реалізації методу `outline_print` ми хочемо використати функціональність трейту `Display`. Відповідно, нам потрібно вказати що трейт `OutlinePrint` буде працювати тільки для типів, які також реалізують `Display` і надають функціональність, потрібну `OutlinePrint`. Ми можемо зробити це у визначенні трейта, вказавши `OutlinePrint: Display`. Ця техніка схожа на додавання до трейта трейтового обмеження. Блок коду 19-22 показує реалізацію трейту `OutlinePrint`.

Файл: `src/main.rs`

```

use std::fmt;

trait OutlinePrint: fmt::Display {
    fn outline_print(&self) {
        let output = self.to_string();
        let len = output.len();
        println!("{}", "*".repeat(len + 4));
        println!("*{}*", " ".repeat(len + 2));
        println!("* {} *", output);
        println!("*{}*", " ".repeat(len + 2));
        println!("{}", "*".repeat(len + 4));
    }
}

```

Блок коду 19-22: реалізація трейту `OutlinePrint`, який вимагає функціональності `Display`

Оскільки ми вказали, що `OutlinePrint` потребує трейту `Display`, ми можемо використовувати функцію `to_string`, автоматично реалізовану для будь-якого типу, що реалізовує `Display`. Якби ми спробували використати `to_string`, не додавши двокрапки і трейту `Display` після назви трейту, ми б отримали помилку про те, що метод `to_string` не був знайдений для типу `&Self` у поточній області видимості.

Подивімося, що станеться, коли ми спробуємо реалізувати `OutlinePrint` для типу, що не реалізує `Display`, такому як структура `Point`:

Файл: `src/main.rs`

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

impl OutlinePrint for Point {}
```

Ми отримуємо помилку, яка повідомляє, що `Display` є потрібним, але не реалізованим:

```
$ cargo run
Compiling traits-example v0.1.0 (file:///projects/traits-example)
error[E0277]: `Point` doesn't implement `std::fmt::Display`
--> src/main.rs:20:6
|
20 | impl OutlinePrint for Point {}
|     ^^^^^^^^^^^^^^^^^ `Point` cannot be formatted with the default formatter
|
= help: the trait `std::fmt::Display` is not implemented for `Point`
= note: in format strings you may be able to use `{:?}` (or `{:#?}` for pretty-
print) instead
note: required by a bound in `OutlinePrint`
--> src/main.rs:3:21
3 | trait OutlinePrint: fmt::Display {
|                       ^^^^^^^^^^^ required by this bound in `OutlinePrint`

For more information about this error, try `rustc --explain E0277`.
error: could not compile `traits-example` due to previous error
```

Щоб виправити це, ми реалізуємо `Display` для `Point` і задовольняємо обмеження для `OutlinePrint` ось таким чином:

Файл: src/main.rs

```
use std::fmt;

impl fmt::Display for Point {
    fn fmt(&self, f: &mut fmt::Formatter) -> fmt::Result {
        write!(f, "({}, {})", self.x, self.y)
    }
}
```

To fix this, we implement `Display` on `Point` and satisfy the constraint that `OutlinePrint` requires, like so:

Використання Шаблону "Новий Тип" для Реалізації Зовнішніх Трейтів на Зовнішніх Типах

У Розділі 10, у підрозділі "Реалізація трейта для типу", ми згадали правило сироти, яке каже, що ми можемо реалізовувати трейт для типу, якщо трейт або тип є локальним для нашого крейта. Це обмеження можна обійти за допомогою *паттерна "новий тип"*, що передбачає створення нового типу у структурі-кортежі. (Про структури-кортежі ми говорили у підрозділі "Використання структур-кортежів без названих полів для створення нових типів" Розділу 5.) Структури-кортежі мають одне поле і є тонкою обгорткою для типу, для якого ми хочемо реалізувати трейт. Тоді тип-обгортка є локальним для нашого крейта, і ми можемо реалізувати трейт для обгортки.

Новий тип - це термін, який походить з мови програмування Haskell. Використання цього шаблону не призводить до втрат швидкодії, а тип обгортки приховується під час компіляції.

Наприклад, скажімо, ми хочемо реалізувати `Display` для `Vec<T>`, що безпосередньо заборонено правилом сироти, тому що трейт `Display` і тип `Vec<T>` визначається поза нашим крейтом. Ми можемо зробити структуру `Wrapper`, що містить екземпляр `Vec<T>`; тоді ми можемо реалізувати `Display` для `Wrapper` використати значення `Vec<T>`, як показано в Блоці коду 19-23.

Файл: src/main.rs

```
use std::fmt;

struct Wrapper(Vec<String>);

impl fmt::Display for Wrapper {
    fn fmt(&self, f: &mut fmt::Formatter) -> fmt::Result {
        write!(f, "[{}]", self.0.join(", "))
    }
}

fn main() {
    let w = Wrapper(vec![String::from("hello"), String::from("world")]);
    println!("w = {}", w);
}
```

Блок коду 19-23: створення типу `Wrapper` навколо `Vec<String>` для реалізації `Display`

Реалізація `Display` використовує `self.0` для доступу до внутрішнього `Vec<T>`, оскільки `Wrapper` - це структура-кортеж, а `Vec<T>` - це елемент з індексом 0 в кортежі. Тоді ми можемо використати функціонал типу `Display` на `Wrapper`.

Недоліком використання цієї техніки є те, що `Wrapper` є новим типом, тож він не має методів значення, яке він містить. Ми мали б реалізувати всі методи `Vec<T>` безпосередньо на `Wrapper`, делегуючи всі методи `self.0`, що дозволить нам використовувати `Wrapper` точно як і `Vec<T>`. Якби ми хотіли, щоб новий тип мав кожен метод, який має внутрішній тип, то реалізація трейту `Deref` (про який йдеться у Розділі 15 у підрозділі “Використання розумних вказівників як звичайних посилань за допомогою трейта `Deref`”) для `Wrapper`, щоб повертав внутрішній тип, могла б бути розв'язанням проблеми. Якщо ж ми не хочемо, щоб тип `Wrapper` мав усі методи внутрішнього типу - наприклад, для обмеження поведінки типу `Wrapper` - то нам треба реалізувати потрібні нам методи вручну.

Цей паттерн "новий тип" також корисний навіть без залучення трейтів. Змінімо фокус і погляньмо на деякі поглиблені способи взаємодії з системою типів Rust. [ch10-02-traits.html#implementing-a-trait-on-a-type](https://pavloslav.github.io/RustBookUkr/print.html#implementing-a-trait-on-a-type) [ch10-02-traits.html#traits-defining-shared-behavior](https://pavloslav.github.io/RustBookUkr/print.html#traits-defining-shared-behavior)

Поглиблено про Типи

Система типів Rust має певні особливості, про які ми вже згадували, але не обговорювали. Почнемо з обговорення нових типів у цілому, розбираючи, чому нові типи корисні як типи. Тоді ми перейдемо до псевдонімів типів, функціоналу, подібному до нових типів, але з трохи іншою семантикою. Також ми обговоримо тип `!` і типи з динамічним розміром.

Використання Шаблону "Новий Тип" для Безпеки і Абстракції Типів

Примітка: цей підрозділ передбачає, що ви вже прочитали попередній підрозділ ["Використання паттерну "новий тип" для реалізації зовнішніх трейтів на зовнішніх типах."](#)

Паттерн "новий тип" також корисний для задач поза тими, які ми досі обговорили, включно зі статичним гарантуванням, що значення не переплутаються, і вказанням одиниць значення. Ви бачили приклад використання нових типів для позначення типів у Блоці коду 19-15: згадайте структури `Millimeters` і `Meters`, що обгортали значення `u32` у новий тип. Якщо ми напишемо функцію з параметром типу `Millimeters`, то не зможемо скомпілювати програму, де випадково спробуємо викликати цю функцію зі значенням типу `Meters` або просто `u32`.

Ми також можемо використовувати патерн "новий тип", щоб абстрагуватися від деталей реалізації типу: новий тип може надати публічний API, що відрізняється від API приватного внутрішнього типу.

Нові типи також можуть приховувати внутрішню реалізацію. Наприклад, ми могли б надати тип `People` для того, щоб загорнути `HashMap<i32, String>`, що зберігає ID людини, пов'язаний з її ім'ям. Код, що використовує `People`, взаємодітиме лише з наданим нами публічним API, таким як метод, щоб додати ім'я - стрічку до колекції `People`; тому коду не треба знати, що внутрішньо ми присвоюємо іменам ID типу `i32`. Паттерн "новий тип" є простим способом досягти інкапсуляції, щоб приховати деталі реалізації, про яку ми говорили у підрозділі ["Інкапсуляція, яка приховує деталі реалізації"](#) Розділу 17.

Створення Синонімів Типів з Псевдонімами Типів

Rust надає можливість проголосити *псевдонім типу*, щоб надати типу, що існує, іншу назву. Для цього використовується ключове слово `type`. Наприклад, ми можемо створити псевдонім `Kilometers` для `i32` ось таким чином:

```
type Kilometers = i32;
```

Тепер псевдонім `Kilometers` є *синонімом* для `i32`; на відміну від типів `Millimeters` і `Meters`, які ми створили в Блоці коду 19-15, `Kilometers` не є окремим новим типом. Значення, що мають тип `Kilometers` будуть оброблятися так само як і значення типу `i32`:

```
type Kilometers = i32;

let x: i32 = 5;
let y: Kilometers = 5;

println!("x + y = {}", x + y);
```

Оскільки `Kilometers` та `i32` є одним типом, ми можемо додавати значення обох типів, і ми можемо передавати значення `Kilometers` у функції, що приймають параметром `i32`. Однак, за допомогою цього методу ми не отримуємо переваг перевірки типів, які ми отримали від паттерну "новий тип", про який говорили раніше. Іншими словами, якщо ми десь переплутаємо значення `Kilometers` і `i32`, компілятор не повідомить нам про помилку.

Основним випадком використання синонімів типу є зменшення повторень. Наприклад, у нас може бути такий довгий тип:

```
Box<dyn Fn() + Send + 'static>
```

Писати цей довгий тип у сигнатурах функцій і анотаціях типів по всьому коду утомлює і призводить до помилок. Уявімо, що в нас є проєкт, повний коду, подібного до Блоку коду 19-24.


```
let f: Box<dyn Fn() + Send + 'static> = Box::new(|| println!("hi"));

fn takes_long_type(f: Box<dyn Fn() + Send + 'static>) {
    // --snip--
}

fn returns_long_type() -> Box<dyn Fn() + Send + 'static> {
    // --snip--
}
```

Блок коду 19-25: використання довгого типу в багатьох місцях

Псевдонім типу робить цей код більш керованим шляхом зменшення повторень. У Блоці коду 19-25 ми ввели псевдонім з назвою `Thunk` для багатослівного типу і можемо замінити всі використання цього типу на короткий псевдонім `Thunk`.

```
type Thunk = Box<dyn Fn() + Send + 'static>;

let f: Thunk = Box::new(|| println!("hi"));

fn takes_long_type(f: Thunk) {
    // --snip--
}

fn returns_long_type() -> Thunk {
    // --snip--
}
```

Блок коду 19-25: введення псевдоніма типу `Thunk` для зменшення повторень

Цей код набагато легше читати і писати! Вибір осмисленої назви для псевдоніма типу також може допомогти передати ваш намір (*think* означає код для обчислення пізніше, то ж це доречна назва для замикання, що зберігається).

Псевдоніми типів також широко використовуються з типом `Result<T, E>` для зменшення повторень. Подивімося на модуль `std::io` зі стандартної бібліотеки. Операції введення-виведення часто повертають `Result<T, E>`, щоб обробити ситуації, де операції не вдалися. Ця бібліотека має структуру `std::io::Error`, що представляє всі можливі помилки введення-виведення. Багато з функцій з `std::io` повертають `Result<T, E>`, де `E` - це `std::io::Error`, наприклад ці функції у трейті `Write`:

```
use std::fmt;
use std::io::Error;

pub trait Write {
    fn write(&mut self, buf: &[u8]) -> Result<usize, Error>;
    fn flush(&mut self) -> Result<(), Error>;

    fn write_all(&mut self, buf: &[u8]) -> Result<(), Error>;
    fn write_fmt(&mut self, fmt: fmt::Arguments) -> Result<(), Error>;
}
```

`Result<..., Error>` повторюється багато разів. Тому `std::io` проголошує псевдонім цього типу:

```
type Result<T> = std::result::Result<T, std::io::Error>;
```

Оскільки це проголошення знаходиться в модулі `std::io`, ми можемо використовувати повний кваліфікований псевдонім `std::io::Result<T>`, тобто `Result<T, E>`, в якому `E` визначено як `std::io::Error`. Сигнатури функцій трейту `Write` в результаті виглядають ось так:

```
pub trait Write {
    fn write(&mut self, buf: &[u8]) -> Result<usize>;
    fn flush(&mut self) -> Result<()>;

    fn write_all(&mut self, buf: &[u8]) -> Result<()>;
    fn write_fmt(&mut self, fmt: fmt::Arguments) -> Result<()>;
}
```

Псевдоніми типів допомагають у два способи: спрощують написання коду і надають нам цілісний інтерфейс у всьому `std::io`. Оскільки це псевдонім, це лише звичайний `Result<T, E>`, що означає, що ми можемо використовувати для нього будь-які методи, що працюють з `Result<T, E>`, а також особливий синтаксис на кшталт оператора `?`.

Тип "Ніколи", що Ніколи не Повертається

Rust має спеціальний тип, що зветься `!`, також відомий у термінології теорії типів як *empty type*, бо він не має значень. Ми радше називаємо його *тип "ніколи"*, бо він стоїть замість типу, що повертається, коли функція ніколи не повертає значення. Ось приклад:

```
fn bar() -> ! {
    // --snip--
}
```

Цей код читається як "функція `bar` ніколи не повертає." Функції, що ніколи не повертають, зветься *розбіжними функціями*. Ми не можемо створювати значень типу `!`, тож `bar` ніколи не може нічого повернути.

Але яка користь від типу, для якого неможливо створити значення? Згадайте код з Блоку коду 2-5, частину гри "Відгадай число"; ми відтворимо частину його тут, у Блоці коду 19-26.

```
let guess: u32 = match guess.trim().parse() {
    Ok(num) => num,
    Err(_) => continue,
};
```

Блок коду 19-26: `match` з рукавом, що закінчується на `continue`

Цього разу ми пропустили деякі деталі в цьому коді. У Розділі 6 у підрозділі "Конструкція управління `match`" ми говорили, що рукави `match` мають усі повертати один і той самий тип. Тож, наприклад, цей код не працює:

```
let guess = match guess.trim().parse() {
    Ok(_) => 5,
    Err(_) => "hello",
};
```

Тип `guess` у цьому коді має бути цілим числом і стрічкою, а Rust вимагає, щоб `guess` був лише одного типу. То що ж повертає `continue`? Як у нас вийшло повернути `u32` з одного рукава та мати інший рукав, що закінчується на `continue` у Блоці коду 19-26?

Як ви вже мабуть здогадалися, `continue` має значення `!`. Тобто, коли Rust обчислює тип `guess`, він перевіряє обидва рукави `match`, перший зі значенням `u32` і другий зі значенням `!`. Оскільки `!` ніколи не має значення, Rust вирішує, що типом `guess` є `u32`.

Формальним ця поведінка описується так: вираз типу `!` може бути приведений до будь-якого іншого типу. Ми можемо поставити `continue` в кінці рукава `match`, бо `continue` не повертає значення; натомість, він передає управління назад на початок циклу, тож у випадку `Err` ми ніколи не присвоїмо значення `guess`.

Тип "ніколи" також використовується у макросі `panic!`. Згадайте функцію `unwrap`, яку ми викликаємо для значень типу `Option<T>`, щоб отримати значення чи запанікувати; ось її визначення:

```
impl<T> Option<T> {
    pub fn unwrap(self) -> T {
        match self {
            Some(val) => val,
            None => panic!("called `Option::unwrap()` on a `None` value"),
        }
    }
}
```

У цьому коді відбувається те ж саме, що й у `match` з Блоку коду 19-26: Rust бачить, що `val` має тип `T` а `panic!` має тип `!`, отже, результат усього виразу `match` є `T`. Цей код працює, оскільки `panic!` не виробляє значення; він завершує програму. У випадку `None`, ми не повертаємо значення з `unwrap`, тож цей код є коректним.

Ще один останній вираз, що має значення `!` - це `loop`:

```
print!("forever ");

loop {
    print!("and ever ");
}
```

Тут цикл ніколи не закінчується, тож значенням виразу є `!`. Однак це не було б так, якби ми додали `break`, оскільки цикл завершиться, коли дістанеться до `break`.

Типи з Динамічним Розміром і Трейт `Sized`

Rust має знати деякі деталі про типи, такі, як скільки місця розподілити під значення певного типу. Це лишає один куток системи типів, на перший погляд, незрозумілим: концепцію *типів з динамічним розміром*. Ці типи, які іноді зветься *DST (dynamically sized types)* чи *безрозмірні типи*, дозволяють нам писати код з використанням значень, розмір яких ми можемо дізнатися лише під час виконання.

Копнімо деталі типу з динамічним розміром, що зветься `str`, який ми використовуємо скрізь у книзі. Саме так, не `&str`, а `str` як такий, що є DST. Ми не можемо знати довжину стрічки до часу виконання, що означає, що ми не можемо створити змінну типу `str`, ані прийняти аргумент типу `str`. Розгляньмо такий код, що не працює:

```
let s1: str = "Hello there!";
let s2: str = "How's it going?";
```

Rust має знати, скільки пам'яті виділяти для будь-якого значення певного типу, і всі значення цього типу мають використовувати однакову кількість пам'яті. Якби Rust

дозволив нам написати такий код, ці два значення `str` мали б займати однакову кількість місця в пам'яті. Але вони мають різні довжини: `s1` потребує 12 байтів пам'яті, а `s2` - 15. Ось чому неможливо створити змінну, що міститиме тип з динамічним розміром.

То що ж нам робити? В цьому випадку ви вже знаєте відповідь: ми робимо типи `s1` і `s2` `&str` замість `str`. Згадайте з підрозділу "Стрічкові слайси" Розділу 4, що структура даних слайс зберігає лише початкове положення і довжину слайса. Тож хоча `&T` і є одним значенням, що зберігає адресу в пам'яті, де знаходиться `T`, `&str` є двома значеннями: адресою `str` і її довжиною. Таким чином ми можемо знати розмір значення `&str` під час компіляції: два розміри `usize`. Тобто ми завжди знаємо розмір `&str`, не важливо якою довгою буде стрічка, на яку воно посилається. В цілому типи з динамічним розміром у Rust використовуються саме у такий спосіб: вони мають додаткову крихту метаданих, що зберігають розмір динамічної інформації. Золоте правило типів із динамічним розміром є те, що ми завжди маємо ховати значення типів з динамічним розміром за вказівником певного роду.

Ми можемо комбінувати `str` з усіма видами вказівників: наприклад, `Box<str>` чи `Rc<str>`. Фактично ви вже бачили це раніше, але з іншими типами з динамічним розміром: трейтами. Будь-який трейт є типом із динамічним розміром, до якого ми можемо звертатися за допомогою назви трейту. У Розділі 17, підрозділі "Використання трейт-об'єктів, які допускають значення різних типів", ми згадали, що для використання трейтів як трейтових об'єктів ми маємо сховати їх за вказівником, таким як

`&dyn Trait` чи `Box<dyn Trait>` (`Rc<dyn Trait>` теж підійде).

Щоб працювати з DST, Rust надає трейт `Sized` для визначення, чи розмір типу відомий під час компіляції. Цей трейт автоматично реалізується для усього, чий розмір є відомим під час компіляції. Крім того, Rust неявно додає обмеження `Sized` на кожну узагальнену функцію. Тобто визначення ось таке узагальненої функції:

```
fn generic<T>(t: T) {
    // --snip--
}
```

насправді розглядається, ніби ми написали таке:

```
fn generic<T: Sized>(t: T) {
    // --snip--
}
```

За замовчуванням узагальнені функції працюють лише з типами, чий розмір є відомим під час компіляції. Однак ви можете застосувати наступний спеціальний синтаксис для послаблення цього обмеження:

```
fn generic<T: ?Sized>(t: &T) {  
    // --snip--  
}
```

Трейтове обмеження `?Sized` означає “`T` може бути чи не бути `Sized`” і цей запис знімає обмеження за замовчуванням, що узагальнені типи мусять мати відомий розмір під час компіляції. Синтаксис `?Trait` із цим значенням можна застосовувати лише для `Sized`, але не для решти трейтів.

Також зауважте, що ми змінили тип параметра `t` з `T` на `&T`. Оскільки тип може не бути `Sized`, ми маємо використати його, сховавши за якогось роду вказівником. У цьому випадку ми обрали посилання.

Далі ми поговоримо про функції та замикання! [ch17-01-what-is-oo.html#encapsulation-that-hides-implementation-details](https://pavloslav.github.io/RustBookUkr/print.html#ch17-01-what-is-oo.html#encapsulation-that-hides-implementation-details) [ch06-02-match.html#the-match-control-flow-operator](https://pavloslav.github.io/RustBookUkr/print.html#ch06-02-match.html#the-match-control-flow-operator) [ch17-02-trait-objects.html#using-trait-objects-that-allow-for-values-of-different-types](https://pavloslav.github.io/RustBookUkr/print.html#ch17-02-trait-objects.html#using-trait-objects-that-allow-for-values-of-different-types)

Поглиблено про Функції та Замикання

Цей підрозділ поглиблено досліджує функціонал, що стосується функцій та замикань, і включає з вказівники на функції та повернення замикань.

Вказівники на Функції

Ми говорили про те, як передати замикання до функцій; ви також можете передати звичайні функції до функцій! Ця техніка є корисною, коли ви хочете передати вже визначену функцію, а не визначати нове замикання. Функції приводяться до типу `fn` (у нижньому регістрі), не плутайте з трейтом замикань `Fn`. Тип `fn` зветься *вказівником на функцію*. Передача функцій за допомогою вказівників на функції дозволяє вам використовувати функції як аргументи до інших функцій.

Синтаксис для зазначення, що параметр є вказівником на функцію, схожий на замикання, як показано у Блоці коду 19-27, де ми визначили функцію `add_one`, яка додає один до свого параметра. Функція `do_twice` приймає два параметри: вказівник на функцію для будь-якої функції, що приймає параметр `i32` і повертає `i32`, та інше значення `i32`. Функція `do_twice` викликає функцію `f` двічі, передаючи їй значення `arg`, а потім додає результати двох викликів. Функція `main` викликає `do_twice` з аргументами `add_one` та `5`.

Файл: src/main.rs

```
fn add_one(x: i32) -> i32 {
    x + 1
}

fn do_twice(f: fn(i32) -> i32, arg: i32) -> i32 {
    f(arg) + f(arg)
}

fn main() {
    let answer = do_twice(add_one, 5);

    println!("The answer is: {}", answer);
}
```

Блок коду 19-27: використання типу `fn` для прийняття вказівник на функцію як аргументу

Цей код виводить `The answer is: 12`. Ми вказуємо, що параметр `f` у `do_twice` є `fn`, що приймає один параметр `i32` і повертає `i32`. Тоді ми можемо викликати `f` у тілі

`do_twice`. У `main` ми можемо передати назву функції `add_one` першим аргументом `do_twice`.

На відміну від замикань, `fn` є типом, а не трейтом, тож ми вказуємо `fn` як тип параметра безпосередньо, а не заявляємо узагальнений параметр типу одного з трейтів `Fn`, як обмеження трейту.

Вказівники на функції реалізують усі три трейти замикань (`Fn`, `FnMut` і `FnOnce`), тобто ви завжди можете передати вказівник на функції аргументом до функції, що очікує на замикання. Найкраще писати функції, використовуючи узагальнений тип і один з трейтів замикань, щоб ваші функції могли приймати і функції, і замикання.

До слова, один приклад того, де ви хочете приймати лише `fn`, а не замикання - це коли ви надаєте інтерфейс зовнішньому коду, що не має замикань: функції C можуть приймати функції як аргументи, але у C немає замикань.

Як приклад того, де ви можете використовувати або визначене на місці замикання, або функцію, подивімося на використання методу `map` з трейту `Iterator` у стандартній бібліотеці. Щоб використати функцію `map` для перетворення вектора чисел на вектор стрічок, ми можемо використати замикання, ось так:

```
let list_of_numbers = vec![1, 2, 3];
let list_of_strings: Vec<String> =
    list_of_numbers.iter().map(|i| i.to_string()).collect();
```

Або ж ми можемо передати функцію аргументом до `map` замість замикання, ось так:

```
let list_of_numbers = vec![1, 2, 3];
let list_of_strings: Vec<String> =
    list_of_numbers.iter().map(ToString::to_string).collect();
```

Зверніть увагу, що ми повинні використовувати повністю кваліфікований синтаксис, про який ми говорили раніше у підрозділі "Поглиблено про трейти", бо існує багато доступних функцій, що звуться `to_string`. Тут ми використовуємо функцію `to_string`, визначену у трейті `ToString`, який стандартна бібліотека реалізує для будь-якого типу, що реалізує `Display`.

Згадайте з підрозділу "Значення еnumів" Розділу 6, що назва кожного варіанту еnumу, який ми визначаємо, також стає функцією ініціалізації. Ми можемо використовувати ці функції ініціалізації як вказівники на функції, які реалізують трейти замикань, що значить, що функції ініціалізації можуть бути аргументами для методів, що приймають замикання, ось так:


```
enum Status {  
    Value(u32),  
    Stop,  
}  
  
let list_of_statuses: Vec<Status> = (0u32..20).map(Status::Value).collect();
```

Тут ми створюємо екземпляри `Status::Value`, використовуючи кожне значення `u32` у діапазоні, для якого викликається `map`, використовуючи функцію ініціалізації `Status::Value`. Деякі люди надають перевагу цьому стилю, а деякі люди вважають за краще використовувати замикання. Вони компілюються в однаковий код, тому використовуйте стиль, зрозуміліший для вас.

Повертання Замикань

Закриття представлено трейтами, що означає, що ви не можете повертати замикання безпосередньо. У більшості випадків, коли ви могли б повернути трейт, ви можете натомість використовувати конкретний тип, який реалізує трейт, як значення, що повертає функція. Однак ви не можете зробити цього з замиканнями, оскільки у них немає конкретного типу, який можна було б повернути; наприклад, ви не можете використовувати вказівник на функцію `fn` як типу, що повертається.

Наступний код намагається повернути замикання безпосередньо, але він не компілюється:

```
fn returns_closure() -> dyn Fn(i32) -> i32 {  
    |x| x + 1  
}
```

Ось помилка компілятора:

Макроси

Ми використовували макроси на кшталт `println!` по всій книзі, але досі повністю не розкривали, що таке макроси і як вони працюють. Термін *макрос* стосується родини особливостей Rust: *декларативні* макроси за допомогою `macro_rules!` і три типи **процедурних** макросів:

- Користувацькі макроси `#[derive]`, які визначають код, що додається з атрибутом `derive`, застосованим на структурах та енумах
- Атрибутоподібні макроси, що визначають користувацькі атрибути, застосовані до будь-чого
- Функцієподібні макроси, що виглядають як функції, але оперують переданими ним як аргумент мовними конструкціями

Ми поговоримо про кожен з них по черзі, але спершу погляньмо, чому нам взагалі потрібні макроси, коли ми вже маємо функції.

Відмінність між Макросами та Функціями

Зasadничо макроси є способом писати код, що пише інший код, що також відомо як *метапрограмування*. У Додатку C ми обговорюємо атрибут `derive`, який генерує для вас реалізацію різних трейтів. Ми також використовували макроси `println!` і `vec!` по всій книзі. Всі ці макроси *розгортаються*, виробляючи більше коду, ніж написаний вами вручну.

Метапрограмування є корисним для зменшення кількості коду, що вам треба писати та підтримувати, що також є однією з ролей функцій. Однак макроси мають деякі додаткові здібності, яких бракує функціям.

Сигнатура функції має проголосити число і тип її параметрів. Макрос, з іншого боку, може приймати довільне число параметрів: ми можемо викликати `println!("hello")` з одним аргументом чи `println!("hello {}", name)` з двома. Також макроси розгортаються до того, як компілятор інтерпретує значення коду, тож макрос може, наприклад, реалізувати трейт на заданому типі. Функція не може такого, бо її викликають під час виконання, а трейт має бути реалізованим під час компіляції.

Недоліком реалізації макросу замість функції є те, що визначення макросів складніші, ніж визначення функцій, бо ви пишете код на Rust, що пише код на Rust. Через таку

опосередкованість визначення макросів у цілому складніше читати, розуміти та підтримувати, ніж визначення функцій.

Ще однією важливою відмінністю між макросами та функціями є те, що ви маєте визначити макроси або принести їх в область видимості до їхнього виклику у файлі, на відміну від функцій, які ви можете визначити будь-де і викликати звідки завгодно.

Декларативні Макроси, Проголошені за допомогою `macro_rules!`, для Загального Метапрограмування

Найчастіше використана форма макросів у Rust - *декларативні макроси*. Їх також іноді називають "макросами за прикладом", "макросами `macro_rules!`" чи просто "макросами." За своєю суттю декларативні макроси дозволяють вам написати щось подібне до виразу Rust `match`. Як говорилося в Розділі 6, вирази `match` - це керівні структури, які приймають вираз, зіставляють результат обчислення виразу з шаблонами, а потім виконують код, пов'язаний з відповідним шаблоном. Макроси так само порівнюють значення з шаблонами, які пов'язані з певним кодом: в цій ситуації значенням є літерал початкового коду Rust, переданого макросу; шаблони зіставляються зі структурою цього початкового коду; і код, пов'язаний з кожним шаблоном, коли збігається, замінює код, переданий в макрос. Це все відбувається під час компіляції.

Щоб визначити макрос, використовується конструкція `macro_rules!`. Дослідимо, як користуватися `macro_rules!`, подивившись, як визначений макрос `vec!`. В Розділі 8 розповідалося, як використовувати макрос `vec!` для створення нового вектора з конкретними значеннями. Наприклад, наступний макрос створить новий вектор, що містить три цілі числа:

```
let v: Vec<u32> = vec![1, 2, 3];
```

Ми також можемо використати макрос `vec!` для створення вектора з двох цілих чисел або вектора з 5 стрічкових слайсів. Ми б не змогли скористатися функцією, щоб зробити те саме, оскільки не знали б кількості або типу значень наперед.

Блок коду 19-28 показує трохи спрощене визначення макросу `vec!`.

Файл: `src/lib.rs`

```
#[macro_export]
macro_rules! vec {
    ( $( $x:expr ),* ) => {
        {
            let mut temp_vec = Vec::new();
            $(
                temp_vec.push($x);
            )*
            temp_vec
        }
    };
}
```

Блок коду 19-28: спрощена версія визначення макросу `vec!`

Примітка: Справжнє визначення макросу `vec!` зі стандартної бібліотеки містить спершу код для розподілу необхідної кількості пам'яті. Цей код є оптимізацією, яку ми не включаємо тут для спрощення прикладу.

Анотація `#[macro_export]` вказує, що цей макрос слід зробити доступним кожного разу, коли крейт, у якому його визначено вводиться до області видимості. Без цієї анотації макрос не було б введено до області видимості.

Потім ми почнемо визначення макросу за допомогою `macro_rules!` і назви макросу, який ми визначаємо, без знаку оклику. За назвою, у цьому випадку `vec`, слідує фігурні дужки, що позначають тіло визначення макросу.

Структура тіла `vec!` подібна до структури виразу `match`. Тут ми маємо один рукав із шаблоном `( $( $x:expr ),* )`, за яким іде `=>` і блок коду, пов'язаний із цим шаблоном. Якщо шаблон зіставляється, буде видано пов'язаний блок коду. Оскільки це є єдиним шаблоном у цьому макросі, є лише один коректний спосіб зіставлення; будь-який інший шаблон призведе до помилки. Складніші макроси матимуть більше ніж один рукав.

Правильний синтаксис шаблону у макросах відрізняється від синтаксису шаблону, розглянутого у Розділі 18, оскільки шаблони макросів зіставляються зі структурою коду Rust, а не значеннями. Розберімо, що означають фрагменти шаблону в Блоці коду 19-28; повний синтаксис шаблонів макросів ви можете подивитися в [Rust Reference](https://rust-lang.org/reference/macros).

Спочатку ми використовуємо набір дужок для того, щоб охопити весь шаблон. Ми використовуємо знак долара (`$`) для проголошення змінної у системі макросів, що міститиме код Rust, що відповідає шаблону. Знак долара дає зрозуміти, що це змінна макросу, а не звичайна змінна Rust. Далі іде набір дужок, що містять значення, що відповідають шаблону в дужках, для використання в коді для заміщення. У `$( )`

знаходиться `$x:expr`, що зіставляється з будь-яким виразом Rust і дає цьому виразу назву `$x`.

Кома після `$()` позначає, що символ-розділювач кома може опціонально з'явитися після коду, що зіставляється з кодом у `$()`. `*` позначає, що шаблон зіставляється з нулем чи більше того, що іде перед `*`.

Коли ми викликаємо цей макрос за допомогою `vec![1, 2, 3];`, шаблон `$x` зіставляється три рази з трьома виразами `1`, `2` і `3`.

Тепер погляньмо на шаблон у тілі коду, пов'язаного з цим рукавом: `temp_vec.push()` у `$()*` генерується для кожної частини, що зіставляється з `$()` у шаблоні нуль чи більше разів, залежно від того, скільки разів зіставляється шаблон. `$x` замінюється у кожному зіставленому виразі. Коли ми викликаємо цей макрос за допомогою `vec![1, 2, 3];`, згенерований код, що замінює виклик макросу, буде таким:

```
{  
    let mut temp_vec = Vec::new();  
    temp_vec.push(1);  
    temp_vec.push(2);  
    temp_vec.push(3);  
    temp_vec  
}
```

Ми визначили макрос, який може прийняти будь-яку кількість аргументів будь-якого типу і може згенерувати код для створення вектора, що містить зазначені елементи.

Щоб дізнатися більше про те, як писати макроси, зверніться до документації в Інтернеті або інших ресурсів, таких як "[Маленька книжка макросів Rust](#)", яку розпочав Деніел Кіп та продовжує Лукас Вірт.

Процедурні Макроси для Генерації Коду з Атрибутів

Друга форма макросів - це *процедурні макроси*, які працюють більш схоже на функції (і є типом процедур). Процедурні макроси беруть певний код на вході, працюють над цим кодом і виробляють певний код на виході замість зіставлення з шаблонами і заміни коду іншим кодом, як роблять декларативні макроси. Три види процедурних макросів це користувацькі вивідні, атрибутоподібні та функцієподібні макроси, і всі працюють схожим чином.

При створенні процедурних макросів визначення мають розміщуватися у їхньому власному крейті з особливим типом крейта. Так зроблено зі складних технічних причин, які ми сподіваємося усунути в майбутньому. У Блоці коду 19-29 ми показуємо, як

визначити процедурний макрос, де `some_attribute` є заповнювачем для використання конкретного різновиду макросу.

Файл: `src/lib.rs`

```
use proc_macro;  
  
#[some_attribute]  
pub fn some_name(input: TokenStream) -> TokenStream {  
}
```

Блок коду 19-29: приклад визначення процедурного макросу

Функція, що визначає процедурний макрос, приймає `TokenStream` на вхід і продукує `TokenStream` на виході. Тип `TokenStream` визначений у крейті `proc_macro`, що постачається разом із Rust, і являє собою послідовність токенів. Це основа макросу: початковий код, з яким працює макрос, є вхідним `TokenStream`, а код, що макрос продукує, є вихідним `TokenStream`. Функція також має атрибут, що визначає, який вид процедурного макросу ми створюємо. Можна мати багато видів процедурних макросів в одному крейті.

Подивімося на різні види процедурних макросів. Ми почнемо з користувальницького вивідного макросу, а потім пояснимо невеликі розбіжності, що роблять інші форми відмінними.

Як Писати Користувацький Макрос `derive`

Створімо крейт з назвою `hello_macro`, що визначає трейт з назвою `HelloMacro` з однією асоційованою функцією з назвою `hello_macro`. Замість примушувати користувачів крейту реалізовувати трейт `HelloMacro` для кожного з їхніх типів, ми надамо процедурний макрос, щоб користувачі могли анотувати свої типи `#[derive>HelloMacro)]` і отримувати реалізацію функції `hello_macro` за замовчуванням. Реалізація за замовчуванням виведе `Hello, Macro! My name is TypeName!`, де `TypeName` - це назва типу, для якого цей трейт визначено. Іншими словами, ми створимо крейт, за допомогою якого інші програмісти зможуть писати код на кшталт Блоку коду 19-30.

Файл: `src/main.rs`

```
use hello_macro::HelloMacro;
use hello_macro_derive::HelloMacro;

#[derive(HelloMacro)]
struct Pancakes;

fn main() {
    Pancakes::hello_macro();
}
```

Блок коду 19-30: Код, який користувачі нашого крейту зможуть писати, використовуючи наш процедурний макрос

Коли ми закінчимо, цей код виведе `Hello, Macro! My name is Pancakes!` Перший крок - це створити новий бібліотечний крейт, ось так:

```
$ cargo new hello_macro --lib
```

Далі ми визначаємо трейт `HelloMacro` і асоційовану функцію:

Файл: `src/lib.rs`

```
pub trait HelloMacro {
    fn hello_macro();
}
```

Ми маємо трейт і його функцію. Зараз користувач нашого крейта може реалізувати трейт для досягнення бажаної функціональності, ось так:

```
use hello_macro::HelloMacro;

struct Pancakes;

impl HelloMacro for Pancakes {
    fn hello_macro() {
        println!("Hello, Macro! My name is Pancakes!");
    }
}

fn main() {
    Pancakes::hello_macro();
}
```

Однак для кожного типу, для якого хочеться використовувати `hello_macro`, треба написати блок реалізації, а ми хочемо позбавити їх від необхідності це робити.

Додатково, ми ще не в змозі надати реалізацію за замовчуванням для функції `hello_macro`, яка надрукує назву типу, для якого її реалізовано: Rust не має можливостей рефлексії, так що неможливо дізнатися назву типу під час виконання. Нам потрібен макрос для генерації коду під час компіляції.

Наступний крок - визначити процедурний макрос. На час написання цього, процедурні макроси мають міститися у своїх власних крейтах. Згодом це обмеження може бути зняте. За угодою, крейти і крейти для макросів мають бути такі: для крейту, що зветься `foo`, крейт з користувацьким вивідним макросом має зватися `foo_derive`. Почнімо новий крейт, що зветься `hello_macro_derive`, усередині нашого проєкту `hello_macro`:

```
$ cargo new hello_macro_derive --lib
```

Наші два крейти тісно пов'язані, тому ми створюємо крейт для процедурного макросу в каталозі нашого крейта `hello_macro`. Якщо ми змінимо визначення трейту в `hello_macro`, то мусимо також змінити реалізацію процедурного макросу в `hello_macro_derive`. Два крейти треба буде публікувати окремо, і програмістам, що використовують ці крейти, доведеться додавати обидва як залежності і вводити обидва до області видимості. Ми могли б натомість додати `hello_macro_derive` як залежність у `hello_macro` і реекспортувати код процедурного макросу. Однак те, як ми структурували проєкт, надає програмістам можливість використовувати `hello_macro` навіть якщо вони не хочуть мати функціонал `derive`.

Нам треба оголосити крейт `hello_macro_derive` як крейт процедурного макросу. Нам також знадобиться функціонал крейтів `syn` та `quote`, як ви побачите за хвилину, тому ми маємо додати їх як залежності. Додайте наступне у файл *Cargo.toml* для `hello_macro_derive`:

Файл: `hello_macro_derive/Cargo.toml`

```
[lib]
proc-macro = true

[dependencies]
syn = "1.0"
quote = "1.0"
```

Щоб почати визначення процедурного макросу, розмістіть код з Блоку коду 19-31 у файлі *src/lib.rs* з крейту `hello_macro_derive`. Зверніть увагу, що цей код не компілюватиметься, поки ми не додамо визначення для функції `impl_hello_macro`.

Файл: `hello_macro_derive/src/lib.rs`

```

use proc_macro::TokenStream;
use quote::quote;
use syn;

#[proc_macro_derive(HelloMacro)]
pub fn hello_macro_derive(input: TokenStream) -> TokenStream {
    // Construct a representation of Rust code as a syntax tree
    // that we can manipulate
    let ast = syn::parse(input).unwrap();

    // Build the trait implementation
    impl_hello_macro(&ast)
}

```

Блок коду 19-31: код, який потребують більшість крейтів з процедурними макросами для того, щоб обробляти код Rust

Зверніть увагу, що ми розділили код на функцію `hello_macro_derive`, що відповідає за аналіз `TokenStream`, і функцію `impl_hello_macro` що відповідає за перетворення синтаксичного дерева: це робить написання процедурного макросу зручнішим. Код у зовнішній функції (у цьому випадку `hello_macro_derive`) буде однаковим для майже кожного крейта процедурного макросу, що ви зустрінете або створення. Код, який ви вкажете у тілі внутрішньої функції (у цьому випадку `impl_hello_macro`), буде різним залежно від призначення вашого процедурного макросу.

Ми додали три нові крейти: `proc_macro`, `syn`, а `quote`. Крейт `proc_macro` постачається з Rust, тож нам не потрібно додавати його до залежностей у *Cargo.toml*. Крейт `proc_macro` - це API компілятора, що дозволяє нам читати та маніпулювати кодом Rust у нашому коді.

Крейт `syn` розбирає код Rust зі стрічки у структуру даних, з якою ми можемо виконувати операції. Крейт `quote` перетворює структуру даних `syn` назад у код Rust. Ці крейти дуже спрощують розбір будь-якого коду Rust, який нам треба обробити: написати повний аналізатор коду Rust - це непросте завдання.

Функцію `hello_macro_derive` буде викликано, коли користувач нашої бібліотеки зазначить `#[derive(HelloMacro)]` для типу. Це можливо, тому що ми анотували функцію `hello_macro_derive` за допомогою `proc_macro_derive` і вказали назву `HelloMacro`, яка відповідає назві нашого трейта; цій угоді слідує більшість процедурних макросів.

Функція `hello_macro_derive` спочатку перетворює `input` з `TokenStream` у структуру даних, яку ми потім можемо інтерпретувати та виконувати операції над нею. І тут вступає в гру `syn`. Функція `parse` із `syn` приймає `TokenStream` і повертає структуру `DeriveInput`, що представляє розібраний код Rust. Блок коду 19-32 показує відповідні частини структури `DeriveInput`, яку ми отримали розбором стрічки `struct Pancakes;`:

```

DeriveInput {
    // --snip--

    ident: Ident {
        ident: "Pancakes",
        span: #0 bytes(95..103)
    },
    data: Struct(
        DataStruct {
            struct_token: Struct,
            fields: Unit,
            semi_token: Some(
                Semi
            )
        }
    )
}

```

Блок коду 19-32: екземпляр `DeriveInput`, який ми отримаємо розбором коду, що має атрибут макросу з Блоку коду 19-30

Поля цієї структури показують, що код Rust, який ми розібрали, є одиничною структурою з `ident` (ідентифікатором, тобто назвою) `Pancakes`. У цієї структури є більше полів для опису різноманітних кодів Rust; зверніться до документації `syn` про `DeriveInput` для детальнішої інформації.

Незабаром ми визначатимемо функцію `impl_hello_macro`, де ми зберемо новий код Rust, який ми хочемо додати. Але перед тим зауважте, що вихід для нашого макросу - це також `TokenStream`. Повернутий `TokenStream` додається до коду, написаного користувачами нашого крейту, тому коли вони компілюватимуть свої крейти, вони отримають додатковий функціонал, наданий нами в зміненому `TokenStream`.

Ви могли не звернути увагу, що ми викликаємо `unwrap`, щоб функція `hello_macro_derive` запанікувала, якщо виклик функції `syn::parse` буде невдалим. Необхідно, щоб наш процедурний макрос панікував при помилках, бо функції `proc_macro_derive` мають повертати `TokenStream`, а не `Result`, щоб відповідати API процедурних макросів. Ми спростили цей приклад, використовуючи `unwrap`; у реальному коді ви маєте забезпечувати конкретніші повідомлення про помилки, описуючи, що саме пішло не так за допомогою `panic!` або `expect`.

Тепер, коли у нас є код, що перетворює анотований код Rust з `TokenStream` на екземпляр `DeriveInput`, згенеруймо код, що реалізує трейт `HelloMacro` на анотованому типі, як показано у Блоці коду 19-33.

Файл: `hello_macro_derive/src/lib.rs`

```
fn impl_hello_macro(ast: &syn::DeriveInput) -> TokenStream {
    let name = &ast.ident;
    let gen = quote! {
        impl HelloMacro for #name {
            fn hello_macro() {
                println!("Hello, Macro! My name is {}", stringify!(#name));
            }
        }
    };
    gen.into()
}
```

Блок коду 19-33: Реалізація трейту `HelloMacro` за допомогою розібраного коду Rust

Ми отримуємо екземпляр структури `Ident`, що містить назву (ідентифікатор) анотованого типу, використовуючи `ast.ident`. Структура у Блоці коду 19-32 показує, що коли ми запускаємо функцію `impl_hello_macro` на код з Блоку коду 19-30, `ident`, що ми отримуємо, має поле `ident` зі значенням `"Pancakes"`. Таким чином, змінна `name` з Блоку коду 19-33 міститиме екземпляр структури `Ident`, який при виведенні стане стрічкою `"Pancakes"`, назвою структури з Блоку коду 19-30.

Макрос `quote!` дозволяє нам визначати код Rust, який ми хочемо повернути. Компілятор очікує на щось відмінне від безпосереднього результату виконання макросу `quote!`, тож ми маємо конвертувати його у `TokenStream`. Ми це робимо викликом методу `into`, який `TokenStream`.

Макрос `quote!` також надає дуже круті механізми шаблонізації: ми можемо ввести `#name` і `quote!` замінить його на значення у змінній `name`. Ви можете навіть зробити деякі повторення, схожі на те, як працюють звичайні макроси. Зверніться до [документації крейту quote](#) для детального ознайомлення.

Ми хочемо, щоб наш процедурний макрос генерував реалізацію трейту `HelloMacro` для анотованого користувачем типу, який ми можемо отримати за допомогою `#name`. Реалізація трейту має одну функцію `hello_macro`, чие тіло містить функціональність, яку ми хочемо надати: виведення в консолі `Hello, Macro! My name is` і потім назву анотованого типу.

Використаний тут макрос `stringify!` вбудований у Rust. Він приймає вираз Rust, такий як `1 + 2`, і під час компіляції перетворює цей вираз у стрічковий літерал на кшталт `"1 + 2"`. Це відрізняється від макросів `format!` чи `println!`, які обчислюють вираз і потім перетворюють результат на `String`. Є можливість, що вхідний `#name` може бути виразом, який треба вивести буквально, тож ми використовуємо `stringify!`. Використання `stringify!` також економить розподілену пам'ять, бо перетворює `#name` на стрічковий літерал під час компіляції.

На цей момент `cargo build` має успішно завершуватися для обох `hello_macro` та `hello_macro_derive`. Під'єднаймо ці крейти до коду з Блок коду 19-30, щоб побачити процедурні макроси в дії! Створіть новий двійковий проєкт у каталозі *projects* командою `cargo new pancakes`. Нам треба додати `hello_macro` та `hello_macro_derive` як залежності до файлу *Cargo.toml* крейту `pancakes`. Якщо ви публікуєте ваші версії `hello_macro` та `hello_macro_derive` на crates.io, вони будуть звичайними залежностями; якщо ж ні, ви можете зазначити як залежності із `path`, ось так:

```
hello_macro = { path = "../hello_macro" }
hello_macro_derive = { path = "../hello_macro/hello_macro_derive" }
```

Помістіть код з Блоку коду 19-30 до *src/main.rs* і запустіть `cargo run`; має вивестися `Hello, Macro! My name is Pancakes!` Реалізація трейту `HelloMacro` з процедурного макросу була включена без потреби в реалізації у крейті `pancakes`; анотація `#[derive(HelloMacro)]` додала реалізацію трейту.

Далі дослідімо, як інші види процедурних макросів відрізняються від користувацьких вивідних макросів.

Атрибуто-Подібні Макроси

Атрибутоподібні макроси схожі на користувацькі вивідні макроси, але замість генерації коду для атрибута `derive` вони дозволяють вам створювати нові атрибути. Вони також гнучкіші: `derive` працює лише зі структурами та енумами; атрибути можна застосовувати також і до інших елементів, наприклад функцій. Ось приклад використання атрибутоподібного макросу: скажімо, ви маєте атрибут з назвою `route`, що анотує функції при використанні фреймворку вебзастосунків:

```
#[route(GET, "/")]
fn index() {
```

Цей атрибут `#[route]` буде визначено фреймворком як процедурний макрос. Сигнатура функції, що визначає макрос, виглядатиме ось так:

```
#[proc_macro_attribute]
pub fn route(attr: TokenStream, item: TokenStream) -> TokenStream {
```

Тут ми маємо два параметри типу `TokenStream`. Перший для вмісту атрибута, тобто частини `GET, "/"`. Другий - це тіло елемента, до якого застосований атрибут, у цьому випадку `fn index() {}` і решта тіла функції.

Окрім цього, атрибутоподібні макроси працюють так само, як і користувацькі вивідні макроси: ви створюєте крейт із типом крейту `proc-macro` і реалізуєте функцію, що створює потрібний вам код!

Функціє-Подібні Макроси

Функцієподібні макроси визначають макроси, що виглядають, як виклики функцій. Подібно до макросів `macro_rules!`, вони гнучкіші за функції; наприклад, вони можуть приймати довільну кількість аргументів. Однак `macro_rules!` можуть бути визначені лише за допомогою синтаксису, схожого на `match`, про який ми говорили раніше у підрозділі [Декларативні макроси, проголошені за допомогою `macro\_rules!`, для загального метапрограмування](#). Функцієподібні макроси приймають параметр `TokenStream`, а їхнє визначення маніпулює цим `TokenStream` за допомогою коду Rust, як і два інші типи процедурних макросів. Прикладом функцієподібних макросів може бути макрос `sql!`, який міг би бути викликаний таким чином:

```
let sql = sql!(SELECT * FROM posts WHERE id=1);
```

Цей макрос розбирає інструкції SQL і перевіряє їх на синтаксичну коректність, що значно складніше, ніж обробка, яку може здійснювати `macro_rules!`. Макрос `sql!` був би визначений ось так:

```
#[proc_macro]
pub fn sql(input: TokenStream) -> TokenStream {
```

Це визначення схоже на сигнатуру користувацького вивідного макросу: ми отримаємо токени, що знаходяться в дужках, і повертаємо код, який нам треба створити.

Підсумок

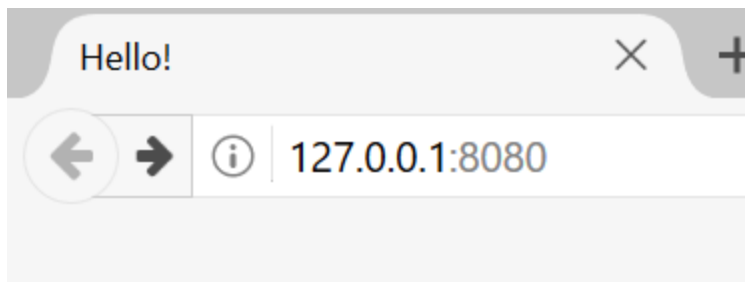
Хух! Тепер у вашому інструментарії є функціонал Rust, який ви навряд чи часто використовуватимете, але ви знатимете, що він доступний за певних обставин. Ми ознайомили вас із кількома складними темами, щоб зустрівши їх у пропозиціях у повідомленнях про помилки або в коді інших людей ви могли розпізнати ці концепції та синтаксис. Використовуйте цей розділ як довідник, що приведе вас до рішення.

Далі ми покажемо на практиці все, що ми обговорювали протягом усієї книги, і зробимо ще один проєкт!

Останній Проєкт: Збірка Багатопотокового Вебсервера

Це була довга подорож, та нарешті ми дійшли до кінця книжки. У цьому розділі ми разом зберемо ще один проєкт для демонстрації деяких концепцій, про які йшлося в останніх розділах, а також згадаємо деякі раніші уроки.

Для нашого останнього проєкту ми зробимо вебсервер, що скаже "привіт" і виглядає як Рисунок 20-1 у веббраузері.



Hello!

Hi from Rust

Рисунок 20-1: Наш останній спільний проєкт

Ось наш план збірки вебсервера:

1. Дізнатися трохи про TCP та HTTP.
2. Прослуховувати TCP-підключення на сокеті.
3. Розібрати невелику кількість HTTP-запитів.
4. Створити коректну HTTP відповідь.
5. Поліпшити пропускну здатність нашого сервера за допомогою пула потоків.

Перед тим як розпочати, ми маємо згадати про одну деталь: метод, який ми будемо використовувати, не буде найкращим способом створення вебсервера на Rust. Учасники спільноти опублікували ряд готових для використання у виробництві крейтів, доступних на crates.io, які забезпечують повніші реалізації вебсервера та пула потоків, ніж те, що ми збираємо. Однак, наш намір у цьому розділі допомогти вам вчитися, а не іти легким шляхом. Оскільки Rust є системною мовою програмування, ми можемо вибрати рівень

абстракції, з яким ми хочемо працювати й можемо піти на нижчий рівень, ніж це можливо чи практично в інших мовах. Тому ми напишемо базовий HTTP-сервер і пул потоків вручну, щоб ви могли вивчити загальні ідеї та техніки, застосовані в крейтах, які ви можете використати в майбутньому.

Збірка Однопоточкового Вебсервера

Ми розпочнемо з запуску однопоточкового вебсервера. Перш ніж почати, розглянемо короткий огляд протоколів, залучених до створення вебсерверів. Деталі цих протоколів лежать поза межами цієї книги, але короткий огляд надасть вам потрібну інформацію.

Два основні протоколи, залучені у вебсерверах, це *Протокол передачі гіпертексту (HTTP)* і *Протокол керування передаванням (TCP)*. Обидва протоколи є *протоколами відповіді на запит*, тобто клієнт ініціює запити, а сервер слухає запити та надає відповідь клієнту. Вміст цих запитів та відповідей визначається протоколами.

TCP - це протокол нижчого рівня, який описує деталі того, як інформація дістається від одного сервера до іншого, але не вказує, що це за інформація. HTTP є надбудовою над TCP і визначає зміст запитів та відповідей. Технічно можливо використовувати HTTP з іншими протоколами, але переважній більшості випадків HTTP відправляє його дані через TCP. Ми працюватимемо з необробленими байтами TCP та запитами і відповідями HTTP.

Прослуховування З'єднання TCP

Наш вебсервер має прослуховувати TCP-з'єднання, тож це буде першою частиною над якою ми працюватимемо. Стандартна бібліотека надає модуль `std::net`, який дозволить нам це зробити. Створімо новий проєкт у звичний спосіб:

```
$ cargo new hello
    Created binary (application) `hello` project
$ cd hello
```

Тепер введіть код з Блока коду 20-1 у `src/main.rs` для початку. Цей код прослуховує локальну адресу `127.0.0.1:7878` на вхідні потоки TCP. Коли він отримує вхідний потік, то виведе `Connection established!`.

Файл: `src/main.rs`

```
use std::net::TcpListener;

fn main() {
    let listener = TcpListener::bind("127.0.0.1:7878").unwrap();

    for stream in listener.incoming() {
        let stream = stream.unwrap();

        println!("Connection established!");
    }
}
```

Блок коду 20-1: прослуховування вхідних потоків і виведення повідомлення, коли прийняли потік

За допомогою `TcpListener` ми можемо прослуховувати TCP-з'єднання за адресою `127.0.0.1:7878`. У адресі розділ перед двокрапкою є IP-адресою, що представляє ваш комп'ютер (вона однакова для всіх комп'ютерів і не представляє конкретно комп'ютер автора), а `7878` - порт. Ми обрали цей порт з двох причин: HTTP зазвичай не приймається на цьому порті, тож наш сервер навряд чи конфліктуватиме з іншим вебсервером, що може працювати на вашій машині, і 7878 - це *rust*, набране на кнопках телефона.

Функція `bind` у цьому сценарії працює як функція `new` функція в тому, що повертає новий екземпляр `TcpListener`. Функція називається `bind` тому, що в організації мережі підключення до порту для прослуховування відоме як "прив'язування до порту."

Функція `bind` повертає `Result<T, E>`, що позначає, що зв'язування може бути невдалим. Наприклад, для підключення до порту 80 потрібні права адміністратора (не-адміністратори можуть слухати лише порти вище ніж 1023), так що, якщо ми намагались під'єднатися до порту 80 без прав адміністратора, зв'язування не спрацює. Зв'язування також не спрацює, наприклад, якщо ми запустимо два екземпляри нашої програми і відтак матимемо дві програми, що слухають один порт. Оскільки ми пишемо базовий сервер лише для навчальних цілей, ми не турбуватимемося про обробку таких помилок; натомість, ми використаємо `unwrap`, щоб зупинити програму, якщо виникнуть помилки.

Метод `incoming` для `TcpListener` повертає ітератор, який дає нам послідовність потоків (точніше, потоків типу `TcpStream`). Кожен *stream* представляє відкрите з'єднання між клієнтом і сервером. З'єднання *connection* є назвою для усього процесу запиту та відповіді, в якому клієнт підключається до сервера, сервер генерує відповідь, і сервер же закриває з'єднання. Таким чином ми читатимемо з `TcpStream`, щоб побачити, що надіслав клієнт, а потім писатимемо нашу відповідь до потоку, щоб відправити дані назад до клієнта. Загалом, цей цикл `for` буде обробляти кожне підключення по черзі і створить ряд потоків, які ми оброблятимемо.

Наразі наша обробка потоку складається з виклику `unwrap` для припинення нашої програми, якщо потік має будь-які помилки; якщо помилок немає, програма виводить повідомлення. У наступному блоці коду ми додамо більше функціональності для варіанту вдалого з'єднання. Причина, з якої ми можемо отримувати помилки з методу `incoming`, коли клієнт підключається до сервера це те, що ми насправді ітеруємо не по з'єднаннях. Натомість ми ітеруємо по *спробах з'єднання*. З'єднання може бути невдалим з ряду причин, багато з них специфічні для різних операційних систем. Наприклад, багато операційних систем мають обмеження на кількість одночасних відкритих підключень, які вони можуть підтримувати; нове спроба підключення після цієї кількості призводитиме до помилки, поки якісь з відкритих підключень не закриються.

Спробуймо запустити цей код! Викличте `cargo run` у терміналі та завантажте `127.0.0.1:7878` у веббраузері. Браузер повинен показати повідомлення про помилку на кшталт "З'єднання скинуто", оскільки сервер поки що не надсилає жодних даних. Але поглянувши в термінал, ви маєте побачити кілька повідомлень, які ми виводимо, коли браузер з'єднується із сервером!

```
Running `target/debug/hello`  
Connection established!  
Connection established!  
Connection established!
```

Іноді ви бачитимете кілька виведених повідомлень на один запит браузера; причина може бути в тому, що браузер запитує сторінку, а також деякі інші ресурси на кшталт піктограми *favicon.ico*, що показується у вкладці браузера.

Також можливо, що браузер намагається з'єднатися із сервером багато разів, бо сервер не надіслав у відповідь жодних даних. Коли `stream` виходить з області видимості й очищується в кінці циклу, з'єднання закривається, бо це є частиною реалізації `drop`. Браузери іноді намагаються повторно з'єднатися із закритими підключеннями, оскільки проблема може бути тимчасовою. Але важливим тут є те, що ми успішно отримали TCP-з'єднання!

Не забудьте зупинити програму, натиснувши `ctrl-c`, коли ви закінчили працювати з певною версією коду. Потім перезапустіть програму, запустивши команду `cargo run` після того, як робите кожен набір змін у коді для того, щоб переконатися, що у вас працює найновіший код.

Читання Запиту

А тепер реалізуймо функціональність для читання запиту з браузера! Для поділу інтересів - спершу встановлення з'єднання, а потім вживання якихось дій зі з'єднанням, ми

почнемо нову функцію для обробки з'єднань. У цій новій функції `handle_connection` ми прочитаємо дані з потоку TCP і виведемо їх, щоб ми могли побачити дані, що пересилаються з браузера. Змініть код, щоб він виглядав як у Блоці коду 20-2.

Файл: `src/lib.rs`

```
use std::{
    io::{prelude::*, BufReader},
    net::{TcpListener, TcpStream},
};

fn main() {
    let listener = TcpListener::bind("127.0.0.1:7878").unwrap();

    for stream in listener.incoming() {
        let stream = stream.unwrap();

        handle_connection(stream);
    }
}

fn handle_connection(mut stream: TcpStream) {
    let buf_reader = BufReader::new(&mut stream);
    let http_request: Vec<_> = buf_reader
        .lines()
        .map(|result| result.unwrap())
        .take_while(|line| !line.is_empty())
        .collect();

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

Блок коду 20-2: читання з `TcpStream` і виведення даних

Ми вносимо `std::io::prelude` і `std::io::BufReader` до області видимості, щоб отримати доступ до трейтів і типів, що дозволяють нам читати і писати до потоку. У циклі `for` у функції `main` замість того, щоб виводити повідомлення про те, що ми встановили з'єднання, тепер ми викликаємо нову функцію `handle_connection` і передаємо їй `stream`.

У функції `handle_connection` ми створюємо новий екземпляр `BufReader`, який огортає мутабельне посилання на `stream`. `BufReader` додає буферизацію, керуючи викликами до трейтових методів `std::io::Read` замість нас.

Ми створюємо змінну з назвою `http_request` для збору рядків запиту, що браузер відправляє на наш сервер. Ми позначаємо, що хочемо зібрати ці рядки у вектор, додавши анотацію типу `Vec<_>`.

`BufReader` реалізує трейт `std::io::BufRead`, що надає метод `lines`. Метод `lines` повертає ітератор `Result<String, std::io::Error>`, розділяючи потік даних кожного разу, коли він бачить байт нового рядка. Щоб отримати кожен `String`, ми відображаємо і робимо `unwrap` для кожного `Result`. `Result` може бути помилкою, якщо дані не є коректним UTF-8 або виникли проблеми із читанням з потоку. Знову ж таки, готова програма повинна обробляти ці помилки більш майстерно, але для простоти ми просто зупиняємо програму у випадку помилки.

Браузер сигналізує про кінець запиту на HTTP, надіславши поспіль два символи нового рядка, тож щоб отримати один запит з потоку, ми беремо рядки, доки не отримаємо рядок, що є порожньою стрічкою. Коли ми зберемо рядки у вектор, ми виводимо їх за допомогою гарного форматування для налагодження, щоб ми могли подивитися на інструкції, що веббраузер надсилає на наш сервер.

Спробуймо цей код! Запустіть програму і знову зробіть запит у веббраузері. Зверніть увагу, що ми все ще бачимо сторінку помилку в браузері, але виведення від нашої програми в термінал виглядатиме тепер схожим на це:

```
$ cargo run
Compiling hello v0.1.0 (file:///projects/hello)
Finished dev [unoptimized + debuginfo] target(s) in 0.42s
Running `target/debug/hello`
Request: [
  "GET / HTTP/1.1",
  "Host: 127.0.0.1:7878",
  "User-Agent: Mozilla/5.0 (Macintosh; Intel Mac OS X 10.15; rv:99.0)
Gecko/20100101 Firefox/99.0",
  "Accept:
text/html,application/xhtml+xml,application/xml;q=0.9,image/avif,image/webp,*/*;q=
0.8",
  "Accept-Language: en-US,en;q=0.5",
  "Accept-Encoding: gzip, deflate, br",
  "DNT: 1",
  "Connection: keep-alive",
  "Upgrade-Insecure-Requests: 1",
  "Sec-Fetch-Dest: document",
  "Sec-Fetch-Mode: navigate",
  "Sec-Fetch-Site: none",
  "Sec-Fetch-User: ?1",
  "Cache-Control: max-age=0",
]
```

Залежно від вашого браузера ви можете отримати трохи інше виведення. Тепер, коли ми виводимо дані запиту, ми бачимо, чому ми отримуємо декілька підключень з одного запиту до браузера, дивлячись на шлях за `GET` в першому рядку запиту. Якщо повторні з'єднання всі запитують `/`, то ми знатимемо, що браузер повторно намагається отримати `/`, бо не отримує відповіді від нашої програми.

Розберімо дані цього запиту, щоб зрозуміти, що саме браузер запитує в нашої програми.

Докладніший Розгляд HTTP-Запиту

HTTP - це протокол на основі тексту і запит використовує такий формат:

```
Метод URI-запит HTTP-версія CRLF
заголовки CRLF
тіло повідомлення
```

Перший рядок це рядок *рядок запиту*, який містить інформацію про те, що саме клієнт запитує. Перша частина рядка запиту позначає на *метод*, наприклад `GET` чи `POST`, який описує як клієнт робить цей запит. Наш клієнт використав запит `GET`, що означає, що він запитує інформацію.

Наступна частина рядка запиту - це `/`, що є *уніфікованим ідентифікатором ресурсу (Uniform Resource Identifier, URI)*, який запитує клієнт: URI це майже те, хоча й не зовсім, що й *уніфікований локатор ресурсу (Uniform Resource Locator, URL)*. Різниця між URI і URL не є важливою для наших цілей у цьому розділі, але специфікація HTTP використовує термін URI, тому ми тут можемо просто думати про URL замість URI.

Остання частина - це версія HTTP, яку використовує клієнт, а потім рядок запиту закінчується *послідовністю CRLF*. (CRLF означає *повернення каретки* і *зміна рядка*, тобто терміни з часів друкарських машинок!) Послідовність CRLF також записується як `\r\n`, де `\r` - повернення каретки, а `\n` - зміна рядка. Послідовність CRLF відділяє рядок запиту від решти даних запиту. Зверніть увагу, що коли виводиться CRLF, ми бачимо початок нового рядка, а не `\r\n`.

Дивлячись на дані рядка запиту, який ми отримали, запустивши нашу програму, ми бачимо, що `GET` - це метод, `/` - URI запиту і `HTTP/1.1` - це версія.

Рядки після рядка запиту, починаючи від `Host:` і далі - це заголовки. Запити `GET` не мають тіла.

Спробуйте запит з іншого браузера або запросіть іншу адресу, наприклад, `127.0.0.1:78/test`, щоб побачити, як змінюються дані запиту.

Тепер, коли ми знаємо, що браузер запитує, спробуймо відправити трохи даних у відповідь!

Написання Відповіді

Ми збираємось реалізувати виправлення даних у відповідь на запит клієнта. Відповіді мають такий формат:

```
HTTP-версія статус-код фраза-прояснення CRLF
заголовки CRLF
тіло повідомлення
```

Перший рядок - це *рядок стану*, що містить версію HTTP, використану у відповіді, числовий код стану, що підсумовує результат запиту, і фразу-пояснення з текстовим описом коду статусу. Після послідовності CRLF ідуть заголовками, ще одна послідовність CRLF та тіло відповіді.

Ось приклад відповіді, що використовує HTTP версії 1.1, має код стану 200, фразу-пояснення OK, без заголовків і без тіла:

```
HTTP/1.1 200 OK\r\n\r\n
```

Код стану 200 це стандартна відповідь про успіх. Цей текст є крихітною успішною відповіддю HTTP. Запишімо її в потік, як нашу відповідь на успішний запит! З функції `handle_connection` видалімо `println!`, який друкував дані запиту, і замінімо їх кодом з Блоку коду 20-3.

Файл: `src/main.rs`

```
fn handle_connection(mut stream: TcpStream) {
    let buf_reader = BufReader::new(&mut stream);
    let http_request: Vec<_> = buf_reader
        .lines()
        .map(|result| result.unwrap())
        .take_while(|line| !line.is_empty())
        .collect();

    let response = "HTTP/1.1 200 OK\r\n\r\n";

    stream.write_all(response.as_bytes()).unwrap();
}
```

Блок коду 20-3: написанні крихітної успішної відповіді HTTP до потоку

Перший новий рядок визначає змінну `response`, яка містить дані повідомлення про успіх. Потім ми викликаємо `as_bytes` для `response`, щоб перетворити стрічку даних на байти. Метод `write_all` для `stream` приймає `&[u8]` і відправляє ці байти безпосередньо у з'єднання. Оскільки операція `write_all` можуть бути невдалою, ми застосовуємо `unwrap`

для будь-яких помилок, як і раніше. Знову ж таки в реальній програмі ви маєте додати тут обробку помилок.

Змінивши так код, запустімо його і зробимо запит. Ми більше не виводимо жодних даних до термінала, тому не побачимо нічого крім того, що виведе Cargo. При завантаженні `127.0.0.1:7878` у веббраузері ви маєте отримати порожню сторінку замість помилки. Ви щойно своїми руками закодували отримання запиту HTTP і відправлення відповіді!

Повертаємо Справжній HTML

Реалізуймо функціональність для повернення чогось більшого за порожню сторінку. Створіть новий файл `hello.html` у кореневій теці вашого проєкту, а не в теці `src`. Ви можете ввести будь-який HTML за вашим бажанням; Блок коду 20-4 показує одну з можливостей.

Файл: `hello.html`

```
<!DOCTYPE html>
<html lang="en">
  <head>
    <meta charset="utf-8">
    <title>Hello!</title>
  </head>
  <body>
    <h1>Hello!</h1>
    <p>Hi from Rust</p>
  </body>
</html>
```

Блок коду 20-4: зразок HTML файлу для повернення у відповідь

Це мінімальний документ HTML5 із заголовком та текстом. Щоб сервер повернув це після отримання запиту, ми змінимо функцію `handle_connection`, як показано у Блоці коду 20-5, щоб вона читала HTML файл, додавала його до відповіді як тіло і відправляла його.

Файл: `src/main.rs`


```

use std::{
    fs,
    io::{prelude::*, BufReader},
    net::{TcpListener, TcpStream},
};
// --snip--

fn handle_connection(mut stream: TcpStream) {
    let buf_reader = BufReader::new(&mut stream);
    let http_request: Vec<_> = buf_reader
        .lines()
        .map(|result| result.unwrap())
        .take_while(|line| !line.is_empty())
        .collect();

    let status_line = "HTTP/1.1 200 OK";
    let contents = fs::read_to_string("hello.html").unwrap();
    let length = contents.len();

    let response =
        format!("{status_line}\r\nContent-Length: {length}\r\n\r\n{contents}");

    stream.write_all(response.as_bytes()).unwrap();
}

```

Блок коду 20-5: відправлення вмісту *hello.html* як тіла відповіді

Ми додали `fs` в інструкцію `use`, щоб ввести в область видимості модуль файлової системи зі стандартної бібліотеки. Код для читання вмісту файлу до стрічки має бути вам знайомим; ми використовували його в Розділі 12, коли читали вміст файлу для нашого проєкту I/O в Блоці коду 12-4.

Далі, ми використовуємо `format!`, щоб додати вміст файлу як тіло успішної відповіді. Для забезпечення коректної HTTP відповіді ми додаємо заголовок `Content-Length`, встановлений у розмір тіла нашої відповіді, у цьому випадку розмір `hello.html`.

Запустіть цей код за допомогою `cargo run` і завантажте `127.0.0.1:78` у браузері; ви повинні побачити зображенням свій HTML!

Наразі ми ігноруємо дані запиту у `http_request` і лише безумовно відправляємо у відповідь вміст HTML файлу. Це означає, що якщо ви спробуєте запитати `127.0.0.1:7878/something-else` у своєму браузері, то все одно отримаєте ту ж саму HTML відповідь. На зараз наш сервер украй обмежений і не робить того, що робить більшість вебсерверів. Ми хочемо налаштувати наші відповіді залежно від запиту і відправляти назад HTML файл лише для правильного сформованого запиту `/`.

Перевірка Запиту і Вибіркова Відповідь

Зараз наш вебсервер поверне HTML з файлу, незалежно від того, що клієнт запитував. Додамо функціональність для перевірки, чи браузер запитує /, перед поверненням HTML файлу і повертатимемо помилку, якщо браузер запитав щось інше. Для цього нам потрібно змінити `handle_connection`, як показано у Блоці коду 20-6. Цей новий код порівнює вміст отриманого запиту із тим, як, як ми знаємо, має виглядати запит до /, і додає блоки `if` та `else`, щоб нарізно обробляти запити.

Файл: `src/main.rs`

```
// --snip--

fn handle_connection(mut stream: TcpStream) {
    let buf_reader = BufReader::new(&mut stream);
    let request_line = buf_reader.lines().next().unwrap().unwrap();

    if request_line == "GET / HTTP/1.1" {
        let status_line = "HTTP/1.1 200 OK";
        let contents = fs::read_to_string("hello.html").unwrap();
        let length = contents.len();

        let response = format!(
            "{status_line}\r\nContent-Length: {length}\r\n\r\n{contents}"
        );

        stream.write_all(response.as_bytes()).unwrap();
    } else {
        // some other request
    }
}
```

Блок коду 20-6: обробка запитів до / окремо від інших запитів

Ми збираємося проглядати лише перший рядок HTTP запиту, тож замість зчитувати весь запит у вектор, ви викликаємо `next`, щоб отримати перший елемент з ітератора. Перший `unwrap` обробляє `Option` і зупиняє програму, якщо ітератор не має елементів. Другий `unwrap` обробляє `Result` і має такий самий ефект, що й `unwrap`, який був у `map`, доданому в Блоці коду 20-2.

Далі ми перевіряємо, чи `request_line` дорівнює рядку запиту для запиту GET до шляху /. Якщо це так, блок `if` поверне вміст нашого HTML файлу.

Якщо `request_line` не дорівнює GET запиту до шляху /, це означає, що ми отримали якийсь інший запит. Ми додамо код блоку `else`, щоб відповісти на всі інші запити, за хвилинку.

Запустіть цей код і запросіть `127.0.0.1:7878`; ви повинні отримати HTML з `hello.html`. Якщо ви зробите будь-який інший запит, наприклад, `127.0.0.1:7878/something-else`, то отримаєте помилку з'єднання, схожу на ті, які ви бачили, коли запускали код з Блоків коду 20-1 і 20-2.

Тепер у Блоці коду 20-7 додамо код до блоку `else`, щоб повернути відповідь з кодом статусу 404, що означає, що запитаний вміст не був знайдений. Також ми повернемо трохи HTML для відображення сторінки в браузері, щоб показати відповідь кінцевому користувачу.

Файл: `src/main.rs`

```
// --snip--
} else {
    let status_line = "HTTP/1.1 404 NOT FOUND";
    let contents = fs::read_to_string("404.html").unwrap();
    let length = contents.len();

    let response = format!(
        "{status_line}\r\nContent-Length: {length}\r\n\r\n{contents}"
    );

    stream.write_all(response.as_bytes()).unwrap();
}
```

Блок коду 20-7: відповідь з кодом стану 404 і сторінкою помилки, якщо було запитано щось відмінне від /

Тут наша відповідь має рядок стану з кодом стану 404 і фразу-пояснення `NOT FOUND`. Тіло відповіді буде HTML з файлу `404.html`. Вам треба створити файл `404.html` поруч із `hello.html` для сторінки помилки; знову ж можете використати будь-який HTML, який бажаєте, чи зразок HTML з Блоку коду 20-8.

Файл: `404.html`

```
<!DOCTYPE html>
<html lang="en">
  <head>
    <meta charset="utf-8">
    <title>Hello!</title>
  </head>
  <body>
    <h1>Oops!</h1>
    <p>Sorry, I don't know what you're asking for.</p>
  </body>
</html>
```

Блок коду 20-8: зразок вмісту для сторінки, яку відправляють у відповідь із кодом 404

Після цих змін запустіть ваш сервер знову. Запит `127.0.0.1:7878` повинен повернути вміст `hello.html`, а будь-який інший запит, наприклад `127.0.0.1:7878/foo`, повинен повернути HTML помилки з `404.html`.

Трохи Рефакторингу

На цей момент блоки `if` та `else` мають багато повторень: обидва читають файли і записують вміст файлів до потоку. Єдиною відмінністю є рядок стану й ім'я файлу. Зробімо код виразнішим, витягши ці відмінності в окремі рядки `if` та `else`, які присвоять значення рядка стану та імені файлу змінним; тоді ми можемо використати ці змінні безумовно в коді, щоб прочитати файл і записати відповідь. Блок коду 20-9 показує отриманий код після заміни великих блоків `if` та `else`.

Файл: `src/main.rs`

```
// --snip--

fn handle_connection(mut stream: TcpStream) {
    // --snip--

    let (status_line, filename) = if request_line == "GET / HTTP/1.1" {
        ("HTTP/1.1 200 OK", "hello.html")
    } else {
        ("HTTP/1.1 404 NOT FOUND", "404.html")
    };

    let contents = fs::read_to_string(filename).unwrap();
    let length = contents.len();

    let response =
        format!("{status_line}\r\nContent-Length: {length}\r\n\r\n{contents}");

    stream.write_all(response.as_bytes()).unwrap();
}
```

Блок коду 20-9: рефакторизація блоків `if` та `else`, щоб містили лише відмінний у двох випадках код

Тепер блоки `if` та `else` лише повертають відповідні значення для рядка стану й імені файлу в кортежі; далі ми використовуємо деструктуризацію, щоб присвоїти ці два значення змінним `status_line` і `filename` скориставшись шаблоном в інструкції `let`, як пояснювалося в Розділі 18.

Цей раніше дубльований код знаходиться поза межами блоків `if` та `else` і використовує змінні `status_line` і `filename`. Це дає змогу легше бачити відмінності між двома

випадками, і це означає, що у нас є тільки одне місце, щоб змінити код, якщо ми хочемо змінити, як працює читання файлів чи відправлення відповіді. Поведінка коду у Блоці коду 20-9 буде такою ж, як у Блоці коду 20-8.

Блискуче! Тепер ми маємо простий вебсервер з приблизно 40 рядків коду на Rust, що відповідає на один запит сторінкою з вмістом і на всі інші запити відповіддю 404.

Наразі наш сервер працює в одному потоці, тобто він може обслуговувати лише один запит за раз. Дослідимо, чому це може бути проблемою, симулюючи повільні запити. Тоді ми полагодимо цю проблему, щоб наш сервер міг обробляти багато запитів одночасно.

Перетворюємо Наш Однопоточковий Сервер на Багатопоточковий

Зараз наш сервер обробляє кожен запит по черзі, тобто він не обробить друге з'єднання, поки не завершить обробку першого. Якщо сервер отримує більше і більше запитів, це послідовне виконання буде все менш і менш оптимальним. Якщо сервер отримує запит, що обробляється довгий час, наступні запити муситимуть чекати, доки довгий запит не буде завершено, навіть якщо нові запити можна обробити швидко. Нам потрібно буде виправити це, але спершу ми поглянемо на цю проблему в дії.

Симуляція Повільного Запиту в Поточній Реалізації Сервера

Ми подивимося на те, як запит з повільною обробкою може вплинути на інші запити, зроблені до нашої поточної реалізації сервера. Блок коду 20-10 реалізує обробку запиту до `/sleep` з симуляцією повільної реакції, що заблокує сервер у режимі сну на 5 секунд до відповіді.

Файл: `src/main.rs`

```
use std::{
    fs,
    io::{prelude::*, BufReader},
    net::{TcpListener, TcpStream},
    thread,
    time::Duration,
};
// --snip--

fn handle_connection(mut stream: TcpStream) {
    // --snip--

    let (status_line, filename) = match &request_line[..] {
        "GET / HTTP/1.1" => ("HTTP/1.1 200 OK", "hello.html"),
        "GET /sleep HTTP/1.1" => {
            thread::sleep(Duration::from_secs(5));
            ("HTTP/1.1 200 OK", "hello.html")
        }
        _ => ("HTTP/1.1 404 NOT FOUND", "404.html"),
    };
    // --snip--
}
```

Блок коду 20-10: симуляція повільного запиту режимом сну на 5 секунд

Тепер ми перейшли з `if` на `match`, бо у нас є три випадки. Ми маємо явно зіставляти слайс з `request_line` із шаблоном зі стрічковими літералами; `match` не робить автоматичних посилань і розіменувань, як методи порівняння на рівність.

Перший рукав той же, що й у блоці `if` з Блоку коду 20-9. Другий рукав зіставляє запит зі `/sleep`. Коли цей запит отримано, сервер спатиме 5 секунд перед передачею успішної HTML сторінки. Третій рукав той же, що й у блоці `else` з Блоку коду 20-9.

Ви можете побачити, наскільки примітивними є наш сервер: справжні бібліотеки оброблять розпізнавання кількох запитів у набагато менш розлогий спосіб!

Запустімо сервер командою `cargo run`. Тоді відкрийте два вікна браузера: одне для `http://127.0.0.1:7878/`, а інше - для `http://127.0.0.1:7878/sleep`. Якщо ви введете URL / кілька разів, то, як і раніше, ви побачите, що відповідь надходить швидко. Але якщо ви введете `/sleep`, а потім завантажите `/`, ви побачите, що `/` чекає, доки `sleep` "проспить" 5 секунд до завантаження.

Існує безліч методів, якими ми могли б скористатися, щоб уникнути гальмування через повільний запит; те, що ми реалізуємо - це пул потоків.

Поліпшення Пропускної Здатності з Пулом Потоків

Пул потоків - це група породжених потоків, що чекають і готові до обробки завдання. Коли програма отримує нове завдання, то призначає один із потоків з пулу на це завдання, і цей потік обробляє завдання. Решта потоків у пулі доступні, щоб обробити будь-яке інше завдання, що надійде, поки перший потік зайнятий обробкою. Коли перший потік завершить обробку свого завдання, то повернеться до пулу незайнятих потоків, готовий обробляти нове завдання. Пул потоків дозволяє вам обробляти з'єднання конкурентно, збільшуючи пропускну здатність вашого сервера.

Ми обмежимо кількість потоків в пулі невеликим числом, щоб захистити нас від атак на відмову в обслуговуванні (Denial of Service, DoS); якби наша програма створювала по потоку на кожен вхідний запит, то хтось, створивши 10 мільйонів запитів до нашого сервера, може обвалити його, вичерпавши всі його ресурси та призвівши до повної зупинки обробки запитів.

Замість необмеженого породження потоків ми матимемо фіксовану кількість потоків, що чекатимуть у пулі. Вхідні запити надсилатимуться в пул для обробки. Пул підтримуватиме чергу вхідних запитів. Кожен з потоків у пулі братиме запит з цієї черги, оброблятиме його і запитуватиме наступний запит з черги. З таким дизайном ми можемо обробити до `N` запитів конкурентно, де `N` є кількістю потоків. Якщо кожен потік відповідатиме на

довгий запит, наступні запити все ж накопичуватиметься в черзі, але ми збільшили кількість довгих запитів, які ми можемо обробити до досягнення цього моменту.

Ця техніка - лише один із багатьох способів покращити пропускну здатність вебсервера. Інші варіанти, які ви можете дослідити, включають *модель fork/join*, *однопоточкова модель асинхронного I/O* та **багатопотокова модель асинхронного I/O **. Якщо ви зацікавилися цією темою, то можете прочитати більше про інші рішення і спробувати реалізувати їх; усі ці варіанти доступні низькорівневій мові на кшталт Rust.

Перед тим, як почати реалізовувати пул тредів, поговоримо про те, як має виглядати його використання. Коли ви намагаєтеся проєктувати код, написання спершу клієнтського інтерфейсу може допомогти керувати проєктуванням. Напишіть API коду, щоб він був структурованим відносно способу його виклику; тоді реалізуйте функціональність відповідно до цієї структури, а не спершу реалізуйте функціональність, а тоді проєктуйте публічний API.

Подібно до того, як ми використовували керовану тестами розробку у проєкті з Розділу 12, ми використаємо тут керовану компілятором розробку. Ми напишемо код, що викликає потрібні нам функції, а потім ми подивимося на помилки компілятора, щоб вирішити, що ми маємо далі змінити, щоб цей код працював. Але перед цим ми дослідимо техніку, яку ми не збираємося використовувати, як відправну точку.

Створення Потoku для Кожного Запиту

Спершу дослідимо, як наш код міг би виглядати, якби створював новий потік для кожного з'єднання. Як зазначено раніше, ми не плануємо так робити через проблеми з потенційним породженням нескінченної кількості потоків, але це вихідна точка, щоб спершу отримати робочий багатопотоковий сервер. Тоді ми покращимо код, додавши пул потоків, і відмінності між двома рішеннями стануть очевиднішими. Блок коду 20-11 показує зміни, які треба внести, щоб `main` породжував новий потік для обробки кожного потоку у циклі `for`.

Файл: `src/main.rs`

```
fn main() {  
    let listener = TcpListener::bind("127.0.0.1:7878").unwrap();  
  
    for stream in listener.incoming() {  
        let stream = stream.unwrap();  
  
        thread::spawn(|| {  
            handle_connection(stream);  
        });  
    }  
}
```


Блок коду 20-11: породження нового потоку виконання для кожного вхідного потоку

Як ви дізналися у Розділі 16, `thread::spawn` створить новий потік, а потім запустить код у замиканні в цьому новому потоці. Якщо ви запустите цей код і завантажите в браузері */sleep*, а тоді / у двох додаткових вкладках браузера, ви й справді побачите, що запит до / не мусить чекати, доки не завершиться */sleep*. Однак, як ми згадували, це кінець-кінцем перенавантажить систему, бо нові потоки створюються без будь-яких обмежень.

Створення Скінченної Кількості Потоків

Ми хочемо, щоб наш пул потоків працював у схожий, знайомий спосіб, щоб перехід з потоків до пулу потоків не вимагав значних змін у коді, що використовує наш API. Блок коду 20-12 показує гіпотетичний інтерфейс для структури `ThreadPool`, яку ми хочемо використати замість `thread::spawn`.

Файл: `src/main.rs`

```
fn main() {  
    let listener = TcpListener::bind("127.0.0.1:7878").unwrap();  
    let pool = ThreadPool::new(4);  
  
    for stream in listener.incoming() {  
        let stream = stream.unwrap();  
  
        pool.execute(|| {  
            handle_connection(stream);  
        });  
    }  
}
```

Блок коду 20-12: наш ідеальний інтерфейс `ThreadPool`

Ми використовуємо `ThreadPool::new` для створення нового пулу потоків налаштуванням числом потоків, у цьому випадку чотирма. Тоді, в циклі `for` циклу, `pool.execute` має інтерфейс, подібний до `thread::spawn` у тому, що він для кожного вхідного потоку приймає замикання, яке пул має виконати. Ми маємо реалізувати `pool.execute` так, щоб він приймав замикання і передавав його треду в пулі на виконання. Цей код ще не компілюється, але ми спробуємо це зробити, щоб компілятор міг підказати, як це виправити.

Збірка `ThreadPool` з Керованим Компілятором Розробки

Внесіть зміни з Блоку коду 20-12 до `src/main.rs`, а потім використаймо помилки компілятора з `cargo check` для керування розробкою. Ось яку першу помилку ми

отримуємо:

```
$ cargo check
  Checking hello v0.1.0 (file:///projects/hello)
error[E0433]: failed to resolve: use of undeclared type `ThreadPool`
  --> src/main.rs:11:16
   |
11 |         let pool = ThreadPool::new(4);
   |                        ^^^^^^^^^^^^^ use of undeclared type `ThreadPool`

For more information about this error, try `rustc --explain E0433`.
error: could not compile `hello` due to previous error
```

Чудово! Ця помилка говорить, що нам потрібен тип чи модуль `ThreadPool`, тож ми його створимо. Наша реалізація `ThreadPool` буде незалежною від виду роботи, що її виконує наш вебсервер. Отже, переробимо крейт `hello` з двійкового крейта на бібліотеку, де міститиметься наша реалізація `ThreadPool`. Після перероблення на бібліотечний крейт ми також могли б використати окрему бібліотеку пулу потоків для будь-якої роботи, яку ми хочемо виконати за допомогою пулу потоків, а не лише для обслуговування вебзапитів.

Створіть `src/lib.rs`, що містить найпростіше визначення структури `ThreadPool`, яке ми можемо наразі мати:

Файл: `src/lib.rs`

```
pub struct ThreadPool;
```

Далі відредагуйте файл `main.rs`, щоб ввести `ThreadPool` до області видимості з бібліотечного крейта, додавши наступний код зверху `src/main.rs`:

Файл: `src/lib.rs`

```
use hello::ThreadPool;
```

Цей код все ще не працює, але перевірмо його ще раз, щоб отримати наступну помилку, над якою нам потрібно буде працювати:

```
$ cargo check
    Checking hello v0.1.0 (file:///projects/hello)
error[E0599]: no function or associated item named `new` found for struct
`ThreadPool` in the current scope
--> src/main.rs:12:28
|
12 |         let pool = ThreadPool::new(4);
|                                ^^^ function or associated item not found in
`ThreadPool`

For more information about this error, try `rustc --explain E0599`.
error: could not compile `hello` due to previous error
```

Ця помилка означає, що нам необхідно створити для `ThreadPool` асоційовану функцію з назвою `new`. Ми також знаємо, що `new` повинна мати один параметр, який може прийняти `4` як аргумент і має повернути екземпляр `ThreadPool`. Реалізуймо найпростішу функцію `new`, що матиме такі характеристики:

Файл: `src/lib.rs`

```
pub struct ThreadPool;

impl ThreadPool {
    pub fn new(size: usize) -> ThreadPool {
        ThreadPool
    }
}
```

Ми обрали типом параметра `size` тип `usize`, бо ми знаємо що від'ємна кількість потоків не має сенсу. Ми також знаємо, що ми використаємо `4` як число елементів у колекції потоків, а це саме те, для чого призначений тип `usize`, як говорилося у підрозділі “Цілі типи” Розділу 3.

Ще раз перевіримо код:

```
$ cargo check
    Checking hello v0.1.0 (file:///projects/hello)
error[E0599]: no method named `execute` found for struct `ThreadPool` in the
current scope
--> src/main.rs:17:14
|
17 |         pool.execute(|| {
|                ^^^^^^^ method not found in `ThreadPool`

For more information about this error, try `rustc --explain E0599`.
error: could not compile `hello` due to previous error
```

Тепер стається помилка, бо ми не маємо методу `execute` на `ThreadPool`. Згадайте з підрозділу "Створення скінченної кількості потоків", що ми вирішили, що інтерфейс нашого пула тредів має бути схожим на `thread::spawn`. На додачу ми реалізуємо функцію `execute`, щоб приймала передане їй замикання і передавало її вільному потоку з пула на виконання.

Ми визначимо метод `execute` для `ThreadPool` так, щоб він приймав параметром замикання. Згадайте з підрозділу "Переміщення захоплених значень із замикання та трейти `Fn`" Розділу 13, що ми можемо приймати замикання параметрами за допомогою трьох різних трейтів: `Fn`, `FnMut` і `FnOnce`. Ми маємо вирішити, який тип замикань використовується тут. Ми знаємо, що в результаті вийде щось схоже на реалізацію `thread::spawn` зі стандартної бібліотеки, тож можемо подивитися на обмеження на параметр з сигнатури `thread::spawn`. Документація показує нам таке:

```
pub fn spawn<F, T>(f: F) -> JoinHandle<T>
    where
        F: FnOnce() -> T,
        F: Send + 'static,
        T: Send + 'static,
```

Тип-параметр `F` - це те, що нас тут цікавить; тип-параметр `T` стосується значення, що повертається, і він нас не цікавить. Ми бачимо, що `spawn` використовує `FnOnce` як обмеження трейту для `F`. Ймовірно, це те саме, що нам треба, тому що ми зрештою передамо аргумент, який отримали у `execute`, до `spawn`. Ми можемо бути впевнені, що `FnOnce` - це трейт, який ми хочемо використовувати, оскільки потік для виконання запиту виконає замикання цього запиту тільки один раз, що відповідає `Once` у `FnOnce`.

Тип-параметр `F` також має трейтове обмеження `Send` і обмеження часу існування `'static`, що є корисним у нашій ситуації: нам потрібен `Send`, щоб передавати замикання від одного потоку до іншого, і `'static`, бо ми не знаємо, скільки часу виконуватиметься потік. Створімо метод `execute` для `ThreadPool`, що прийматиме узагальнений параметр типу `F` із цими обмеженнями:

Файл: `src/lib.rs`

```
impl ThreadPool {
    // --snip--
    pub fn execute<F>(&self, f: F)
    where
        F: FnOnce() + Send + 'static,
    {
    }
}
```

Ми все ще використовуємо `()` після `FnOnce`, бо `FnOnce` представляє замикання, що не приймає параметрів і повертає одиничний тип `()`. Як і у визначеннях функцій, тип, що повертається, можна не вказувати у сигнатурі, але навіть якщо ми не маємо параметрів, то все одно потребуємо дужки.

Знову ж таки, це найпростіша реалізація методу `execute`: вона не робить нічого, але ми намагаємося лише змусити наш код компілюватися. Ще раз перевіримо:

```
$ cargo check
Checking hello v0.1.0 (file:///projects/hello)
Finished dev [unoptimized + debuginfo] target(s) in 0.24s
```

Компілюється! Але зверніть увагу, що якщо ви спробуєте запустити `cargo run` і зробити запит у браузері, то побачите в браузері помилки, які ми вже бачили на початку розділу. Наша бібліотека ще не викликає замикання, передане до `execute`!

Примітка: ви могли чути, що про мови з жорсткими компіляторами, такими як Haskell and Rust, кажуть "якщо код компілюється, то він працює." Але це твердження не завжди правильне. Наш проєкт компілюється, але абсолютно нічого не робить! Якби ми збирали реальний, повний проєкт, це був би вдалий час почати написати юніт-тести, щоб перевірити, що код компілюється і має бажану поведінку.

Валідація Числа Потоків у `new`

Ми ще нічого не робимо з параметрами `new` та `execute`. Реалізуймо тіла цих функцій з бажаною для нас поведінкою. Для початку, подумаємо про `new`. Раніше ми вибрали беззнаковий тип для параметра `size`, бо пул з від'ємним числом потоків не має сенсу. Однак пул з нулем потоків також не має жодного сенсу, проте нуль є абсолютно валідним `usize`. Ми додамо код, щоб перевірити, чи `size` є більшим, ніж нуль, перш ніж повертати екземпляр `ThreadPool` і змусимо програму паніку якщо вона отримує нуль, використовуючи макрос `assert!`, як показано в Блоці коду 20-13.

Файл: `src/lib.rs`

```
impl ThreadPool {
    /// Create a new ThreadPool.
    ///
    /// The size is the number of threads in the pool.
    ///
    /// # Panics
    ///
    /// The `new` function will panic if the size is zero.
    pub fn new(size: usize) -> ThreadPool {
        assert!(size > 0);

        ThreadPool
    }

    // --snip--
}
```

Блок коду 20-13: реалізація `ThreadPool::new`, що панікує, якщо `size` буде нулем

Ми також додали трохи документації до нашого `ThreadPool` документаційним коментарем. Зверніть увагу, що ми слідували за хорошими практиками документації, додавши розділ, який описує ситуації, в яких наша функція може панікувати, як обговорювалося в Розділі 14. Спробуйте запустити `cargo doc --open` і натисніть на структуру `ThreadPool`, щоб побачити як виглядає документація для `new`!

Замість додавати макрос `assert!`, як ми зробили тут, ми могли б змінити `new` на `build` і повертати `Result`, як ми робили з `Config::build` у проєкті I/O з Блоку коду 12-9. Але ми вирішили, що в цьому випадку створити пул потоків без жодного потоку має бути невинною помилкою. Якщо ви почуваєтеся амбітним, спробуйте написати функцію, що зветься `build`, щоб порівняти з функцією `new`, з такою сигнатурою:

```
pub fn build(size: usize) -> Result<ThreadPool, PoolCreationError> {
```

Створення Місця для Зберігання Потоків

Тепер, коли ми можемо переконатися, що маємо валідну кількість потоків для зберігання в пулі, ми можемо створити ці потоки і зберегти їх у структурі `ThreadPool` перед тим, як її повертати. Але як нам "зберегти" потік? Ще раз погляньмо на сигнатуру `thread::spawn`:

```
pub fn spawn<F, T>(f: F) -> JoinHandle<T>
where
    F: FnOnce() -> T,
    F: Send + 'static,
    T: Send + 'static,
```

Функція `spawn` повертає `JoinHandle<T>`, де `T` - тип, що повертає замикання. Спробуймо також використати `JoinHandle` і побачимо, що вийде. У нашому випадку, замикання, які ми передаємо до пулу потоків, будуть обробляти з'єднання, нічого не повертаючи, так що `T` буде одиничним типом `()`.

Код у Блоці коду 20-14 скомпілюється, але ще не створює жодних потоків. Ми змінили визначення `ThreadPool`, додавши в нього вектор екземплярів `thread::JoinHandle<()>`, ініціалізували цей вектор об'ємом `size`, організували цикл `for`, який виконуватиме певний код для створення потоків, та повернули екземпляр `ThreadPool`, що містить їх.

Файл: `src/lib.rs`

```
use std::thread;

pub struct ThreadPool {
    threads: Vec<thread::JoinHandle<()>>,
}

impl ThreadPool {
    // --snip--
    pub fn new(size: usize) -> ThreadPool {
        assert!(size > 0);

        let mut threads = Vec::with_capacity(size);

        for _ in 0..size {
            // create some threads and store them in the vector
        }

        ThreadPool { threads }
    }
    // --snip--
}
```

Блок коду 20-14: створення вектора, що містить потоки, у `ThreadPool`

Ми ввели до області видимості `std::thread` з бібліотечного крейта, бо ми використовуємо `thread::JoinHandle` як тип елементів у векторі у `ThreadPool`.

Коли отримано валідний розмір, наш `ThreadPool` створює новий вектор, що може містити `size` елементів. Функція `with_capacity` виконує те саме завдання, що й `Vec::new`, але з важливою відмінністю: вона наперед виділяє місце у векторі. Оскільки ми знаємо, що нам потрібно зберігати `size` елементів у векторі, цей розподіл наперед є дещо ефективнішим, ніж використання `Vec::new`, який змінює розмір при вставленні елементів.

Коли ви знову запустите `cargo check`, він має відпрацювати успішно.

Структура `Worker`, Відповідальна за Пересилання Коду з `ThreadPool` до Потоків

Ми залишили коментар у циклі `for` у Блоці коду 20-14 про створення потоків. Тут ми розберемо, як насправді створювати потоки. Стандартна бібліотека уможливорює створення потоків через `thread::spawn`, який очікує отримати якийсь код, який потік має запустити, щойно його було створено. Однак у нашому випадку ми хочемо створити потоки, що *очікують* на код, який ми надішлемо пізніше. Реалізація зі стандартної бібліотеки не надає жодного способу це зробити; ми маємо реалізувати його вручну.

Ми реалізуємо таку поведінку, впровадивши нову структуру даних між `ThreadPool` і потоками, що оброблятиме цю поведінку. Ми назвемо цю структуру даних `Worker`, що є звичним терміном у реалізації пула. `Worker` бере код, який потрібно запустити і запускає його в своєму потоці. Уявіть працівників кухні в ресторані: вони чекають, поки не прийде замовлення від клієнтів, і тоді вони відповідають за прийняття цих замовлень і їхнє виконання.

Замість зберігання вектора екземплярів `JoinHandle<()>` у пулі потоків, Ми зберігатимемо екземпляри структури `Worker`. Кожен `Worker` зберігатиме один екземпляр `JoinHandle<()>`. Тоді ми реалізуємо метод для `Worker`, який прийматиме замикання з кодом для запуску і відправлятиме його в уже робочий потік на виконання. Також ми надамо кожному `worker` `id`, щоб ми могли розрізняти різних `worker` в пулі для журналювання або налагодження.

Ось новий процес, що відбудеться, коли ми створимо `ThreadPool`. Ми реалізуємо код, який відправляє замикання в потік після того, як в нас вже є `Worker`, налаштований таким чином:

1. Визначимо структуру `Worker`, яка містить `id` і `JoinHandle<()>`.
2. Змінимо `ThreadPool`, щоб він містив вектор екземплярів `Worker`.
3. Визначимо функцію `Worker::new`, що приймає номер `id` і повертає екземпляр `Worker`, що містить `id` та потік, породжений із порожнім замиканням.
4. У `ThreadPool::new` ми використовуємо лічильник циклу `for`, щоб згенерувати `id`, створити нового `Worker` з цим `id`, і зберегти `worker` у векторі.

Якщо ви готові до випробовування, застосуйте ці зміни самі перед переглядом Блока коду 20-15.

Готові? Ось Блок коду 20-15 з одним із можливих способів зробити описані зміни.

Файл: `src/lib.rs`


```

use std::thread;

pub struct ThreadPool {
    workers: Vec<Worker>,
}

impl ThreadPool {
    // --snip--
    pub fn new(size: usize) -> ThreadPool {
        assert!(size > 0);

        let mut workers = Vec::with_capacity(size);

        for id in 0..size {
            workers.push(Worker::new(id));
        }

        ThreadPool { workers }
    }
    // --snip--
}

struct Worker {
    id: usize,
    thread: thread::JoinHandle<()>,
}

impl Worker {
    fn new(id: usize) -> Worker {
        let thread = thread::spawn(|| {});

        Worker { id, thread }
    }
}

```

Блок коду 20-15: зміни до `ThreadPool`, щоб містив екземпляри `Worker` замість безпосередньо потоків

Ми змінили назву поля у `ThreadPool` з `threads` на `workers`, бо воно тепер містить екземпляри `Worker` замість `JoinHandle<()>`. Ми використовуємо лічильник циклу `for` циклі як аргумент `Worker::new`, і зберігаємо кожен новий `Worker` у векторі під назвою `workers`.

Зовнішній код (скажімо, наш сервер з `src/main.rs`) не має знати деталей реалізації стосовно використання структури `Worker` у `ThreadPool`, тож ми робимо структуру `Worker` і її функцію `new` приватними. Функція `Worker::new` використовує `id`, що ми їй передаємо, і зберігає екземпляр `JoinHandle<()>`, створений породженням нового потоку за допомогою порожнього замикання.

Примітка: якщо операційна система не може створити потік через нестачу системних ресурсів, `thread::spawn` панікуватиме. Це призведе до паніки усього нашого сервера, навіть якщо створення деяких потоків і буде вдалим. Заради простоти ця поведінка прийнятна, але у виробничій реалізації пулу потоків ви, швидше за все, захочете скористатися `std::thread::Builder` і його методом `spawn`, що повертає натомість `Result`.

Цей код скомпілюється і зберігатиме кількість екземплярів `Worker`, яку ми передали як аргумент для `ThreadPool::new`. Але ми *все ще* не обробляємо замикавання, які ми отримали у `execute`. Подивімося, як це зробити, далі.

Надсилання Запитів до Потоків Через Канали

Наступна проблема, якою ми займемося, полягає в тому, що замикавання, передані `thread::spawn`, не роблять абсолютно нічого. Наразі ми отримуємо замикавання, що хочемо виконати, у методі `execute`. Але ми маємо передати до `thread::spawn` якесь замикавання, коли ми створюємо кожного `Worker` при створенні `ThreadPool`.

Ми хочемо, щоб щойно створені структури `Worker` отримували код з черги, що міститься в `ThreadPool`, і відправляли цей код у свій потік на виконання.

Канали, про які ми дізналися у Розділі 16 — простий спосіб спілкування між двома потоками — ідеально підходять для цього випадку. Ми скористаємося каналом як чергою завдань, і `execute` відправить завдання з `ThreadPool` до екземплярів `Worker`, які перешлють завдання до своїх потоків. Ось наш план:

1. `ThreadPool` створить канал і утримуватиме відправника.
2. Кожен `Worker` утримуватиме отримувача.
3. Ми створимо нову структуру `Job`, що міститиме замикавання, що їх ми хочемо відправити каналом.
4. Метод `execute` відправить завдання, яке треба виконати, через відправника.
5. У своєму потоці `Worker` буде в циклі запитувати свого отримувача і виконувати замикавання з отриманих завдань.

Почнімо з створення каналу в `ThreadPool::new` та утримання відправника у екземплярі `ThreadPool`, як показано у Блоці коду 20-16. Структура `Job` наразі не містить нічого, але буде типом елементів, що їх ми відправляємо каналом.

Файл: `src/lib.rs`

```
use std::{sync::mpsc, thread};

pub struct ThreadPool {
    workers: Vec<Worker>,
    sender: mpsc::Sender<Job>,
}

struct Job;

impl ThreadPool {
    // --snip--
    pub fn new(size: usize) -> ThreadPool {
        assert!(size > 0);

        let (sender, receiver) = mpsc::channel();

        let mut workers = Vec::with_capacity(size);

        for id in 0..size {
            workers.push(Worker::new(id));
        }

        ThreadPool { workers, sender }
    }
    // --snip--
}
```

Блок коду 20-16: змінюємо `ThreadPool`, зберігаючи відправника каналу, що передає екземпляри `Job`

У `ThreadPool::new` ми створюємо новий канал і пул тепер містить відправника. Це успішно компілюється.

Спробуймо передати отримувача каналу усім worker, коли пул потоків створює канал. Ми знаємо, що хотіли б використати приймач у потоці, породженому worker, тож ми посилатимемося на параметр `receiver` у замиканні. Код у Блоці коду 20-17 поки що не компілюється.

Файл: src/lib.rs

```

impl ThreadPool {
    // --snip--
    pub fn new(size: usize) -> ThreadPool {
        assert!(size > 0);

        let (sender, receiver) = mpsc::channel();

        let mut workers = Vec::with_capacity(size);

        for id in 0..size {
            workers.push(Worker::new(id, receiver));
        }

        ThreadPool { workers, sender }
    }
    // --snip--
}

// --snip--

impl Worker {
    fn new(id: usize, receiver: mpsc::Receiver<Job>) -> Worker {
        let thread = thread::spawn(|| {
            receiver;
        });

        Worker { id, thread }
    }
}

```

Блок коду 20-17: передавання приймача до worker

Ми зробили деякі дрібні і очевидні зміни: ми передаємо приймач до `Worker::new`, а потім використовуємо його всередині замикання.

Коли ми спробуємо перевірити цей код, то отримаємо таку помилку:

```
$ cargo check
  Checking hello v0.1.0 (file:///projects/hello)
error[E0382]: use of moved value: `receiver`
  --> src/lib.rs:26:42
   |
21 |         let (sender, receiver) = mpsc::channel();
   |         ----- move occurs because `receiver` has type
   |         `std::sync::mpsc::Receiver<Job>`, which does not implement the `Copy` trait
...
26 |         workers.push(Worker::new(id, receiver));
   |                                ^^^^^^^^^ value moved here, in
previous iteration of loop

For more information about this error, try `rustc --explain E0382`.
error: could not compile `hello` due to previous error
```

Код намагається передати `receiver` кільком екземплярам `Worker`. Так не виходить, бо, як ви пам'ятаєте з Розділу 16, реалізація каналу в Rust має багатьох *виробників* і одного *споживача*. Це означає, що ми не можемо просто клонувати споживацький вихід каналу, щоб виправити код. Також ми не хочемо надсилати повідомлення кілька разів декільком споживачам; ми хочемо єдиниц список повідомлень з декількома worker, таким чином, щоб кожне повідомлення було оброблене один раз.

На додачу, приймання завдання з черги в каналі включає зміну `receiver`, тож потокам потрібен безпечний спосіб спільно використовувати та змінювати `receiver`; інакше ми можемо отримати стан гонитви (як розповідалося в Розділі 16).

Згадайте потокобезпечні розумні вказівники, про які йшлося в Розділі 16: щоб розділити володіння між кількома потоками і дозволити потокам змінювати значення, нам треба було скористатися `Arc<Mutex<T>>`. Тип `Arc` дозволить кільком worker володіти приймачем, а `Mutex` гарантує, що лише один worker отримує завдання з приймача за раз. Блок коду 20-18 показує зміни, які ми маємо зробити.

Файл: `src/lib.rs`

```

use std::{
    sync::{mpsc, Arc, Mutex},
    thread,
};
// --snip--

impl ThreadPool {
    // --snip--
    pub fn new(size: usize) -> ThreadPool {
        assert!(size > 0);

        let (sender, receiver) = mpsc::channel();

        let receiver = Arc::new(Mutex::new(receiver));

        let mut workers = Vec::with_capacity(size);

        for id in 0..size {
            workers.push(Worker::new(id, Arc::clone(&receiver)));
        }

        ThreadPool { workers, sender }
    }

    // --snip--
}

// --snip--

impl Worker {
    fn new(id: usize, receiver: Arc<Mutex<mpsc::Receiver<Job>>>) -> Worker {
        // --snip--
    }
}

```

Блок коду 20-18: спільне використання приймача worker за допомогою `Arc` і `Mutex`

У `ThreadPool::new`, ми розміщуємо приймач у `Arc` і `Mutex`. Для кожного нового worker ми клонуємо `Arc`, щоб збільшити лічильник посилань, щоб worker могли спільно володіти приймачем.

З цими змінами, код компілюється! Ми вже близько!

Реалізація Методу `execute`

Нарешті реалізуємо метод `execute` для `ThreadPool`. Також ми змінимо `Job` зі структури на псевдонім типу для трейтового об'єкта, який містить тип замикання, яку приймає `execute`. Як уже говорилося в підрозділі “Створення типів-синонімів за допомогою

псевдонімів типів” Розділу 19, псевдоніми типів дозволяють нам скорочувати довгі типи для простоти використання. Подивіться на Блок коду 20-19.

Файл: src/lib.rs

```
// --snip--

type Job = Box<dyn FnOnce() + Send + 'static>;

impl ThreadPool {
    // --snip--

    pub fn execute<F>(&self, f: F)
    where
        F: FnOnce() + Send + 'static,
    {
        let job = Box::new(f);

        self.sender.send(job).unwrap();
    }
}

// --snip--
```

Блок коду 20-19: створення псевдоніма типу `Job` для `Box`, що містить кожне замикання, і відправлення завдання каналом

Після створення нового екземпляра `Job` за допомогою замикання, що ми отримали в `execute`, ми підправляємо це завдання через вхід каналу. Ми викликаємо `unwrap` для `send` на випадок, якщо відправлення буде невдалим. Це може статися якщо, наприклад, ми зупинимо всі потоки, тобто вихід каналу припинить отримувати нові повідомлення. На цей час ми не можемо зупинити наші потоки: вони продовжують виконуватись, доки пул існує. Причина, чому ми використовуємо `unwrap`, полягає в тому, що ми знаємо, що невдача тут неможлива, але компілятор цього не знає.

Та ми ще не зовсім закінчили! У `worker` наше замикання, що передається до `thread::spawn`, лише *посилається* на вихід каналу. Натомість нам треба, щоб замикання у вічному циклі отримувало з вихідного кінця каналу завдання і після отримання виконувало його. Зробімо зміни, показані в Блоці коду 20-20, у `Worker::new`.

Файл: src/lib.rs

```
// --snip--

impl Worker {
    fn new(id: usize, receiver: Arc<Mutex<mpsc::Receiver<Job>>>) -> Worker {
        let thread = thread::spawn(move || loop {
            let job = receiver.lock().unwrap().recv().unwrap();

            println!("Worker {id} got a job; executing.");

            job();
        });

        Worker { id, thread }
    }
}
```

Блок коду 20-20: отримання і виконання завдань у потоці worker

Тут ми спершу викликаємо `lock` для `receiver`, щоб отримати м'ютекс, а потім викликаємо `unwrap` для паніки, якщо сталася якась помилка. Здійснення блокування може призвести до невдачі, якщо м'ютекс знаходиться у стані *poisoned*, що може статися, якщо якийсь інший потік запанікував, поки утримував блокування, а не відпустив його. У цій ситуації виклик `unwrap` для паніки є правильною дією. Можете за бажання змінити `unwrap` на `expect` зі змістовним для вас повідомленням про помилку.

Якщо ми отримали блокування м'ютекса, то викличемо `recv`, щоб отримати `Job` з каналу. Останній `unwrap` також покриває всі помилки, що могли виникнути якщо потік, що утримує відправника, завершився, так само як метод `send` повертає `Err`, якщо отримувач завершився.

Виклик `recv` блокує, тож якщо завдань немає, поточний потік чекатиме, доки не з'явиться доступне завдання. `Mutex<T>` гарантує, що лише один потік `worker` за раз намагатиметься отримати завдання.

Наш пул потоків нарешті у робочому стані! Виконайте `cargo run` і зробіть кілька запитів:


```

$ cargo run
   Compiling hello v0.1.0 (file:///projects/hello)
warning: field is never read: `workers`
  --> src/lib.rs:7:5
  |
7 |     workers: Vec<Worker>,
  |     ^^^^^^^^^^^^^^^^^^^^^
  |
= note: `[warn(dead_code)]` on by default

warning: field is never read: `id`
  --> src/lib.rs:48:5
  |
48 |     id: usize,
  |     ^^^^^^^^^
  |

warning: field is never read: `thread`
  --> src/lib.rs:49:5
  |
49 |     thread: thread::JoinHandle<()>,
  |     ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
  |

warning: `hello` (lib) generated 3 warnings
   Finished dev [unoptimized + debuginfo] target(s) in 1.40s
   Running `target/debug/hello`
Worker 0 got a job; executing.
Worker 2 got a job; executing.
Worker 1 got a job; executing.
Worker 3 got a job; executing.
Worker 0 got a job; executing.
Worker 2 got a job; executing.
Worker 1 got a job; executing.
Worker 3 got a job; executing.
Worker 0 got a job; executing.
Worker 2 got a job; executing.

```

Успіх! Тепер у нас є пул потоків, який виконує з'єднання асинхронно. Також ніколи не буде створено більше ніж чотири потоки, тому наша система не перенавантажиться, якщо сервер отримає забагато запитів. Якщо ми робимо запит до `/sleep`, сервер буде мати можливість обслуговувати інші запити, бо їх виконувати буде інший потік.

Примітка: якщо ви відкриєте `/sleep` в декількох вікнах браузера одночасно, вони можуть вантажитися по одному з 5-секундним інтервалом. Деякі веббраузери виконують кілька екземплярів одного запиту послідовно для потреб кешування. Це обмеження не викликане нашим вебсервером.

Після вивчення циклу `while let` у Розділі 18, ви можете поцікавитися, чому ми не написали код потоку `worker`, як показано в Блоці коду 20-21.

Файл: src/lib.rs

```
// --snip--

impl Worker {
    fn new(id: usize, receiver: Arc<Mutex<mpsc::Receiver<Job>>>) -> Worker {
        let thread = thread::spawn(move || {
            while let Ok(job) = receiver.lock().unwrap().recv() {
                println!("Worker {id} got a job; executing.");

                job();
            }
        });

        Worker { id, thread }
    }
}
```

Блок коду 20-21: альтернативна реалізація `Worker::new` за допомогою `while let`

Цей код компілюється і запускається, але не дає бажаного багатопотокового результату: повільний запит усе ще змушує інші потоки чекати на обробку. Причина дещо тонка: структура `Mutex` не має публічного методу `unlock`, тому що володіння блокуванням базується на часі існування `MutexGuard<T>` у `LockResult<MutexGuard<T>>`, повернутим методом `lock`. Під час компіляції borrow checker може гарантувати правило, що ресурс, захищений `Mutex`, не буде доступним, якщо ми не маємо блокування. Однак ця реалізація також призведе до того, що блокування буде утримуватися довше, ніж потрібно, якщо ми не пам'ятаємо про час існування `MutexGuard<T>`.

Код у Блоці коду 20-20, що робить `let job = receiver.lock().unwrap().recv().unwrap();`, працює, бо в `let` будь-які тимчасові значення, використані у правій стороні знаку рівності, негайно очищуються, коли завершується інструкція `let`. Проте, `while let` (і `if let` та `match`) не очищують тимчасові значення до кінця відповідного блоку. У Блоці коду 20-21 блокування утримується на час виклику `job()`, тобто інші worker не можуть отримувати завдання. [ch19-04-advanced-types.html#creating-type-synonyms-with-type-aliases](https://pavloslav.github.io/RustBookUkr/print.html#creating-type-synonyms-with-type-aliases) [ch13-01-closures.html#moving-captured-values-out-of-the-closure-and-the-fn-traits](https://pavloslav.github.io/RustBookUkr/print.html#moving-captured-values-out-of-the-closure-and-the-fn-traits)

Плавне Вимкнення та Очищення

Код у Блоці коду 20-20 відповідає на запити асинхронно, використовуючи пулу потоків, так, як ми й планували. Ми отримуємо деякі попередження про поля `workers`, `id`, і `threads`, які ми не використовуємо напямую, що нагадує нам ми нічого не очищуємо. Коли ми використовуємо менш елегантний метод зупинки основного потоку за допомогою `ctrl-c`, решта потоків також негайно зупиняється, навіть якщо ми посередині обробки запиту.

Наступним ми реалізуємо трейт `Drop`, щоб викликати `join` для кожного з потоків у пулі, щоб вони могли завершити запити, над якими працюють, перед закриттям. Потім ми реалізуємо спосіб повідомити потокам, що вони мають припинити отримувати нові запити і вимкнутися. Щоб побачити цей код у дії, ми змінимо наш сервер, щоб він приймав лише два запити перед плавним вимиканням пулу потоків.

Реалізація Трейту `Drop` для `ThreadPool`

Почнімо з реалізації `Drop` для нашого пулу потоків. Коли пул очищується, всі потоки повинні приєднатися до основного, щоб переконатися, що вони завершили роботу. Блок коду 20-22 показує першу спробу реалізації `Drop`; цей код ще не зовсім працює.

Файл: `src/lib.rs`

```
impl Drop for ThreadPool {  
    fn drop(&mut self) {  
        for worker in &mut self.workers {  
            println!("Shutting down worker {}", worker.id);  
  
            worker.thread.join().unwrap();  
        }  
    }  
}
```

Блок коду 20-22: приєднання всіх потоків, коли пул потоків виходить з області видимості

Спершу ми в циклі перебираємо всі `workers` в пулі потоків. Для цього ми використовуємо `&mut`, бо `self` є мутабельним посиланням, і ми також мусимо мати можливість змінити `worker`. Для кожного `worker` ми виводимо повідомлення, що цей конкретний `worker` вимикається, а потім викликаємо `join` для потоку цього `worker`. Якщо виклик `join` буде невдалим, ми використовуємо `unwrap`, щоб Rust запанікував і грубо припинив роботу.

Ось помилка, яку ми отримуємо, коли ми компілюємо цей код:

```
$ cargo check
  Checking hello v0.1.0 (file:///projects/hello)
error[E0507]: cannot move out of `worker.thread` which is behind a mutable
reference
  --> src/lib.rs:52:13
52 |         worker.thread.join().unwrap();
   |         ^^^^^^^^^^^^^^^^^^ ----- `worker.thread` moved due to this method
call
   |         |
   |         move occurs because `worker.thread` has type `JoinHandle<()>`,
   |         which does not implement the `Copy` trait
note: this function takes ownership of the receiver `self`, which moves
`worker.thread`

For more information about this error, try `rustc --explain E0507`.
error: could not compile `hello` due to previous error
```

Ця помилка каже нам, що ми не можемо викликати `join`, бо ми маємо лише мутабельне позичання кожного `worker`, а `join` перебирає володіння своїм аргументом. Щоб вирішити цю проблему, ми маємо перемістити потік з екземпляра `Worker`, що володіє цим `thread`, щоб `join` міг поглинути потік. Ми робили це у Блоці коду 17-15: якщо `Worker` містить `Option<thread::JoinHandle<()>>`, ми можемо викликати метод `take` для `Option`, щоб перемістити значення з варіанту `Some` і залишити варіант `None` на своєму місці. Іншими словами, `Worker`, який працює, матиме варіант `Some` у `thread`, і коли ми хочемо очистити `Worker`, то ми замінимо `Some` на `None`, тож `Worker` не матиме потоку для виконання.

Отож ми знаємо, що хочемо оновити визначення `Worker` наступним чином:

Файл: `src/lib.rs`

```
struct Worker {
    id: usize,
    thread: Option<thread::JoinHandle<()>>,
}
```

Тепер покладемося на компілятор, щоб знайти інші місця, які потрібно змінити. При перевірці цього коду ми отримуємо дві помилки:

```

$ cargo check
  Checking hello v0.1.0 (file:///projects/hello)
error[E0599]: no method named `join` found for enum `Option` in the current scope
--> src/lib.rs:52:27
   |
52 |         worker.thread.join().unwrap();
   |                        ^^^^^ method not found in `Option<JoinHandle<()>>`
   |
note: the method `join` exists on the type `JoinHandle<()>`
help: consider using `Option::expect` to unwrap the `JoinHandle<()>` value,
panicking if the value is an `Option::None`
   |
52 |         worker.thread.expect("REASON").join().unwrap();
   |                        ++++++
   |

error[E0308]: mismatched types
--> src/lib.rs:72:22
   |
72 |         Worker { id, thread }
   |                  ^^^^^^^ expected enum `Option`, found struct `JoinHandle`
   |
= note: expected enum `Option<JoinHandle<()>>`
       found struct `JoinHandle<_>`
help: try wrapping the expression in `Some`
   |
72 |         Worker { id, thread: Some(thread) }
   |                        ++++++          +
   |

Some errors have detailed explanations: E0308, E0599.
For more information about an error, try `rustc --explain E0308`.
error: could not compile `hello` due to 2 previous errors

```

Розберімося з другою помилкою, що вказує, на код в кінці `Worker::new`; ми маємо обгорнути значення `thread` у `Some`, коли ми створюємо нового `Worker`. Зробіть такі зміни, щоб виправити цю помилку:

Файл: `src/lib.rs`

```

impl Worker {
    fn new(id: usize, receiver: Arc<Mutex<mpsc::Receiver<Job>>>) -> Worker {
        // --snip--

        Worker {
            id,
            thread: Some(thread),
        }
    }
}

```

Перша помилка знаходиться у нашій реалізації `Drop`. Ми вже згадували раніше, що збиралися викликати `take` для значення `Option`, щоб перемістити `thread` з `worker`. Наступні зміни роблять це:

Файл: `src/lib.rs`

```
impl Drop for ThreadPool {
    fn drop(&mut self) {
        for worker in &mut self.workers {
            println!("Shutting down worker {}", worker.id);

            if let Some(thread) = worker.thread.take() {
                thread.join().unwrap();
            }
        }
    }
}
```

Як обговорювалося в Розділі 17, метод `take` для `Option` забирає варіант `Some` і залишає `None` на своєму місці. Ми використовуємо `if let` для деструктуризації `Some` і отримуємо потік; тоді ми викликаємо `join` для потоку. Якщо потік `worker` вже `None`, то ми знаємо, що цей `worker` уже очистив свій потік, тож в цьому випадку нічого не відбудеться.

Подавання Сигналів Потокам Припинити Чекати на Завдання

Після всіх змін, які ми зробили, наш код компілюється без попереджень. Однак, погана новина в тому, що цей код ще не функціонує так, як ми цього хочемо. Причина в логіці в замиканнях, що виконуються в потоках екземплярів `Worker`: наразі, ми викликаємо `join`, але це не вимикає потоки, бо їхні цикли `loop` постійно шукають завдання. Якщо ми спробуємо очистити `ThreadPool` з нашою поточною реалізацією `drop`, головний потік заблокується назавжди, чекаючи на завершення першого потоку.

Щоб розв'язати цю проблему нам знадобиться зміна в реалізації `drop` для `ThreadPool`, а також зміна в циклі `Worker`.

Спершу ми змінимо реалізацію `drop` для `ThreadPool`, щоб явно очищати `sender` перед очікуванням на завершення потоків. Блок коду 20-23 показує зміни до `ThreadPool` для явного очищення `sender`. Ми використовуємо ту ж техніку `Option` і `take`, якою вже користувалися з потоком, щоб перемістити `sender` зі `ThreadPool`:

Файл: `src/lib.rs`

```

pub struct ThreadPool {
    workers: Vec<Worker>,
    sender: Option<mpsc::Sender<Job>>,
}
// --snip--
impl ThreadPool {
    pub fn new(size: usize) -> ThreadPool {
        // --snip--

        ThreadPool {
            workers,
            sender: Some(sender),
        }
    }

    pub fn execute<F>(&self, f: F)
    where
        F: FnOnce() + Send + 'static,
    {
        let job = Box::new(f);

        self.sender.as_ref().unwrap().send(job).unwrap();
    }
}

impl Drop for ThreadPool {
    fn drop(&mut self) {
        drop(self.sender.take());

        for worker in &mut self.workers {
            println!("Shutting down worker {}", worker.id);

            if let Some(thread) = worker.thread.take() {
                thread.join().unwrap();
            }
        }
    }
}

```

Блок коду 20-23: явне очищення `sender` перед приєднанням потоків `worker`

Очищення `sender` закриває канал, що позначає, що більше повідомлень не буде надіслано. Коли це стається, всі виклики до `recv`, зроблені `worker` в нескінченному циклі повернуть помилку. У Блоці коду 20-24 ми змінюємо цикл у `Worker` на для плавного виходу з циклу в цьому випадку, тобто потоки завершаться, коли реалізація `drop` для `ThreadPool` викличе для них `join`.

Файл: `src/lib.rs`

```
impl Worker {
    fn new(id: usize, receiver: Arc<Mutex<mpsc::Receiver<Job>>>) -> Worker {
        let thread = thread::spawn(move || loop {
            let message = receiver.lock().unwrap().recv();

            match message {
                Ok(job) => {
                    println!("Worker {id} got a job; executing.");

                    job();
                }
                Err(_) => {
                    println!("Worker {id} disconnected; shutting down.");
                    break;
                }
            }
        });

        Worker {
            id,
            thread: Some(thread),
        }
    }
}
```

Блок коду 20-24: явне переривання циклу, коли `recv` повертає помилку

Щоб побачити цей код в дії, змінімо `main`, щоб приймати лише два запити перед плавним вимиканням сервера, як показано в Блоці коду 20-25.

Файл: `src/lib.rs`

```
fn main() {
    let listener = TcpListener::bind("127.0.0.1:7878").unwrap();
    let pool = ThreadPool::new(4);

    for stream in listener.incoming().take(2) {
        let stream = stream.unwrap();

        pool.execute(|| {
            handle_connection(stream);
        });
    }

    println!("Shutting down.");
}
```

Блок коду 20-25: вимикання сервера виходом з циклу після обслуговування двох запитів

Вам би не сподобалося, якби справжній вебсервер вимикався після обслуговування лише двох запитів. Це код демонструє, що плавне вимикання і очищення працюють, як слід.

Метод `take`, визначений в трейті `Iterator`, обмежує ітерації максимум першими двома елементами. `ThreadPool` вийде з області видимості в кінці `main` і запуститься реалізація `drop`.

Запустіть сервер за допомогою `cargo run` і зробіть три запити. Третій запит призведе до помилки, і у вашому терміналі ви маєте побачити виведення, схоже на це:

```
$ cargo run
  Compiling hello v0.1.0 (file:///projects/hello)
  Finished dev [unoptimized + debuginfo] target(s) in 1.0s
  Running `target/debug/hello`
Worker 0 got a job; executing.
Shutting down.
Shutting down worker 0
Worker 3 got a job; executing.
Worker 1 disconnected; shutting down.
Worker 2 disconnected; shutting down.
Worker 3 disconnected; shutting down.
Worker 0 disconnected; shutting down.
Shutting down worker 1
Shutting down worker 2
Shutting down worker 3
```

Ви можете побачити іншу послідовність worker і виведених повідомлень. Ми бачимо цих повідомлень, як працює цей код; worker 0 і 3 отримали два перші запити. Сервер припинив приймати з'єднання після другого з'єднання, і реалізація `Drop` для `ThreadPool` почала виконуватися до того, як worker 3 розпочав роботу. Очищення `sender` від'єднує всіх workers і наказує їм вимкнутися. Кожен worker виводить повідомлення при роз'єднанні, і тоді пул потоків викликає `join`, чекаючи, доки кожен worker завершиться.

Зверніть увагу на один цікавий аспект конкретно цього виконання: `ThreadPool` очистив `sender`, і до того, як будь-який worker отримав помилку, ми намагалися приєднати worker 0. Worker 0 ще не отримав помилку від `recv`, тому основний потік заблокувався, чекаючи на завершення worker 0. Тим часом worker 3 отримав завдання, а потім всі потоки отримали помилку. Коли worker 0 завершив роботу, основний потік зачекав на завершення роботи решти workers. У цей момент, вони всі вийшли з циклів і зупинилися.

Вітання! Ми завершили наш проєкт; у нас є примітивний вебсервер, який використовує пул потоків для асинхронних відповідей. Ми можемо виконати плавне вимикання нашого сервера, яке очищує потоки в пулі.

Ось повний код для зв'язки:

Файл: src/main.rs

```
use hello::ThreadPool;
use std::fs;
use std::io::prelude::*;
use std::net::TcpListener;
use std::net::TcpStream;
use std::thread;
use std::time::Duration;

fn main() {
    let listener = TcpListener::bind("127.0.0.1:7878").unwrap();
    let pool = ThreadPool::new(4);

    for stream in listener.incoming().take(2) {
        let stream = stream.unwrap();

        pool.execute(|| {
            handle_connection(stream);
        });
    }

    println!("Shutting down.");
}

fn handle_connection(mut stream: TcpStream) {
    let mut buffer = [0; 1024];
    stream.read(&mut buffer).unwrap();

    let get = b"GET / HTTP/1.1\r\n";
    let sleep = b"GET /sleep HTTP/1.1\r\n";

    let (status_line, filename) = if buffer.starts_with(get) {
        ("HTTP/1.1 200 OK", "hello.html")
    } else if buffer.starts_with(sleep) {
        thread::sleep(Duration::from_secs(5));
        ("HTTP/1.1 200 OK", "hello.html")
    } else {
        ("HTTP/1.1 404 NOT FOUND", "404.html")
    };

    let contents = fs::read_to_string(filename).unwrap();

    let response = format!(
        "{}\r\nContent-Length: {}\r\n\r\n{}",
        status_line,
        contents.len(),
        contents
    );

    stream.write_all(response.as_bytes()).unwrap();
    stream.flush().unwrap();
}
```

Файл: src/lib.rs

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

pub struct ThreadPool {
    workers: Vec<Worker>,
    sender: Option<mpsc::Sender<Job>>,
}

type Job = Box<dyn FnOnce() + Send + 'static>;

impl ThreadPool {
    /// Create a new ThreadPool.
    ///
    /// The size is the number of threads in the pool.
    ///
    /// # Panics
    ///
    /// The `new` function will panic if the size is zero.
    pub fn new(size: usize) -> ThreadPool {
        assert!(size > 0);

        let (sender, receiver) = mpsc::channel();

        let receiver = Arc::new(Mutex::new(receiver));

        let mut workers = Vec::with_capacity(size);

        for id in 0..size {
            workers.push(Worker::new(id, Arc::clone(&receiver)));
        }

        ThreadPool {
            workers,
            sender: Some(sender),
        }
    }

    pub fn execute<F>(&self, f: F)
    where
        F: FnOnce() + Send + 'static,
    {
        let job = Box::new(f);

        self.sender.as_ref().unwrap().send(job).unwrap();
    }
}

impl Drop for ThreadPool {
    fn drop(&mut self) {
        drop(self.sender.take());
    }
}
```

```

        for worker in &mut self.workers {
            println!("Shutting down worker {}", worker.id);

            if let Some(thread) = worker.thread.take() {
                thread.join().unwrap();
            }
        }
    }

    struct Worker {
        id: usize,
        thread: Option<thread::JoinHandle<()>>,
    }

    impl Worker {
        fn new(id: usize, receiver: Arc<Mutex<mpsc::Receiver<Job>>>) -> Worker {
            let thread = thread::spawn(move || loop {
                let message = receiver.lock().unwrap().recv();

                match message {
                    Ok(job) => {
                        println!("Worker {} got a job; executing.", id);

                        job();
                    }
                    Err(_) => {
                        println!("Worker {} disconnected; shutting down.", id);
                        break;
                    }
                }
            });

            Worker {
                id,
                thread: Some(thread),
            }
        }
    }
}

```

Ми могли б зробити ще більше! Якщо ви хочете продовжити покращувати цей проєкт, ось деякі ідеї:

- Додати більше документації до `ThreadPool` та його публічних методів.
- Додати тести для функціонала бібліотеки.
- Замінити виклики `unwrap` надійнішою обробкою помилок.
- Використати `ThreadPool` для виконання інших завдань, крім обслуговування вебзапитів.
- Знайти крейт пулу потоків на crates.io та реалізувати аналогічний вебсервер за допомогою цього крейта. Тоді порівняти його API і надійність з пулом потоків, реалізованим нами.

Підсумок

Хороша робота! Ви дісталися кінця книги! Ми хочемо подякувати вам за те, що приєдналися до нас у цій подорожі по Rust. Тепер ви готові реалізовувати свої власні проекти Rust і допомагати іншим людям у їхніх проєктах. Не забувайте, що існує гостинна спільнота Растацеанців, які залюбки допоможуть вам з будь-якими викликами, з якими ви стикаєтеся у подорож по Rust.

Додатки

Наступні розділи містять довідковий матеріал, що стане в пригоді у вашій подорожі мовою Rust.

Додаток А: Ключові Слова

Цей список містить ключові слова, зарезервовані для поточного або майбутнього використання в мові Rust. Відтак, вони не можуть використовуватися як ідентифікатори (за винятком сирих ідентифікаторів, про які ми поговоримо в розділі “[Сирі ідентифікатори](#)”). Ідентифікатори - це імена функцій, змінних, параметрів, полів структур, модулів, крейтів, констант, макросів, статичних значень, атрибутів, типів, трейтів і часів існування.

Ключові Слова, що Наразі Використовуються

Нижче наведено список ключових слів, що наразі використовуються, з описом їхнього призначення.

- `as` - виконує примітивне перетворення типу, прибирає неоднозначність трейта, що містить елемент, або перейменовує елементи в інструкціях `use`
- `async` - повертає `Future` замість блокування поточного потоку
- `await` - призупиняє виконання, допоки результат `Future` не буде готовим
- `break` - негайно завершує цикл
- `const` - визначає константу чи константний вказівник
- `continue` - переходить до наступної ітерації циклу
- `crate` - у шляху модуля посилається на корінь крейта
- `dyn` - динамічна диспетчеризація трейтового об'єкта
- `else` - вказує альтернативний рукав для конструкцій керування `if` та `if let`
- `enum` - визначає енул
- `extern` - підключає зовнішню функцію або змінну
- `false` - булевий літерал "хиба"
- `fn` - визначає функцію чи тип вказівника на функцію
- `for` - перебирає елементи з ітератора, реалізує трейт або вказує більш значущий час існування
- `if` - виконує код залежно від умовного виразу
- `impl` - реалізує притаманну функціональність чи трейт
- `in` - частина синтаксису циклу `for`
- `let` - зв'язує змінну
- `loop` - створює цикл без умов зупинки
- `match` - зіставляє значення з шаблонами
- `mod` - визначає модуль

- `move` - передає замиканню володіння усіма захопленими значеннями
- `mut` - позначає мутабельність у посиланнях, вказівниках чи шаблонних зв'язуваннях
- `pub` - позначає публічну видимість полів структур, блоків `impl` або модулів
- `ref` - зв'язує за посиланням
- `return` - повертає значення з функції
- `Self` - псевдонім для типу, який ми визначаємо або реалізуємо
- `self` - суб'єкт методу або поточного модуля
- `static` - визначає час існування, що триває протягом усього часу виконання програми або глобальну змінну
- `struct` - визначає структуру
- `super` - посилається на батьківський модуль відносно поточного
- `trait` - визначає трейт
- `true` - булевий літерал "правда"
- `type` - визначає псевдонім типу чи асоційований тип
- `union` - визначає **об'єднання**; є ключовим словом виключно при проголошенні об'єднання
- `unsafe` - позначає небезпечний код, функції, трейти чи реалізації
- `use` - вводить символи в область видимості
- `where` - позначає обмеження типу
- `while` - цикл з умовою зупинки

Зарезервовані на Майбутнє Ключові Слова

Нижче наведено список ключових слів, що ще не мають функціональності, та є зарезервованими в Rust для можливого використання у майбутньому.

- `abstract`
- `become`
- `box`
- `do`
- `final`
- `macro`
- `override`
- `priv`
- `try`
- `typeof`
- `unsized`
- `virtual`
- `yield`

Сирі Ідентифікатори

Сирі ідентифікатори - це синтаксис, що дозволяє використовувати ключові слова там, де зазвичай це заборонено. Для використання сирого ідентифікатора, додайте до ключового слова префікс `r#`.

Наприклад `match` є ключовим словом. Якщо ви спробуєте скомпілювати цю функцію, що використовує `match` як ім'я:

Файл: `src/main.rs`

```
fn match(needle: &str, haystack: &str) -> bool {  
    haystack.contains(needle)  
}
```

то отримаєте таку помилку:

```
error: expected identifier, found keyword `match`  
--> src/main.rs:4:4  
   |  
4 | fn match(needle: &str, haystack: &str) -> bool {  
   |     ^^^^^ expected identifier, found keyword
```

Ця помилка показує, що не можна використовувати ключове слово `match` як ідентифікатор функції. Щоб використати `match` як назву функції, вам доведеться використати синтаксис сирого ідентифікатора, ось так:

Файл: `src/main.rs`

```
fn r#match(needle: &str, haystack: &str) -> bool {  
    haystack.contains(needle)  
}  
  
fn main() {  
    assert!(r#match("foo", "foobar"));  
}
```

Цей код компілюється без помилок. Зверніть увагу, що префікс `r#` в імені функції є як у визначенні, так і там, де ми викликаємо цю функцію в `main`.

Сирі ідентифікатори дозволяють вам використовувати будь-яке слово як ідентифікатор, навіть якщо воно зарезервоване як ключове слово. Це надає нам більше свободи для вибору назв ідентифікаторів, а також дозволяє інтегруватися з програмами, написаними мовами, де ці слова не є ключовими. Крім того, сирі ідентифікатори дозволяють використовувати бібліотеки, що написані в іншому виданні Rust, ніж ваше. Наприклад,

`try` не було ключовим словом у редакції 2015, але стало у редакції 2018. Якщо ви залежите від бібліотеки, що написана в редакції 2015 і має функцію `try`, вам знадобиться синтаксис сирого ідентифікатора, в цьому випадку `r#try`, щоб викликати цю функцію з коду в редакції 2018. Див. [Додаток E](#) щоб отримати більше інформації про видання.

Додаток В: Оператори та Символи

Цей додаток містить словник синтаксису Rust, включно з операторами та іншими символами, що вживаються самостійно або в контексті шляхів, узагальнених типів, обмежень трейтів, макросів, атрибутів, коментарів, кортежів і дужок.

Оператори

Таблиця В-1 містить оператори Rust, приклади, як ці оператори вживаються, коротке пояснення, і чи можна перевантажити оператор. Якщо оператор можна перевантажити, то вказується відповідний трейт, який треба використати для перевантаження.

Таблиця В-1: Оператори

Оператор	Приклад	Пояснення	Перевантаження?
<code>!</code>	<code>ident!(...)</code> <code>ident!{...}</code> <code>ident![...]</code>	Макрос	
<code>!</code>	<code>!expr</code>	Побітове чи логічне доповнення	<code>Not</code>
<code>!=</code>	<code>expr != expr</code>	Порівняння на нерівність	<code>PartialEq</code>
<code>%</code>	<code>expr % expr</code>	Арифметична остача	<code>Rem</code>
<code>%=</code>	<code>var %= expr</code>	Арифметична остача з присвоєнням	<code>RemAssign</code>
<code>&</code>	<code>&expr</code> , <code>&mut expr</code>	Позичання	
<code>&</code>	<code>&type</code> , <code>&mut type</code> , <code>&'a type</code> , <code>&'a mut type</code>	Тип позиченого вказівника	
<code>&</code>	<code>expr & expr</code>	Побітове AND	<code>BitAnd</code>
<code>&=</code>	<code>var &= expr</code>	Побітове AND з присвоєнням	<code>BitAndAssign</code>
<code>&&</code>	<code>expr && expr</code>	Логічне AND зі скороченим обчисленням	

Оператор	Приклад	Пояснення	Перевантаження?
<code>*</code>	<code>expr * expr</code>	Арифметичне множення	<code>Mul</code>
<code>*=</code>	<code>var *= expr</code>	Арифметичне множення з присвоєнням	<code>MulAssign</code>
<code>*</code>	<code>*expr</code>	Розіменування	<code>Deref</code>
<code>*</code>	<code>*const type</code> , <code>*mut type</code>	Сирий вказівник	
<code>+</code>	<code>trait + trait</code> , <code>'a + trait</code>	Комбіноване обмеження типу	
<code>+</code>	<code>expr + expr</code>	Арифметичне додавання	<code>Add</code>
<code>+=</code>	<code>var += expr</code>	Арифметичне додавання з присвоєння	<code>AddAssign</code>
<code>,</code>	<code>expr, expr</code>	Роздільник аргументів чи елементів	
<code>-</code>	<code>- expr</code>	Обчислення арифметичного протилежного	<code>Neg</code>
<code>-</code>	<code>expr - expr</code>	Арифметичне віднімання	<code>Sub</code>
<code>-=</code>	<code>var -= expr</code>	Арифметичне віднімання з присвоєнням	<code>SubAssign</code>
<code>-></code>	<code>fn(...) -></code> <code>type</code> , <code> ... -></code> <code>type</code>	Тип, що повертає функція чи замикання	
<code>.</code>	<code>expr.ident</code>	Доступ до складової	
<code>..</code>	<code>..</code> , <code>expr..</code> , <code>..expr</code> , <code>expr..expr</code>	Діапазонний літерал, не включає праву межу	<code>PartialOrd</code>
<code>..=</code>	<code>..=expr</code> , <code>expr..=expr</code>	Діапазонний літерал, включає праву межу	<code>PartialOrd</code>
<code>..</code>	<code>..expr</code>	Оновлення структурного літералу	

Оператор	Приклад	Пояснення	Перевантаження?
<code>..</code>	<code>variant(x, ..), struct_type { x, .. }</code>	Шаблон зв'язування "і решта"	
<code>...</code>	<code>expr...expr</code>	(Застарілий, використовуйте натомість <code>..=</code>) У шаблоні: діапазонний шаблон, включає межу	
<code>/</code>	<code>expr / expr</code>	Арифметичне ділення	<code>Div</code>
<code>/=</code>	<code>var /= expr</code>	Арифметичне ділення з присвоєнням	<code>DivAssign</code>
<code>:</code>	<code>pat: type, ident: type</code>	Обмеження	
<code>:</code>	<code>ident: expr</code>	Ініціалізатор поля структури	
<code>:</code>	<code>'a: loop {...}</code>	Мітка циклу	
<code>;</code>	<code>expr;</code>	Завершення структури чи елементу	
<code>;</code>	<code>[...; len]</code>	Частина синтаксису масиву фіксованого розміру	
<code><<</code>	<code>expr << expr</code>	Зсув ліворуч	<code>Shl</code>
<code><<=</code>	<code>var <<= expr</code>	Зсув ліворуч із присвоєнням	<code>ShlAssign</code>
<code><</code>	<code>expr < expr</code>	Порівняння менше	<code>PartialOrd</code>
<code><=</code>	<code>expr <= expr</code>	Порівняння менше або дорівнює	<code>PartialOrd</code>
<code>=</code>	<code>var = expr, ident = type</code>	Присвоєння/еквівалентність	
<code>==</code>	<code>expr == expr</code>	Порівняння рівності	<code>PartialEq</code>
<code>=></code>	<code>pat => expr</code>	Частина синтаксису рукава match	
<code>></code>	<code>expr > expr</code>	Порівняння більше	<code>PartialOrd</code>
<code>>=</code>	<code>expr >= expr</code>	Порівняння більше або дорівнює	<code>PartialOrd</code>
<code>>></code>	<code>expr >> expr</code>	Зсув праворуч	<code>Shr</code>

Оператор	Приклад	Пояснення	Перевантаження?
<code>>>=</code>	<code>var >>= expr</code>	Зсув праворуч із присвоєнням	<code>ShrAssign</code>
<code>@</code>	<code>ident @ pat</code>	Зв'язування шаблону	
<code>^</code>	<code>expr ^ expr</code>	Побітове виключне OR	<code>BitXor</code>
<code>^=</code>	<code>var ^= expr</code>	Побітове виключне OR з присвоєнням	<code>BitXorAssign</code>
<code> </code>	<code>pat pat</code>	Альтернативні шаблони	
<code> </code>	<code>expr expr</code>	Побітове OR	<code>BitOr</code>
<code> =</code>	<code>var = expr</code>	Побітове OR із присвоєнням	<code>BitOrAssign</code>
<code> </code>	<code>expr expr</code>	Логічне OR зі скороченим обчисленням	
<code>?</code>	<code>expr?</code>	Передавання помилки	

Неоператорні Символи

Наступний список містить усі символи, що не працюють як оператори; тобто, вони не поведуться як виклик функції чи методу.

Таблиця В-2 містить символи, що вживаються самостійно і є коректними у різних місцях.

Таблиця В-2: Окремий синтаксис

Символ	Пояснення
<code>'ident</code>	Іменований час існування чи мітка циклу
<code>...u8</code> , <code>...i32</code> , <code>...f64</code> , <code>...usize</code> і т.д.	Числовий літерал певного типу
<code>"..."</code>	Стрічковий літерал
<code>r"..."</code> , <code>r#"..."#</code> , <code>r##"..."##</code> і т.д.	Сирий стрічковий літерал, символи екранування не обробляються
<code>b"..."</code>	Байтовий стрічковий літерал; створює масив байтів замість стрічки
<code>br"..."</code> , <code>br#"..."#</code> , <code>br##"..."##</code> і т.д.	Сирий байтовий стрічковий літерал, комбінація сирого і байтового стрічкових літералів
<code>'...'</code>	Символьний літерал

Символ	Пояснення
<code>b'...'</code>	Байтовий літерал ASCII
<code> ... expr</code>	Замикання
<code>!</code>	Завжди порожній нижній тип для функцій, що не завершуються
<code>_</code>	“Ігнороване” зв'язування шаблону; також використовується для покращення читабельності цілочисельних літералів

Таблиця В-3 містить символи, що зустрічаються в контексті шляхів до елемента в ієрархії модулів.

Таблиця В-3: Синтаксис, що стосується шляхів

Символ	Пояснення
<code>ident::ident</code>	Шлях до простору імен
<code>::path</code>	Шлях відносно кореня крейта (тобто явно заданий абсолютний шлях)
<code>self::path</code>	Шлях відносно поточного модуля (тобто явно заданий відносний шлях).
<code>super::path</code>	Шлях відносно батьківського модуля
<code>type::ident</code> , <code><type as trait>::ident</code>	Асоційовані константи, функції та типи
<code><type>::...</code>	Асоційований елемент для типу, що не можна прямо назвати (наприклад <code><&T>::...</code> , <code><[T]>::...</code> і т.д..)
<code>trait::method(...)</code>	Уточнення неоднозначного виклику методу називанням трейту, що визначає його
<code>type::method(...)</code>	Уточнення неоднозначного виклику методу називанням типу, для якого він визначений
<code><type as trait>::method(...)</code>	Уточнення неоднозначного виклику методу називанням трейту і типу

Таблиця В-4 містить символи, що зустрічаються в контексті параметрів узагальнених типів.

Таблиця В-4: Узагальнені типи

Символ	Пояснення
<code>path<...></code>	Вказує параметри до узагальненого типу в типі (наприклад <code>Vec<u8></code>)
<code>path::<code><...></code>, method::<code><...></code></code>	Вказує параметри до узагальненого типу, функції чи методу у виразі; часто зветься "турборибкою" (наприклад, <code>"42".parse::<i>i32>()</i></code>)
<code>fn ident<...> ...</code>	Визначення узагальненої функції
<code>struct ident<...> ...</code>	Визначення узагальненої структури
<code>enum ident<...> ...</code>	Визначення узагальненого енуму
<code>impl<...> ...</code>	Визначення узагальненої реалізації
<code>for<...> type</code>	Обмеження часу існування вищого рівня
<code>type<ident=type></code>	Узагальнений тип, де один чи більше асоційованих типів мають конкретні значення (наприклад, <code>Iterator<Item=T></code>)

Таблиця В-5 містить символи, що зустрічаються в контексті обмеження параметрів узагальненого типу обмеженнями трейта.

Таблиця В-5: Обмеження трейтів

Символ	Пояснення
<code>T: U</code>	Узагальнений параметр <code>T</code> обмежений типами, що реалізують <code>U</code>
<code>T: 'a</code>	Узагальнений тип <code>T</code> має пережити час існування <code>'a</code> (тобто тип не може транзитивно містити посилання з часом існування меншим ніж <code>'a</code>)
<code>T: 'static</code>	Узагальнений тип <code>T</code> не містить позичених посилань, окрім <code>'static</code>
<code>'b: 'a</code>	Узагальнений час існування <code>'b</code> має перевищувати час існування <code>'a</code>
<code>T: ?Sized</code>	Узагальнений тип <code>T</code> має бути типом з динамічним розміром
<code>'a + trait, trait + trait</code>	Комбіноване обмеження типу

Таблиця В-6 містить символи, що зустрічаються в контексті виклику чи визначення макросів і зазначення атрибутів елементу.

Таблиця В-6: Макроси та атрибути

Символ	Пояснення
<code>#[meta]</code>	Зовнішній атрибут
<code>#![meta]</code>	Внутрішній атрибут
<code>\$ident</code>	Підставлення в макросі
<code>\$ident:kind</code>	Захоплення в макросі
<code>\$(...)...</code>	Повторення в макросі
<code>ident!(...)</code> , <code>ident!{...}</code> , <code>ident![...]</code>	Виклик макросу

Таблиця В-7 містить символи для створення коментарів.

Таблиця В-7: Коментарі

Символ	Пояснення
<code>//</code>	Коментар-рядок
<code>///!</code>	Внутрішній документаційний коментар-рядок
<code>///</code>	Зовнішній документаційний коментар-рядок
<code>/*...*/</code>	Коментар-блок
<code>/*!...*/</code>	Внутрішній документаційний коментар-блок
<code>**...*/</code>	Зовнішній документаційний коментар-блок

Таблиця В-8 містить символи, що зустрічаються в контексті використання кортежів.

Таблиця В-8: Кортежі

Символ	Пояснення
<code>()</code>	Порожній кортеж (також відомий як одиничний тип), як літерал, так і тип
<code>(expr)</code>	Вираз у дужках
<code>(expr,)</code>	Вираз - кортеж з одного елементу
<code>(type,)</code>	Тип - кортеж з одного елементу
<code>(expr, ...)</code>	Вираз - кортеж з декількома елементами
<code>(type, ...)</code>	Тип - кортеж з декількома елементами
<code>expr(expr, ...)</code>	Виклик функції; також використовується для ініціалізації кортежів-структур і кортежів-варіантів еnumів
<code>expr.0</code> , <code>expr.1</code> , і т.д.	Індексація кортежу

Таблиця В-9 містить контексти, в яких застосовуються фігурні дужки.

Таблиця В-9: Фігурні Дужки

Контекст	Пояснення
<code>{...}</code>	Вираз-блок
<code>Type {...}</code>	Літерал структури

Таблиця В-10 містить контексти, в яких застосовуються квадратні дужки.

Таблиця В-10: Квадратні Дужки

Контекст	Пояснення
<code>[...]</code>	Літерал масиву
<code>[expr; len]</code>	Літерал масиву, що містить <code>len</code> копій <code>expr</code>
<code>[type; len]</code>	Тип масиву, що містить <code>len</code> екземплярів типу <code>type</code>
<code>expr[expr]</code>	Індексація колекції. Може бути перевантаженою (<code>Index</code> , <code>IndexMut</code>)
<code>expr[..]</code> , <code>expr[a..]</code> , <code>expr[..b]</code> , <code>expr[a..b]</code>	Індексація колекції, що має виробляти слайс за допомогою <code>Range</code> , <code>RangeFrom</code> , <code>RangeTo</code> , or <code>RangeFull</code> як індексу

Додаток С: Похідні Трейти

У різних місцях книги ми обговорювали атрибут `derive`, який можна застосувати до визначення структури або енуму. Атрибут `derive` генерує код, що реалізує трейт з власною реалізацією за замовчуванням для типу, який ви позначили за допомогою синтаксичної конструкції `derive`.

У цьому додатку наведено перелік усіх трейтів стандартної бібліотеки, які ви можете застосовувати з `derive`. Кожен розділ покриває:

- Які оператори та методи дозволить застосування цього трейту
- Що робить реалізація трейту, створена за допомогою `derive`
- Що реалізація трейту позначає для типу
- Умови, за яких вам можна чи не можна реалізовувати трейт
- Приклади операцій, що вимагають цього трейту

Якщо ви хочете отримати поведінку, відмінну від наданої атрибутом `derive`, зверніться до [документації стандартної бібліотеки](#) для кожного трейту, щоб дізнатися подробиці, як реалізувати їх вручну.

Тут наведений повний список трейтів, визначених у стандартній бібліотеці, які можуть бути реалізовані для ваших типів за допомогою `derive`. Інші трейти, визначені в стандартній бібліотеці, не мають притомної поведінки за замовчуванням, тому ви повинні реалізувати їх так, щоб вони мали сенс для досягнення вашої конкретної мети.

Приклад трейту, який не можна вивести, це `Display`, що обробляє форматування для кінцевих користувачів. Ви завжди маєте продумати відповідний спосіб, як показати ваш тип кінцевому користувачеві. Які частини типу має кінцевий користувач право бачити? Які частини будуть для них актуальними? Який формат даних буде для них найбільш адекватним? Компілятор Rust не може цього знати, тож не може й забезпечити відповідну поведінку за замовчуванням.

Список вивідних трейтів, наданий у цьому додатку, не є вичерпним: бібліотеки можуть реалізувати `derive` для своїх власних трейтів, що робить список трейтів, які ви можете використовувати з `derive`, повністю відкритим. Реалізація `derive` включає в себе використання процедурного макросу, про що розповідається в підрозділі "[Макроси](#)" Розділу 19.

Debug - Форматування для Програмістів

Трейт `Debug` надає зневаджувальний формат у рядках форматування, який зазначається додаванням `:?` у заповнювач `{}`.

Трейт `Debug` дозволяє виводити екземпляри типу для цілей зневадження, щоб ви та інші програмісти, які використовують ваш тип, могли переглянути екземпляр у певному місці виконання програми.

Трейт `Debug` потрібен, наприклад, при використанні макросу `assert_eq!`. Цей макрос виводить значення екземплярів, переданих йому аргументами, якщо перевірка на рівність не пройшла, щоб програмісти могли побачити, чому два екземпляри не були однаковими.

PartialEq та Eq для Порівняння на Рівність

Трейт `PartialEq` дозволяє вам порівнювати екземпляри типу, щоб перевірити на рівність, і дозволяє використання операторів `==` та `!=`.

Виведення `PartialEq` реалізує метод `eq`. Коли `PartialEq` виведено для структури, два екземпляри рівні лише тоді, коли *всі* поля є рівними, і не рівні, якщо хоча б в одному полі розрізняються. При виведенні на енумах кожен варіант дорівнює собі і не дорівнює іншим варіантам.

Трейт `PartialEq` потрібен, наприклад, для макросу `assert_eq!`, який має бути в змозі порівняти два екземпляри типу на рівність.

Трейт `Eq` не має методів. Його мета - позначити, що кожне значення цього типу дорівнює самому собі. Трейт `Eq` може застосовуватися лише для типів, які також реалізують `PartialEq`, хоча не всі типи, що реалізують `PartialEq`, можуть реалізовувати `Eq`. Одним прикладом такого типу є числа з рухомою комою: реалізація чисел з рухомою комою позначає, що два екземпляри зі значенням не-число (`NaN`) не рівні між собою.

Приклад, коли `Eq` є необхідним, це ключі у `HashMap<K, V>`, щоб `HashMap<K, V>` завжди міг визначити, чи два ключі є однаковими.

PartialOrd та Ord для Порівнянь Упорядкування

Трейт `PartialOrd` дозволяє порівнювати екземпляри типу з метою сортування. Для типу, що реалізує `PartialOrd`, можуть застосовуватися оператори `<`, `>`, `<=` та `>=`. Ви можете застосувати трейт `PartialOrd` лише для типів, що також реалізують `PartialEq`.

Виведення `PartialOrd` реалізує метод `partial_cmp`, який повертає `Option<Ordering>`, що буде `None`, якщо вказані значення неможливо впорядкувати. Приклад значення, яке не можна впорядкувати, навіть якщо більшість значень такого типу можуть бути порівнянні, це значення не-число (`NaN`) чисел з рухомою комою. Виклик `partial_cmp` для будь-якого числа з рухомою комою і значення `NaN` поверне `None`.

При виведенні для структур `PartialOrd` порівнює два екземпляри, порівнюючи значення кожного поля у порядку, в якому ці поля присутні у проголошенні структури. При виведенні для еnumів, варіанти еnumу, проголошені раніше, вважаються меншими, ніж вказані пізніше.

Трейт `PartialOrd` потрібен, наприклад, методу `gen_range` з крейту `rand`, що генерує випадкові значення в інтервалі, заданому інтервальним виразом.

Трейт `Ord` вказує, для будь-яких двох значень анотованого типу буде існувати коректний порядок. Трейт `Ord` реалізує метод `cmp`, який повертає `Ordering`, а не `Option<Ordering>`, бо правильний порядок є завжди можливим. Ви можете застосувати трейт `Ord` лише для типів, які також реалізують `PartialOrd` і `Eq` (а `Eq` вимагає `PartialEq`). При виведенні на структурах і еnumах `cmp` поводитьься так само, як і виведена реалізація `partial_cmp` для `PartialOrd`.

Приклад потреби трейту `Ord` - зберігання значень у `BTreeSet<T>`, структурі даних, що зберігає дані на основі порядку сортування значень.

Clone і Copy для Дублікації Даних

Трейт `Clone` дозволяє явно створити глибоку копію значення, і процес дублікації може містити виконання довільного коду і копіювання даних у купі. Дивіться підрозділ “Як взаємодіють змінні з даними: клонування” Розділу 4 для додаткової інформації про `Clone`.

Виведення `Clone` реалізує метод `clone`, який при реалізації для всього типу викликає `clone` для кожної частини типу. Це означає, що всі поля і значення типу мають також реалізовувати `Clone`, щоб можна було вивести `Clone`.

Приклад, коли потрібен `Clone`, це виклик методу `to_vec` для слайса. Слайс не володіє екземплярами типу, які він містить, але вектор, повернутий з `to_vec`, мусить володіти своїми екземплярами, тож `to_vec` викликає `clone` для кожного елемента. Тож тип, що зберігається в слайсі, має реалізовувати `Clone`.

Трейт `Copy` дозволяє вам дублікацію значення, копіюючи біти, збережені в стеку, без жодного довільного коду. Дивіться підрозділ “Дані в стеку: копіювання” Розділу 4 для додаткової інформації про `Copy`.

Трейт `Copy` не визначає жодних методів, щоб не дозволити програмістам перевантажити ці методи і порушити припущення, що додатковий код не буде виконано. Таким чином, всі програмісти можуть виходити з припущення, що копіювання значення є дуже швидким.

Ви можете вивести `Copy` для будь-якого типу, всі частини якого реалізують `Copy`. Тип, що реалізує `Copy`, також має реалізовувати `Clone`, `Copy` має тривіальну реалізацію `Clone`, що робить те саме, що й `Copy`.

Трейт `Copy` рідко коли буває потрібна; типи, що реалізують `Copy`, мають доступні оптимізації, завдяки яким не треба викликати `clone`, що робить код більш виразним.

Все, що можливо з `Copy`, ви також можете досягти за допомогою `Clone`, але код може бути повільнішим і вам доведеться місцями використовувати `clone`.

`Hash` для Перетворення Значення у Значення Фіксованого Розміру

Трейт `Hash` дозволяє взяти екземпляр типу довільного розміру і перетворити його у значення фіксованого розміру за допомогою хеш-функції. Виведення `Hash` реалізовує метод `hash`. Виведена реалізація методу `hash` комбінує результати викликів `hash` для кожної частини типу, що означає, що всі поля і значення також мають реалізовувати `Hash` для виведення `Hash`.

Приклад, коли потрібен `Hash`, це зберігання ключів у `HashMap<K, V>`, щоб ефективно зберігати дані.

`Default` для Значень за Замовчуванням

Трейт `Default` дозволяє вам створювати значення за замовчуванням для типу. Виведення `Default` реалізовує функцію `default`. Виведена реалізація функції `default` викликає функцію `default` для кожної частини типу, що означає, що всі поля або значення в типі також повинні реалізовувати `Default`, щоб можна було вивести `Default`.

Функція `Default::default` зазвичай використовується у поєднанні з синтаксисом оновлення структури, про який ідеться в підрозділі "Створення екземплярів з інших екземплярів за допомогою синтаксису оновлення структур" Розділу 5. Ви можете виставити кілька полів конструкції, а потім встановити і використати значення за замовчуванням для решти полів за допомогою `..Default::default()`.

Наприклад, трейт `Default` необхідний, коли ви використовуєте метод `unwrap_or_default` для екземплярів `Option<T>`. Якщо `Option<T>` має значення `None`, метод

`unwrap_or_default` поверне результат `Default::default` для типу `T`, що знаходиться в `Option<T>`. [ch05-01-defining-structs.html#creating-instances-from-other-instances-with-struct-update-syntax](#) [ch04-01-what-is-ownership.html#stack-only-data-copy](#) [ch04-01-what-is-ownership.html#ways-variables-and-data-interact-clone](#)

Додаток D - Корисні Інструменти Розробки

В цьому додатку ми говоримо про деякі корисні інструменти розробки, які надає проєкт Rust. Ми оглянемо автоматичне форматування, швидкі способи застосувати виправлення для попереджень, linter і інтеграцію з IDE.

Автоматичне Форматування з `rustfmt`

Інструмент `rustfmt` переформатує ваш код відповідно до стилю коду спільноти. Багато спільних проєктів використовують `rustfmt` для запобігання суперечкам, який стиль використовувати під час написання Rust: всі формтують свій код за допомогою цього інструменту.

Щоб встановити `rustfmt`, введіть наступне:

```
$ rustup component add rustfmt
```

Ця команда дає вам `rustfmt` і `cargo-fmt`, подібно до того, як Rust дає вам `rustc` та `cargo`. Щоб відформатувати будь-який проєкт Cargo, введіть наступне:

```
$ cargo fmt
```

Запуск цієї команди переформатує весь код Rust в поточному крейті. Це має змінити лише стиль коду, а не його семантику. Для отримання додаткової інформації по `rustfmt` перегляньте [його документацію](#).

Виправте Ваш Код з `rustfix`

Інструмент `rustfix` постачається разом із встановленим Rust і може автоматично виправляти попередження компілятора, які мають чіткий спосіб виправлення проблеми, що, ймовірно, є саме те, що ви хочете. Ймовірно, ви вже бачили попередження компілятора. Наприклад, розглянемо цей код:

Файл: `src/main.rs`

```
fn do_something() {}

fn main() {
    for i in 0..100 {
        do_something();
    }
}
```

Тут ми викликаємо функцію `do_something` 100 разів. але ми ніколи не використовуємо змінну `i` в тілі циклу `for`. Rust попереджає нас про це:

```
$ cargo build
   Compiling myprogram v0.1.0 (file:///projects/myprogram)
warning: unused variable: `i`
--> src/main.rs:4:9
   |
4  |     for i in 0..100 {
   |         ^ help: consider using `_i` instead
   |
= note: #[warn(unused_variables)] on by default

Finished dev [unoptimized + debuginfo] target(s) in 0.50s
```

Попередження пропонує нам використати `_i` як назву змінної: підкреслення вказує на те, що не збираємося використовувати цю змінну. Ми можемо автоматично застосувати цю пропозицію, використовуючи інструмент `rustfix`, запустивши команду `cargo fix`:

```
$ cargo fix
   Checking myprogram v0.1.0 (file:///projects/myprogram)
   Fixing src/main.rs (1 fix)
Finished dev [unoptimized + debuginfo] target(s) in 0.59s
```

Коли ми знову подивимось на `src/main.rs`, то побачимо, що `cargo fix` змінив код:

Файл: `src/main.rs`

```
fn do_something() {}

fn main() {
    for _i in 0..100 {
        do_something();
    }
}
```

Змінна циклу `for` тепер називається `_i`, і попередження більше не з'являється.

Ви також можете використовувати команду `cargo fix` для перенесення коду між різними виданнями Rust. Про видання йдеться у Додатку E.

Більше Підказок від Clippy

Clippy - це інструмент, що містить набір підказок для аналізу вашого коду, щоб ви могли виявити загальні помилки та поліпшити ваш Rust-код.

Щоб встановити Clippy, введіть наступне:

```
$ rustup component add clippy
```

Щоб запустити lint Clippy для будь-якого проєкту Cargo, введіть наступне:

```
$ cargo clippy
```

Наприклад, ви пишете програму, що використовує наближення математичної константи, такої як π , як ця програма:

Файл: `src/main.rs`

```
fn main() {  
    let x = 3.1415;  
    let r = 8.0;  
    println!("the area of the circle is {}", x * r * r);  
}
```

Запуск `cargo clippy` на цьому проєкті призводить до помилки:

```
error: approximate value of `{32, 64}::consts::PI` found  
--> src/main.rs:2:13  
2 |         let x = 3.1415;  
  |                 ^^^^^^^  
= note: `[deny(clippy::approx_constant)]` on by default  
= help: consider using the constant directly  
= help: for further information visit https://rust-lang.github.io/rust-clippy/master/index.html#approx_constant
```

Ця помилка повідомляє, що Rust вже має більш точну константу `pi` і що ваша програма буде коректнішою, якщо ви скористаєтеся цією константою. Тоді ви зміните свій код, щоб використовувати константу `pi`. Наступний код не призводить до помилок або попереджень від Clippy:

Файл: src/main.rs

```
fn main() {  
    let x = std::f64::consts::PI;  
    let r = 8.0;  
    println!("the area of the circle is {}", x * r * r);  
}
```

Для отримання додаткової інформації про Clippy перегляньте [його документацію](#).

Інтеграція з IDE з `rust-analyzer`

Для покращення інтеграції в IDE спільнота Rust рекомендує використовувати `rust-analyzer`. Цей інструмент є набором доовколокомпіляторних утиліт, що спілкуються за допомогою [Language Server Protocol](#), що є специфікацією для IDE і мов програмування для взаємного спілкування. Різні клієнти можуть використовувати

`rust-analyzer`, наприклад [the Rust analyzer plug-in for Visual Studio Code](#).

Відвідайте [домашню сторінку проєкту Rust-analyzer](#) на для інструкцій з установки, потім встановіть підтримку мовного сервера у вашому IDE. Ваше IDE набуде можливостей, таких, як автодоповнення, перехід до визначення і вбудовані помилки.

Додаток Е - Видання

У розділі 1 ви бачили, що `cargo new` додає трохи метаданих до файлу *Cargo.toml* стосовно видання (edition). Цей додаток пояснює, що це означає!

Мова Rust і компілятор мають шеститижневий цикл випуску, що означає, що користувачі отримують постійний потік нового функціоналу. Інші мови програмування випускають великі зміни і рідше; Rust випускає менші оновлення частіше. За певний час, усі ці маленькі зміни накопичуються. Але від випуску до випуску може бути складно озирнутися і сказати "Ух, між Rust 1.10 та Rust 1.31, Rust так сильно змінився!"

Кожні два чи три роки команда Rust випускає нове *видання* Rust. Кожне видання збирає функціонал, що утворює чіткий пакет, з повністю оновленою документацією та інструментарієм. Нові видання постачаються як частина звичайного шеститижневого процесу випусків.

Видання слугують різним цілям для різних людей:

- Для активних користувачів Rust нове видання збирає накопичені зміни у легкозрозумілий пакет.
- Для некористувачів, нове видання подає сигнал, що сталися якісь суттєві досягнення, завдяки чому Rust, можливо, став вартим більшої уваги.
- Для тих, хто розробляє Rust, нове видання надає точку відліку для процесу в цілому.

На час написання цього, доступні три видання Rust: Rust 2015, Rust 2018 і Rust 2021. Ця книжка написана з використанням ідіом видання Rust 2021.

Ключ `edition` у *Cargo.toml* указує, яке видання компілятор має використати для вашого коду. Якщо ключа немає, Rust використовує `2015` як значення видання з міркувань зворотної сумісності.

Кожен проєкт може обрати видання, відмінне від видання 2015 за умовчанням. Видання можуть містити несумісні зміни, такі як появу нового ключового слова, що конфліктує із ідентифікаторами в коді. Однак, якщо ви не погодитеся на ці зміни, ваш код буде продовжувати компілюватися, навіть коли ви оновите версію компілятора Rust, яку ви використовуєте.

Всі версії компілятора Rust підтримують усі видання, що існували до випуску цього компілятора, і вони можуть зв'язувати крейти усіх підтримуваних видань. Зміни видання лише впливають на те, як компілятор початково розбирає код. Таким чином, якщо ви використовуєте Rust 2015, а одна з ваших залежностей використовує Rust 2018, ваш

проект скомпільюється і зможе використовувати цю залежність. Зворотна ситуація, де ваш проект використовує Rust 2018, а залежність використовує Rust 2015, теж працює.

Одразу зазначимо: більшість функціонала доступно у всіх виданнях. Розробники, що використовують будь-яку редакцію Rust, продовжать бачити покращення, коли виходитимуть нові стабільні випуски. Однак у певних випадках, переважно коли додаються нові ключові слова, деякий новий функціонал може бути доступним лише в пізніших виданнях. Вам доведеться перемкнути видання, щоб мати повну змогу використовувати такий функціонал.

Для більш докладної інформації, *Edition Guide* є вичерпною книжкою про видання, де перелічуються відмінності між виданнями та пояснюється, як автоматично оновити код до нової редакції за допомогою `cargo fix`.

Додаток F: Переклади Книги

Ресурси іншими мовами. Більшість з них не завершена; прогляньте [позначки стану перекладу](#), щоб долучитися чи дати нам знати про новий переклад!

- [Português \(BR\)](#)
- [Português \(PT\)](#)
- [简体中文](#)
- [正體中文](#)
- [Українська](#)
- [Español, альтернативна](#)
- [Italiano](#)
- [Русский](#)
- [한국어](#)
- [日本語](#)
- [Français](#)
- [Polski](#)
- [Cebuano](#)
- [Tagalog](#)
- [Esperanto](#)
- [ελληνική](#)
- [Svenska](#)
- [Farsi](#)
- [Deutsch](#)
- [हिंदी](#)
- [ไทย](#)
- [Danske](#)

Додаток G - як Розробляється Rust і "Нічний Rust"

Цей додаток розповідає про те, як робиться Rust і як це впливає на вас як на розробника на Rust.

Стабільність без Застою

Як мова, Rust *багато* піклується про стабільність вашого коду. Ми хочемо, щоб Rust був якомога надійнішим фундаментом, на якому ви зможете будувати, і якби все постійно змінювалося, це було б неможливо. У той самий час ми, якщо ми не зможемо експериментувати з новим функціоналом, то можемо пропустити важливі недоліки аж до їхнього релізу, коли ми вже не зможемо це змінити.

Наше розв'язання цієї проблеми - це те, що ми зовемо "стабільність без застою", і наш керівний принцип такий: ви ніколи не маєте боятися оновлення до нової версії стабільного Rust. Кожне оновлення має бути безболісним, але також має приносити нові можливості, менше помилок і швидший час компіляції.

Ту-туу! Канали Випуску і Залізничний Розклад

Розробка Rust відбувається за *залізничним розкладом*. Тобто вся розробка робиться в гілці **master** репозиторію Rust. Релізи слідує залізничній моделі випусків програмного забезпечення (software release train model), яку використовують Cisco IOS та інші проекти програмного забезпечення. Існують три *канали релізів* Rust:

- Щонічний (nightly)
- Бета (beta)
- Стабільний (stable)

Більшість розробників Rust в переважно використовують стабільний канал, але ті, хто хоче спробувати експериментальні нові функції, можуть використовувати щонічний або бету.

Ось приклад того, як працює процес розробки та релізів: припустімо, що команда Rust працює над релізом Rust 1.5. Цей реліз відбувся у грудні 2015 року, але він забезпечить нам реалістичні номери версій. У Rust додається новий функціонал: новий коміт з'являється у гілці **master**. Кожної ночі виробляється нова щонічна версія Rust. Кожен

день відбувається реліз, і ці релізи створюються автоматично нашою інфраструктурою релізів. Тож із плином часу наші релізи виглядають ось так, по одному за ніч:

```
nightly: * - - * - - *
```

Кожні шість тижнів настає час підготувати новий реліз! Гілка **beta** у репозиторію Rust відгалужується від гілки **master**, що належить щонічній версії. Тепер є два релізи:

```
nightly: * - - * - - *
          |
beta:    *
```

Більшість користувачів Rust не використовують бета-релізи активно, а лише тестують на беті у своїх системах неперервної інтеграції (CI), щоб допомогти Rust знайти можливі регресії. Тим часом нові щонічні релізи з'являються кожної ночі:

```
nightly: * - - * - - * - - * - - *
          |
beta:    *
```

Припустимо, було знайдено регресію. Добре, що ми мали якийсь час для перевірки бета-релізу перед тим, як регресія прокралася до стабільного релізу! Виправлення застосовується до **master**, тож тепер щонічна версія виправлена, а потім виправлення переноситься (backport) у **бета-гілку** і робиться реліз:

```
nightly: * - - * - - * - - * - - * - - *
          |
beta:    * - - - - - - - - *
```

Шість тижнів минуло після створення першої бети, настав час для стабільної версії!

Стабільна гілка робиться з гілки **beta**:

```
nightly: * - - * - - * - - * - - * - - * - * - *
          |
beta:    * - - - - - - - - *
          |
stable:  *
```

Ура! Rust 1.5 зроблено! Проте ми забули одну річ: оскільки минуло шість тижнів, нам також потрібна нова бета *наступної* версії Rust, 1.6. Тож після того, як **стабільна** версія відгалужується від **бети**, наступна версія **бети** знову відгалужується від **щонічної** версії:

```

nightly: * - - * - - * - - * - - * - - * - * - *
          |                                     |
beta:    * - - - - - - - *                   *
          |
stable:   *

```

Це зветься "залізничною моделлю", тому що кожні шість тижнів реліз "залишає станцію", але все ще потрібно здійснити подорож в бета-каналі, перш ніж він прибуде як стабільна версія.

Релізи Rust випускаються що шість тижнів, як годинник. Якщо ви знаєте дату одного релізу Rust, то можете дізнатися дату наступного: за шість тижнів. Гарний аспект запланованих що шість тижнів релізів полягає в тому, що наступний потяг прибуде незабаром. Якщо певний функціонал пропускає якийсь певний реліз, немає потреби хвилюватися: інший відбудеться за короткий час! Це допомагає зменшити тиск, що хтось спробує протягти, можливо, недошліфований функціонал близько до строку релізу.

Завдяки цьому процесу, ви завжди можете перевірити наступну збірку Rust і впевнитись, що до неї легко оновитися: якщо бета реліз не працює так, як очікувалося, ви можете повідомити про це команді, і його відремонтують до наступного стабільного релізу! Аварії в бета релізі порівняно рідкісні, але `rustc` є лише програмою, і вади існують.

Нестабільний Функціонал

Є ще одна хитрість у цій моделі релізів: нестабільний функціонал. Rust використовує техніку, що зветься "прапорці функціонала", щоб визначити, який функціонал увімкнено в даному релізі. Якщо новий функціонал перебуває в активній розробці, він опиняється в `master`, а, відтак, у щонічних релізах, але поза *прапорцем функціонала*. Якщо ви, як користувач, захочете спробувати функціонал, над яким ведеться робота, то можете це зробити, але ви маєте використовувати нічний реліз Rust і позначити свій вихідний файл відповідним прапорцем, щоб погодитися на цей функціонал.

Якщо ви використовуєте бета або стабільний реліз Rust, то не можете використовувати прапорці функціонала. Це ключ, що дозволяє нам отримати практичний досвід нового функціонала до того, як його оголосять стабільним назавжди. Ті, хто бажає бути на передньому краю, можуть підписатися на це, а ті, хто бажає надійного досвіду, може залишитися на стабільному релізі і знати, що їхній код не зламається. Стабільність без застою.

Ця книга містить інформацію лише про стабільний функціонал, оскільки в процесі функціонал все ще змінюється, і, безумовно, він буде різним у той час, коли була написана ця книжка, і коли він буде включеним до стабільних збірок. Ви можете знайти документацію про функціонал, доступний лише в щонічних релізах, онлайн.

Rustup і Роль Щонічного Rust

Rustup дозволяє легко перемикатися між різними каналами релізів Rust, глобально чи для окремих проєктів. За замовчуванням буде встановлено стабільний Rust. Для встановлення, наприклад, щонічного, запустіть:

```
$ rustup toolchain install nightly
```

Ви також можете побачити всі *ланцюжки інструментів* (toolchain, релізи Rust і пов'язаних компонентів), що ви встановили за допомогою `rustup`. Ось приклад на комп'ютері одного з авторів з Windows:

```
> rustup toolchain list
stable-x86_64-pc-windows-msvc (default)
beta-x86_64-pc-windows-msvc
nightly-x86_64-pc-windows-msvc
```

Як ви можете бачити, стабільний ланцюжок інструментів є замовчуванням. Більшість користувачів Rust використовують переважно стабільний реліз. Ви можете використовувати стабільний реліз більшу частину часу, але використовувати щонічний у конкретному проєкті, якщо вам потрібен функціонал з переднього краю. Для цього, ви можете запустити `rustup override` в теці цього проєкту, щоб встановити щонічний ланцюжок інструментів для використання `rustup`, коли ви в цій теці:

```
$ cd ~/projects/needs-nightly
$ rustup override set nightly
```

Відтепер кожного разу як ви викликаєте `rustc` чи `cargo` всередині `~/projects/needs-nightly`, `rustup` переконається, що ви використовуєте щонічний Rust, а не стабільний Rust за замовчуванням. Це стає в пригоді, коли ви маєте багато проєктів Rust!

Команди і Процес RFC

То як же вам дізнатися про цей новий функціонал? Модель розробки Rust слідує процесу "прохання прокоментувати" (RFC, Request For Comments). Якщо ви хочете покращення в Rust, то можете написати пропозицію, що зветься RFC.

Будь-хто може написати RFC для покращення Rust, і пропозиції розглядаються і обговорюються командою Rust, що складається з багатьох тематичних підкоманд. Повний список команд знаходиться [на вебсайті Rust](#) і включає команди для кожної області проєкт: дизайн мови, реалізація компілятора, Інфраструктура, документація та

інші. Відповідна команда читає пропозицію і коментарі, пише деякі власні коментарі, і врешті-решт виникає консенсус - прийняти або відхилити цей функціонал.

Якщо функціонал буде прийнятий, то в репозиторії Rust відкривається задача, і хтось може її виконати. Особа, що реалізує функціонал цілком може бути не тою особою, що його взагалі запропонувала! Коли реалізація готова, вона додається в гілку `master` за бар'єром функціонала, про який ми говорили в підрозділі "[Нестабільний функціонал](#)".

За деякий час, коли розробники Rust, які використовують щонічні релізи, зможуть спробувати новий функціонал, члени команди обговорять, як функціонал працює на щонічному релізі, і вирішать, чи варто перенести його в стабільний Rust чи ні. Якщо буде ухвалено рішення просуватися, то бар'єр функціонала знімають, і функціонал тепер вважається стабільним! І він їде залізницею у новий стабільний реліз Rust.