Comprehensive Rust 🦀

Martin Geisler

차 례

Comprehensive Rust에 오신 것을 환영합니다🦀

이 강의는 무료이며, Google 의 Android 팀이 만들었습니다. 기본 문법부터제네릭, 에러 핸들링과 같은 고급주제까지 러스트의 모든 것을 포함합니다.

> 이 과정의 최신 버전은 https://google.github.io/comprehensive-rust/ 에서 확인할 수있습니다. 다른 곳에서 읽고 있는 경우 이곳에서 업데이트를 확인하시기바랍니다.
>
> The course is also available as a PDF.

강의는 당신이 러스트에 대해서 아무것도 모른다고 가정하고 아래의 목표를가지고 있습니다:

- 러스트 구문과 언어에 대한 포괄적인 이해를 제공합니다.
- 기존 프로그램을 수정하고 러스트에서 새 프로그램을 작성할 수 있습니다.
- 일반적인 러스트 관용구를 보여줍니다.

We call the first four course days Rust Fundamentals.

그 후에는, 아래와 같은 개별 주제를 심화해서 공부할 수 있습니다:

- Android: Android 플랫폼 개발 (AOSP) 시 Rust 사용에 관한 반나절 과정입니다. 여기에는 C, C++, Java 와의 상호 운용성이 포함됩니다.
- Chromium 의 Rust 심층 분석은 Chromium 브라우저의 일부로 Rust 를 사용하는 방법에관한 반나절 과정입니다. 여기에는 Chromium 의'gn' 빌드 시스템에서 Rust 를 사용하여서드 파티 라이브러리 (”crates”) 와 C++ 상호 운용성을 가져오는 방법이포함되어 있습니다.
- Bare-metal: bare-metal(임베디드) 개발 시 Rust 사용에 관한 종일 과정입니다. 마이크로컨트롤러와 애플리케이션 프로세서를 모두 다룹니다.
- 동시성: Rust 의동시성에 관한 종일 과정입니다. 여기서는 고전적인 동시성 (스레드와뮤텍스를 사용하여 선점형 스케줄링을 하는 것) 과 async/await 동시성 (future 를 사용하는 협력적인 멀티태스킹) 을 모두 다룹니다.

제외사항

러스트는 며칠만에 모든 것을 다루기에는 너무 큰 언어입니다. 그래서 아래와같은것을 목표로 하지 않습니다:

- 매크로 만들기: 매크로와 관련한 자세한 내용은러스트프로그래밍 언어, 19.1 절과 Rustonomicon 를 참조하세요.

독자 수준에 대한가정

본 강의는 여러분이 프로그래밍 자체에 대해서는 알고 있다고 가정합니다. 러스트는 정적타입 언어이며, 강좌에서는 C/C++ 와의 비교, 대조를 통해러스트를 설명할 것입니다.

C/C++을 모르더라도 동적 타입 언어 (Python 이나 JavaScript 등) 프로그래밍경험이 있다면 따라오는데 큰 문제는 없을 것입니다.

이것은 ”발표자 노트”의 예제입니다. 이 부분을 이용해서 추가 정보를제공합니다. 주로 강의실에서 제기되는 일반적인 질문에 대한 답변과 강사가다루어야 할 키 포인트일 수 있습니다.

제 1 장

강의진행

강사를 위한 안내 페이지입니다.

다음은 구글 내부에서 이 과정을 어떤식으로 운영해왔는지에 대한 배경정보입니다.

수업은 보통 오전 9 시부터 오후 4 시까지 진행되며, 중간에 1 시간의점심시간을 제공합니다. 따라서 오전 수업이 3 시간, 오후 수업이 3 시간입니다. 두 세션에는 여러 번의 휴식 시간 및 학생들이 연습문제를 풀수 있는 시간이 포함되어 있습니다.

강의를 실행하기 위한 준비:

1. 강의 자료를 숙지합니다. 주요 요점을 강조하기 위해 강의 참조 노트를포함하였습니다. (추가적인 노트를 작성하여 제공해 주시면감사하겠습니다.) 강의 참조 노트의 링크를 누르면 별도의 팝업으로 분리가되며, 메인 화면에서는 사라집니다. 깔끔한 화면으로 강의를 진행할 수있습니다.
2. Decide on the dates. Since the course takes four days, we recommend that you schedule the days over two weeks. Course participants have said that they find it helpful to have a gap in the course since it helps them process all the information we give them.
3. 충분한 공간을 확보합니다. 15 에서 20 명 규모의 공간을 추천합니다. 수강생과 강사가 질의를 하기에 충분한 시간과 공간이어야 합니다. 강사나수강생 모두 _책상_을 사용할 수 있는 강의실이면 좋습니다. 강의 중에강사가 라이브 코딩을 하게 될 경우가 많으며, 이때 자리에 앉아 노트북을사용하는 것이 도움이 됩니다.
4. 강의 당일 조금 일찍 와서 준비합니다. 강사 노트북에서 `mdbook serve` 를 이용해 직접 프레젠테이션 하면 페이지 이동시의 지연이없습니다.(설치방법을 참조하세요.) 또한, 그렇게 하면 강의 도중 오타를 발견했을 때 그자리에서 바로 수정 가능하다는 장점도 있습니다.
5. 수강생들이 직접 (개별 혹은 그룹으로) 연습문제를 풀도록 합니다. 대체로오전, 오후에 각각 30-45 분 정도를 연습문제에 할당합니다 (이는 해답을보고 설명하는 시간까지 포함합니다). 막혀 도움을 필요로 하는 수강생이없는지 수시로 확인합니다. 만약 같은 문제를 여러 사람이 겪고 있다면, 그 문제를 강의실 전체 인원에게 알리고 해결책을 제시합니다. 예를 들어 표준라이브러리 어디에 가면 그 문제에 대한 해답을 찾을 수 있는지 알려줍니다.

이제 준비는 끝났습니다. 우리가 그랬듯이 여러분들도 이 강의를 즐기시길바랍니다!

강의를 계속 개선할 수 있도록피드백을제공해 주십시오. 우리는 무엇이 좋았고, 무엇이 모자랐는지 듣고 싶습니다. 수강생들로 부터의피드백도환영합니다!

1.1 강의구성

강사를 위한 안내 페이지입니다.

Rust 기초

첫 4 일은 Rust 기초로 이루어지며 짧은 시간 안에 많은 내용을 다루게 됩니다.

Course schedule:

- Day 1 Morning (2 hours and 10 minutes, including breaks)

Segment	Duration
개요	5 minutes
Hello World!	15 minutes
타입 및 값	45 minutes
흐름 제어	40 minutes

- Day 1 Afternoon (2 hours and 15 minutes, including breaks)

Segment	Duration
튜플 및 배열	35 minutes
참조	35 minutes
사용자 정의 타입	50 minutes

- Day 2 Morning (2 hours and 55 minutes, including breaks)

Segment	Duration
개요	3 minutes
패턴 매칭	1 hour
메소드와 트레이트	50 minutes
제네릭	40 minutes

- Day 2 Afternoon (3 hours and 10 minutes, including breaks)

Segment	Duration
표준 라이브러리	1 hour and 20 minutes
표준 라이브러리	1 hour and 40 minutes

- Day 3 Morning (2 hours and 20 minutes, including breaks)

Segment	Duration
개요	3 minutes
메모리 관리	1 hour
스마트 포인터	55 minutes

- Day 3 Afternoon (2 hours and 10 minutes, including breaks)

Segment	Duration
빌림	50 minutes
수명	1 hour and 10 minutes

- Day 4 Morning (2 hours and 40 minutes, including breaks)

Segment	Duration
개요	3 minutes
Iterators	45 minutes
모듈	40 minutes
테스트	45 minutes

- Day 4 Afternoon (2 hours and 10 minutes, including breaks)

Segment	Duration
오류처리	55 minutes
안전하지 않은 러스트	1 hour and 5 minutes

심화학습

In addition to the 4-day class on Rust Fundamentals, we cover some more specialized topics:

Rust in Android

The Rust in Android deep dive is a half-day course on using Rust for Android platform development. This includes interoperability with C, C++, and Java.

AOSP 코드를 여러분의 컴퓨터에 체크아웃해야 합니다. 그런 다음, 그 컴퓨터에서과정 저장소를체크아웃하고 `src/android/` 디렉터리를 AOSP 코드의 루트로이동합니다. 이렇게 하면 안드로이드 빌드 시스템에서 과제용으로 추가된 `Android.bp` 파일을 인식할 수 있습니다.

`adb sync` 명령어가 에뮬레이터 혹은 실제 장치와 작동하는지확인합니다. 그리고 `src/android/build_all.sh` 를 수행해서 모든안드로이드 예제를 미리 빌드해 보세요. 그 쉘 스크립트를 읽고, 그 안에서수행되는 명령어들을 확인한 후 각 명령어들을 수동으로 실행해도 잘 되는지확인하세요.

Chromium 에서 Rust 사용

Chromium 에서 Rust 사용 이 과정은 Chromium 브라우저의 일부로 Rust 를 사용하는 방법에 관한반나절 과정입니다. 여기에는 Chromium 의 gn 빌드 시스템에서 Rust 를사용하는 방법과 서드 파티 라이브러리 (”crates”) 를 가져오는 방법, C++ 상호 운용성 등이 포함되어 있습니다.

Chromium 을 빌드할 수 있어야 합니다. 디버그, 구성요소 빌드는 속도를높이기 위해권장되지만모든 빌드가 작동합니다. 빌드한 Chromium 브라우저를 실행할 수 있는지확인합니다.

Bare-Metal Rust

The Bare-Metal Rust deep dive is a full day class on using Rust for bare-metal (embedded) development. Both microcontrollers and application processors are covered.

마이크로컨트롤러 파트를 진행하기 위해서는 BBC micro:bit v2 개발 보드를 미리 구매해야합니다. 모든 사용자는시작 페이지에설명된 대로 각종 패키지를 설치해야 합니다.

Concurrency in Rust

The Concurrency in Rust deep dive is a full day class on classical as well as `async`/`await` concurrency.

새 크레이트를 설정하고 몇 가지 의존성을 다운로드해 두어야 합니다. 그런다음 예제를 `src/main.rs`에 복사/붙여넣기 하여 테스트 해 볼 수있습니다:

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

강의형식

이 강의는 강사와 수강생이 양방향으로 소통하면서 진행하도록 디자인되었습니다. 다양한 질문을 통해 러스트의 여러 부분을 탐험할 수 있도록하세요!

1.2 단축키

다음은, mdBook 시스템 (현 사이트) 에서 유용한 단축키들 입니다:

- 왼쪽 화살표: 이전 페이지로 이동합니다.
- 오른쪽 화살표: 다음 페이지로 이동합니다.
- Ctrl + Enter: 현재 포커스를 받은 코드 샘플 블록을 실행합니다.
- s: 검색창을 활성화합니다.(mdBook 문제로 23.01.19 기준 영어로만가능합니다.)

1.3 다른언어들

이 과정은 다른 언어로도 제공됩니다. 괄호 안은 번역에 도움 주신분들입니다:

- Brazilian Portuguese by @rastringer, @hugojacob, @joaovicmendes, and @henrif75.
- Chinese (Simplified) by @suetfei, @wnghl, @anlunx, @kongy, @noahdragon, @superwhd, @SketchK, and @nodmp.
- Chinese (Traditional) by @hueich, @victorhsieh, @mingyc, @kuanhungchen, and @johnathan79717.
- Korean by @keispace, @jiyongp, @jooyunghan, and @namhyung.
- Spanish by @deavid.

페이지 오른쪽 위의 메뉴를 통해 다른 언어로 전환할 수 있습니다.

번역문제

진행 중인 번역이 많습니다. 최근에 업데이트된 번역본으로 연결되는 링크는다음과 같습니다:

- 벵갈어: @raselmandol 제공.
- 프랑스어: @KookaS 및@vcaen 제공.
- 독일어: @Throvn 및@ronaldfw 제공.
- 일본어: @CoinEZ-JPN 및@momotaro1105 제공.
- Italian by @henrythebuilder and @detro.

이 과정의 번역 작업에 도움을 주고 싶다면여기설명된 내용을 참고하세요. 진행 중인 번역 작업에 대한 내용은이슈트래커를 참고하세요.

제 2 장

카고사용하기

러스트를 시작하려고하면 당신은 곧 Cargo 라는, 러스트 생태계에서사용하는 표준 빌드/실행 도구를 만날 것 입니다. 여기서는 카고가 무엇인지, 그리고 카고가 러스트 생태계에서 어떤 역할을 하는지, 그리고 이 강의에서어떻게 사용될 지에 대해 간략히 설명하겠습니다.

설치하기

> **https://rustup.rs/ 의 설치 방법을 따르세요.**

This will give you the Cargo build tool (`cargo`) and the Rust compiler (`rustc`). You will also get `rustup`, a command line utility that you can use to install to different compiler versions.

Rust 를 설치한 후에는 Rust 와 호환되도록 편집기나 IDE 를 구성해야 합니다. 대부분의 편집기는 VS Code, Emacs, Vim/Neovim 등에 자동 완성 및 정의로 이동 기능을 제공하는 rust-analyzer 와 통신하여 이를실행합니다. RustRover 라는 다른 IDE 도 사용할 수 있습니다.

- 데비안/우분투 시스템에서는 `apt` 를 이용해서 카고, 러스트 소스, 러스트 포매터를 설치할 수있습니다. 그러나 이 방법을 따를 경우 최신 버전이 아닌 러스트를사용게되며, 그 결과 예상치 못한 문제를 겪을 수도 있습니다. 설치명령어는 아래와 같습니다:

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

2.1 러스트생태계

러스트의 생태계는 여러가지 도구들로 구성되어 있으며, 그 중 중요한 것들은아래와 같습니다:

- `rustc`: `.rs` 확장자 파일을 바이너리 혹은 다른 중간형식으로 변환해주는 Rust 컴파일러입니다.
- `cargo`: 러스트 의존성 관리자이자 빌드 시스템 입니다. 여러분의프로젝트에 명시된 의존성들을 https://crates.io 에서 자동으로다운로드 받고, 그 소스코드를 `rustc` 로 전달하여 빌드를시킵니다. 또한 유닛 테스트를 실행하는 테스트 러너를 내장하고 있습니다.
- `rustup`: the Rust toolchain installer and updater. This tool is used to install and update `rustc` and `cargo` when new versions of Rust are released. In addition, `rustup` can also download documentation for the standard library. You can have multiple versions of Rust installed at once and `rustup` will let you switch between them as needed.

키 포인트:

- 러스트는 6 주마다 새로운 릴리즈가 발표되며 이전 릴리즈와의 호환성을유지하고 있습니다.
- 릴리즈는 3 가지 버전으로 제공됩니다: ”stable”, ”beta” 그리고 ”nightly”.
- 새로운 기능은 ”nightly” -> ”beta” -(6 주후)-> ”stable” 로 변경됩니다.
- 의존성은 다양한저장소, git 프로젝트, 디렉터리 등에서 제공될 수 있습니다.
- 러스트는 에디션으로구분됩니다. 현재는 Rust 2021 에디션입니다. 이 전 에디션으로 Rust 2015 와 Rust 2018 이 있습니다.
  - 에디션은 이전 에디션과 호환이 되지 않을 수 있습니다.
  - 에디션이 바뀌면서 프로그램이 의도치 않게 깨지는 문제를 막기 위해, 각프로그램은 자신이 빌드될 에디션을 명시적으로 `Cargo.toml` 에지정해야 합니다.
  - 러스트 생태계가 에디션 별로 파편회 되는 것을 막기 위해, 러스트컴파일러는 서로 다른 에디션에서 작성된 코드들을 하나의 바이너리로묶을 수 있습니다.
  - `cargo` 를 사용하지 않고 컴파일러를 직접 사용하는 경우는 거의없음을 언급해 주시기 바랍니다.
  - It might be worth alluding that Cargo itself is an extremely powerful and comprehensive tool. It is capable of many advanced features including but not limited to:
    * 프로젝트/패키지 구조화
    * 워크스페이스
    * 개발/런타임 종속성 관리 및 캐싱
    * 빌드스크립트
    * 전역설치
    * cargo clippy 와같은 하위 플러그인으로 확장 가능.
  - 공식 Cargo Book 에서 자세한사항을 확인하시기 바랍니다.

2.2 강의에서의 코드샘플

이 강의자료에 있는 모든 예제는 브라우저에서 바로 수행 가능합니다. 이렇게한 이유는, 준비 과정을 단순화 시키고, 모두가 같은 환경에서 작업할 수있도록 하기 위함입니다.

그럼에도 불구하고, 카고 (cargo) 를 직접 설치하는 것을 강력 권장합니다. 이게 과제 작성에 더 도움이 될겁니다. 또한, 마지막 날에는 의존성이 있는예제를 작업하게 될 텐데, 그 때에는 어차피 카고가 필요합니다.

이 강의 자료의 코드 블록들은 전부 인터엑티브 합니다:

```
fn main() {
    println!("수정해 주세요!");
}
```

코드 블록에 포커스를 두고 Ctrl + Enter 를 눌러 실행해 볼 수 있습니다.

강의에서 대부분의 코드 샘플은 위와 같이 수정할수 있지만 일부 코드는다음과 같은 이유로 수정할 수 없습니다:

- 유닛 테스트는 내장 플레이그라운드에서 실행이 안됩니다. 외부플레이그라운드 사이트에 붙여넣어 테스트를 실행하시기 바랍니다.
- 내장된 플레이그라운드에서는 페이지 이동시 작성된 모든 내용이사라집니다. 따라서 로컬 환경이나 외부 플레이그라운드 사이트에서연습문제를 해결하는 것이 좋습니다.

2.3 로컬 환경의카고

만약 개인용 컴퓨터에서 코드를 실행해보려면 먼저 러스트를 설치해야합니다. Rust Book 의 지침에 따라 `rustc` 와 `cargo` 를 함께 설치 하시기바랍니다. 설치 후 아래 커맨드를 통해 각 툴의 버전을 확인 할 수 있습니다:

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

이 버전보다 더 최신의 버전이어도 상관 없습니다. 러스트는 하위 호환성을지원합니다.

정상적으로 설치가 되었으면, 강의 예제중 하나를 러스트 바이너리로 빌드해봅시다:

1. 예시 블록에 있는 "Copy to clipboard" 버튼을 클릭해서 복사합니다.
2. 터미널에서 `cargo new exercise` 를 입력해서 새로운 `exercise/` 폴더를 만듭니다:

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

3. `exercise/` 폴더로 이동한 후, `cargo run` 커맨드로코드를 실행합니다:

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

4. `src/main.rs` 에 코드를 작성합니다. 예를 들어 이전 페이지의 소스를 아래와 같이 `src/main.rs` 에 작성합니다

   ```
   fn main() {
       println!("수정해 주세요!");
   }
   ```

5. `cargo run` 커맨드로 소스를 빌드하고 실행합니다:

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

6. `cargo check` 커맨드는 빠르게 에러를 확인할 수 있습니다. `cargo build` 는 실행없이 컴파일만 합니다. 이 경우에 `target/debug/` 폴더에서 output 을 확인 할 수 있습니다. `cargo build --release` 커맨드는 릴리즈 버전용 최적화를 켜서컴파일하며 `target/release/` 폴더에서 확인 할 수 있습니다.
7. `Cargo.toml` 파일에는 의존성 패키지를 추가할 수 있습니다. `cargo` 커맨드를 실행하면 자동으로 의존성 패키지를 다운로드하고컴파일 까지 해 줍니다.

수강생들이 카고를 설치하고 로컬 편집기를 이용하도록 독려하세요. 조금귀찮을 수도 있지만, 이렇게 해야만 좀 더 실제와 가까운 개발환경을 갖추게되는 것입니다.

제 I 편

1일차 오전

제 3 장

1 일차개요

강의 첫 날입니다. 오늘 배울 것이 참 많습니다:

- 러스트 기본 문법: 변수, 스칼라 / 복합 타입, 열거형, 구조체, 참조형, 함수와 메서드.
- Types and type inference.
- 제어 흐름은 루프, 조건문 등으로 구성됩니다.
- 사용자 정의 타입: 구조체 및 enum
- 패턴 매칭: 열거형, 구조체 그리고 배열 분해.

일정예약

Including 10 minute breaks, this session should take about 2 hours and 10 minutes. It contains:

Segment	Duration
개요	5 minutes
Hello World!	15 minutes
타입 및 값	45 minutes
흐름 제어	40 minutes

학생들에게 다음을 상기시켜 주시기 바랍니다:

- 궁금한 점이 있으면 주저하지 말고 질문 해야 합니다.
- 이 수업은 상호작용이 요합니다. 토론을 망설이지 마세요!
  - As an instructor, you should try to keep the discussions relevant, i.e., keep the discussions related to how Rust does things vs some other language. It can be hard to find the right balance, but err on the side of allowing discussions since they engage people much more than one-way communication.
- 질문이 슬라이드보다 앞서가도 괜찮습니다.
  - 학습에 있어서 반복은 매우 중요합니다. 슬라이드는 그저 도움을 줄 뿐, 원하는 대로 건너띄어도 됩니다.

첫날에는 다른 언어에서도 직접적인 관련이 있는 Rust 의' 기본사항' 을 보여드리고자 합니다. Rust 의 고급 부분은 그다음에 제공됩니다.

교실에서 가르치는 경우 여기에서 일정을 검토하는 것이 좋습니다. 각세그먼트가 끝나면 연습문제가 있고 그 뒤에 휴식이 이어집니다. 휴식 후연습문제 해답을 다룰 계획입니다. 여기에 표시된 시간은 과정을 일정에 맞게진행하기 위한 권장 시간입니다. 필요에 따라 유연하게 조정하시기 바랍니다.

제 4 장

Hello World!

This segment should take about 15 minutes. It contains:

Slide	Duration
러스트란?	10 minutes
Rust 의 이점	3 minutes
플레이그라운드	2 minutes

4.1 러스트란?

러스트는 2015 년에버전 1.0 을릴리즈 한 새로운 프로그램 언어입니다:

- 러스트는 C++와 유사한 정적 컴파일 언어입니다
  - `rustc` 는 LLVM 을 백엔드로 사용합니다.
- 러스트는 다양한 플랫폼과 아키텍쳐를 지원합니다:
  - x86, ARM, WebAssembly, ...
  - Linux, Mac, Windows, ...
- 러스트는 다양한 장치에서 사용될 수 있습니다:
  - 펌웨어와 부트로더 (임베디드)
  - 스마트 디스플레이,
  - 스마트폰,
  - 데스크탑,
  - 서버.

러스트는 C++가 사용되는 대부분의 곳에서 사용 가능합니다:

- 높은 유연성.
- 높은 수준의 제어.
- 마이크로컨트롤러 같은 매우 제한된 장치로 스케일 다운 가능.
- 별도의 런타임을 필요로 하지 않으며, 가비지 컬렉션도 없음.
- 성능을 타협하지 않으면서도 안정성과 안전에 중점을 둠.

4.2 Rust 의이점

러스트만의 독특한 세일즈 포인트 (장점):

- - 메모리 버그의 전체 클래스가 컴파일시간에 방지됩니다.
  - 초기화되지 않는 변수가 없습니다.
  - 메모리 이중 해제가 원천적으로 불가능 합니다.
  - 메모리 해제 후 사용이 원천적으로 불가능 합니다.
  - NULL 포인터는 없습니다.
  - 뮤텍스를 잠궈 놓고 여는 것을 잊는 실수를 할 수 없습니다.
  - 스레드간 데이터 레이스를 막아줍니다.
  - 반복자가 갑자기 무효화 되는 경우가 없습니다.
- - Rust 문이 실행하는 작업은지정되지 않은 상태로 두지 않습니다.
  - 배열 접근시 경계 체크.
  - 정수형 타입의 변수에서 오버플로우 발생시 동작이 잘 정의되어있습니다.
- - 상위 수준 언어만큼 표현력이 뛰어나고인체공학적입니다.
  - 열거형과 패턴 매칭.
  - 제네릭.
  - FFI 런타임 오버헤드 없음.
  - 비용이 들지 않는 추상화.
  - 친절한 컴파일러 오류메시지.
  - 내장 종속성 관리자.
  - 내장 테스트 지원.
  - LSP (Language Server Protocol, 언어 서버 프로토콜) 지원이 잘되어있음.

여기에서 많은 시간을 보내지 마세요. 이 모든 사항은 나중에 자세히다룹니다.

수강생들에게 어떤 프로그래밍 언어를 사용했는지 물어보시기 바랍니다. 어떤언어를 사용했느냐에 따라 러스트에서 어떤 점을 강조해야 할지를 고민해보세요:

- C/C++: 러스트는 ' 빌림' 검사기를 통해서수행중에 발생할 수 있는 모든 에러를 제거합니다. 러스트는 C 와 C++과비슷한 수준의 성능을 보여주면서도, 그 언어들에서 종종 발생하는 메모리관련 오류가 없습니다. 또한, 패턴 매칭이나, 기본적으로 제공되는 종속성관리와 같은 현대적인 언어의 기능들을 제공합니다.
- Java, Go, Python, JavaScript: 이 언어들과 동일한 메모리 안정성과 함께, ' 하이레벨' 언어의 느낌을 느낄 수있습니다. 거기에 더해, 가비지 컬렉터가 없는 C/C++와 유사한 수준의빠르고 예측 가능한 성능을 기대할 수 있습니다. 그리고 필요한 경우저수준 하드웨어를 다루는 코드로 작성할 수 있습니다.

4.3 플레이그라운드

The Rust Playground provides an easy way to run short Rust programs, and is the basis for the examples and exercises in this course. Try running the ”hello-world” program it starts with. It comes with a few handy features:

- ' 도구' 에서 `rustfmt` 옵션을사용하여 ' 표준' 방식으로 코드 형식을지정합니다.
- Rust 에는 코드를 생성하기 위한 두 가지 기본' 프로필' 이 있습니다. 디버그 (추가런타임 검사, 최적화 감소) 및 출시 (더 적은 런타임 검사, 최적화증가) 입니다. 상단의 ' 디버그' 에서액세스할 수 있습니다.

- 관심이 있으시면 '...' 아래의'ASM' 을 사용해 생성된 어셈블리 코드를확인하세요.

As students head into the break, encourage them to open up the playground and experiment a little. Encourage them to keep the tab open and try things out during the rest of the course. This is particularly helpful for advanced students who want to know more about Rust's optimizations or generated assembly.

제 5 장

타입 및값

This segment should take about 45 minutes. It contains:

Slide	Duration
Hello World!	5 minutes
변수	5 minutes
값	5 minutes
연산	3 minutes
문자열	5 minutes
타입 추론	3 minutes
연습문제: 피보나치	15 minutes

5.1 Hello World!

가장 간단한 러스트 프로그램으로써, 고전적인 Hello World 를 작성해보겠습니다:

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

확인할 수 있는 것들:

- 함수는 `fn` 으로 선언합니다.
- C/C++ 와 마찬가지로 중괄호`{}`로 블록을 표시합니다.
- `main` 함수는 프로그램 진입점입니다.
- 러스트는 똑똑한 매크로 (hygienic macros) 시스템을 가지고 있습니다. `println!`는 그 예시입니다.
- 러스트의 문자열은 UTF-8 로 인코딩되며 이모지와 같은 유니코드 문자를포함할 수 있습니다.

This slide tries to make the students comfortable with Rust code. They will see a ton of it over the next four days so we start small with something familiar.

키 포인트:

- 러스트는 C/C++/Java 와 같은 전통적인 다른 언어와 매우 유사합니다. 러스트는 절차적 언어입니다. 정말로 필요한 경우가 아니라면, 러스트는이미 존재하는 것을 새로 구현하려고 하지 않습니다.

- 러스트는 유니코드 지원과 같은 현대 언어의 특징을 전부 지원합니다.
- Rust uses macros for situations where you want to have a variable number of arguments (no function overloading).
- 똑똑한 매크로 (hygienic macro) 는 매크로가 사용되는 스코프에서 의도치않게 변수를 가로채지 않습니다. 사실 러스트 매크로는 완전히 hygenic 하지는 않습니다. 링크를참고하세요.
- 러스트는 멀티 패러다임 언어입니다. 예를 들어 강력한객체 지향프로그래밍 기능을 지원하기도 하며, 함수형 언어로 분류되지는 않지만폭넓은 범위의함수형컨셉을 지원합니다.

5.2 변수

Rust provides type safety via static typing. Variable bindings are made with `let`:

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

- `x = 20` 의 주석 처리를 삭제하여 변수가 기본적으로 불변임을보여줍니다. `mut` 키워드를 추가하여 변경을 허용합니다.
- 여기서 'i32' 는 변수의 타입입니다. 이는컴파일 시간에 알려져야 하지만, 타입 추론 (나중에 설명)을 사용하면프로그래머가 이를 생략할 수 있는 경우가 많습니다.

5.3 값

다음은 몇 가지 기본 내장 타입과 각 타입의 리터럴 값 문법입니다.

	타입	리터럴 값
부호있는 정수	`i8`, `i16`, `i32`, `i64`, `i128`, `isize`	`-10`, `0`, `1_000`, `123_i64`
부호없는 정수	`u8`, `u16`, `u32`, `u64`, `u128`, `usize`	`0`, `123`, `10_u16`
부동소수	`f32`, `f64`	`3.14`, `-10.0e20`, `2_f32`
유니코드 문자	`char`	`'a'`, `'α'`, `'∞'`
불리언	`bool`	`true`, `false`

각 타입의 크기는 다음과 같습니다:

- `iN`, `uN`, `fN` 은 모두 _N_비트 입니다.
- `isize` 와 `usize` 는 포인터와 같은 크기입니다,
- `char` 32 비트 입니다,
- `bool` 은 8 비트 입니다.

위에 표시되지 않은 몇 가지 문법이 있습니다:

- All underscores in numbers can be left out, they are for legibility only. So `1_000` can be written as `1000` (or `10_00`), and `123_i64` can be written as `123i64`.

5.4 연산

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

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

main 이외의 함수는 이번이 처음이지만 의미는 명확합니다. 세 개의정수를 사용하고 정수를 반환합니다. 함수는 나중에 더 자세히 다루겠습니다.

산술연산은 다른 언어와 매우 유사하며 우선순위가 비슷합니다.

정수 오버플로는 어떻게 되나요? C 및 C++에서 정수의오버플로는 실제로 정의되지 않으며, 다른 플랫폼이나 컴파일러에서 다른작업을 실행할 수 있습니다. Rust 에서는 정수 오버플로 시의 동작이 정의되어있습니다.

Change the i32's to i16 to see an integer overflow, which panics (checked) in a debug build and wraps in a release build. There are other options, such as overflowing, saturating, and carrying. These are accessed with method syntax, e.g., `(a * b).saturating_add(b * c).saturating_add(c * a)`.

사실 컴파일러는 상수 표현식의 오버플로를 감지하므로 이 예에서는 별도의함수가 필요합니다.

5.5 문자열

Rust has two types to represent strings, both of which will be covered in more depth later. Both *always* store UTF-8 encoded strings.

- `String` - a modifiable, owned string.
- `&str` - 읽기 전용 문자열입니다. 문자열 리터럴은 이 타입을가집니다.

```
fn main() {
    let greeting: &str = "  인사말";
    let planet: &str = "🪐";
    let mut sentence = String::new();
    sentence.push_str(greeting);
    sentence.push_str(", ");
    sentence.push_str(planet);
    println!("마지막 문장: {}", sentence);
    println!("{:?}", &sentence[0..5]);
    //println!("{:?}", &sentence[12..13]);
}
```

이 슬라이드는 문자열을 소개합니다. 여기에 있는 모든 내용은 나중에 자세히다루겠지만 후속 슬라이드와 연습문제에서 문자열을 사용하는 데는 이것으로충분합니다.

- 문자열 내에 잘못된 UTF-8 인코딩이 있는 것은 정의되지 않은 동작이며, 이는 안전한 Rust 에서는 허용되지 않습니다.
- `String` 은 생성자 (`::new()`) 및 `s.push_str(..)` 과 같은 메서드가 포함된 사용자 정의타입입니다.

- `&str` 의 &는 참조임을 나타냅니다. 참조는 나중에다루므로 지금은 `&str` 을 '읽기 전용문자열' 을 의미하는 단위로 생각하세요.
- 주석 처리된 줄은 바이트 위치별로 문자열의 색인을 생성합니다. `12..13` 은 문자 경계에서 끝나지 않으므로 프로그램이 패닉 상태가됩니다. 오류 메시지에 따라 문자 경계에서 끝나는 범위로 조정합니다.
- Raw strings allow you to create a `&str` value with escapes disabled: `r"\n" == "\\n"`. You can embed double-quotes by using an equal amount of # on either side of the quotes:

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

- Using `{:?}` is a convenient way to print array/vector/struct of values for debugging purposes, and it's commonly used in code.

5.6 타입추론

러스트는 변수가 어떻게 사용되는지를 보고 그 변수의 타입을 추론합니다:

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

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

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

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

이 슬라이드는, 러스트 컴파일러가 변수가 어떻게 선언되어 있고, 어떻게사용되는지를 제약 조건으로 삼아서 변수의 타입을 추론하는 모습을보여줍니다.

여기서 중요한 것은, 이렇게 명시적인 타입을 생략하고 선언되었다고 해서"어떤 타입"이라도 다 담을 수 있는 타입이 되는 것은 아니라는 점입니다. 명시적인 타입 선언이 있던 없던, 컴파일러가 생성한 머신코드는 동일합니다. 컴파일러는 단지 타입 선언을 생략할 수 있도록 해서 프로그래머가 더 간결한코드를 쓸 수 있도록 도와줄 뿐입니다.

아무것도 정수 리터럴의 타입을 제한하지 않는 경우 Rust 는 기본적으로 `i32` 를 사용합니다. 그러면 오류 메시지에 `{integer}`로표시될 수 있습니다. 마찬가지로 부동 소수점 리터럴의 기본값은 `f64` 입니다.

```
fn main() {
    let x = 3.14;
    let y = 20;
    assert_eq!(x, y);
```

```
    // ERROR: `{float} == {integer}` 구현이 없음
}
```

5.7 연습문제: 피보나치

The first and second Fibonacci numbers are both 1. For n>2, the n'th Fibonacci number is calculated recursively as the sum of the n-1'th and n-2'th Fibonacci numbers.

Write a function `fib(n)` that calculates the n'th Fibonacci number. When will this function panic?

```
fn fib(n: u32) -> u32 {
    if n <= 2 {
        // 기본 사례입니다.
        todo!("Implement this")
    } else {
        // 재귀 사례입니다.
        todo!("Implement this")
    }
}

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

5.7.1 해답

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

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

제 6 장

흐름제어

This segment should take about 40 minutes. It contains:

Slide	Duration
if 표현식	4 minutes
배열과 for 반복문	5 minutes
break 와 continue	4 minutes
블록 및 범위	5 minutes
함수	3 minutes
매크로	2 minutes
연습문제: 콜라츠 수열	15 minutes

6.1 `if` 표현식

다른 언어의 `if` 문과 똑같이 `if` 표현식을 사용합니다:

```
fn main() {
    let x = 10;
    if x == 0 {
        println!("zero!");
    } else if x < 100 {
        println!("큰");
    } else {
        println!("거대한");
    }
}
```

게다가 `if` 는 표현식으로 사용할 수도 있습니다. 아래 코드는 위와동일합니다:

```
fn main() {
    let x = 10;
    let size = if x < 20 { "작은" } else { "대형" };
    println!("숫자 크기: {}", size);
}
```

Because `if` is an expression and must have a particular type, both of its branch blocks must have the same type. Show what happens if you add `;` after `"small"` in the second example.

표현식에 'if' 가 사용된 경우 다음 문과구분하기 위해 표현식에 `;` 이 있어야 합니다. 컴파일러 오류를보려면 `println!` 앞의 `;` 을 삭제하세요.

6.2 배열과 for 반복문

Rust 에는 `while`, `loop`, `for` 라는 세 가지 반복키워드가 있습니다.

while

The `while` keyword works much like in other languages, executing the loop body as long as the condition is true.

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

6.2.1 for

The `for` loop iterates over ranges of values or the items in a collection:

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

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

- Under the hood `for` loops use a concept called ”iterators” to handle iterating over different kinds of ranges/collections. Iterators will be discussed in more detail later.
- `for` 반복문은 4 까지만 실행됩니다. 마지막 값을포함시키는 방법으로 `1..=5` 와 같은 문법을 보여주세요.

6.2.2 loop

The `loop` statement just loops forever, until a `break`.

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

```
            break;
        }
    }
}
```

6.3 break와 continue

다음 반복을 즉시 시작하려면 continue 를사용합니다.

If you want to exit any kind of loop early, use break. For loop, this can take an optional expression that becomes the value of the loop expression.

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

6.3.1 Labels

사용합니다. 중첩 루프에서는 레이블과 함께 사용할 수 있습니다:

```
fn main() {
    let s = [[5, 6, 7], [8, 9, 10], [21, 15, 32]];
    let mut elements_searched = 0;
    let target_value = 10;
    'outer: for i in 0..=2 {
        for j in 0..=2 {
            elements_searched += 1;
            if s[i][j] == target_value {
                break 'outer;
            }
        }
    }
    print!("elements searched: {elements_searched}");
}
```

- loop 는 non-trivial 값을 반환하는 유일한 반복문입니다. 이는 while 및 for 반복문과 달리 최소한 한 번은 루프문을수행하는 것이 보장되기 때문입니다.

6.4 블록 및범위

블록

A block in Rust contains a sequence of expressions, enclosed by braces `{}`. Each block has a value and a type, which are those of the last expression of the block:

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

위의 `main` 함수는 마지막 표현식이 `;` 로 끝나기 때문에반환되는 값과 타입이 `()` 입니다.

- 블록 마지막 줄을 수정하면서 블록의 값이 어떻게 바뀌는지 보여주세요. 예를 들어, 세미콜론을 넣거나 뺀다든지, 아니면 `return` 을 사용해보세요.

6.4.1 범위 (Scopes) 와쉐도잉 (Shadowing)

변수의 범위는 자신을 포함하는 블록으로 제한됩니다.

현재 범위에 있는 변수와, 바깥 범위에 있는 변수 모두 가릴 (쉐도잉) 수있습니다:

```
fn main() {
    let a = 10;
    println!("이전: {a}");
    {
        let a = "hello";
        println!("내부 범위: {a}");

        let a = true;
        println!("내부 범위 섀도 처리됨: {a}");
    }

    println!("이후: {a}");
}
```

- Show that a variable's scope is limited by adding a `b` in the inner block in the last example, and then trying to access it outside that block.
- Shadowing is different from mutation, because after shadowing both variable's memory locations exist at the same time. Both are available under the same name, depending where you use it in the code.
- A shadowing variable can have a different type.
- 처음에 쉐도잉을 보면 코드를 더 모호하게 만든다고 생각할 수 도있습니다. 그러나 실제로 쉐도잉을 이용하면, 어떤 변수에서 `.unwrap()` 된 값을 새로운 변수에 담을 경우 새로운 이름을 지을필요 없이 기존 이름을 유지할 수 있어서 편리합니다.

6.5 함수

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

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

- 매개변수를 선언할 때에는 이름을 먼저 쓰고, 타입을 나중에 씁니다. 이름과 타입은 : 로 구분합니다. 이는 일부 언어 (예를 들어 C) 와반대임에 유의하시기 바랍니다. 마찬가지로, 리턴 타입도 함수의 시작이아닌 가장 뒷부분에 선언합니다.
- The last expression in a function body (or any block) becomes the return value. Simply omit the `;` at the end of the expression. The `return` keyword can be used for early return, but the "bare value" form is idiomatic at the end of a function (refactor `gcd` to use a `return`).
- 반환값이 없는 함수의 경우, 유닛 타입 `()` 을 반환합니다. `-> ()` 가 생략된 경우 컴파일러는 이를 추론합니다.
- Overloading is not supported – each function has a single implementation.
  - 항상 고정된 개수의 매개변수를 사용합니다. 기본 인수는 지원되지않습니다. 매크로는 가변 함수를 지원하는 데 사용할 수 있습니다.
  - Always takes a single set of parameter types. These types can be generic, which will be covered later.

6.6 매크로

매크로는 컴파일 중에 Rust 코드로 확장되며 다양한 수의 인수를 사용할 수있습니다. 끝에 `!`로 구분됩니다. Rust 표준 라이브러리에는 여러가지 유용한 매크로가 포함되어 있습니다.

- `println!(format, ..)` prints a line to standard output, applying formatting described in `std::fmt`.
- `format!(format, ..)` 은 `println!`처럼 작동하지만결과를 문자열로 반환합니다.
- `dbg!(expression)` 은 표현식의 값을 기록하고 반환합니다.
- `todo!()` 는 일부 코드를 아직 구현되지 않은 것으로 표시합니다. 실행하면 패닉이 발생합니다.
- `unreachable!()` 은 일부 코드를 연결할 수 없다고 표시합니다. 실행하면 패닉이 발생합니다.

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

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

```
}

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

이 섹션에서는 이러한 일반적인 편의 기능이 있으며 이를 사용하는 방법을기억해야 합니다. 매크로로 정의되는 이유와 확장 대상은 특별히 중요하지않습니다.

이 과정에서는 매크로 정의를 다루지 않지만 이후 섹션에서는 파생 매크로의사용에 관해 설명합니다.

6.7 연습문제: 콜라츠수열

콜라츠 수열은임의의 n 에 대해 다음과 같이 정의됩니다.10 보다 큰 경우:- _ni_이 1 이면수열은 _n 에서 종료됩니다.i_.

- _ni_이 (가) 짝수면 _ni+1= ni/ 2_입니다.
- _ni_이 (가) 홀수이면 _ni+1= 3 * ni
  - 1_입니다.

예를 들어 _n1_ = 3: 으로 시작하면- 3 is odd, so _n2_ = 3 * 3 + 1 = 10 이며

- 10 is even, so _n3_ = 10 / 2 = 5 이며
- 5 is odd, so *n4* = 3 * 5 + 1 = 16;
- 16 is even, so *n5* = 16 / 2 = 8;
- 8 is even, so *n6* = 8 / 2 = 4;
- 4 is even, so *n7* = 4 / 2 = 2;
- 2 is even, so _n8_ = 1 이 되어
- 수열이 종료됩니다.

주어진 첫 번째 n 에 대해 콜라츠 수열의 길이를 계산하는 함수를작성합니다.

```
/// `n`에서 시작하는 콜라츠 수열의 길이를 결정합니다.
fn collatz_length(mut n: i32) -> u32 {
  todo!("Implement this")
}

fn main() {
  todo!("Implement this")
}
```

6.7.1 해답

```
/// `n`에서 시작하는 콜라츠 수열의 길이를 결정합니다.
fn collatz_length(mut n: i32) -> u32 {
    let mut len = 1;
    while n > 1 {
        n = if n % 2 == 0 { n / 2 } else { 3 * n + 1 };
        len += 1;
    }
    len
}
```

```rust
#[test]
fn test_collatz_length() {
    assert_eq!(collatz_length(11), 15);
}

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

제 II 편

1일차 오후

제 7 장

Welcome Back

Including 10 minute breaks, this session should take about 2 hours and 15 minutes. It contains:

Segment	Duration
튜플 및 배열	35 minutes
참조	35 minutes
사용자 정의 타입	50 minutes

제 8 장

튜플 및배열

This segment should take about 35 minutes. It contains:

Slide	Duration
배열	5 minutes
튜플	5 minutes
Cargo 에 통합됨	3 minutes
열거형 분해 (역구조화)	5 minutes
연습문제: 중첩 배열	15 minutes

8.1 배열

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

- A value of the array type `[T; N]` holds `N` (a compile-time constant) elements of the same type `T`. Note that the length of the array is *part of its type*, which means that `[u8; 3]` and `[u8; 4]` are considered two different types. Slices, which have a size determined at runtime, are covered later.
- 범위를 벗어난 배열 요소에 액세스해 보세요. 배열 액세스는 런타임에확인됩니다. Rust 는 일반적으로 이러한 확인을 최적화할 수 있으며, 안전하지 않은 Rust 를 사용하면 범위 확인을 하지 않을 수도 있습니다.
- 리터럴을 사용하여 배열에 값을 할당할 수 있습니다.
- The `println!` macro asks for the debug implementation with the ? format parameter: `{}` gives the default output, `{:?}` gives the debug output. Types such as integers and strings implement the default output, but arrays only implement the debug output. This means that we must use debug output here.
- #을 추가하면 (`{a:#?}`) 좀 더 읽기 쉬운 ”이쁜” 형태로 출력이 됩니다.

8.2 튜플

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

- 배열과 마찬가지로 튜플은 고정 길이를 갖습니다.
- 튜플은 서로 다른 타입의 값들을 하나의 복합 타입으로 묶습니다.
- 튜플에 속한 값은 `t.0`, `t.1` 과 같이 인덱스로 접근할 수있습니다.
- The empty tuple `()` is referred to as the "unit type" and signifies absence of a return value, akin to `void` in other languages.

8.3 Cargo 에통합됨

`for` 문은 배열 반복을 지원하지만 튜플은 지원하지 않습니다.

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

이 기능은 `IntoIterator` 트레잇을 사용하지만 여기서는 아직 다루지않았습니다.

`assert_ne!` 매크로가 여기에 새로 추가되었습니다. `assert_eq!` 및 `assert!` 매크로도 있습니다. 이매크로들은 항상 값을 확인하지만, `debug_assert!`와 같은 디버그전용 매크로는 출시 빌드에서 컴파일되지 않고 사라집니다.

8.4 열거형분해 (역구조화)

When working with tuples and other structured values it's common to want to extract the inner values into local variables. This can be done manually by directly accessing the inner values:

```
fn print_tuple(tuple: (i32, i32)) {
    let left = tuple.0;
    let right = tuple.1;
    println!("left: {left}, right: {right}");
}
```

However, Rust also supports using pattern matching to destructure a larger value into its constituent parts:

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

```
    println!("left: {left}, right: {right}");
}
```

This works with any kind of structured value:

```
struct Foo {
    a: i32,
    b: bool,
}

fn print_foo(foo: Foo) {
    let Foo { a, b } = foo;
    println!("a: {a}, b: {b}");
}
```

- The patterns used here are ”irrefutable”, meaning that the compiler can statically verify that the value on the right of = has the same structure as the pattern.
- A variable name is an irrefutable pattern that always matches any value, hence why we can also use `let` to declare a single variable.
- Rust also supports using patterns in conditionals, allowing for equality comparison and destructuring to happen at the same time. This form of pattern matching will be discussed in more detail later.
- Edit the examples above to show the compiler error when the pattern doesn't match the value being matched on.

8.5 연습문제: 중첩배열

배열에는 다음과 같은 다른 배열이 포함될 수 있습니다.

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

What is the type of this variable?

Use an array such as the above to write a function `transpose` which will transpose a matrix (turn rows into columns):

```
           ⌧  1 2 3⌧          1 4 7
"transpose"⌧  4 5 6⌧    "=="  2 5 8
           ⌧  7 8 9⌧          3 6 9
```

두 함수 모두 행렬의 크기는 3 x 3 으로 하드코딩 합니다.

아래 코드를 https://play.rust-lang.org/ 에 복사해서 구현하시면됩니다:

```
// TODO: 구현이 완료되면 이를 삭제합니다.
#![allow(unused_variables, dead_code)]

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

#[test]
fn test_transpose() {
    let matrix = [
        [101, 102, 103], //
        [201, 202, 203],
```

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

fn main() {
    let matrix = [
        [101, 102, 103], // <-- 주석으로 rustfmt가 줄바꿈을 추가합니다.
        [201, 202, 203],
        [301, 302, 303],
    ];

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

8.5.1 해답

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

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

```
        ]
    );
}

fn main() {
    let matrix = [
        [101, 102, 103], // <-- 주석으로 rustfmt가 줄바꿈을 추가합니다.
        [201, 202, 203],
        [301, 302, 303],
    ];

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

제 9 장

참조

This segment should take about 35 minutes. It contains:

Slide	Duration
공유 참조	10 minutes
허상 (dangling) 참조	10 minutes
연습문제: 도형	15 minutes

9.1 공유참조

참조는 값에 대한 책임을 지지 않고 다른 값에 액세스하는 방법을 제공하며' 빌림' 이라고도 합니다. 공유 참조는 읽기전용이며 참조된 데이터는 변경할 수 없습니다.

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

T 타입에 대한 공유 참조는 &T 타입을 갖습니다. 참조값은 & 연산자로 작성됩니다. * 연산자는 참조를' 역참조' 하여 값을 산출합니다.

러스트는 허상 (dangling) 참조를 컴파일러 단계에서 찾아내고 금지합니다:

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

- 참조는 참조되는 값을 ' 빌린다' 고 하며, 이는 포인터에 익숙하지 않은 학생들에게 좋은 모델입니다. 코드에서는참조를 사용하여 값에 액세스할 수 있지만 여전히 원래 변수가 값을' 소유' 합니다. 소유권에 관한 자세한내용은 3 일 차에 다룹니다.

- 참조는 포인터로 구현되며 주요 이점은 가리키는 대상보다 훨씬 작을 수있다는 것입니다. C 또는 C++에 익숙한 학생은 참조를 포인터로인식합니다. 과정의 후반부에서는 Rust 가 원시 포인터 사용에서 비롯되는메모리 안전 버그를 방지하는 방법을 다룹니다.
- Rust 는 참조를 자동으로 생성하지 않습니다. &가 항상필요합니다.
- Rust will auto-dereference in some cases, in particular when invoking methods (try `r.is_ascii()`). There is no need for an `->` operator like in C++.
- 이 예에서 `r` 은 변경 가능하므로 다시 할당할 수있습니다 (`r = &b`). 이렇게 하면 `r` 이 다시 바인딩되어다른 것을 참조합니다. 이는 참조에 할당하면 참조된 값이 변경되는 C++와는 다릅니다.
- 공유 참조에서는 참조 값이 변경 가능하더라도 이를 수정할 수 없습니다. `*r = 'X'` 를 입력해 보세요.
- Rust 는 모든 참조의 전체 기간을 추적하여 충분히 오래 지속되는지확인합니다. 안전한 Rust 에서는 댕글링 참조가 발생할 수 없습니다. `x_axis` 는 `point` 에 대한 참조를 반환하지만 `point` 는 함수가 반환되면 할당이 해제되므로 컴파일되지않습니다.
- 소유권에 대한 주제를 다룰 때 이 빌림에 대해 더 자세히 이야기하겠습니다.

9.2 허상 (dangling) 참조

변경 가능한 참조라고도 하는 배타적 참조를 사용하면 참조되는 값을 변경할수 있습니다. 타입은 `&mut T` 입니다.

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

키 포인트:

- '배타적' 이란 값에 액세스하는 데 이참조만 사용할 수 있음을 의미합니다. 동시에 다른 참조 (공유 또는배타적) 가 존재할 수 없으며, 배타적 참조가 존재하는 동안에는 참조된값에 액세스할 수 없습니다. `x_coord` 가 활성화되어 있는 동안`&point.0` 을 만들거나 `point.0` 을 변경해 보세요.
- Be sure to note the difference between `let mut x_coord: &i32` and `let x_coord: &mut i32`. The first one represents a shared reference which can be bound to different values, while the second represents an exclusive reference to a mutable value.

9.3 연습문제: 도형

3 차원 도형을 위한 몇 가지 유틸리티 함수를 만들어 보겠습니다. 3 차원 상의한 점을 `[f64;3]` 으로 나타내도록 합니다. 함수 서명은 개발자가결정합니다.

```
// 해당 좌표의 제곱을 더하고
// 제곱근을 사용하여 벡터의 크기를 계산합니다. `v.sqrt()`와 같은 `sqrt()` 메서드를 사용하여 제곱근을
// 계산합니다.


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

```
}

// 벡터의 크기를 계산하고 모든 좌표를 해당 크기로 나눠서
// 벡터를 정규화합니다.


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

// 다음 `main`을 사용하여 작업을 테스트합니다.

fn main() {
    println!("단위 벡터의 크기: {}", magnitude(&[0.0, 1.0, 0.0]));

    let mut v = [1.0, 2.0, 9.0];
    println!("{v:?} 크기: {}", magnitude(&v));
    normalize(&mut v);
    println!("정규화 후 {v:?}의 크기: {}", magnitude(&v));
}
```

9.3.1 해답

```
/// 지정된 벡터의 크기를 계산합니다.
fn magnitude(vector: &[f64; 3]) -> f64 {
    let mut mag_squared = 0.0;
    for coord in vector {
        mag_squared += coord * coord;
    }
    mag_squared.sqrt()
}

/// 방향 변경 없이 벡터 크기를 1.0으로 변경합니다.
fn normalize(vector: &mut [f64; 3]) {
    let mag = magnitude(vector);
    for item in vector {
        *item /= mag;
    }
}

fn main() {
    println!("단위 벡터의 크기: {}", magnitude(&[0.0, 1.0, 0.0]));

    let mut v = [1.0, 2.0, 9.0];
    println!("{v:?} 크기: {}", magnitude(&v));
    normalize(&mut v);
    println!("정규화 후 {v:?}의 크기: {}", magnitude(&v));
}
```

제 10 장

사용자 정의타입

This segment should take about 50 minutes. It contains:

Slide	Duration
구조체	10 minutes
튜플	10 minutes
열거형	5 minutes
정적변수 (static) 와 상수 (const)	5 minutes
타입 별칭	2 minutes
연습문제: 엘리베이터 이벤트	15 minutes

10.1 구조체

C/C++ 와 마찬가지로 러스트는 커스텀 구조체를 지원합니다:

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

fn describe(person: &Person) {
    println!("{}은 (는) {}세입니다.", person.name, person.age);
}

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

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

    let name = String::from("에이버리");
    let age = 39;
```

```
    let avery = Person { name, age };
    describe(&avery);

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

키 포인트:

- 구조체는 C/C++ 와 유사합니다.
  - C++ 와 같지만 C 와는 달리 타입을 정의하기 위해'typedef' 가 필요하지 않습니다.
  - C++ 와 달리 구조체 간 상속은 없습니다.
- This may be a good time to let people know there are different types of structs.
  - Zero-sized structs (e.g. `struct Foo;`) might be used when implementing a trait on some type but don't have any data that you want to store in the value itself.
  - 다음 슬라이드에서는 필드 이름이 덜 중요할 때 사용할 수 있는 튜플구조체를 소개합니다.
- If you already have variables with the right names, then you can create the struct using a shorthand.
- The syntax `..avery` allows us to copy the majority of the fields from the old struct without having to explicitly type it all out. It must always be the last element.

10.2 튜플

각 필드 이름이 중요하지 않다면 튜플 구조체를 사용할 수 있습니다:

```
struct Point(i32, i32);

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

튜플 구조체는 종종 단일 필드의 래퍼 (wrapper, 러스트에서뉴타입 (newtype) 이라고 부름) 로 사용됩니다:

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

fn compute_thruster_force() -> PoundsOfForce {
    todo!("NASA 로켓 과학자에게 물어보세요")
}

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

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

- 뉴타입은 기본 타입에 부가적인 의미를 더하는 좋은 방법입니다. 예를들어:
  - 숫자값에 단위를 표시할 수 있음: 위에서 `Newtons` 이 그예입니다.

  - The value passed some validation when it was created, so you no longer have to validate it again at every use: `PhoneNumber(String)` or `OddNumber(u32)`.
- `Newtons` 타입의 값에 `f64` 값을 더하는 방법을보여주세요.
  - 러스트는 분명하지 않은 것을 싫어합니다. 예를 들면 자동으로 unwrap 하거나 불리언 값을 정수 값으로 사용하는 것들이 그렇습니다.
  - 연산자 재정의는 3 일차 제네릭 부분에서 다룹니다.
- 이는 화성기후 궤도선 (Mars Climate Orbiter) 의 실패 원인으로 지목된 도량형 입력오류를 보여줍니다.

10.3 열거형

enum 키워드는 몇가지 변형 (variant) 으로 표현되는 열거형 타입을생성합니다:

```
#[derive(Debug)]
enum Direction {
    Left,
    Right,
}

#[derive(Debug)]
enum PlayerMove {
    Pass,                          // 단순 변형
    Run(Direction),                // 튜플 변형
    Teleport { x: u32, y: u32 }, // 구조체 변형
}

fn main() {
    let m: PlayerMove = PlayerMove::Run(Direction::Left);
    println!("이번 차례: {:?}", m);
}
```

키 포인트:

- Enumerations allow you to collect a set of values under one type.
- `Direction` 은 변형을 가지는 열거형 타입입니다. 여기에는 `Direction::Left` 와 `Direction::Right` 의 두 값이포함됩니다.
- `PlayerMove` is a type with three variants. In addition to the payloads, Rust will store a discriminant so that it knows at runtime which variant is in a `PlayerMove` value.
- This might be a good time to compare structs and enums:
  - In both, you can have a simple version without fields (unit struct) or one with different types of fields (variant payloads).
  - You could even implement the different variants of an enum with separate structs but then they wouldn’ t be the same type as they would if they were all defined in an enum.
- Rust 는 판별식을 저장하는 데 최소한의 공간을 사용합니다.
  - 필요한 경우 필요한 가장 작은 크기의 정수를 저장합니다.
  - 허용된 변형 값이 모든 비트 패턴을 포함하지 않는 경우 잘못된 비트패턴을 사용하여 판별식을 인코딩합니다 (' 틈새최적화'). 예를 들어 `Option<&u8>`은 정수를 가리키는포인터나 None 변형의 경우 `NULL` 을 저장합니다.
  - C 와의 연동을 위해 식별자 값을 직접 지정할 수도 있습니다:
    ```
    #[repr(u32)]
    enum Bar {
    ```

```
    A, // 0
    B = 10000,
    C, // 10001
}

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

repr 속성이 없다면 10001 이 2 바이트로 표현가능하기 때문에식별자의 타입 크기는 2 바이트가 됩니다.

더살펴보기

Rust 에는 enum 이 더 적은 공간을 차지하도록 하는 데 사용할 수 있는 여러최적화가 있습니다.

- 널포인터 최적화: 어떤타입들에 대해서 러스트는 `size_of::<T>()` 가 `size_of::<Option<T>>()` 와같은 것을 보장합니다.

  실제로 널포인터 최적화가 적용된 것을 확인하고 싶다면 아래의 예제코드를사용하세요. 주의할 점은, 여기에서 보여주는 비트 패턴이 컴파일러가보장해 주는 것은 아니라는 점입니다. 여기에 의존하는 것은 완전히 unsafe 합니다.

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

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

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

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

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

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

```
        }
    }
```

10.4 정적변수 (static) 와상수 (const)

Static and constant variables are two different ways to create globally-scoped values that cannot be moved or reallocated during the execution of the program.

상수 (const)

상수는 컴파일 할 때 그 값이 정해집니다. 그리고 그 값은 그 상수가사용되는 모든 부분에서 인라인 됩니다:

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

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

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

Rust RFC Book 에 따르면 상수는, 그 상수가 사용되는 곳에 인라인 됩니다.

`const` 값을 생성할 때에는 `const` 로 마킹된 함수만이 호출가능하며, 이 함수들은 컴파일 시에 호출이 됩니다. 물론 `const` 함수들을 런타임에 호출하는 것도 가능합니다.

정적변수 (static)

정적 변수는 프로그램이 수행되는 동안 유지가 됩니다. 그러므로 다른 변수로이동 (move) 되지 않습니다:

```
static BANNER: &str = "Welcome to RustOS 3.14";

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

As noted in the Rust RFC Book, these are not inlined upon use and have an actual associated memory location. This is useful for unsafe and embedded code, and the variable lives through the entirety of the program execution. When a globally-scoped value does not have a reason to need object identity, `const` is generally preferred.

- 러스트의 `const` 는 C++의 `constexpr` 과 매우 비슷합니다.
- 반면에 러스트의 `static` 은 C++의 `const` 나 가변 정적변수 (mutable global variable) 와 훨씬 더 유사합니다.

- `static` 은 객체에 정체성을 부여합니다. 정체정이란 메모리상에서의 주소, 그리고 내부 상태를 의미합니다.
- 프로그램 수행시 그 값이 정해지는 상수가 필요한 경우는 드뭅니다. 그러나그렇다고 해도, 정적 변수를 사용하는 것 보다는 더 유용하고 안전합니다.

속성 비교테이블:

속성	정적 (static) 변수	상수 (constant)
메모리 상에 주소가 있는가	예	아니오 (인라인 됨)
프로그램이 수행되는 동안 계속 살아있는가	예	아니오
변경 가능한가	예 (그러나 안전하지 않음)	아니오
컴파일시 그 값이 결정되는가	예 (컴파일시 초기화 됨)	예
사용되는 곳에 인라인 되는가	아니오	예

더살펴보기

Because `static` variables are accessible from any thread, they must be Sync. Interior mutability is possible through a `Mutex`, atomic or similar.

Thread-local data can be created with the macro `std::thread_local`.

10.5 타입별칭

타입 별칭은 다른 타입의 이름을 생성합니다. 두 타입은 서로 바꿔서 사용할수 있습니다.

```
enum CarryableConcreteItem {
    Left,
    Right,
}

type Item = CarryableConcreteItem;

// 별칭은 다음과 같이 길고 복잡한 타입에서 더 유용합니다.
use std::cell::RefCell;
use std::sync::{Arc, RwLock};
type PlayerInventory = RwLock<Vec<Arc<RefCell<Item>>>>;
```

C 프로그래머는 이를 `typedef` 와 유사한 것으로 인식합니다.

10.6 연습문제: 엘리베이터이벤트

엘리베이터 제어 시스템의 이벤트를 나타내는 데이터 구조를 만들어보겠습니다. 다양한 이벤트를 구성하기 위해 타입과 함수를 정의하는 것은개발자의 몫입니다. 해당 타입을 `{:?}` 문법을 통해 출력할 수있도록 `#[derive(Debug)]` 를 사용합시다.

이 연습에서는 `main` 함수가 에러없이 동작하도록 데이터 구조만만들면 됩니다. 이 과정의 다음 부분에서는 이러한 구조에서 데이터를가져오는 방법을 다룹니다.

```
#[derive(Debug)]
/// 컨트롤러가 반응해야 하는 엘리베이터 시스템의 이벤트입니다.
enum Event {
    // TODO: 필요한 변형들을 추가하세요.
}

/// 이동 방향입니다.
#[derive(Debug)]
enum Direction {
    Up,
    Down,
}

/// 엘리베이터가 지정된 층에 도착했습니다.
fn car_arrived(floor: i32) -> Event {
    todo!()
}

/// 엘리베이터 문이 열렸습니다.
fn car_door_opened() -> Event {
    todo!()
}

/// 엘리베이터 문이 닫혔습니다.
fn car_door_closed() -> Event {
    todo!()
}

/// 지정된 층의 엘리베이터 로비에서 방향 버튼을 눌렀습니다.
fn lobby_call_button_pressed(floor: i32, dir: Direction) -> Event {
    todo!()
}

/// 엘리베이터에서 층 버튼을 눌렀습니다.
fn car_floor_button_pressed(floor: i32) -> Event {
    todo!()
}

fn main() {
    println!(
        "1층 승객이 위쪽 버튼을 눌렀습니다. {:?}",
        lobby_call_button_pressed(0, Direction::Up)
    );
```

```
    println!("엘리베이터가 1층에 도착했습니다: {:?}", car_arrived(0));
    println!("엘리베이터 문이 열렸습니다. {:?}", car_door_opened());
    println!(
        "승객이 3층 버튼을 눌렀습니다. {:?}",
        car_floor_button_pressed(3)
    );
    println!("엘리베이터 문이 닫혔습니다: {:?}", car_door_closed());
    println!("엘리베이터가 3층에 도착했습니다. {:?}", car_arrived(3));
}
```

10.6.1 해답

```
#[derive(Debug)]
/// 컨트롤러가 반응해야 하는 엘리베이터 시스템의 이벤트입니다.
enum Event {
    /// 버튼을 눌렀습니다.
    ButtonPressed(Button),

    /// 엘리베이터가 지정된 층에 도착했습니다.
    CarArrived(Floor),

    /// 엘리베이터 문이 열렸습니다.
    CarDoorOpened,

    /// 엘리베이터 문이 닫혔습니다.
    CarDoorClosed,
}

/// 층은 정수로 표시됩니다.
type Floor = i32;

/// 이동 방향입니다.
#[derive(Debug)]
enum Direction {
    Up,
    Down,
}

/// 사용자가 액세스할 수 있는 버튼입니다.
#[derive(Debug)]
enum Button {
    /// 특정 층의 엘리베이터 로비에 있는 버튼입니다.
    LobbyCall(Direction, Floor),

    /// 엘리베이터 내부의 층 버튼입니다.
    CarFloor(Floor),
}

/// 엘리베이터가 지정된 층에 도착했습니다.
fn car_arrived(floor: i32) -> Event {
    Event::CarArrived(floor)
```

```
}

/// 엘리베이터 문이 열렸습니다.
fn car_door_opened() -> Event {
    Event::CarDoorOpened
}

/// 엘리베이터 문이 닫혔습니다.
fn car_door_closed() -> Event {
    Event::CarDoorClosed
}

/// 지정된 층의 엘리베이터 로비에서 방향 버튼을 눌렀습니다.
fn lobby_call_button_pressed(floor: i32, dir: Direction) -> Event {
    Event::ButtonPressed(Button::LobbyCall(dir, floor))
}

/// 엘리베이터에서 층 버튼을 눌렀습니다.
fn car_floor_button_pressed(floor: i32) -> Event {
    Event::ButtonPressed(Button::CarFloor(floor))
}

fn main() {
    println!(
        "1층 승객이 위쪽 버튼을 눌렀습니다. {:?}",
        lobby_call_button_pressed(0, Direction::Up)
    );
    println!("엘리베이터가 1층에 도착했습니다: {:?}", car_arrived(0));
    println!("엘리베이터 문이 열렸습니다. {:?}", car_door_opened());
    println!(
        "승객이 3층 버튼을 눌렀습니다. {:?}",
        car_floor_button_pressed(3)
    );
    println!("엘리베이터 문이 닫혔습니다: {:?}", car_door_closed());
    println!("엘리베이터가 3층에 도착했습니다. {:?}", car_arrived(3));
}
```

제 III 편

2일차 오전

제 11 장

2 일차개요

Now that we have seen a fair amount of Rust, today will focus on Rust's type system:

- Pattern matching: extracting data from structures.
- 메서드: 함수를 타입과 연결
- 트레잇: 여러 타입에서 공유하는 동작
- 제네릭: 다른 타입의 타입 매개변수화
- 표준 라이브러리 타입 및 트레잇: Rust 의 풍부한 표준 라이브러리 둘러보기

일정예약

Including 10 minute breaks, this session should take about 2 hours and 55 minutes. It contains:

Segment	Duration
개요	3 minutes
패턴 매칭	1 hour
메소드와 트레이트	50 minutes
제네릭	40 minutes

제 12 장

패턴매칭

This segment should take about 1 hour. It contains:

Slide	Duration
Matching Values	10 minutes
열거형 분해 (역구조화)	10 minutes
흐름 제어	10 minutes
연습문제: 표현식 평가	30 minutes

12.1 Matching Values

The `match` keyword lets you match a value against one or more *patterns*. The comparisons are done from top to bottom and the first match wins.

C/C++의 `switch` 와 비슷하게 값을 패턴으로 사용할 수도 있습니다:

```
#[rustfmt::skip]
fn main() {
    let input = 'x';
    match input {
        'q'                       => println!("Quitting"),
        'a' | 's' | 'w' | 'd'     => println!("이리저리 이동"),
        '0'..='9'                 => println!("숫자 입력"),
        key if key.is_lowercase() => println!("소문자: {key}"),
        _                         => println!("기타"),
    }
}
```

The _ pattern is a wildcard pattern which matches any value. The expressions *must* be exhaustive, meaning that it covers every possibility, so _ is often used as the final catch-all case.

Match can be used as an expression. Just like `if`, each match arm must have the same type. The type is the last expression of the block, if any. In the example above, the type is `()`.

패턴의 변수 (이 예에서는 `key`) 는 일치 부문 내에서 사용할 수 있는바인딩을 만듭니다.

일치 가드는 조건이 참인 경우에만 부분이 일치하도록 합니다.

키 포인트:

- 패턴에서 사용되는 특수 문자들을 알려주세요
  - |: or 기호입니다
  - ..: 필요한 만큼 확장합니다
  - 1..=5: 끝 값 (여기서는 5) 을 포함하는 범위를 나타냅니다
  - _: 와일드카드입니다
- 매치 가드는 별도의 문법 요소로서 패턴 자체만으로 표현하기 어려운복잡한 경우를 간결하게 표현하고자 할 때 유용합니다.
- 매치의 각 팔 (혹은 가지) 안에 따로 `if` 를 사용한 것과 다릅니다. 매치 가지의 => 뒤에 사용된 `if` 표현식은해당 가지가 선택된 다음에 실행됩니다. 따라서 여기서 `if` 조건이실패하더라도 원래 `match` 의 다른 가지는 고려되지 않습니다.
- 가드에 정의된 조건은 | 를 포함하는 패턴의 모든표현식에 적용됩니다.

12.2 열거형분해 (역구조화)

튜플과 마찬가지로 구조체 및 enum 도 다음을 일치시켜 디스트럭처링할 수있습니다.

구조체

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

#[rustfmt::skip]
fn main() {
    let foo = Foo { x: (1, 2), y: 3 };
    match foo {
        Foo { x: (1, b), y } => println!("x.0 = 1, b = {b}, y = {y}"),
        Foo { y: 2, x: i }   => println!("y = 2, x = {i:?}"),
        Foo { y, .. }        => println!("y = {y}, 다른 필드는 무시됨"),
    }
}
```

열거형

구조체나 열거형 값의 일부를 패턴 매치를 통해 변수에 바인딩할 수있습니다. 간단한 enum 타입을 먼저 살펴보겠습니다:

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

fn divide_in_two(n: i32) -> Result {
    if n % 2 == 0 {
```

```
        Result::Ok(n / 2)
    } else {
        Result::Err(format!("{n}을 (를) 두 개의 동일한 부분으로 나눌 수 없음"))
    }
}

fn main() {
    let n = 100;
    match divide_in_two(n) {
        Result::Ok(half) => println!("{n}을 (를) 둘로 나눈 값은 {half}입니다."),
        Result::Err(msg) => println!("죄송합니다. 오류가 발생했습니다. {msg}"),
    }
}
```

match 구문에서 divide_in_two 함수에서 반환되는 Result 값을 두 개의 팔 (혹은 가지) 로 *(destructure)* 하였습니다. 첫번째 팔에서 half 는 Ok variant 에 담긴값으로 바인딩됩니다. 두번째 팔에서 msg 는 오류 메시지 문자열에바인딩됩니다.

구조체

- foo 의 리터럴 값을 다른 패턴과 일치하도록 변경합니다.
- Foo 에 새 필드를 추가하고 필요에 따라 패턴을 변경합니다.
- 캡처와 상수 표현식은 구분하기 어려울 수 있습니다. 두 번째 부문의 2 를 변수로 변경해 보고 작동하지 않는 것을 확인하세요. const 로 변경하고 다시 작동하는지 확인합니다.

열거형

키 포인트:

- if/else 표현식은 열거형을 반환하고, 이 값은 나중에 match 로 분해됩니다.
- 열거형에 세번째 variant 를 추가하고 코드를 실행하여 오류를표시해보세요. 코드 어느 부분에 누락이 있는지, 그리고 컴파일러가 어떤식으로 힌트를 주는지 같이 살펴보세요.
- The values in the enum variants can only be accessed after being pattern matched.
- Demonstrate what happens when the search is inexhaustive. Note the advantage the Rust compiler provides by confirming when all cases are handled.
- Save the result of divide_in_two in the result variable and match it in a loop. That won't compile because msg is consumed when matched. To fix it, match &result instead of result. That will make msg a reference so it won't be consumed. This "match ergonomics" appeared in Rust 2018. If you want to support older Rust, replace msg with ref msg in the pattern.

12.3 흐름제어

Rust 에는 다른 언어와는 다른 몇 가지 제어 흐름 구조가 있으며 패턴 일치에사용됩니다.

- if let 표현식
- while let expressions
- match 표현식

`if let` 표현식

`if let` 표현식을 사용하면 값이 패턴과 일치하는지에 따라 다른 코드를 실행할 수있습니다:

```
fn sleep_for(secs: f32) {
    let dur = if let Ok(dur) = std::time::Duration::try_from_secs_f32(secs) {
        dur
    } else {
        std::time::Duration::from_millis(500)
    };
    std::thread::sleep(dur);
    println!("{:?} 동안 잠들었습니다.", dur);
}

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

`let else` expressions

패턴을 일치시키고 함수에서 반환하는 일반적인 경우에는 `let else` 를사용합니다. 'else' 사례는 해당 코드를벗어나야 합니다 (return, break 또는 패닉 - 블록의 다음위치로 이동하는 것만 아니면 됩니다).

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

    let first_byte_char = if let Some(first_byte_char) = s.chars().next() {
        first_byte_char
    } else {
        return Err(String::from("got empty string"));
    };

    if let Some(digit) = first_byte_char.to_digit(16) {
        Ok(digit)
    } else {
        Err(String::from("16 진수가 아님"))
    }
}

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

마지막으로, 무한 루프를 만드는 `loop` 키워드가 있습니다:

```
fn main() {
```

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

Here `String::pop` returns `Some(c)` until the string is empty, after which it will return `None`. The `while let` lets us keep iterating through all items.

if-let

- `if let` 이 `match` 보다 더 간결할 수 있습니다 (예: 한가지브랜치만 흥미로운 경우). 이와 달리 `match` 에서는 모든 브랜치가처리되어야 합니다.
- 일반적 사용법은 `Option` 을 사용할 때 `Some` 값을처리하는 것입니다.
- `match` 와 달리 `if let` 은 패턴 일치를 위한 보호 절을지원하지 않습니다.

let-else

위에서 본 것처럼 `if-let` 은 중첩할 수 있습니다. `let-else` 구조는 이 중첩된 코드의 평면화를 지원합니다. 코드가 어떻게 변화하는지학생들이 볼 수 있도록 어색한 버전을 다시 작성하세요.

다시 작성된 버전은 다음과 같습니다.

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

    let Some(first_byte_char) = s.chars().next() else {
        return Err(String::from("got empty string"));
    };

    let Some(digit) = first_byte_char.to_digit(16) else {
        return Err(String::from("16 진수가 아님"));
    };

    return Ok(digit);
}
```

while-let

- `while let` 은 값이 패턴에 매치되는 동안 계속됩니다.
- You could rewrite the `while let` loop as an infinite loop with an if statement that breaks when there is no value to unwrap for `name.pop()`. The `while let` provides syntactic sugar for the above scenario.

12.4 연습문제: 표현식평가

Let's write a simple recursive evaluator for arithmetic expressions.

여기서 Box 타입은 스마트 포인터이며 이 과정의 후반부에서 자세히다룹니다. 테스트에서 볼 수 있듯이 표현식은 Box::new 를 사용하여"박스로 표시"할 수 있습니다. 박스로 표시된 표현식을 평가하려면 deref 연산자 (*) 를 사용하여 "박스 표시를 해제"합니다: eval(*boxed_expr).

일부 표현식은 평가할 수 없으며 오류를 반환합니다. 표준 Result<Value, String> 타입은 성공한 값을 나타내거나 (Ok(Value)) 메시지와 함께 오류를나타냅니다 (Err(String)). 나중에 이 Result 타입에 대해서 자세히살펴보겠습니다.

코드를 복사하여 Rust 플레이그라운드에 붙여넣고 eval 구현을시작합니다. 최종 생성물은 테스트를 통과해야 합니다. todo!() 를사용하고 테스트를 하나씩 통과하도록 하면 도움이 될 수 있습니다. #[ignore] 를 사용하여 테스트를 일시적으로 건너뛸 수도있습니다.

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

일찍 완료한 경우 0 으로 나누기나 정수 오버플로가 발생하는 테스트를 작성해보세요. 패닉 대신 Result 로 이 문제를 어떻게 처리할 수 있을까요?

```
/// 두 개의 하위 표현식에서 실행할 연산입니다.
#[derive(Debug)]
enum Operation {
    Add,
    Sub,
    Mul,
    Div,
}

/// 트리 형식의 표현식입니다.
#[derive(Debug)]
enum Expression {
    /// 두 개의 하위 표현식에 관한 연산입니다.
    Op { op: Operation, left: Box<Expression>, right: Box<Expression> },

    /// 리터럴 값
    Value(i64),
}

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

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

#[test]
fn test_sum() {
    assert_eq!(
        eval(Expression::Op {
            op: Operation::Add,
            left: Box::new(Expression::Value(10)),
            right: Box::new(Expression::Value(20)),
```

```
        }),
        Ok(30)
    );
}

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

#[test]
fn test_error() {
    assert_eq!(
        eval(Expression::Op {
            op: Operation::Div,
            left: Box::new(Expression::Value(99)),
            right: Box::new(Expression::Value(0)),
        }),
        Err(String::from("0으로 나누기"))
    );
}
```

12.4.1 해답

```
/// 두 개의 하위 표현식에서 실행할 연산입니다.
#[derive(Debug)]
enum Operation {
    Add,
    Sub,
    Mul,
```

```
    Div,
}

/// 트리 형식의 표현식입니다.
#[derive(Debug)]
enum Expression {
    /// 두 개의 하위 표현식에 관한 연산입니다.
    Op { op: Operation, left: Box<Expression>, right: Box<Expression> },

    /// 리터럴 값
    Value(i64),
}

fn eval(e: Expression) -> Result<i64, String> {
    match e {
        Expression::Op { op, left, right } => {
            let left = match eval(*left) {
                Ok(v) => v,
                e @ Err(_) => return e,
            };
            let right = match eval(*right) {
                Ok(v) => v,
                e @ Err(_) => return e,
            };
            Ok(match op {
                Operation::Add => left + right,
                Operation::Sub => left - right,
                Operation::Mul => left * right,
                Operation::Div => {
                    if right == 0 {
                        return Err(String::from("0 으로 나누기"));
                    } else {
                        left / right
                    }
                }
            })
        }
        Expression::Value(v) => Ok(v),
    }
}

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

#[test]
fn test_sum() {
    assert_eq!(
        eval(Expression::Op {
            op: Operation::Add,
```

```
            left: Box::new(Expression::Value(10)),
            right: Box::new(Expression::Value(20)),
        }),
        Ok(30)
    );
}

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

#[test]
fn test_error() {
    assert_eq!(
        eval(Expression::Op {
            op: Operation::Div,
            left: Box::new(Expression::Value(99)),
            right: Box::new(Expression::Value(0)),
        }),
        Err(String::from("0 으로 나누기"))
    );
}

fn main() {
    let expr = Expression::Op {
        op: Operation::Sub,
        left: Box::new(Expression::Value(20)),
        right: Box::new(Expression::Value(10)),
    };
```

```
    println!("expr: {:?}", expr);
    println!("결과: {:?}", eval(expr));
}
```

제 13 장

메소드와트레이트

This segment should take about 50 minutes. It contains:

Slide	Duration
메서드	10 minutes
트레잇 (Trait)	15 minutes
트레잇 상속하기	3 minutes
연습문제: 일반 min	20 minutes

13.1 메서드

러스트에서 선언된 타입에 대해 `impl` 블록에 함수를 선언하여메서드를 연결 할 수 있습니다:

```
#[derive(Debug)]
struct Race {
    name: String,
    laps: Vec<i32>,
}

impl Race {
    // 수신자 없음, 정적 메서드입니다.
    fn new(name: &str) -> Self {
        Self { name: String::from(name), laps: Vec::new() }
    }

    // self에 대한 독점적 빌림 읽기/쓰기 액세스입니다.
    fn add_lap(&mut self, lap: i32) {
        self.laps.push(lap);
    }

    // self에 대한 공유 및 읽기 전용 빌림 액세스입니다.
    fn print_laps(&self) {
        println!("랩 타임 {}회, {} 경기:", self.laps.len(), self.name);
```

```
        for (idx, lap) in self.laps.iter().enumerate() {
            println!("{idx}랩: {lap}초");
        }
    }

    // self의 독점적 소유권
    fn finish(self) {
        let total: i32 = self.laps.iter().sum();
        println!("{} 레이스 종료, 총 랩 시간: {}", self.name, total);
    }
}

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

The `self` arguments specify the "receiver" - the object the method acts on. There are several common receivers for a method:

- `&self`: 호출자로부터 공유가능한 불변 참조 방식으로 객체를빌려옴을 나타냅니다. 객체는 메소드 호출 뒤에도 사용될 수 있습니다.
- `&mut self`: 호출자로부터 유일한 가변 참조 방식으로 객체를빌려옴을 나타냅니다. 객체는 메소드 호출 뒤에도 사용될 수 있습니다.
- `self`: 호출자로부터 객체의 소유권을 가져오고 객체는호출자로부터 메소드로 이동됩니다. 메소드가 객체를 소유하게 되며 따라서명시적으로 소유권을 다른 곳으로 전달하지 않는다면 메서드 종료와 함께객체는 drop(해제) 됩니다.
- `mut self`: same as above, but the method can mutate the object.
- 리시버 없음: 구조체의 정적 메서드가 됩니다. 주로 생성자를 만들때사용하게 되며, 생성자는 흔히 new 라고 이름붙입니다.

키 포인트:

- 메서드를 함수와 비교하여 소개하는 것도 도움이 될 수 있습니다.
  - 메서드는 구조체나 열거형과 같은 타입의 인스턴스에서 호출 되며, 첫번째 매개변수 (파라메터) 는 인스턴스를 `self` 로 표기합니다.
  - 메서드를 이용하면 receiver 문법을 사용할 수 있고 코드를 좀더체계적으로 정리할 수 있습니다. 메서드들이 예측 가능한 위치에 모여있으니 찾기 쉽습니다.
- 메서드 receiver 인 `self` 키워드 사용을 언급해 주시기 바랍니다.
  - 예제의 경우 `self: &Self` 의 줄인 버전임을 알려주고, 구조체의 이름을 직접 사용하면 어떻게 되는지 보여주는 것도 좋습니다.
  - `impl` 블록 내부에서는 `Self` 가 해당 타입의 이름대용으로 사용될 수 있음을 알려주세요.
  - 구조체의 필드를 접근할 때 점 표기를 사용하듯이 `self` 에 점표기를 사용하여 개별 필드들을 접근할 수 있습니다.
  - This might be a good time to demonstrate how the `&self` differs from `self` by trying to run `finish` twice.
  - `self` 를 사용하는 이같은 변형들 외에도 `Box<Self>`와 같이 리시버 타입으로허용되는특

별한래퍼 타입이 있습니다.

13.2 트레잇 (Trait)

트레잇은 타입을 추상화 하는데 사용됩니다. 인터페이스와 비슷합니다:

```
trait Pet {
    /// Return a sentence from this pet.
    fn talk(&self) -> String;

    /// Print a string to the terminal greeting this pet.
    fn greet(&self);
}
```

- 트레잇은 해당 트레잇을 구현하기 위해 타입이 가져야 하는 여러 메서드를정의합니다.
- In the "Generics" segment, next, we will see how to build functionality that is generic over all types implementing a trait.

13.2.1 Implementing Traits

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

    fn greet(&self) {
        println!("오, 귀여워! 이름이 뭐야? {}", self.talk());
    }
}

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

impl Pet for Dog {
    fn talk(&self) -> String {
        format!("멍멍, 제 이름은 {}입니다.", self.name)
    }
}

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

- To implement `Trait` for `Type`, you use an `impl Trait for Type { .. }` block.
- Unlike Go interfaces, just having matching methods is not enough: a `Cat` type with a `talk()` method would not automatically satisfy `Pet` unless it is in an `impl Pet` block.
- Traits may provide default implementations of some methods. Default implementations can rely on all the methods of the trait. In this case, `greet` is provided, and relies on `talk`.

13.2.2 트레잇 (Trait)

A trait can require that types implementing it also implement other traits, called *supertraits*. Here, any type implementing `Pet` must implement `Animal`.

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

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

struct Dog(String);

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

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

fn main() {
    let puppy = Dog(String::from("렉스"));
    println!("{} has {} legs", puppy.name(), puppy.leg_count());
}
```

This is sometimes called ”trait inheritance” but students should not expect this to behave like OO inheritance. It just specifies an additional requirement on implementations of a trait.

13.2.3 공유타입

Associated types are placeholder types which are supplied by the trait implementation.

```
#[derive(Debug)]
struct Meters(i32);
#[derive(Debug)]
struct MetersSquared(i32);

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

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

```
    }
}

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

- Associated types are sometimes also called ”output types”. The key observation is that the implementer, not the caller, chooses this type.
- Many standard library traits have associated types, including arithmetic operators and `Iterator`.

13.3 트레잇상속하기

지원되는 트레잇은 다음과 같이 맞춤 타입에 자동으로 구현할 수 있습니다.

```
#[derive(Debug, Clone, Default)]
struct Player {
    name: String,
    strength: u8,
    hit_points: u8,
}

fn main() {
    let p1 = Player::default(); // 기본 트레잇은 `default` 생성자를 추가합니다.
    let mut p2 = p1.clone(); // Clone 트레잇은 `clone` 메서드를 추가합니다.
    p2.name = String::from("EldurScrollz");
    // Debug 트레잇은 `{:?}` 표현을 사용한 출력을 지원합니다.
    println!("{:?} 대 {:?}", p1, p2);
}
```

상속은 매크로를 사용하여 구현되며 많은 크레이트가 유용한 상속 매크로를제공하여 유용한 기능을 추가합니다. 예를 들어 `serde` 는`#[derive(Serialize)]` 를 사용하여 구조체의 직렬화 지원을상속할 수 있습니다.

13.4 Exercise: Logger Trait

Let's design a simple logging utility, using a trait `Logger` with a `log` method. Code which might log its progress can then take an `&impl Logger`. In testing, this might put messages in the test logfile, while in a production build it would send messages to a log server.

However, the `StderrLogger` given below logs all messages, regardless of verbosity. Your task is to write a `VerbosityFilter` type that will ignore messages above a maximum verbosity.

This is a common pattern: a struct wrapping a trait implementation and implementing that same trait, adding behavior in the process. What other kinds of wrappers might be useful in a logging utility?

```
use std::fmt::Display;

pub trait Logger {
    /// Log a message at the given verbosity level.
```

```
    fn log(&self, verbosity: u8, message: impl Display);
}

struct StderrLogger;

impl Logger for StderrLogger {
    fn log(&self, verbosity: u8, message: impl Display) {
        eprintln!("verbosity={verbosity}: {message}");
    }
}

fn do_things(logger: &impl Logger) {
    logger.log(5, "FYI");
    logger.log(2, "Uhoh");
}

// TODO: Define and implement `VerbosityFilter`.

fn main() {
    let l = VerbosityFilter { max_verbosity: 3, inner: StderrLogger };
    do_things(&l);
}
```

13.4.1 해답

```
use std::fmt::Display;

pub trait Logger {
    /// Log a message at the given verbosity level.
    fn log(&self, verbosity: u8, message: impl Display);
}

struct StderrLogger;

impl Logger for StderrLogger {
    fn log(&self, verbosity: u8, message: impl Display) {
        eprintln!("verbosity={verbosity}: {message}");
    }
}

fn do_things(logger: &impl Logger) {
    logger.log(5, "FYI");
    logger.log(2, "Uhoh");
}

/// Only log messages up to the given verbosity level.
struct VerbosityFilter {
    max_verbosity: u8,
    inner: StderrLogger,
}
```

```
impl Logger for VerbosityFilter {
    fn log(&self, verbosity: u8, message: impl Display) {
        if verbosity <= self.max_verbosity {
            self.inner.log(verbosity, message);
        }
    }
}

fn main() {
    let l = VerbosityFilter { max_verbosity: 3, inner: StderrLogger };
    do_things(&l);
}
```

제 14 장

제네릭

This segment should take about 40 minutes. It contains:

Slide	Duration
외부 (다른언어) 함수들	5 minutes
제네릭 데이터 타입	10 minutes
제네릭 타입 제한 (트레잇 경계)	10 minutes
트레잇 구현하기 (impl Trait)	5 minutes
연습문제: 일반 min	10 minutes

14.1 외부 (다른언어) 함수들

Rust supports generics, which lets you abstract algorithms or data structures (such as sorting or a binary tree) over the types used or stored.

```
/// `n` 값에 따라 `even` 또는 `odd`를 선택합니다.
fn pick<T>(n: i32, even: T, odd: T) -> T {
    if n % 2 == 0 {
        even
    } else {
        odd
    }
}

fn main() {
    println!("선택한 숫자: {:?}", pick(97, 222, 333));
    println!("선택한 튜플: {:?}", pick(28, ("개", 1), ("고양이", 2)));
}
```

- Rust 는 인수 및 반환 값의 타입을 기반으로 T 의 타입을 추론합니다.
- 이는 C++ 템플릿과 비슷하지만, Rust 는 제네릭 함수를 즉시 부분적으로컴파일하므로 제약 조건과 일치하는 모든 타입에 함수가 유효해야 합니다. 예를 들어 `n == 0` 인 경우 `even + odd` 를 반환하도록 `pick` 을 수정하세요. 정수가 포함된 `pick` 인스턴스화만사용되더라도 Rust 는 이를 유효하지 않은 것으로 간주합니다. C++에서는이를 허용합니다.

- Generic code is turned into non-generic code based on the call sites. This is a zero-cost abstraction: you get exactly the same result as if you had hand-coded the data structures without the abstraction.

14.2 제네릭 데이터타입

제네릭을 사용하여 필드의 타입을 추상화 할 수 있습니다:

```
#[derive(Debug)]
struct Point<T> {
    x: T,
    y: T,
}

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

    // fn set_x(&mut self, x: T)
}

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

- : `impl<T> Point<T> {}`에서 `T`가 왜 두 번 사용됩니까?
  - 제네릭 타입에 대한 제네릭 구현 이기 때문입니다. 이 두 제네릭은 서로독립적입니다.
  - 이는 임의의 모든 `T`에 대해서 이 메소드들이 정의된다는 것을의미합니다.
  - It is possible to write `impl Point<u32> { .. }`.
    * `Point`는 여전히 제네릭이며 `Point<f64>`를 사용할 수도 있지만 이블록의 메서드는 `Point<u32>`만 쓸 수있습니다.
- 새 변수 `let p = Point { x: 5, y: 10.0 };`를 선언해보세요. 다음과 같은 두 개의 타입 변수를 사용하여 다른 타입의 요소를가진 포인트를 허용하도록 코드를 업데이트합니다. `T`와 `U`

14.3 제네릭

Traits can also be generic, just like types and functions. A trait's parameters get concrete types when it is used.

```
#[derive(Debug)]
struct Foo(String);

impl From<u32> for Foo {
    fn from(from: u32) -> Foo {
        Foo(format!("Converted from integer: {from}"))
    }
```

```
}

impl From<bool> for Foo {
    fn from(from: bool) -> Foo {
        Foo(format!("Converted from bool: {from}"))
    }
}

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

- The `From` trait will be covered later in the course, but its definition in the `std` docs is simple.
- Implementations of the trait do not need to cover all possible type parameters. Here, `Foo::From("hello")` would not compile because there is no `From<&str>` implementation for `Foo`.
- Generic traits take types as "input", while associated types are a kind of "output type. A trait can have multiple implementations for different input types.
- In fact, Rust requires that at most one implementation of a trait match for any type `T`. Unlike some other languages, Rust has no heuristic for choosing the "most specific" match. There is work on adding this support, called specialization.

14.4 제네릭 타입 제한 (트레잇경계)

제네릭을 이용하다 보면 타입이 어떤 트레잇을 구현하고 있어야 하는 경우가있습니다. 그래야 그 트레잇의 메서드를 호출할 수 있기 때문입니다.

`T: Trait` 혹은 `impl Trait` 를 사용하면 됩니다:

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

// struct NotClonable;

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

- `NonClonable` 을 만들어 `duplicate` 에 전달해 보세요.
- 여러 트레잇이 필요한 경우 `+`를 사용하여 트레잇을 결합합니다.
- `where` 문법을 사용할 수도 있습니다. 수강생들도 코드를 읽다가그 문법을 마주할 수 있습니다.

```
fn duplicate<T>(a: T) -> (T, T)
where
    T: Clone,
```

```
{
    (a.clone(), a.clone())
}
```

  - 이를 이용하면 타입 파라메터가 많은 경우 함수 시그니처를 간결하게정리하는 데 도움이 됩니다.
  - 좀 더 강력한 추가 기능도 제공합니다.
    * : 왼쪽에 임의의 타입 (예를 들어 `Option<T>`) 을 사용할 수 있습니다.

- Rust 는 아직 특수화를 지원하지 않습니다. 예를 들어 원본 `duplicate` 가 있는 경우 특수 `duplicate(a: u32)` 를추가하는 것은 유효하지 않습니다.

14.5 트레잇구현하기 (`impl Trait`)

트레잇 바운드와 유사하게 `impl Trait` 문법은 함수의 인자와반환값에도 적용 가능합니다:

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

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

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

`impl Trait` allows you to work with types which you cannot name. The meaning of `impl Trait` is a bit different in the different positions.

- 함수 인자의 타입으로 사용되었을 경우에는 `impl Trait` 는 트레잇경계가 있는 익명의 제네릭 타입을 의미합니다.
- 리턴 타입으로 사용되었을 경우에는, 그 트레잇을 구현하는 구체적인타입인데, 타입 이름을 프로그래머가 짓지 않았다는 것을 의미합니다. 이는그 구체적인 타입 이름을 API 로 공개하고 싶지 않은 경우에 유용합니다.

  함수가 리턴되는 곳에서의 타입 추론은 어렵습니다. 어떤 함수의 리턴타입이 `impl Foo` 로 선언되어 있을 경우, 그 함수가 실제로리턴하는 타입은 소스 코드 상 어디에도 나타나 있지 않습니다. `collect<B<() -> B` 와 같이제너릭 타입을 리턴하는 함수는 `B` 를 만족하는 어떤 타입도 리턴할수 있습니다. 이 경우, 호출하는 측에서는 `let x: Vec<_> = foo.collect()` 나터보피시 문법을 써서 `foo.collect::<Vec<_>>()` 와같이 리턴 타입을 명시적으로 써 주어야 할 수도 있습니다.

`debuggable` 타입은 무엇인가요? `let debuggable: () = ..` 을 시도하여 오류 메시지가 어떻게 표시되는지 확인합니다.

14.6 Exercise: Generic `min`

In this short exercise, you will implement a generic `min` function that determines the minimum of two values, using the `Ord` trait.

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

// TODO: `main`에 사용되는 `min` 함수를 구현합니다.

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

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

    assert_eq!(min("hello", "goodbye"), "goodbye");
    assert_eq!(min("bat", "armadillo"), "armadillo");
}
```

- Show students the `Ord` trait and `Ordering` enum.

14.6.1 해답

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

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

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

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

    assert_eq!(min("hello", "goodbye"), "goodbye");
    assert_eq!(min("bat", "armadillo"), "armadillo");
}
```

제 IV 편

2일차 오후

제 15 장

Welcome Back

Including 10 minute breaks, this session should take about 3 hours and 10 minutes. It contains:

Segment	Duration
표준 라이브러리	1 hour and 20 minutes
표준 라이브러리	1 hour and 40 minutes

제 16 장

표준라이브러리

This segment should take about 1 hour and 20 minutes. It contains:

Slide	Duration
표준 라이브러리	3 minutes
문서화주석 테스트	5 minutes
Duration	10 minutes
Option, Result	10 minutes
String	10 minutes
Vec	10 minutes
HashMap	10 minutes
연습문제: 카운터	20 minutes

이 섹션의 각 슬라이드에서는 문서 페이지를 검토하고 보다 일반적인메서드를 중점적으로 살펴봅니다.

16.1 표준라이브러리

Rust comes with a standard library which helps establish a set of common types used by Rust libraries and programs. This way, two libraries can work together smoothly because they both use the same `String` type.

In fact, Rust contains several layers of the Standard Library: `core`, `alloc` and `std`.

- `core` includes the most basic types and functions that don't depend on `libc`, allocator or even the presence of an operating system.
- `alloc` 은 `Vec`, `Box`, `Arc` 와 같이 전역힙 할당이 필요한 타입을 포함합니다.
- 임베디드 러스트 응용프로그램은 주로 `core` 만 사용하거나 가끔 `alloc` 을 함께 사용합니다.

16.2 문서화주석테스트

Rust comes with extensive documentation. For example:

- All of the details about loops.
- u8 과같은 기본 타입

- Standard library types like `Option` or `BinaryHeap`.

사실 자체 코드를 문서화할 수 있습니다.

```
/// 첫 번째 인수를 두 번째 인수로 나눌 수 있는지 확인합니다.
///
/// 두 번째 인수가 0 이면 결과는 false 입니다.
fn is_divisible_by(lhs: u32, rhs: u32) -> bool {
    if rhs == 0 {
        return false;
    }
    lhs % rhs == 0
}
```

콘텐츠는 마크다운으로 처리됩니다. 게시된 모든 Rust 라이브러리 크레이트는 rustdoc 도구를 사용하여 `docs.rs` 에 자동으로문서화됩니다. 일반적으로 API 의 모든 공개 항목은 이 패턴을 사용하여문서화됩니다.

항목 내부 (예: 모듈 내부) 의 항목을 문서화하려면 ' 내부문서 주석' 이라고 하는 `//!` 혹은`/*! .. */` 를 사용하세요.

```
//! 이 모듈에는 정수의 분할 가능성과 관련된 기능이 포함되어 있습니다.
```

- Show students the generated docs for the `rand` crate at https://docs.rs/rand.

16.3 Duration

`Option<T>`의 일부 사용법은 이미살펴봤습니다. 'T' 타입의 값을저장하거나 아무것도 저장하지 않습니다. 예를 들어 `String::find` 는 `Option<usize>`를 반환합니다.

```
fn main() {
    let name = "Löwe 老虎 Léopard Gepardi";
    let mut position: Option<usize> = name.find('é');
    println!("find 의 반환값 {position:?}");
    assert_eq!(position.unwrap(), 14);
    position = name.find('Z');
    println!("find 의 반환값 {position:?}");
    assert_eq!(position.expect("문자를 찾을 수 없음"), 0);
}
```

- `Option` is widely used, not just in the standard library.
- `unwrap` 은 `Option` 의 값을 반환하거나 패닉을 반환합니다. `expect` 도 비슷하지만 오류 메시지가 표시됩니다.
  - None 발생 시 패닉 상태가 될 수 있지만' 실수' 로 None 을 체크하는 것을 잊을수는 없습니다.
  - 무언가를 함께 해킹할 때 모든 곳에서 `unwrap/expect` 를 실행하는 것이 일반적이지만프로덕션 코드는 일반적으로 더 나은 방식으로 None 을처리합니다.
- 틈새 최적화란 `Option<T>`가 메모리에서 `T` 와 크기가 같은 경우가 많다는 것을 의미합니다.

16.4 Option, Result

`Result` 는 `Option` 과 유사하지만 작업의 성공 또는 실패를나타내며, 각각 타입이 다릅니다. 이는 표현식 연습에서 정의된 `Res` 와 유사하지만 제네릭: `Result<T, E>`입니다. 여기서 `T` 는 `Ok` 변형에 사용되고 `E` 는 `Err` 변형에 표시됩니다.

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

fn main() {
    let file: Result<File, std::io::Error> = File::open("diary.txt");
    match file {
        Ok(mut file) => {
            let mut contents = String::new();
            if let Ok(bytes) = file.read_to_string(&mut contents) {
                println!("다이어리: {contents}({bytes}바이트)");
            } else {
                println!("파일 콘텐츠를 읽을 수 없습니다.");
            }
        }
        Err(err) => {
            println!("다이어리를 열 수 없습니다. {err}");
        }
    }
}
```

- Option 와 마찬가지로, 성공한 경우의 값은 Result 내부에 있습니다. 그래서, 개발자는 명시적으로 이를 추출하여야 합니다. 이렇게 함으로써 값을 읽기 전에 오류 발생 여부를 반드시 체크하도록유도하고 있습니다. 만일 오류가 절대 발생하지 않는 경우라면 unwrap() 이나 expect() 를 사용할 수 있으며, 이는개발자의 의도 (: 오류가 발생할 수 없음) 을 명시적으로나타내는 방법이기도 합니다.
- Result documentation is a recommended read. Not during the course, but it is worth mentioning. It contains a lot of convenience methods and functions that help functional-style programming.
- Result 는 오류 처리를 위한 표준 타입입니다. 3 일차 과정에서살펴봅니다.

16.5 String

String 은힙에 할당되고 가변 길이의 표준 UTF-8 문자열 버퍼입니다:

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

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

    let s3 = String::from("  ");
    println!("s3: len = {}, 문자 수 = {}", s3.len(), s3.chars().count());
}
```

String 은 Deref<Target = str>을구현합니다. 이는 , String 값에 대해서도 str 의 모든메서드를 호출 할 수 있다는 의미 입니다.

- String::new 는 새로운 빈 문자열을 반환합니다. String::with_capacity 는 새로 만들

문자열 버퍼에 넣을 데이터크기를 알고 있는 경우에 사용할 수 있습니다.

- `String::len` 은 `String` 의 바이트 크기를 반환합니다. (실제 문자 개수와는 다를 수 있습니다.)
- `String::chars` 는 실제 문자 (character) 들에 대한 이터레이터를반환합니다. `char` 로 표현되는 문자는 우리가 실제로 인식하고사용하는 문자와는 다를 수 있습니다. 자소 결합으로 문자를 표현하는경우가 있기 때문입니다. 이에 대해서는 Grapheme Cluster 를 참고하세요.
- 사람들이 문자열이라고 말할 때에는 `&str` 이거나 `String` 일 수 있습니다.
- 어떤 타입이 `Deref<Target = T>`를구현하고 있으면, 컴파일러는 여러분이 `T` 의 메소드들을 호출할 수있게 도와줍니다.
  - `Deref` 트레잇에 관해서 아직 다루지 않았으므로 이 시점에서는이것으로 문서의 사이드바 구조가 대부분 설명됩니다.
  - `String` 은 `Deref<Target = str>`을 구현하고있기 때문에 `String` 에 대해서도 `str` 메소드들을호출할 수 있습니다.
  - Write and compare `let s3 = s1.deref();` and `let s3 = &*s1;`.
- `String` 은 바이트 벡터의 래퍼로 구현되어 있습니다. 벡터가지원하는 여러가지 연산들을 `String` 도 지원합니다. 다만 `String` 은 몇가지 보장 내용이 더 있습니다.
- `String` 을 인덱스로 접근하는 방법들을 비교해보세요:
  - `s3.chars().nth(i).unwrap()` 를 이용하여 한 문자를 선택하는경우, `i` 값이 범위를 벗어날 때, 벗어나지 않을 때 동작을설명하세요.
  - `s3[0..4]` 를 이용해서 문자열의 일부를 선택하는데, 슬라이스가 유니코드 문자열 경계에 딱 맞지 않을 경우 어떻게 되는지설명하세요.
- Many types can be converted to a string with the `to_string` method. This trait is automatically implemented for all types that implement `Display`, so anything that can be formatted can also be converted to a string.

16.6 Vec

`Vec` 는 힙에 할당된 표준 가변 크기 버퍼입니다:

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

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

    // 요소가 있는 벡터를 초기화하는 표준 매크로입니다.
    let mut v3 = vec![0, 0, 1, 2, 3, 4];

    // 짝수 요소만 유지합니다.
    v3.retain(|x| x % 2 == 0);
    println!("{v3:?}");

    // 연속 중복 삭제
    v3.dedup();
    println!("{v3:?}");
}
```

Vec 은 Deref<Target = [T]>를구현합니다. 이는 Vec 에서 슬라이스 메서드를 호출 할 수 있다는 의미입니다.

- Vec is a type of collection, along with String and HashMap. The data it contains is stored on the heap. This means the amount of data doesn't need to be known at compile time. It can grow or shrink at runtime.
- Vec<T>는 제네릭 타입이기도 합니다. 하지만 T 를 꼭 지정해줄 필요는 없습니다. 이 경우, 러스트 타입추론이 벡터에 처음 push 하는 데이터로 T 를 알 수있었습니다.
- vec![...] 는 Vec::new() 대신 쓸 수 있는 표준매크로로서, 초기 데이터를 추가한 벡터를 생성할 수 있습니다.
- 벡터는 [ ] 를 사용하여 인덱스로 접근할 수있습니다. 하지만 범위를 벗어나면 패닉이 발생합니다. 대신 get 을 사용하면 Option 을 반환합니다. pop 함수는 마지막 요소를 제거합니다.
- Slices are covered on day 3. For now, students only need to know that a value of type Vec gives access to all of the documented slice methods, too.

16.7 HashMap

HashDoS 공격으로부터 보호되는 표준 해시 맵입니다:

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

fn main() {
    let mut page_counts = HashMap::new();
    page_counts.insert("허클베리 핀의 모험".to_string(), 207);
    page_counts.insert("그림 동화".to_string(), 751);
    page_counts.insert("오만과 편견".to_string(), 303);

    if !page_counts.contains_key("레 미제라블") {
        println!(
            "{}의 책은 알고 있지만 레 미제라블은 알지 못합니다.",
            page_counts.len()
        );
    }

    for book in ["오만과 편견", "이상한 나라의 앨리스"] {
        match page_counts.get(book) {
            Some(count) => println!("{book}: {count}페이지"),
            None => println!("{book}을(를) 알 수 없습니다."),
        }
    }

    // 값을 찾을 수 없는 경우 .entry() 메서드를 사용하여 값을 삽입합니다.
    for book in ["오만과 편견", "이상한 나라의 앨리스"] {
        let page_count: &mut i32 = page_counts.entry(book.to_string()).or_insert(0);
        *page_count += 1;
    }

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

- HashMap 은 prelude 에 정의되어 있지 않기 때문에 명시적으로추가해줘야 합니다.

- 아래 코드를 테스트해보세요. 첫 문장에서는 해시맵에 책이 있는지검사하여, 없으면 디폴트 값을 반환합니다. 두번 째 문장에서는 해시맵에해당 책이 없는 경우, 지정한 값을 해시맵에 추가한 뒤 그 값을반환합니다.

```
let pc1 = page_counts
    .get("해리 포터와 마법사의 돌")
    .unwrap_or(&336);
let pc2 = page_counts
    .entry("헝거게임".to_string())
    .or_insert(374);
```

- 안타깝지만 hashmap!같은 매크로가 없습니다.
  - 러스트 1.56 부터는 HashMap 이 From<[(K, V); N]>을구현하기 때문에 배열 리터럴을 이용하여 쉽게 해시맵을 초기화할 수있습니다:

```
let page_counts = HashMap::from([
  ("해리 포터와 마법사의 돌".to_string(), 336),
  ("헝거게임".to_string(), 374),
]);
```

- 키-값 쌍에 대한 Iterator 로 해시맵을 만들 수도 있습니다.
- 예제 코드에서는 편의상 해시맵의 키로 &str 를 사용하지않았습니다. 물론 컬렉션에 참조를 사용할 수도 있습니다. 다만 참조를사용하게 되면 빌림 검사기 때문에 복잡해 질 수 있습니다.
  - 예제 코드에서 to_string() 을 없애도 컴파일에 문제가 없는지확인해보세요. 어떤 문제에 부딪힐까요?
- 해시맵의 몇 몇 메서드는 해시맵 내부의 특별한 타입 (예를 들어 std::collections::hash_map::Keys) 들을 리턴합니다. 이러한타입들은 Rust 문서에서도 검색할 수 있습니다. 수강생들에게 이 타입들에대한 문서를 보여주고, 이 문서에 keys 메서드로의 역 링크가있음을 알려주세요.

16.8 연습문제: 카운터

이 연습에서는 매우 간단한 데이터 구조를 사용하여 제네릭으로 만듭니다. std::collections::HashMap 을사용하여 어떤 값이 표시되었는지, 각각 얼마나 표시되었는지 추적합니다.

Counter 의 초기 버전은 u32 값에만 작동하도록 하드코딩되어 있습니다. 추적 중인 값 타입에 대해 구조체 및 메서드를제네릭으로 만듭니다. 그러면 Counter 가 모든 타입의 값을 추적할수 있습니다.

일찍 완료한 경우 entry 메서드를 사용하여, count 메서드를 구현하는 데 필요한 해시 조회횟수를 절반으로 줄여보세요.

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

/// Counter는 각 T 타입 값이 표시된 횟수를 계산합니다.
struct Counter {
    values: HashMap<u32, u64>,
}

impl Counter {
    /// 새 Counter를 만듭니다.
    fn new() -> Self {
        Counter {
            values: HashMap::new(),
```

```
        }
    }

    /// 지정된 값의 발생 횟수를 셉니다.
    fn count(&mut self, value: u32) {
        if self.values.contains_key(&value) {
            *self.values.get_mut(&value).unwrap() += 1;
        } else {
            self.values.insert(value, 1);
        }
    }

    /// 지정된 값이 표시된 횟수를 반환합니다.
    fn times_seen(&self, value: u32) -> u64 {
        self.values.get(&value).copied().unwrap_or_default()
    }
}

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

    for i in 10..20 {
        println!("{} 개의 {} 값을 발견했습니다.", ctr.times_seen(i), i);
    }

    let mut strctr = Counter::new();
    strctr.count("사과");
    strctr.count("오렌지");
    strctr.count("사과");
    println!("사과 {}개 받음", strctr.times_seen("사과"));
}
```

16.8.1 해답

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

/// Counter는 각 T 타입 값이 표시된 횟수를 계산합니다.
struct Counter<T: Eq + Hash> {
    values: HashMap<T, u64>,
}

impl<T: Eq + Hash> Counter<T> {
    /// 새 Counter를 만듭니다.
    fn new() -> Self {
```

```
        Counter { values: HashMap::new() }
    }

    /// 지정된 값의 발생 횟수를 셉니다.
    fn count(&mut self, value: T) {
        *self.values.entry(value).or_default() += 1;
    }

    /// 지정된 값이 표시된 횟수를 반환합니다.
    fn times_seen(&self, value: T) -> u64 {
        self.values.get(&value).copied().unwrap_or_default()
    }
}

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

    for i in 10..20 {
        println!("{} 개의 {} 값을 발견했습니다.", ctr.times_seen(i), i);
    }

    let mut strctr = Counter::new();
    strctr.count("사과");
    strctr.count("오렌지");
    strctr.count("사과");
    println!("사과 {}개 받음", strctr.times_seen("사과"));
}
```

제 17 장

표준라이브러리

This segment should take about 1 hour and 40 minutes. It contains:

Slide	Duration
비교	10 minutes
Iterators	10 minutes
From 과 Into	10 minutes
테스트	5 minutes
Read 와 Write	10 minutes
Default, 구조체 업데이트 문법	5 minutes
클로저 (Closure)	20 minutes
연습문제: 바이너리 트리	30 minutes

표준 라이브러리 타입과 마찬가지로 각 트레잇에 관한 문서를 검토하는 데시간을 할애하세요.

이 섹션은 깁니다. 중간에 휴식을 취하세요.

17.1 비교

이러한 트레잇은 값 간의 비교를 지원합니다. 모든 트레잇은 이러한 트레잇을구현하는 필드를 포함하는 타입에 대해 상속될 수 있습니다.

`PartialEq` and `Eq`

`PartialEq` 는 필수 메서드인 `eq` 및 제공된 `ne` 메서드를 사용하는 부분 등가 관계입니다. `==` 및 `!=` 연산자는 이러한 메서드를 호출합니다.

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

```
    }
}
```

Eq is a full equivalence relation (reflexive, symmetric, and transitive) and implies `PartialEq`. Functions that require full equivalence will use `Eq` as a trait bound.

`PartialOrd` 및 `Ord`

`PartialOrd` 는 `partial_cmp` 메서드를 사용하여 부분순서를 정의합니다. <, <=, >=, > 연산자를 구현하는 데사용됩니다.

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

`Ord` 는 전체 순서 지정이며 `cmp` 는 `Ordering` 을반환합니다.

`PartialEq` 는 서로 다른 타입 간에 구현될 수 있지만 `Eq` 는구현될 수 없습니다. 반사적이기 때문입니다.

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

실제로 이러한 트레잇을 상속하는 것은 일반적이지만 구현하는 것은 드문일입니다.

17.2 Iterators

연산자 오버로드는 `std::ops` 에 있는 다양한 트레잇들을 통해구현됩니다:

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

impl std::ops::Add for Point {
```

```
    type Output = Self;

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

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

논의점:

- You could implement Add for &Point. In which situations is that useful?
  - 답: Add:add 는 self 를 소모합니다. 만약 타입 T 가 Copy 트레잇을 구현하고 있지 않다면&T 에 대해서도 연산자 오버로딩을 고려해야 합니다. 이렇게하면 호출부에서 불필요한 복사를 피할 수 있습니다.
- 왜 Output 이 연관된 타입인가요? 타입 파라메터로 만들 수있을까요?
  - Short answer: Function type parameters are controlled by the caller, but associated types (like Output) are controlled by the implementer of a trait.
- Add 를 이용해서 서로 다른 두 개의 타입을 더할 수도 있습니다. 예를 들어 `impl Add<(i32, i32)> for Point` 는튜플을 Point 에 더할 수 있게 해 줍니다.

17.3 From 과 Into

타입은 용이한 형변환을 위해 From 과 Into 를구현합니다:

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

From 이구현되면 Into 역시 자동으로 구현됩니다:

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

- 그렇기 때문에 사용자 정의 타입의 경우에도 From 만 구현하는것이 일반적입니다.
- "String 으로 변환할 수 있는 모든 것"과 같은 함수의 인수 타입을선언할 때에는 Into 를 사용해야 함을 조심하세요. 그래야만, 함수는 From 을 구현한 타입과 Into 구현한타입 모두를 인자로 받을 수 있습니다.

17.4 테스트

Rust 에는 타입 변환이 없지만 as 를 사용한 명시적변환은 지원됩니다. 이는 일반적으로 C 의미론을 따라 정의됩니다.

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

as 의 결과는 Rust 에서 정의되며 여러 플랫폼에서일관됩니다. 이는 기호를 변경하거나 더 작은 타입으로 변환할 때의 직관과일치하지 않을 수 있습니다. 문서를 확인하고 명확하게 설명해 주세요.

Casting with as is a relatively sharp tool that is easy to use incorrectly, and can be a source of subtle bugs as future maintenance work changes the types that are used or the ranges of values in types. Casts are best used only when the intent is to indicate unconditional truncation (e.g. selecting the bottom 32 bits of a u64 with as u32, regardless of what was in the high bits).

For infallible casts (e.g. u32 to u64), prefer using From or Into over as to confirm that the cast is in fact infallible. For fallible casts, TryFrom and TryInto are available when you want to handle casts that fit differently from those that don't.

이 슬라이드가 끝난 후 잠시 쉬어가는 것이 좋습니다.

as is similar to a C++ static cast. Use of as in cases where data might be lost is generally discouraged, or at least deserves an explanatory comment.

이는 정수를 usize 로 변환하여 색인으로 사용할 때 일반적입니다.

17.5 Read 와 Write

Read 와 BufRead 를사용하면 u8 타입의 데이터 스트림을 읽을 수 있습니다:

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

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

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

    let file = std::fs::File::open(std::env::current_exe()?)?;
    println!("파일 내 줄: {}", count_lines(file));
    Ok(())
}
```

이와 비슷하게, Write 를 사옹하면 u8 타입의 데이터를 쓸수 있습니다:

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

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

fn main() -> Result<()> {
    let mut buffer = Vec::new();
    log(&mut buffer, "안녕하세요")?;
    log(&mut buffer, "World")?;
    println!("로그 내역: {:?}", buffer);
    Ok(())
}
```

17.6 `Default` 트레잇

`Default` 트레잇은 어떤 타입에 대한 기본값을 제공합니다.

```rust
#[derive(Debug, Default)]
struct Derived {
    x: u32,
    y: String,
    z: Implemented,
}

#[derive(Debug)]
struct Implemented(String);

impl Default for Implemented {
    fn default() -> Self {
        Self("존 스미스".into())
    }
}

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

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

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

- 트레잇을 직접 구현하거나 `#[derive(Default)]` 를 붙여서컴파일러에게 구현을 맡길 수 있습니다.
- 컴파일러가 제공하는 자동 구현의 경우 모든 필드에 대해 기본 값을 설정한새 인스턴스를 반환합니다.

  - 이는 구조체의 각 필드 타입들이 모두 `Default` 트레잇을구현해야 함을 의미합니다.
- 러스트 표준 타입들은 대부분 `Default` 를 구현하고 있으며, 기본값은 `0` 이나 `""`처럼 예상 가능한 값들입니다.
- The partial struct initialization works nicely with default.
- The Rust standard library is aware that types can implement `Default` and provides convenience methods that use it.
- The `..` syntax is called struct update syntax.

17.7 클로저 (Closure)

클로저 혹은 람다표현식은 익명타입입니다. 이들은 Fn,FnMut, FnOnce 라는 특별한 트레잇을 구현합니다:

```
fn apply_with_log(func: impl FnOnce(i32) -> i32, input: i32) -> i32 {
    println!("Calling function on {input}");
    func(input)
}

fn main() {
    let add_3 = |x| x + 3;
    println!("add_3: {}", apply_with_log(add_3, 10));
    println!("add_3: {}", apply_with_log(add_3, 20));

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

    let multiply_sum = |x| x * v.into_iter().sum::<i32>();
    println!("multiply_sum: {}", apply_with_log(multiply_sum, 3));
}
```

Fn(예를 들어 add_3) 은 캡처된 값을 소모도 변경도 하지않고, 혹은 어떤 것도 캡쳐하지 않았을 수도 있기 때문에 동시에 여러번호출할 수 있습니다.

FnMut(예를 들어 accumulate) 는 캡처된 값을 변경할 수있으므로 여러 번 호출은 가능하지만 동시에 호출 할 수는 없습니다.

FnOnce (예를 들어 multiply_sum) 는 한번만 호출되며캡처된 값을 소모합니다.

FnMut 는 FnOnce 의 하위타입입니다. Fn 은 FnMut 과 FnOnce 의 하위 타입입니다. 즉, FnMut 는 FnOnce 가 호출되는 곳이면 어디서나 사용 할 수 있고 Fn 은 FnMut 와 FnOnce 가 호출되는 곳이면 어디든 사용할 수있습니다.

When you define a function that takes a closure, you should take FnOnce if you can (i.e. you call it once), or FnMut else, and last Fn. This allows the most flexibility for the caller.

In contrast, when you have a closure, the most flexible you can have is Fn (it can be passed everywhere), then FnMut, and lastly FnOnce.

컴파일러는 클로저가 무엇을 캡쳐하는지에 따라 Copy(예를 들어 add_3) 과 Clone(예를 들어 multiply_sum) 을알아서 추론합니다.

기본적으로 클로져는, 가능하다면, 참조를 사용하여 캡쳐를 합니다. move 키워드를 쓰면 값으로 캡쳐가 됩니다.

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

fn main() {
    let hi = make_greeter("Hi".to_string());
    hi("Greg");
}
```

17.8 연습문제: 바이너리트리

이 예에서는 기존의 "ROT13" 암호화를 구현합니다. 이 코드를 플레이그라운드에 복사하고 누락된 비트를구현합니다. 결과가 여전히 유효한 UTF-8 인지 확인하려면 ASCII 영문자만회전하세요.

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

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

// `RotDecoder`의 `Read` 트레잇을 구현합니다.

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

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

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

    #[test]
    fn binary() {
        let input: Vec<u8> = (0..=255u8).collect();
        let mut rot = RotDecoder::<&[u8]> { input: input.as_ref(), rot: 13 };
```

```
        let mut buf = [0u8; 256];
        assert_eq!(rot.read(&mut buf).unwrap(), 256);
        for i in 0..=255 {
            if input[i] != buf[i] {
                assert!(input[i].is_ascii_alphabetic());
                assert!(buf[i].is_ascii_alphabetic());
            }
        }
    }
}
```

각각 13 자씩 회전하는 두 개의 RotDecoder 인스턴스를 함께체이닝하면 어떻게 될까요?

17.8.1 해답

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

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

impl<R: Read> Read for RotDecoder<R> {
    fn read(&mut self, buf: &mut [u8]) -> std::io::Result<usize> {
        let size = self.input.read(buf)?;
        for b in &mut buf[..size] {
            if b.is_ascii_alphabetic() {
                let base = if b.is_ascii_uppercase() { 'A' } else { 'a' } as u8;
                *b = (*b - base + self.rot) % 26 + base;
            }
        }
        Ok(size)
    }
}

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

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

    #[test]
    fn joke() {
        let mut rot =
            RotDecoder { input: "Gb trg gb gur bgure fvqr!".as_bytes(), rot: 13 };
```

```
        let mut result = String::new();
        rot.read_to_string(&mut result).unwrap();
        assert_eq!(&result, "To get to the other side!");
    }

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

제 V 편

3일차 오전

제 18 장

3 일차개요

오늘 다룰 내용은 다음과 같습니다.

- 메모리 관리, 수명, 빌림 검사기: Rust 가 메모리 안전을 보장하는 방법
- 스마트 포인터: 표준 라이브러리 포인터 타입

일정예약

Including 10 minute breaks, this session should take about 2 hours and 20 minutes. It contains:

Segment	Duration
개요	3 minutes
메모리 관리	1 hour
스마트 포인터	55 minutes

제 19 장

메모리관리

This segment should take about 1 hour. It contains:

Slide	Duration
프로그램 메모리 검토	5 minutes
자동 메모리 관리	10 minutes
소유권	5 minutes
Move 문법	5 minutes
Clone	2 minutes
복합 타입	5 minutes
Drop	10 minutes
연습문제: 빌드 타입	20 minutes

19.1 프로그램 메모리검토

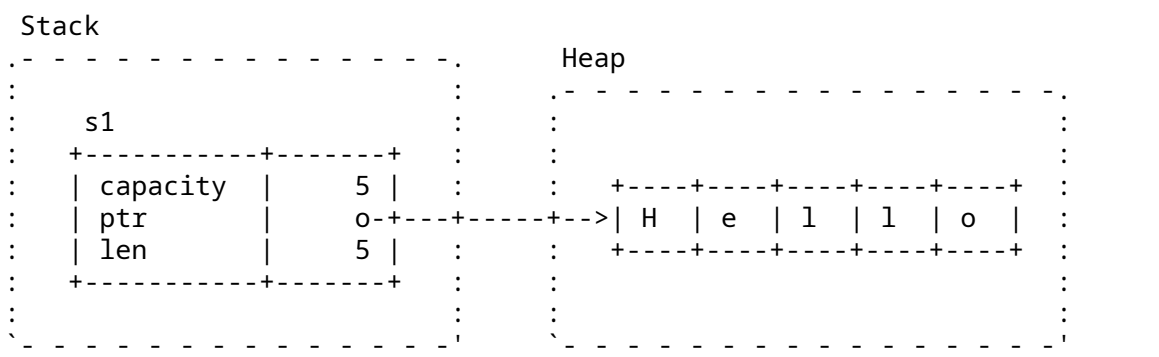
프로그램은 두 가지 방법으로 메모리를 할당합니다.

- 스택: 로컬 변수를 위한 연속적인 메모리 영역.
  - 여기 저장되는 값은 컴파일 시 결정되는 고정 크기를 갖습니다.
  - 매우 빠름: 메모리 할당/반환이 단지 스택 포인터의 이동만으로구현됩니다.
  - 관리가 쉬움: 함수가 호출되면 할당되고, 리턴하면 반환됩니다.
  - 스택에 있는 값들은 매우 높은 메모리 인접성을 가집니다.
- 힙: 함수 호출/리턴과 상관 없이 유지되는 값이 저장되는 곳.
  - 여기 저장되는 값은 프로그램 수행시 그 크기가 결정됩니다.
  - 스택 보다는 느림: 메모리 할당/반환시 해야 할 일이 좀 더 있습니다.
  - 메모리 인접성을 보장하지 않습니다.

예제

`String` 을 하나 만들게 되면, 스택에는 고정된 크기의 메타 데이터가생성되고, 힙에는 가변 크기의 데이터, 즉, 실제 문자열, 이 생성됩니다:

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



- 문자열 (String) 은 실제로는 Vec 입니다. 크기 (capacity) 와 현재 길이 (length) 정보를 가지며, 더 큰 크기가 필요할경우 힙에서 재 할당을 합니다.
- 힙은 기본적으로 System Allocator 를 통해 할당됩니다. 그리고 Allocator API 를이용해서 커스텀 메모리 할당자를 만들 수도 있습니다.

더살펴보기

We can inspect the memory layout with unsafe Rust. However, you should point out that this is rightfully unsafe!

```rust
fn main() {
    let mut s1 = String::from("안녕하세요");
    s1.push(' ');
    s1.push_str("world");
    // 집에서는 하지 마세요. 교육 목적으로만 사용할 수 있습니다.
    // 문자열은 레이아웃을 보장하지 않으므로
    // 정의되지 않은 동작이 발생할 수 있습니다.
    unsafe {
        let (capacity, ptr, len): (usize, usize, usize) = std::mem::transmute(s1);
        println!("capacity = {capacity}, ptr = {ptr:#x}, len = {len}");
    }
}
```

19.2 자동 메모리관리

전통적으로, 두 종류의 프로그래밍 언어가 있습니다:

- 메모리 관리가 프로그래머의 완전한 통제하에 있지만 수동 (그래서 안전하지않을 수 있는) 인 언어: C, C++, Pascal, ...
  - 프로그래머가 힙 메모리를 할당하거나 확보할 시기를 결정합니다.
  - 프로그래머는 포인터가 여전히 유효한 메모리를 가리키는지 확인해야합니다.
  - 연구 결과에 따르면 프로그래머도 실수합니다.
- 메모리 관리가 런타임에 의해 되므로 안전하지만 자동 (그래서 프로그래머가개입할 여지가 적거나 없는) 인 언어: Java, Python, Go, Haskell, ...
  - 런타임 시스템은 메모리를 더 이상 참조할 수 없을 때까지 해제되지않도록 합니다.

– 일반적으로 참조 계산, 가비지 컬렉션 또는 RAII 로 구현됩니다.

러스트는 이 둘을 혼합한 새로운 형태의 메모리 관리 기법을 제공합니다:

컴파일 시 올바른 메모리 관리를 강제함으로써 완전한 통제와 안전성 제공.

이를 가능하게 하는 러스트의 컨셉은 명시적인 소유권입니다.

이 슬라이드는 다른 언어를 사용하는 학생들이 맥락에 따라 Rust 를 사용하는데 도움을 주기 위해 작성되었습니다.

- C 는 `malloc` 및 `free` 를 사용하여 힙을 수동으로 관리해야합니다. 일반적인 오류로는 `free` 호출을 잊어버리거나, 동일한포인터에 대해 여러 번 호출하거나, 포인터가 가리키는 메모리가 해제된 후포인터를 역참조하는 것 등이 있습니다.
- C++에는 함수가 반환될 때 메모리가 해제되도록 소멸자를 호출하는 것에관한 언어 보장을 활용하는 스마트 포인터 (`unique_ptr`, `shared_ptr`) 와 같은 도구가 있습니다. 이러한 도구를 오용하여 C 와 유사한 버그를 생성하는 것은 여전히 매우 쉽습니다.
- Java, Go, Python 은 가비지 컬렉터를 사용해 더 이상 연결할 수 없는메모리를 식별하고 삭제합니다. 이렇게 하면 모든 포인터가 역참조될 수있으므로 use-after-free 및 기타 클래스의 버그를 제거할 수 있습니다. 하지만 GC 는 런타임 비용이 발생하며 제대로 조정하기가 어렵습니다.

Rust 의 소유권 및 빌림 모델은 대부분의 경우 정확히 필요한 곳에 alloc 및 free 작업을 실행하여 C 의 성능을 얻을 수 있습니다. 즉, 비용이 들지않습니다. C++의 스마트 포인터와 유사한 도구도 제공합니다. 필요한 경우참조 계산과 같은 다른 옵션을 사용할 수 있으며, 서드 파티 크레이트를사용하여 런타임 가비지 컬렉션을 지원할 수도 있습니다 (이 클래스에서는다루지 않음).

19.3 소유권

모든 변수 바인딩은 유효한 "범위 (스코프)"를 가지며, 범위 밖에서 변수사용하면 에러가 발생합니다:

```
struct Point(i32, i32);

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

We say that the variable *owns* the value. Every Rust value has precisely one owner at all times.

At the end of the scope, the variable is *dropped* and the data is freed. A destructor can run here to free up resources.

가비지 컬렉션 구현에 익숙한 학생은 가비지 컬렉터가 연결 가능한 모든메모리를 찾기 위해 ' 루트' 세트로시작한다는 사실을 알 수 있을 것입니다. Rust 의 ' 단일소유자' 원칙도 이와 유사합니다.

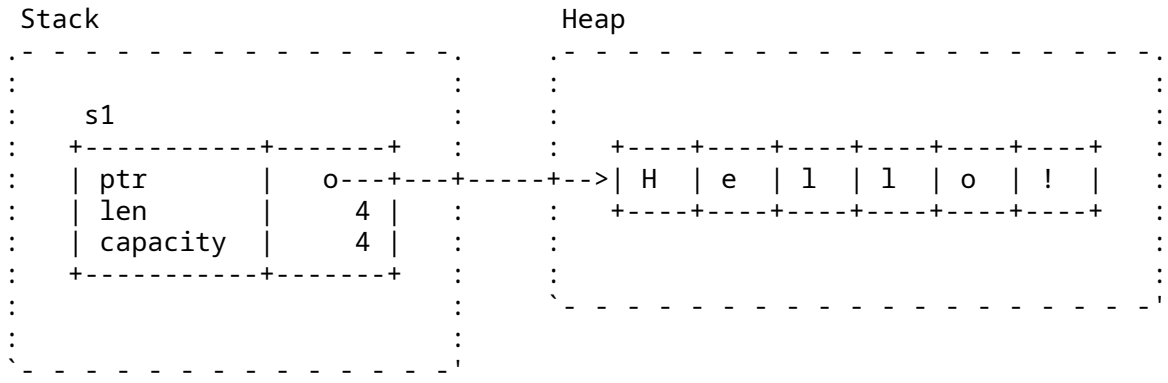
19.4 Move 문법

(변수의) 할당은 _소유권_을 변수 간에 이동시킵니다:

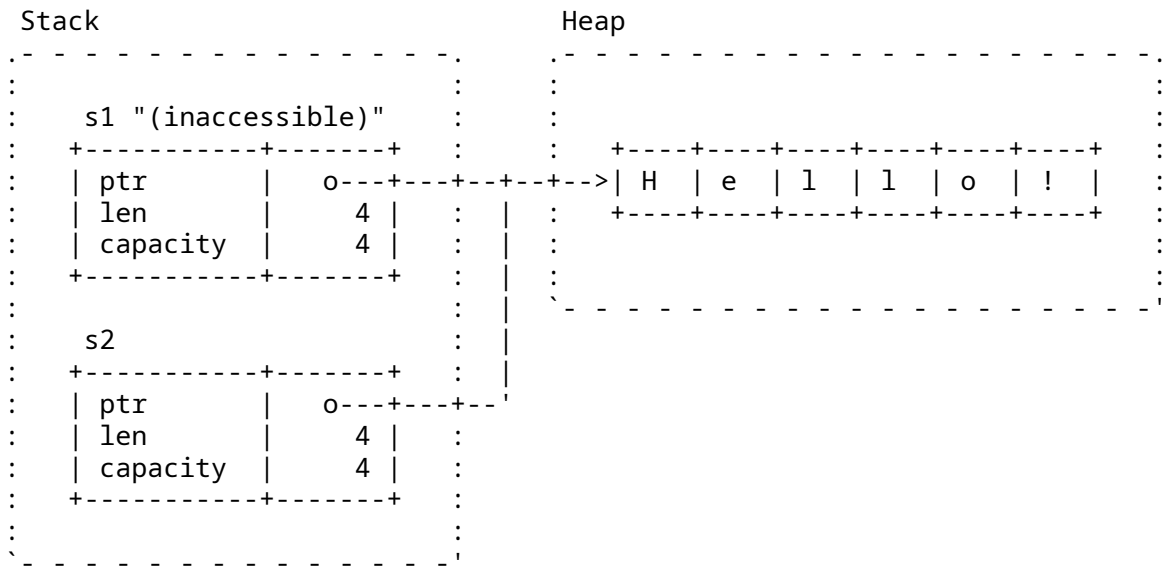
```rust
fn main() {
    let s1: String = String::from("Hello!");
    let s2: String = s1;
    println!("s2: {s2}");
    // println!("s1: {s1}");
}
```

- s1 을 s2 에 할당하여 소유권을 이전시킵니다.
- s1 의 스코프가 종료되면 아무 일도 없습니다: 왜냐하면 s1 은 아무런 소유권이 없기 때문입니다.
- s2 의 스코프가 종료되면 문자열 데이터는 해제됩니다.

s2 로 이동 전 메모리:



s2 로 이동 후 메모리:



값을 함수에 전달할때, 그 값은 매개변수에 할당됩니다. 이때 소유권의이동이 일어납니다:

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

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

```
    say_hello(name);
    // say_hello(name);
}
```

- 이는 C++과 정반대 임을 설명하세요. C++에서는 복사가 기본이고, `std::move` 를 이용해야만 (그리고 이동 생성자가 정의되어있어야만!) 소유권 이전이 됩니다.
- 실제로 이동되는 것은 소유권일 뿐입니다. 머신 코드 레벨에서 데이터복사가 일어날 지 말 지에 대한 것은 컴파일러 내부에서 일어나는 최적화문제입니다. 이런 복사는 최적화 과정에서 제거가 됩니다.
- 정수와 같은 간단한 값들은 Copy (뒤에 설명합니다) 로 마킹될 수있습니다.
- 러스트에서는 복사할때에는 명시적으로 `clone` 을 사용합니다.

`say_hello` 예:

- `say_hello` 함수의 첫번째 호출시 `main` 함수는 자신이가진 `name` 에 대한 소유권을 포기하므로, 이후 `main` 함수에서는 `name` 을 사용할 수 없습니다.
- `name` 에 할당되있는 힙 메모리는 `say_hello` 함수의끝에서 해제됩니다.
- `main` 함수에서 `name` 을 참조로전달 (빌림) 하고 (`&name`), `say_hello` 에서 매개변수를참조형으로 수정한다면 `main` 함수는 `name` 의 소유권을유지할 수 있습니다.
- 또는 첫번째 호출 시 `main` 함수에서 `name` 을 복제하여전달할 수도 있습니다.(`name.clone()`)
- 러스트는 이동을 기본으로 하고 복제를 명시적으로 선언하도록 만듬으로, 의도치 않게 복사본을 만드는 것이 C++에서보다 어렵습니다.

더살펴보기

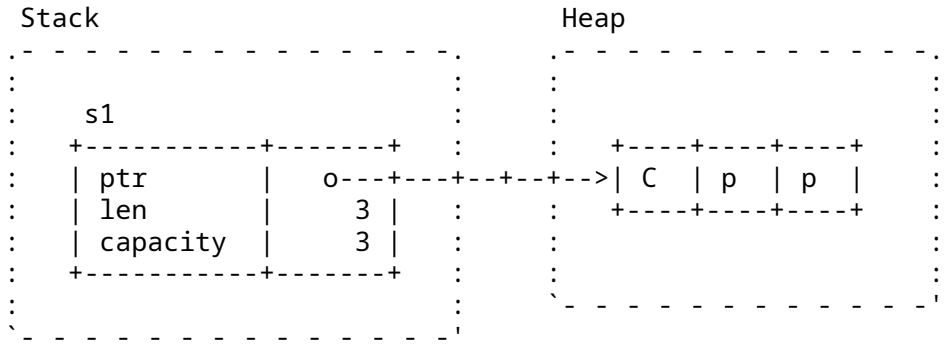
Defensive Copies in Modern C++

Modern C++은 이 문제를 다르게 해결합니다:

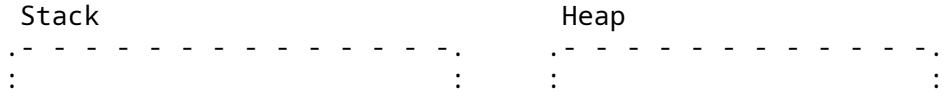
```
std::string s1 = "Cpp";
std::string s2 = s1;  // s1의 데이터를 복제합니다.
```

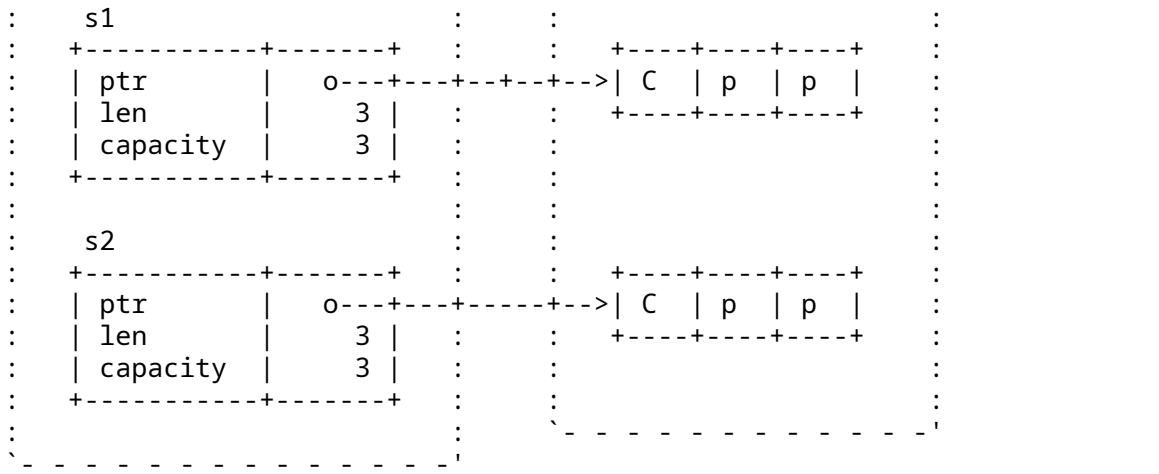
- `s1` 의 힙 데이터는 복제되고, `s2` 는 독립적인 복사본을얻습니다.
- `s1` 와 `s2` 의 스코프가 종료되면 각각의 메모리가해제됩니다.

복사 전:



복사 후:





키 포인트:

- C++는 Rust 와 약간 다른 선택을 했습니다. =는 데이터를복사하므로 문자열 데이터가 클론되어야 합니다. 그렇지 않으면 문자열 중하나가 범위를 벗어날 때 double-free 가 발생합니다.
- C++에는 값을 이동할 수 있는 시점을 나타내는 데 사용되는 `std::move` 도있습니다. 예가 `s2 = std::move(s1)` 이었다면 힙 할당이발생하지 않습니다. 이동 후에는 `s1` 이 유효하지만 지정되지 않은상태가 됩니다. Rust 와 달리 프로그래머는 `s1` 을 계속 사용할 수있습니다.
- Rust 와 달리, C++의 =는 복사되거나 이동되는 타입에 따라 결정된임의의 코드를 실행할 수 있습니다.

19.5 Clone

값을 하는 경우도 있습니다. `Clone` 트레잇을사용하면 됩니다.

```
#[derive(Default)]
struct Backends {
    hostnames: Vec<String>,
    weights: Vec<f64>,
}

impl Backends {
    fn set_hostnames(&mut self, hostnames: &Vec<String>) {
        self.hostnames = hostnames.clone();
        self.weights = hostnames.iter().map(|_| 1.0).collect();
    }
}
```

`Clone` 의 개념은 힙 할당이 발생하는 위치를 쉽게 알아내는것입니다. `.clone()` 및 `Vec::new` 또는 `Box::new` 와 같은 다른 코드도 찾아봅니다.

빌림 검사기로 문제 해결 방법을' 클론' 하고 나중에 다시 방문하여 해당클론을 최적화하려고 시도하는 경우가 많습니다.

19.6 복합타입

이동이 기본 설정이지만, 특정 타입은 복사됩니다:

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

이러한 타입들은 Copy 트레잇을 구현합니다.

직접 만든 타입들도 Copy 트레잇을 구현하여 복사를 할 수 있습니다:

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

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

- 할당 후, p1 와 p2 는 자신의 데이터를 소유합니다.
- 명시적으로 p1.clone() 를 사용하여 데이터를 복사할 수 있습니다.

복사 (copy) 와 복제 (clone) 는 같지 않습니다:

- 복사는 메모리의 내용을 그대로 한 벌 더 만드는 것을 의미하며, 아무객체에서나 다 지원하지는 않습니다.
- 복사는 커스터마이즈 할 수 없습니다. (C++에서 복사 생성자를 통해 복사동작을 임의로 구현할 수 있는 것과 비교가 됩니다.)
- 복제는 보다 일반적인 작업이며, Clone 트레잇을 구현하여 복제시동작을 커스터마이즈 할 수 있습니다.
- Drop 트레잇을 구현한 타입은 복사되지 않습니다.

위의 예시에서 다음을 시도해 보시기 바랍니다:

- Point 구조체에 String 필드를 추가하세요. 컴파일 되지않을 것입니다. 왜냐하면 String 은 Copy 트레잇을구현하고 있지 않기 때문입니다.
- Remove Copy from the derive attribute. The compiler error is now in the println! for p1.
- p1 을 복제하면 잘 동작함을 확인해 보세요.

19.7 Drop 트레잇

Drop 트레잇을구현하면, 그 값이 스코프 밖으로 나갈 때 실행될 코드를 작성할 수있습니다:

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

impl Drop for Droppable {
```

```
    fn drop(&mut self) {
        println!("{} 삭제 중", self.name);
    }
}

fn main() {
    let a = Droppable { name: "a" };
    {
        let b = Droppable { name: "b" };
        {
            let c = Droppable { name: "c" };
            let d = Droppable { name: "d" };
            println!("B 블록에서 나가기");
        }
        println!("A 블록에서 나가기");
    }
    drop(a);
    println!("main 에서 나가기");
}
```

- `std::mem::drop` 은 `std::ops::Drop::drop` 과 같지않습니다.
- 값이 범위를 벗어나면 자동으로 삭제됩니다.
- 값이 삭제될 때 `std::ops::Drop` 을 구현하면 `Drop::drop` 구현이 호출됩니다.
- 그러면 `Drop` 구현 여부와 관계없이 해당 필드도 모두 삭제됩니다.
- `std::mem::drop` 은 값을 사용하는 빈 함수입니다. 중요한 점은값의 소유권을 가지므로 범위 끝에서 삭제된다는 점입니다. 따라서 범위를벗어날 때보다 빨리 값을 명시적으로 삭제할 수 있는 편리한 방법입니다.
  - 이는 `drop` 에서 잠금 해제, 파일 닫기 등의 작업을 실행하는객체에 유용할 수 있습니다.

논의점:

- `Drop::drop` 은 왜 `self` 를 인자로 받지 않습니까?
  - 짧은 대답: 만약 그렇게 된다면 `std::mem::drop` 이 블록의끝에서 호출되고, 다시 `Drop::drop` 을 호출하게 되어, 스택오버플로가 발생합니다!
- `drop(a)` 를 `a.drop()` 로 변경해 보시기 바랍니다.

19.8 연습문제: 빌드타입

이 예에서는 모든 데이터를 소유하는 복잡한 데이터 타입을 구현합니다. 편의함수를 사용하여 새로운 값을 하나씩 빌드하도록 지원하기 위해' 빌더 패턴' 을 사용합니다.

누락된 부분을 채우세요.

```
#[derive(Debug)]
enum Language {
    Rust,
    Java,
    Perl,
}

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

```
    version_expression: String,
}

/// 소프트웨어 패키지를 나타냅니다.
#[derive(Debug)]
struct Package {
    name: String,
    version: String,
    authors: Vec<String>,
    dependencies: Vec<Dependency>,
    language: Option<Language>,
}

impl Package {
    /// Return a representation of this package as a dependency, for use in
    /// building other packages.
    fn as_dependency(&self) -> Dependency {
        todo!("1")
    }
}

/// 패키지용 빌더입니다. `build()`를 사용하여 `Package` 자체를 만듭니다.
struct PackageBuilder(Package);

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

    /// 패키지 버전을 설정합니다.
    fn version(mut self, version: impl Into<String>) -> Self {
        self.0.version = version.into();
        self
    }

    /// 패키지 작성자를 설정합니다.
    fn authors(mut self, authors: Vec<String>) -> Self {
        todo!("3")
    }

    /// 종속 항목을 추가합니다.
    fn dependency(mut self, dependency: Dependency) -> Self {
        todo!("4")
    }

    /// 언어를 설정합니다. 설정하지 않으면 언어가 기본적으로 None 으로 설정됩니다.
    fn language(mut self, language: Language) -> Self {
        todo!("5")
    }

    fn build(self) -> Package {
```

```
        self.0
    }
}

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

19.8.1 해답

```
#[derive(Debug)]
enum Language {
    Rust,
    Java,
    Perl,
}

#[derive(Clone, Debug)]
struct Dependency {
    name: String,
    version_expression: String,
}

/// 소프트웨어 패키지를 나타냅니다.
#[derive(Debug)]
struct Package {
    name: String,
    version: String,
    authors: Vec<String>,
    dependencies: Vec<Dependency>,
    language: Option<Language>,
}

impl Package {
    /// Return a representation of this package as a dependency, for use in
    /// building other packages.
    fn as_dependency(&self) -> Dependency {
        Dependency {
            name: self.name.clone(),
            version_expression: self.version.clone(),
```

```
        }
    }
}

/// 패키지용 빌더입니다. `build()`를 사용하여 `Package` 자체를 만듭니다.
struct PackageBuilder(Package);

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

    /// 패키지 버전을 설정합니다.
    fn version(mut self, version: impl Into<String>) -> Self {
        self.0.version = version.into();
        self
    }

    /// 패키지 작성자를 설정합니다.
    fn authors(mut self, authors: Vec<String>) -> Self {
        self.0.authors = authors;
        self
    }

    /// 종속 항목을 추가합니다.
    fn dependency(mut self, dependency: Dependency) -> Self {
        self.0.dependencies.push(dependency);
        self
    }

    /// 언어를 설정합니다. 설정하지 않으면 언어가 기본적으로 None 으로 설정됩니다.
    fn language(mut self, language: Language) -> Self {
        self.0.language = Some(language);
        self
    }

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

fn main() {
    let base64 = PackageBuilder::new("base64").version("0.13").build();
    println!("base64: {base64:?}");
    let log =
```

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

제 20 장

스마트포인터

This segment should take about 55 minutes. It contains:

Slide	Duration
Box	10 minutes
Rc	5 minutes
트레잇 객체	10 minutes
연습문제: 바이너리 트리	30 minutes

20.1 Box<T>

Box 는힙 데이터에 대한 소유 포인터입니다:

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

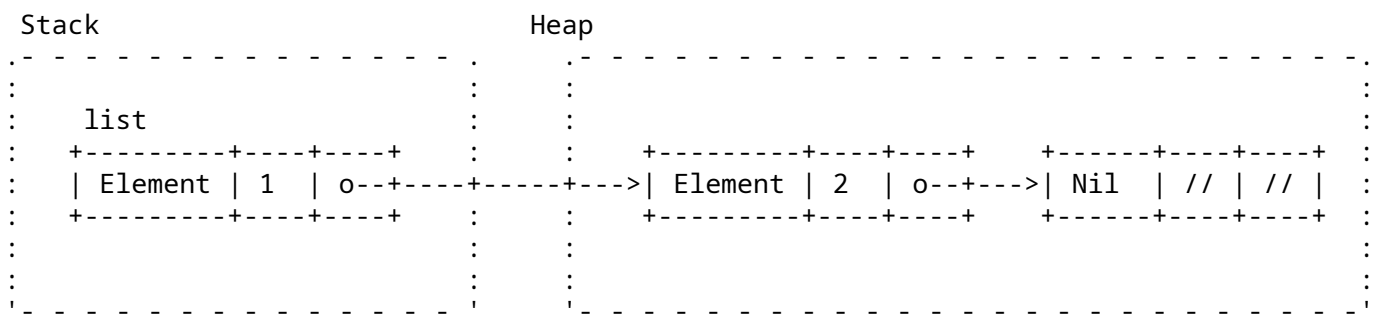
```
 Stack                     Heap
.- - - - - - -.     .- - - - - - -.
:             :     :             :
:    five     :     :             :
:   +-----+   :     :   +-----+   :
:   | o---|---+-----+-->|  5  |   :
:   +-----+   :     :   +-----+   :
:             :     :             :
:             :     :             :
`- - - - - - -'     `- - - - - - -'
```

Box<T>은 Deref<Target = T>를구현합니다. 이는 Box<T>에서 T 메서드를 직접 호출 할 수 있다는 의미입니다.

재귀 데이터나 동적크기의 데이터 타입은 Box 타입을 사용해야합니다:

```
#[derive(Debug)]
enum List<T> {
```

```
    /// A non-empty list: first element and the rest of the list.
    Element(T, Box<List<T>>),
    /// An empty list.
    Nil,
}

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



- Box is like `std::unique_ptr` in C++, except that it's guaranteed to be not null.
- Box 는 아래의 경우에 유용합니다:
  - 타입 크기를 컴파일 시점에 알 수 없는 경우.
  - 아주 큰 데이터의 소유권을 전달하고 싶은 경우. 스택에 있는 큰데이터를 복사하는 대신 Box 를 이용하여 데이터는 힙에저장하고 포인터만 이동하면 됩니다.
- If Box was not used and we attempted to embed a `List` directly into the `List`, the compiler would not be able to compute a fixed size for the struct in memory (the `List` would be of infinite size).
- Box 는 일반 포인터와 크기가 같기 때문에 크기를 계산하는 데문제가 없습니다. 다만 힙에 위치한 `List` 의 다음 요소를 가리킬뿐입니다.
- Remove the Box in the List definition and show the compiler error. We get the message ”recursive without indirection”, because for data recursion, we have to use indirection, a Box or reference of some kind, instead of storing the value directly.

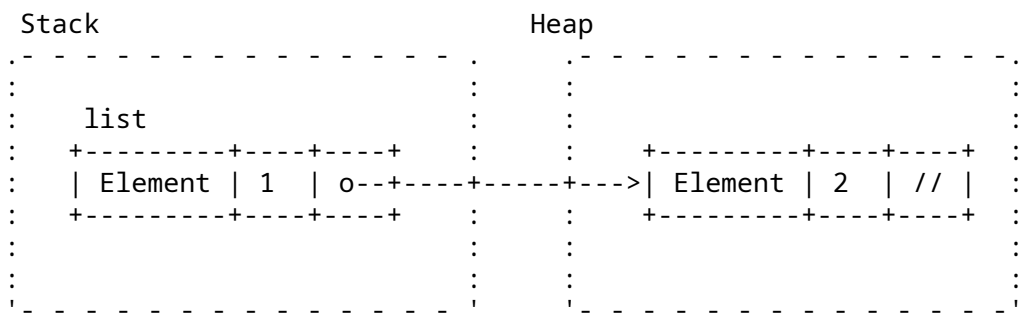
더살펴보기

니치 (틈새) 최적화 (Niche Optimization)

```
#[derive(Debug)]
enum List<T> {
    Element(T, Box<List<T>>),
    Nil,
}

fn main() {
    let list: List<i32> =
```

```
        List::Element(1, Box::new(List::Element(2, Box::new(List::Nil))));
    println!("{list:?}");
}
```

Box 는 비어있을 수 없습니다. 따라서 포인터는 항상 유효하며 null 이 아닙니다. 이는 컴파일러가 메모리 레이아웃을 최적화 할 수있게 해줍니다:



20.2 Rc

Rc 는참조 카운팅 공유 포인터입니다. 여러 위치에서 동일한 데이터를 참조해야할경우 사용합니다:

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

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

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

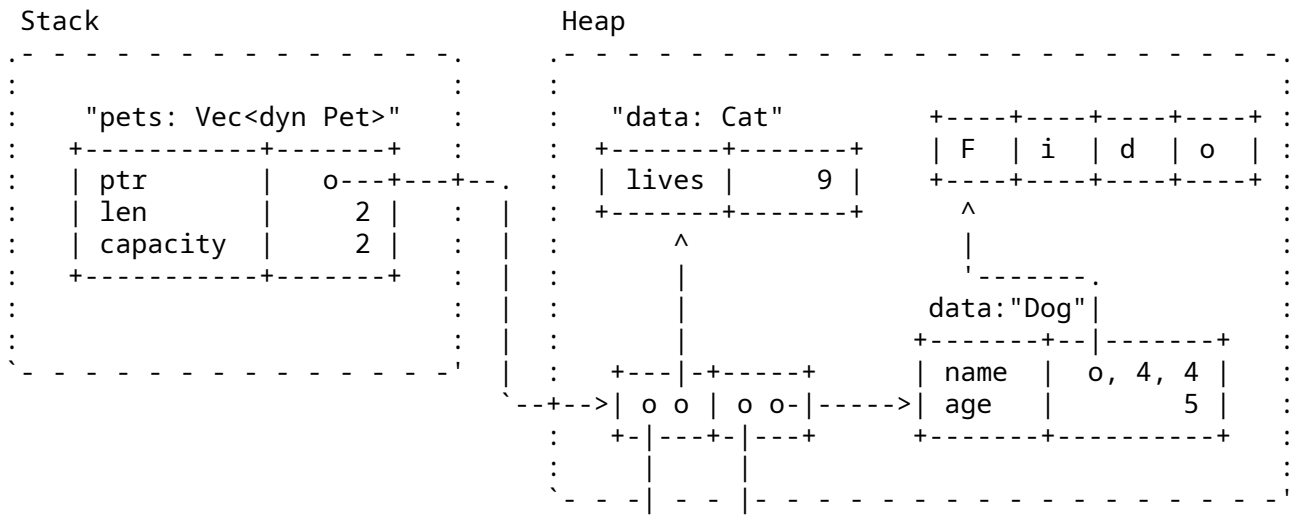
- 멀티스레드 환경에서 작업하는 경우 Arc 와 Mutex 를참조하세요.
- drop 가능한 순환 구조를 만들기 위해 공유 포인터를 Weak 포인터로 _다운그레이드_할 수도 있습니다.
- Rc 는 참조 카운트를 통해 참조가 있는 동안은 Rc 가가리키고 있는 값이 메모리에서 해제되지 않음을 보장합니다.
- C++의 std::shared_ptr 와 유사합니다.
- clone 은 비용이 거의 들지 않습니다. 같은 곳을 가리키는포인터를 하나 더 만들고, 참조 카운트를 늘립니다. 포인터가 가리키는 값자체가 복제 (깊은 복제) 되지는 않으며, 그래서 코드에서 성능 문제가있는지 검토할 때 일반적으로 Rc 를 clone 하는 것은무시할 수 있습니다.
- make_mut 는 실제로 필요한 경우에 내부 값을복제하고 ("clone-on-write") 가변 참조를 반환합니다.
- 참조 카운트를 확인하려면 Rc::strong_count 를 사용하세요.
- Rc::downgrade gives you a *weakly reference-counted* object to create cycles that will be dropped properly (likely in combination with RefCell).

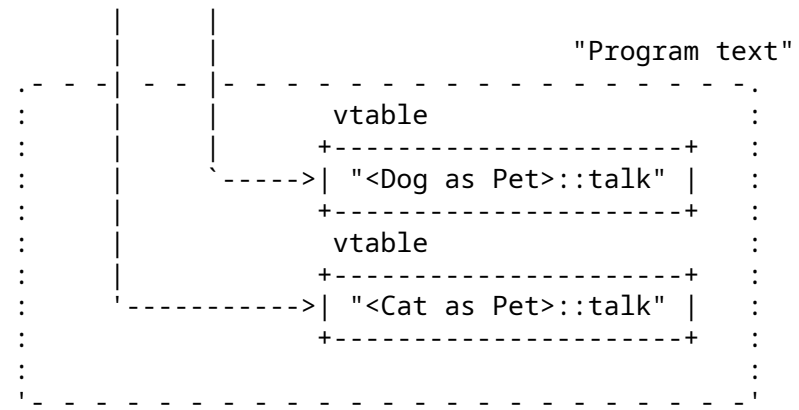
20.3 트레잇객체

트레잇 객체는 타입이 다른 값 (예를 들어 컬렉션에 속한 각 값) 들을 가질 수있습니다:

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

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

impl Pet for Dog {
    fn talk(&self) -> String {
        format!("멍멍, 제 이름은 {}입니다.", self.name)
    }
}

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

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

pets 를 할당한 이후의 메모리 레이아웃:





- Types that implement a given trait may be of different sizes. This makes it impossible to have things like `Vec<dyn Pet>` in the example above.
- `dyn Pet` 이라고 하면이 타입의 크기는 동적이며 `Pet` 을구현하고 있다고 컴파일러에게 알려주는 것입니다.
- 이 예에서는 `pets` 가 스택에 할당되고 벡터 데이터는 힙에있습니다. 두 벡터 요소는 _fat 포인터_ 입니다.
  - A fat pointer is a double-width pointer. It has two components: a pointer to the actual object and a pointer to the virtual method table (vtable) for the `Pet` implementation of that particular object.
  - 이름이 Fido 인 `Dog` 의 데이터는 `name` 및 `age` 필드입니다. `Cat` 에는 `lives` 필드가 있습니다.
- 아래 코드의 결과와 비교해보세요:

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

20.4 연습문제: 바이너리트리

바이너리 트리는 모든 노드에 두 개의 하위 요소 (왼쪽과 오른쪽) 가 있는 트리유형 데이터 구조입니다. 각 노드가 값을 저장하는 트리를 만들겠습니다. 주어진 노드 N 의 경우 N 의 왼쪽 하위 트리에 있는 모든 노드는 더 작은 값을포함하고, N 의 오른쪽 하위 트리에 있는 모든 노드는 더 큰 값을 포함합니다.

지정된 테스트가 통과하도록 다음 타입을 구현합니다.

추가 크레딧: 값을 순서대로 반환하는 바이너리 트리에 대한 반복자를구현합니다.

```
/// A node in the binary tree.
#[derive(Debug)]
struct Node<T: Ord> {
    value: T,
    left: Subtree<T>,
    right: Subtree<T>,
}

/// A possibly-empty subtree.
#[derive(Debug)]
struct Subtree<T: Ord>(Option<Box<Node<T>>>);
```

```rust
/// 바이너리 트리를 사용하여 값 집합을 저장하는 컨테이너입니다.
///
/// 동일한 값이 여러 번 추가되면 한 번만 저장됩니다.
#[derive(Debug)]
pub struct BinaryTree<T: Ord> {
    root: Subtree<T>,
}

// Implement `new`, `insert`, `len`, and `has`.

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

    #[test]
    fn len() {
        let mut tree = BinaryTree::new();
        assert_eq!(tree.len(), 0);
        tree.insert(2);
        assert_eq!(tree.len(), 1);
        tree.insert(1);
        assert_eq!(tree.len(), 2);
        tree.insert(2); // 고유 항목이 아닙니다.
        assert_eq!(tree.len(), 2);
    }

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

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

    #[test]
    fn unbalanced() {
        let mut tree = BinaryTree::new();
        for i in 0..100 {
```

```
            tree.insert(i);
        }
        assert_eq!(tree.len(), 100);
        assert!(tree.has(&50));
    }
}
```

20.4.1 해답

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

/// A node in the binary tree.
#[derive(Debug)]
struct Node<T: Ord> {
    value: T,
    left: Subtree<T>,
    right: Subtree<T>,
}

/// A possibly-empty subtree.
#[derive(Debug)]
struct Subtree<T: Ord>(Option<Box<Node<T>>>);

/// 바이너리 트리를 사용하여 값 집합을 저장하는 컨테이너입니다.
///
/// 동일한 값이 여러 번 추가되면 한 번만 저장됩니다.
#[derive(Debug)]
pub struct BinaryTree<T: Ord> {
    root: Subtree<T>,
}

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

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

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

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

impl<T: Ord> Subtree<T> {
    fn new() -> Self {
```

```
        Self(None)
    }

    fn insert(&mut self, value: T) {
        match &mut self.0 {
            None => self.0 = Some(Box::new(Node::new(value))),
            Some(n) => match value.cmp(&n.value) {
                Ordering::Less => n.left.insert(value),
                Ordering::Equal => {}
                Ordering::Greater => n.right.insert(value),
            },
        }
    }

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

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

impl<T: Ord> Node<T> {
    fn new(value: T) -> Self {
        Self { value, left: Subtree::new(), right: Subtree::new() }
    }
}

fn main() {
    let mut tree = BinaryTree::new();
    tree.insert("foo");
    assert_eq!(tree.len(), 1);
    tree.insert("bar");
    assert!(tree.has(&"foo"));
}

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

```
    #[test]
    fn len() {
        let mut tree = BinaryTree::new();
        assert_eq!(tree.len(), 0);
        tree.insert(2);
        assert_eq!(tree.len(), 1);
        tree.insert(1);
        assert_eq!(tree.len(), 2);
        tree.insert(2); // 고유 항목이 아닙니다.
        assert_eq!(tree.len(), 2);
    }

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

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

    #[test]
    fn unbalanced() {
        let mut tree = BinaryTree::new();
        for i in 0..100 {
            tree.insert(i);
        }
        assert_eq!(tree.len(), 100);
        assert!(tree.has(&50));
    }
}
```

제 VI 편

3일차 오후

제 21 장

Welcome Back

Including 10 minute breaks, this session should take about 2 hours and 10 minutes. It contains:

Segment	Duration
빌림	50 minutes
수명	1 hour and 10 minutes

제 22 장

빌림

This segment should take about 50 minutes. It contains:

Slide	Duration
빌림	10 minutes
빌림	10 minutes
상호운용성	10 minutes
연습문제: 엘리베이터 이벤트	20 minutes

22.1 빌림

As we saw before, instead of transferring ownership when calling a function, you can let a function *borrow* the value:

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

fn add(p1: &Point, p2: &Point) -> Point {
    Point(p1.0 + p2.0, p1.1 + p2.1)
}

fn main() {
    let p1 = Point(3, 4);
    let p2 = Point(10, 20);
    let p3 = add(&p1, &p2);
    println!("{p1:?} + {p2:?} = {p3:?}");
}
```

- `add` 함수는 두 `Point` 객체 값을 _빌려_와서 새로운 `Point` 객체를 반환합니다.
- `p1` 과 `p2` 의 소유권은 여전히호출자 (`main` 함수) 에 있습니다.

이 슬라이드는 1 일 차의 참조를 다룬 자료를 검토하는 것으로, 약간 확장되어함수 인수와 반환 값을 포함합니다.

더살펴보기

스택에 할당된 값을 리턴하는 것에 대한 참고:

- Demonstrate that the return from add is cheap because the compiler can eliminate the copy operation. Change the above code to print stack addresses and run it on the Playground or look at the assembly in Godbolt. In the ”DEBUG” optimization level, the addresses should change, while they stay the same when changing to the ”RELEASE” setting:

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

fn add(p1: &Point, p2: &Point) -> Point {
    let p = Point(p1.0 + p2.0, p1.1 + p2.1);
    println!("&p.0: {:p}", &p.0);
    p
}

pub fn main() {
    let p1 = Point(3, 4);
    let p2 = Point(10, 20);
    let p3 = add(&p1, &p2);
    println!("&p3.0: {:p}", &p3.0);
    println!("{p1:?} + {p2:?} = {p3:?}");
}
```

- 러스트 컴파일러는 반환값 최적화 (RVO) 를 수행할 수 있습니다.
- C++에서 copy elision 은 생성자의 부수효과 가능성이 있어 언어레벨의정의가 필요하지만 러스트에서는 문제가 되지 않습니다. 만약 RVO 가발생하지 않으면 러스트는 항상 간단하고 효율적인 `memcpy` 복사를수행할 것입니다.

22.2 빌림

Rust's *borrow checker* puts constraints on the ways you can borrow values. For a given value, at any time:

- You can have one or more shared references to the value, *or*
- You can have exactly one exclusive reference to the value.

```
fn main() {
    let mut a: i32 = 10;
    let b: &i32 = &a;

    {
        let c: &mut i32 = &mut a;
        *c = 20;
    }

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

- 충돌하는 참조가 같은 지점에 _존재_해서는 안 됩니다. 참조가역참조되는 위치는 중요하지 않습니다.
- 위 코드 컴파일 되지 않습니다. 왜냐하면 c 는 a 를 가변변수로 빌렸고, 이와 동시에 b 는 a 를 불변 변수로빌렸기 때문입니다.
- b 에 대한 `println!` 구분을 c 가 있는 스코프앞으로 이동하면 컴파일이 됩니다.
- 이렇게 바꾸면, 컴파일러는 c 가 a 를 가변 변수로 빌리기전에만 b 가 사용된다는 것을 확인할 수 있습니다. 빌림 검사기의이러한 기능을 ”non-lexical lifetime” 이라고 합니다.
- The exclusive reference constraint is quite strong. Rust uses it to ensure that data races do not occur. Rust also *relies* on this constraint to optimize code. For example, a value behind a shared reference can be safely cached in a register for the lifetime of that reference.
- 빌림 검사기는 구조체의 여러 필드에 대한 배타적 참조를 동시에 가져오는등 여러 일반적인 패턴을 수용하도록 설계되었습니다. 하지만 제대로”인식”하지 못해 ”빌림 검사기와의 충돌”이 발생하는 경우도 있습니다.

22.3 상호운용성

In some situations, it's necessary to modify data behind a shared (read-only) reference. For example, a shared data structure might have an internal cache, and wish to update that cache from read-only methods.

The ”interior mutability” pattern allows exclusive (mutable) access behind a shared reference. The standard library provides several ways to do this, all while still ensuring safety, typically by performing a runtime check.

RefCell

```
use std::cell::RefCell;
use std::rc::Rc;

#[derive(Debug, Default)]
struct Node {
    value: i64,
    children: Vec<Rc<RefCell<Node>>>,
}

impl Node {
    fn new(value: i64) -> Rc<RefCell<Node>> {
        Rc::new(RefCell::new(Node { value, ..Node::default() }))
    }

    fn sum(&self) -> i64 {
        self.value + self.children.iter().map(|c| c.borrow().sum()).sum::<i64>()
    }
}

fn main() {
    let root = Node::new(1);
    root.borrow_mut().children.push(Node::new(5));
    let subtree = Node::new(10);
```

```
    subtree.borrow_mut().children.push(Node::new(11));
    subtree.borrow_mut().children.push(Node::new(12));
    root.borrow_mut().children.push(subtree);

    println!("그래프: {root:#?}");
    println!("그래프 합계: {}", root.borrow().sum());
}
```

`Cell`

`Cell` wraps a value and allows getting or setting the value, even with a shared reference to the `Cell`. However, it does not allow any references to the value. Since there are no references, borrowing rules cannot be broken.

The main thing to take away from this slide is that Rust provides *safe* ways to modify data behind a shared reference. There are a variety of ways to ensure that safety, and `RefCell` and `Cell` are two of them.

- `RefCell` enforces Rust's usual borrowing rules (either multiple shared references or a single exclusive reference) with a runtime check. In this case, all borrows are very short and never overlap, so the checks always succeed.
- `Rc` only allows shared (read-only) access to its contents, since its purpose is to allow (and count) many references. But we want to modify the value, so we need interior mutability.
- `Cell` is a simpler means to ensure safety: it has a `set` method that takes `&self`. This needs no runtime check, but requires moving values, which can have its own cost.
- Demonstrate that reference loops can be created by adding `root` to `subtree.children`.
- `self.value` 를 증가시키는메서드인 `fn inc(&mut self)` 를 추가하고 그 메서드를자식노드에서 호출하세요. 그러면 `thread 'main' panicked at 'already borrowed: BorrowMutError'` 런타임 패닉이 발생함을 보이세요.

22.4 연습문제: 엘리베이터이벤트

당신은 건강 상태를 모니터링하는 시스템을 구현하는 일을 하고 있습니다. 그일환으로 당신은 사용자의 건강 상태 통계를 추적해야합니다.

You'll start with a stubbed function in an `impl` block as well as a `User` struct definition. Your goal is to implement the stubbed out method on the `User` `struct` defined in the `impl` block.

Copy the code below to https://play.rust-lang.org/ and fill in the missing method:

```
// TODO: 구현이 완료되면 이를 삭제합니다.
#![allow(unused_variables, dead_code)]


#![allow(dead_code)]
pub struct User {
    name: String,
    age: u32,
    height: f32,
    visit_count: usize,
```

```
    last_blood_pressure: Option<(u32, u32)>,
}

pub struct Measurements {
    height: f32,
    blood_pressure: (u32, u32),
}

pub struct HealthReport<'a> {
    patient_name: &'a str,
    visit_count: u32,
    height_change: f32,
    blood_pressure_change: Option<(i32, i32)>,
}

impl User {
    pub fn new(name: String, age: u32, height: f32) -> Self {
        Self { name, age, height, visit_count: 0, last_blood_pressure: None }
    }

    pub fn visit_doctor(&mut self, measurements: Measurements) -> HealthReport {
        todo!("병원 방문에 따른 측정값을 기반으로 사용자 통계를 업데이트합니다.")
    }
}

fn main() {
    let bob = User::new(String::from("Bob"), 32, 155.2);
    println!("저는 {}이고, 나이는 {}세입니다.", bob.name, bob.age);
}

#[test]
fn test_visit() {
    let mut bob = User::new(String::from("Bob"), 32, 155.2);
    assert_eq!(bob.visit_count, 0);
    let report =
        bob.visit_doctor(Measurements { height: 156.1, blood_pressure: (120, 80) });
    assert_eq!(report.patient_name, "Bob");
    assert_eq!(report.visit_count, 1);
    assert_eq!(report.blood_pressure_change, None);

    let report =
        bob.visit_doctor(Measurements { height: 156.1, blood_pressure: (115, 76) });

    assert_eq!(report.visit_count, 2);
    assert_eq!(report.blood_pressure_change, Some((-5, -4)));
}
```

22.4.1 해답

```
#![allow(dead_code)]
```

```rust
pub struct User {
    name: String,
    age: u32,
    height: f32,
    visit_count: usize,
    last_blood_pressure: Option<(u32, u32)>,
}

pub struct Measurements {
    height: f32,
    blood_pressure: (u32, u32),
}

pub struct HealthReport<'a> {
    patient_name: &'a str,
    visit_count: u32,
    height_change: f32,
    blood_pressure_change: Option<(i32, i32)>,
}

impl User {
    pub fn new(name: String, age: u32, height: f32) -> Self {
        Self { name, age, height, visit_count: 0, last_blood_pressure: None }
    }

    pub fn visit_doctor(&mut self, measurements: Measurements) -> HealthReport {
        self.visit_count += 1;
        let bp = measurements.blood_pressure;
        let report = HealthReport {
            patient_name: &self.name,
            visit_count: self.visit_count as u32,
            height_change: measurements.height - self.height,
            blood_pressure_change: match self.last_blood_pressure {
                Some(lbp) => {
                    Some((bp.0 as i32 - lbp.0 as i32, bp.1 as i32 - lbp.1 as i32))
                }
                None => None,
            },
        };
        self.height = measurements.height;
        self.last_blood_pressure = Some(bp);
        report
    }
}

fn main() {
    let bob = User::new(String::from("Bob"), 32, 155.2);
    println!("저는 {}이고, 나이는 {}세입니다.", bob.name, bob.age);
}

#[test]
```

```
fn test_visit() {
    let mut bob = User::new(String::from("Bob"), 32, 155.2);
    assert_eq!(bob.visit_count, 0);
    let report =
        bob.visit_doctor(Measurements { height: 156.1, blood_pressure: (120, 80) });
    assert_eq!(report.patient_name, "Bob");
    assert_eq!(report.visit_count, 1);
    assert_eq!(report.blood_pressure_change, None);

    let report =
        bob.visit_doctor(Measurements { height: 156.1, blood_pressure: (115, 76) });

    assert_eq!(report.visit_count, 2);
    assert_eq!(report.blood_pressure_change, Some((-5, -4)));
}
```

제 23 장

수명

This segment should take about 1 hour and 10 minutes. It contains:

Slide	Duration
Slices: &[T]	10 minutes
허상 (dangling) 참조	10 minutes
함수 호출에서의 수명	10 minutes
수명	5 minutes
수명	5 minutes
연습문제: Protobuf 파싱	30 minutes

23.1 슬라이스

슬라이스는 큰 컬랙션의 일부 (혹은 전체) 를 보여주는 뷰 (view) 입니다:

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

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

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

- 슬라이스는 다른 (슬라이스 된) 타입으로부터 데이터를' 빌려' 옵니다.
- 질문: `s` 를 출력하기 전에 `a[3]` 을 수정하면 무슨일이 있어날까요?
- 슬라이스는 우선 `a` 를 빌린다음, 시작과 끝 인덱스를브래킷 (`[]`) 안에 지정해서 만듭니다.
- 슬라이스가 인덱스 0 부터 시작한다면 시작 인덱스는 생략 가능합니다. 즉`&a[0..a.len()]` 와 `&a[..a.len()]` 는동일합니다.
- 마지막 인덱스도 생략 가능합니다. 그래서 `&a[2..a.len()]` 와 `&a[2..]` 는 동일합니다.
- 따라서 전체 배열에 대한 슬라이스는 `&a[..]` 가 됩니다.

- `s` 는 `i32` 들로 이루어진 슬라이스에 대한 참조입니다. `s` 의 타입 (`&[i32]`) 에 배열의 크기가 빠져 있음에주목하시기 바랍니다. 즉, 슬라이스를 이용하면 다양한 길이의 데이터를다룰 수 있습니다.
- Slices always borrow from another object. In this example, `a` has to remain 'alive' (in scope) for at least as long as our slice.
- The question about modifying `a[3]` can spark an interesting discussion, but the answer is that for memory safety reasons you cannot do it through `a` at this point in the execution, but you can read the data from both `a` and `s` safely. It works before you created the slice, and again after the `println`, when the slice is no longer used.

23.2 허상 (dangling) 참조

이제 Rust 의 두 가지 문자열 타입을 이해할 수 있습니다. `&str` 은거의 `&[char]` 와 비슷하지만 데이터가 가변 길이인코딩 (UTF-8) 으로 저장됩니다.

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

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

    let s3: &str = &s2[6..];
    println!("s3: {s3}");
}
```

러스트 용어:

- `&str` 은 문자열 슬라이스에 대한 (불변) 참조입니다.
- `String` 은 문자열을 담을 수 있는 버퍼입니다.
- `&str` 은 문자열 슬라이스입니다. 문자열 슬라이스는 UTF-8 로인코딩된 문자열 데이터를 의미합니다. 문자열 리터럴 (`"Hello"`) 은프로그램 바이너리에 저장됩니다.
- 러스트의 `String` 타입은 실제로는 문자열을 이루는 바이트에 대한백터 (`Vec<u8>`) 입니다. `Vec<T>`가 `T` 를 소유하고 있듯이, `String` 이 가리키고 있는 문자열은 `String` 의 소유입니다.
- As with many other types `String::from()` creates a string from a string literal; `String::new()` creates a new empty string, to which string data can be added using the `push()` and `push_str()` methods.
- The `format!()` macro is a convenient way to generate an owned string from dynamic values. It accepts the same format specification as `println!()`.
- You can borrow `&str` slices from `String` via `&` and optionally range selection. If you select a byte range that is not aligned to character boundaries, the expression will panic. The `chars` iterator iterates over characters and is preferred over trying to get character boundaries right.
- For C++ programmers: think of `&str` as `std::string_view` from C++, but the one that always points to a valid string in memory. Rust `String` is a rough equivalent of `std::string` from C++ (main difference: it can only contain UTF-8 encoded bytes and will never use a small-string optimization).

- Byte strings literals allow you to create a `&[u8]` value directly:

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

23.3 함수 호출에서의수명

A reference has a *lifetime*, which must not "outlive" the value it refers to. This is verified by the borrow checker.

The lifetime can be implicit - this is what we have seen so far. Lifetimes can also be explicit: `&'a Point`, `&'document str`. Lifetimes start with `'` and `'a` is a typical default name. Read `&'a Point` as "a borrowed `Point` which is valid for at least the lifetime `a`".

Lifetimes are always inferred by the compiler: you cannot assign a lifetime yourself. Explicit lifetime annotations create constraints where there is ambiguity; the compiler verifies that there is a valid solution.

수명은 함수에 값을 전달하고 함수에서 값을 반환하는 경우를 고려할 때 더복잡해집니다.

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

fn left_most(p1: &Point, p2: &Point) -> &Point {
    if p1.0 < p2.0 {
        p1
    } else {
        p2
    }
}

fn main() {
    let p1: Point = Point(10, 10);
    let p2: Point = Point(20, 20);
    let p3 = left_most(&p1, &p2); // p3의 수명은 어떻게 되나요?
    println!("p3: {p3:?}");
}
```

In this example, the compiler does not know what lifetime to infer for `p3`. Looking inside the function body shows that it can only safely assume that `p3`'s lifetime is the shorter of `p1` and `p2`. But just like types, Rust requires explicit annotations of lifetimes on function arguments and return values.

`'a` 를 `left_most` 에 적절하게추가합니다.

```
fn left_most<'a>(p1: &'a Point, p2: &'a Point) -> &'a Point {
```

즉, "`a` 보다 오래 지속되는 `p1` 과 `p2` 가 있으면 반환 값은 최소한`'a` 동안 유지됩니다.

일반적으로 수명은 다음 슬라이드에 설명된 대로 생략될 수 있습니다.

23.4 함수 호출에서의수명

Lifetimes for function arguments and return values must be fully specified, but Rust allows lifetimes to be elided in most cases with a few simple rules. This is not inference – it is just a syntactic shorthand.

- 수명 주석이 없는 각 인수에 하나씩 제공됩니다.
- 인수 수명이 하나만 있는 경우 주석 처리되지 않은 모든 반환 값에제공됩니다.
- 인수 수명이 여러 개 있지만 첫 번째가'self' 의 수명이면 해당 전체 기간은주석 처리되지 않은 모든 반환 값에 제공됩니다.

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

fn cab_distance(p1: &Point, p2: &Point) -> i32 {
    (p1.0 - p2.0).abs() + (p1.1 - p2.1).abs()
}

fn nearest<'a>(points: &'a [Point], query: &Point) -> Option<&'a Point> {
    let mut nearest = None;
    for p in points {
        if let Some((_, nearest_dist)) = nearest {
            let dist = cab_distance(p, query);
            if dist < nearest_dist {
                nearest = Some((p, dist));
            }
        } else {
            nearest = Some((p, cab_distance(p, query)));
        };
    }
    nearest.map(|(p, _)| p)
}

fn main() {
    println!(
        "{:?}",
        nearest(
            &[Point(1, 0), Point(1, 0), Point(-1, 0), Point(0, -1),],
            &Point(0, 2)
        )
    );
}
```

이 예에서 cab_distance 는 간단히 생략됩니다.

nearest 함수는 인수에 여러 참조가 포함되어 명시적 주석이 필요한함수의 또 다른 예를 제공합니다.

반환된 수명에 관해 ' 거짓말' 하도록 서명을조정해 보세요.

```
fn nearest<'a, 'q>(points: &'a [Point], query: &'q Point) -> Option<&'q Point> {
```

This won't compile, demonstrating that the annotations are checked for validity by the compiler. Note that this is not the case for raw pointers (unsafe), and this is a common source of errors with unsafe Rust.

Students may ask when to use lifetimes. Rust borrows *always* have lifetimes. Most of the time, elision and type inference mean these don't need to be written out. In more complicated cases, lifetime annotations can help resolve ambiguity. Often, especially when prototyping, it's easier to just work with owned data by cloning values where necessary.

23.5 구조체에서의수명

어떤 타입이 빌려온 데이터를 저장하고 있다면, 반드시 수명을 표시해야합니다:

```
#[derive(Debug)]
struct Highlight<'doc>(&'doc str);

fn erase(text: String) {
    println!("안녕 {text}!");
}

fn main() {
    let text = String::from("The quick brown fox jumps over the lazy dog.");
    let fox = Highlight(&text[4..19]);
    let dog = Highlight(&text[35..43]);
    // erase(text);
    println!("{fox:?}");
    println!("{dog:?}");
}
```

- 위의 예제에서 `Highlight` 의어노테이션 (`<'doc>`) 은적어도 `Highlight` 인스턴스가 살아있는 동안에는 그 내부의`&str` 가 가리키는 데이터 역시 살아있어야 한다는 것을의미합니다.
- 만약 `text` 가 `fox` (혹은 `dog`) 의 수명이 다하기전에 `erase` 함수 호출 등으로 사라지게 된다면 빌림 검사기가에러를 발생합니다.
- 빌린 데이터를 가지고 있는 타입은 사용자로 하여금 원본 데이터를유지하도록 강제합니다. 이런 타입은 경량 뷰 (lightweight view) 를만드는데 유용하지만, 이 제약 조건 때문에 이런 타입을 사용하는 것이쉽지만은 않습니다.
- 따라서, 가능하다면, 구조체가 자신의 데이터를 직접 소유하도록 하는 것이좋습니다.
- 한 구조체안에 여러 참조가 있으면서, 이 참조들의 수명이 서로 다르게지정되는 경우도 있습니다. 이는 참조와 그 구조체 간의 관계 뿐만이아니라, 그 참조들 사이의 수명 관계를 설명해야 할 경우에 필요합니다. 매우 고급 기술입니다.

23.6 연습문제: Protobuf 파싱

이 연습에서는 protobuf 바이너리 인코딩용 파서를 빌드합니다. 생각보다 간단합니다. 이는 데이터슬라이스를 전달하는 일반적인 파싱 패턴을 보여줍니다. 기본 데이터 자체는복사되지 않습니다.

protobuf 메시지를 완전히 파싱하려면 필드 번호로 색인이 생성된 필드의타입을 알아야 합니다. 이는 일반적으로 `proto` 파일에 제공됩니다. 이 연습에서는 이러한 정보를 각 필드에 대해 호출되는 함수의 `match` 문으로 인코딩합니다.

다음 proto 를 사용합니다.

```
message PhoneNumber {
  optional string number = 1;
  optional string type = 2;
}
```

```
message Person {
  optional string name = 1;
  optional int32 id = 2;
  repeated PhoneNumber phones = 3;
}
```

proto 메시지는 일련의 필드로 차례로 인코딩됩니다. 각각은 값이 뒤에 오는' 태그' 로 구현됩니다. 태그에는 필드번호 (예: Person 메시지의 id 필드에 대한 2) 및바이트 스트림에서 페이로드를 결정하는 방법을 정의하는 와이어 타입이포함됩니다.

태그를 포함한 정수는 VARINT 라는 가변 길이 인코딩으로 표시됩니다. 다행히 parse_varint 는 아래에 정의되어 있습니다. 또한 제공된 코드는콜백을 정의하여 Person 및 PhoneNumber 필드를 처리하고메시지를 이러한 콜백에 대한 일련의 호출로 파싱합니다.

이제 parse_field 함수를 구현하고 Person 과 PhoneNumber 구조체애 대해 ProtoMessage 트레잇만구현하면 됩니다.

```
use std::convert::TryFrom;
use thiserror::Error;

#[derive(Debug, Error)]
enum Error {
    #[error("잘못된 varint")]
    InvalidVarint,
    #[error("잘못된 wire-type")]
    InvalidWireType,
    #[error("예상치 못한 EOF")]
    UnexpectedEOF,
    #[error("잘못된 길이")]
    InvalidSize(#[from] std::num::TryFromIntError),
    #[error("예상치 못한 wire-type")]
    UnexpectedWireType,
    #[error("잘못된 문자열 (UTF-8 아님)")]
    InvalidString,
}

/// 와이어에 표시된 와이어 타입입니다.
enum WireType {
    /// Varint WireType 은 값이 단일 VARINT 임을 나타냅니다.
    Varint,
    //I64,  -- not needed for this exercise
    /// The Len WireType indicates that the value is a length represented as a
    /// VARINT followed by exactly that number of bytes.
    Len,
    /// The I32 WireType indicates that the value is precisely 4 bytes in
    /// little-endian order containing a 32-bit signed integer.
    I32,
}

#[derive(Debug)]
/// 와이어 타입에 따라 타입이 지정된 필드 값입니다.
enum FieldValue<'a> {
```

```
    Varint(u64),
    //I64(i64),  -- 이 연습에서는 필요하지 않습니다.
    Len(&'a [u8]),
    I32(i32),
}

#[derive(Debug)]
/// 필드 번호 및 값을 포함하는 필드입니다.
struct Field<'a> {
    field_num: u64,
    value: FieldValue<'a>,
}

trait ProtoMessage<'a>: Default + 'a {
    fn add_field(&mut self, field: Field<'a>) -> Result<(), Error>;
}

impl TryFrom<u64> for WireType {
    type Error = Error;

    fn try_from(value: u64) -> Result<WireType, Error> {
        Ok(match value {
            0 => WireType::Varint,
            //1 => WireType::I64,  -- 이 연습에서는 필요하지 않습니다.
            2 => WireType::Len,
            5 => WireType::I32,
            _ => return Err(Error::InvalidWireType),
        })
    }
}

impl<'a> FieldValue<'a> {
    fn as_string(&self) -> Result<&'a str, Error> {
        let FieldValue::Len(data) = self else {
            return Err(Error::UnexpectedWireType);
        };
        std::str::from_utf8(data).map_err(|_| Error::InvalidString)
    }

    fn as_bytes(&self) -> Result<&'a [u8], Error> {
        let FieldValue::Len(data) = self else {
            return Err(Error::UnexpectedWireType);
        };
        Ok(data)
    }

    fn as_u64(&self) -> Result<u64, Error> {
        let FieldValue::Varint(value) = self else {
            return Err(Error::UnexpectedWireType);
        };
        Ok(*value)
```

```
    }
}

/// VARINT를 파싱하여 파싱된 값과 나머지 바이트를 반환합니다.
fn parse_varint(data: &[u8]) -> Result<(u64, &[u8]), Error> {
    for i in 0..7 {
        let Some(b) = data.get(i) else {
            return Err(Error::InvalidVarint);
        };
        if b & 0x80 == 0 {
            // 이는 VARINT의 마지막 바이트이므로
            // u64로 변환하여 반환합니다.
            let mut value = 0u64;
            for b in data[..=i].iter().rev() {
                value = (value << 7) | (b & 0x7f) as u64;
            }
            return Ok((value, &data[i + 1..]));
        }
    }

    // 7바이트를 초과하면 유효하지 않습니다.
    Err(Error::InvalidVarint)
}

/// 태그를 필드 번호와 WireType으로 변환합니다.
fn unpack_tag(tag: u64) -> Result<(u64, WireType), Error> {
    let field_num = tag >> 3;
    let wire_type = WireType::try_from(tag & 0x7)?;
    Ok((field_num, wire_type))
}


/// 필드를 파싱하여 나머지 바이트를 반환합니다.
fn parse_field(data: &[u8]) -> Result<(Field, &[u8]), Error> {
    let (tag, remainder) = parse_varint(data)?;
    let (field_num, wire_type) = unpack_tag(tag)?;
    let (fieldvalue, remainder) = match wire_type {
        _ => todo!("Based on the wire type, build a Field, consuming as many bytes as ne
    };
    todo!("Return the field, and any un-consumed bytes.")
}

/// 주어진 데이터에서 메시지를 파싱하여 메시지의 각 필드에 대해 `T::add_field`를 호출합니다.
///
/// 전체 입력이 사용됩니다.
fn parse_message<'a, T: ProtoMessage<'a>>(mut data: &'a [u8]) -> Result<T, Error> {
    let mut result = T::default();
    while !data.is_empty() {
        let parsed = parse_field(data)?;
        result.add_field(parsed.0)?;
        data = parsed.1;
```

```
    }
    Ok(result)
}

#[derive(Debug, Default)]
struct PhoneNumber<'a> {
    number: &'a str,
    type_: &'a str,
}

#[derive(Debug, Default)]
struct Person<'a> {
    name: &'a str,
    id: u64,
    phone: Vec<PhoneNumber<'a>>,
}

// TODO: Implement ProtoMessage for Person and PhoneNumber.

fn main() {
    let person: Person = parse_message(&[
        0x0a, 0x07, 0x6d, 0x61, 0x78, 0x77, 0x65, 0x6c, 0x6c, 0x10, 0x2a, 0x1a,
        0x16, 0x0a, 0x0e, 0x2b, 0x31, 0x32, 0x30, 0x32, 0x2d, 0x35, 0x35, 0x35,
        0x2d, 0x31, 0x32, 0x31, 0x32, 0x12, 0x04, 0x68, 0x6f, 0x6d, 0x65, 0x1a,
        0x18, 0x0a, 0x0e, 0x2b, 0x31, 0x38, 0x30, 0x30, 0x2d, 0x38, 0x36, 0x37,
        0x2d, 0x35, 0x33, 0x30, 0x38, 0x12, 0x06, 0x6d, 0x6f, 0x62, 0x69, 0x6c,
        0x65,
    ])
    .unwrap();
    println!("{:#?}", person);
}
```

23.6.1 해답

```
use std::convert::TryFrom;
use thiserror::Error;

#[derive(Debug, Error)]
enum Error {
    #[error("잘못된 varint")]
    InvalidVarint,
    #[error("잘못된 wire-type")]
    InvalidWireType,
    #[error("예상치 못한 EOF")]
    UnexpectedEOF,
    #[error("잘못된 길이")]
    InvalidSize(#[from] std::num::TryFromIntError),
    #[error("예상치 못한 wire-type")]
    UnexpectedWireType,
    #[error("잘못된 문자열 (UTF-8 아님)")]
    InvalidString,
```

```
}

/// 와이어에 표시된 와이어 타입입니다.
enum WireType {
    /// Varint WireType은 값이 단일 VARINT임을 나타냅니다.
    Varint,
    //I64,  -- not needed for this exercise
    /// The Len WireType indicates that the value is a length represented as a
    /// VARINT followed by exactly that number of bytes.
    Len,
    /// The I32 WireType indicates that the value is precisely 4 bytes in
    /// little-endian order containing a 32-bit signed integer.
    I32,
}

#[derive(Debug)]
/// 와이어 타입에 따라 타입이 지정된 필드 값입니다.
enum FieldValue<'a> {
    Varint(u64),
    //I64(i64),  -- 이 연습에서는 필요하지 않습니다.
    Len(&'a [u8]),
    I32(i32),
}

#[derive(Debug)]
/// 필드 번호 및 값을 포함하는 필드입니다.
struct Field<'a> {
    field_num: u64,
    value: FieldValue<'a>,
}

trait ProtoMessage<'a>: Default + 'a {
    fn add_field(&mut self, field: Field<'a>) -> Result<(), Error>;
}

impl TryFrom<u64> for WireType {
    type Error = Error;

    fn try_from(value: u64) -> Result<WireType, Error> {
        Ok(match value {
            0 => WireType::Varint,
            //1 => WireType::I64,  -- 이 연습에서는 필요하지 않습니다.
            2 => WireType::Len,
            5 => WireType::I32,
            _ => return Err(Error::InvalidWireType),
        })
    }
}

impl<'a> FieldValue<'a> {
    fn as_string(&self) -> Result<&'a str, Error> {
```

```
        let FieldValue::Len(data) = self else {
            return Err(Error::UnexpectedWireType);
        };
        std::str::from_utf8(data).map_err(|_| Error::InvalidString)
    }

    fn as_bytes(&self) -> Result<&'a [u8], Error> {
        let FieldValue::Len(data) = self else {
            return Err(Error::UnexpectedWireType);
        };
        Ok(data)
    }

    fn as_u64(&self) -> Result<u64, Error> {
        let FieldValue::Varint(value) = self else {
            return Err(Error::UnexpectedWireType);
        };
        Ok(*value)
    }
}

/// VARINT를 파싱하여 파싱된 값과 나머지 바이트를 반환합니다.
fn parse_varint(data: &[u8]) -> Result<(u64, &[u8]), Error> {
    for i in 0..7 {
        let Some(b) = data.get(i) else {
            return Err(Error::InvalidVarint);
        };
        if b & 0x80 == 0 {
            // 이는 VARINT의 마지막 바이트이므로
            // u64로 변환하여 반환합니다.
            let mut value = 0u64;
            for b in data[..=i].iter().rev() {
                value = (value << 7) | (b & 0x7f) as u64;
            }
            return Ok((value, &data[i + 1..]));
        }
    }

    // 7바이트를 초과하면 유효하지 않습니다.
    Err(Error::InvalidVarint)
}

/// 태그를 필드 번호와 WireType으로 변환합니다.
fn unpack_tag(tag: u64) -> Result<(u64, WireType), Error> {
    let field_num = tag >> 3;
    let wire_type = WireType::try_from(tag & 0x7)?;
    Ok((field_num, wire_type))
}

/// 필드를 파싱하여 나머지 바이트를 반환합니다.
fn parse_field(data: &[u8]) -> Result<(Field, &[u8]), Error> {
```

```
    let (tag, remainder) = parse_varint(data)?;
    let (field_num, wire_type) = unpack_tag(tag)?;
    let (fieldvalue, remainder) = match wire_type {
        WireType::Varint => {
            let (value, remainder) = parse_varint(remainder)?;
            (FieldValue::Varint(value), remainder)
        }
        WireType::Len => {
            let (len, remainder) = parse_varint(remainder)?;
            let len: usize = len.try_into()?;
            if remainder.len() < len {
                return Err(Error::UnexpectedEOF);
            }
            let (value, remainder) = remainder.split_at(len);
            (FieldValue::Len(value), remainder)
        }
        WireType::I32 => {
            if remainder.len() < 4 {
                return Err(Error::UnexpectedEOF);
            }
            let (value, remainder) = remainder.split_at(4);
            // `value`의 길이가 4 바이트이므로 오류를 래핑 해제합니다.
            let value = i32::from_le_bytes(value.try_into().unwrap());
            (FieldValue::I32(value), remainder)
        }
    };
    Ok((Field { field_num, value: fieldvalue }, remainder))
}

/// 주어진 데이터에서 메시지를 파싱하여 메시지의 각 필드에 대해 `T::add_field`를 호출합니다.
///
/// 전체 입력이 사용됩니다.
fn parse_message<'a, T: ProtoMessage<'a>>(mut data: &'a [u8]) -> Result<T, Error> {
    let mut result = T::default();
    while !data.is_empty() {
        let parsed = parse_field(data)?;
        result.add_field(parsed.0)?;
        data = parsed.1;
    }
    Ok(result)
}

#[derive(Debug, Default)]
struct PhoneNumber<'a> {
    number: &'a str,
    type_: &'a str,
}

#[derive(Debug, Default)]
struct Person<'a> {
    name: &'a str,
```

```
    id: u64,
    phone: Vec<PhoneNumber<'a>>,
}

impl<'a> ProtoMessage<'a> for Person<'a> {
    fn add_field(&mut self, field: Field<'a>) -> Result<(), Error> {
        match field.field_num {
            1 => self.name = field.value.as_string()?,
            2 => self.id = field.value.as_u64()?,
            3 => self.phone.push(parse_message(field.value.as_bytes()?)?),
            _ => {} // 나머지는 모두 건너뜁니다.
        }
        Ok(())
    }
}

impl<'a> ProtoMessage<'a> for PhoneNumber<'a> {
    fn add_field(&mut self, field: Field<'a>) -> Result<(), Error> {
        match field.field_num {
            1 => self.number = field.value.as_string()?,
            2 => self.type_ = field.value.as_string()?,
            _ => {} // 나머지는 모두 건너뜁니다.
        }
        Ok(())
    }
}

fn main() {
    let person: Person = parse_message(&[
        0x0a, 0x07, 0x6d, 0x61, 0x78, 0x77, 0x65, 0x6c, 0x6c, 0x10, 0x2a, 0x1a,
        0x16, 0x0a, 0x0e, 0x2b, 0x31, 0x32, 0x30, 0x32, 0x2d, 0x35, 0x35, 0x35,
        0x2d, 0x31, 0x32, 0x31, 0x32, 0x12, 0x04, 0x68, 0x6f, 0x6d, 0x65, 0x1a,
        0x18, 0x0a, 0x0e, 0x2b, 0x31, 0x38, 0x30, 0x30, 0x2d, 0x38, 0x36, 0x37,
        0x2d, 0x35, 0x33, 0x30, 0x38, 0x12, 0x06, 0x6d, 0x6f, 0x62, 0x69, 0x6c,
        0x65,
    ])
    .unwrap();
    println!("{:#?}", person);
}

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

    #[test]
    fn as_string() {
        assert!(FieldValue::Varint(10).as_string().is_err());
        assert!(FieldValue::I32(10).as_string().is_err());
        assert_eq!(FieldValue::Len(b"hello").as_string().unwrap(), "hello");
    }
```

```
    #[test]
    fn as_bytes() {
        assert!(FieldValue::Varint(10).as_bytes().is_err());
        assert!(FieldValue::I32(10).as_bytes().is_err());
        assert_eq!(FieldValue::Len(b"hello").as_bytes().unwrap(), b"hello");
    }

    #[test]
    fn as_u64() {
        assert_eq!(FieldValue::Varint(10).as_u64().unwrap(), 10u64);
        assert!(FieldValue::I32(10).as_u64().is_err());
        assert!(FieldValue::Len(b"hello").as_u64().is_err());
    }
}
```

제 VII 편

1일차 오전

제 24 장

4 일차개요

Today we will cover topics relating to building large-scale software in Rust:

- 반복자: `Iterator` 트레잇 심층 분석
- 모듈과 가시성
- Testing.
- 오류처리 (에러 핸들링): 패닉, `Result`, `?` 연산자.
- 안전하지 않은 Rust: 안전한 Rust 로 원하는 것을 표현할 수 없을 때에만사용하세요.

일정예약

Including 10 minute breaks, this session should take about 2 hours and 40 minutes. It contains:

Segment	Duration
개요	3 minutes
Iterators	45 minutes
모듈	40 minutes
테스트	45 minutes

제 25 장

Iterators

This segment should take about 45 minutes. It contains:

Slide	Duration
Iterator	5 minutes
IntoIterator	5 minutes
FromIterator	5 minutes
연습문제: 반복자 메서드 체이닝	30 minutes

25.1 `Iterator`

컬랙션에 있는 값들을 접근하기 위해서는 `Iterator` 트레잇을 사용합니다. 이 트레잇은 `next` 메서드를 비롯한 많은메서드를 제공합니다. 많은 표준 라이브러리 타입이 `Iterator` 를구현하고 있으며, 여러분도 여러분의 타입이 이 트레잇을 직접 구현하도록 할수 있습니다.

```
struct Fibonacci {
    curr: u32,
    next: u32,
}

impl Iterator for Fibonacci {
    type Item = u32;

    fn next(&mut self) -> Option<Self::Item> {
        let new_next = self.curr + self.next;
        self.curr = self.next;
        self.next = new_next;
        Some(self.curr)
    }
}

fn main() {
    let fib = Fibonacci { curr: 0, next: 1 };
```

```
    for (i, n) in fib.enumerate().take(5) {
        println!("fib({i}): {n}");
    }
}
```

- The `Iterator` trait implements many common functional programming operations over collections (e.g. `map`, `filter`, `reduce`, etc). This is the trait where you can find all the documentation about them. In Rust these functions should produce the code as efficient as equivalent imperative implementations.
- `IntoIterator` 는 루프를 작동하게 만드는 트레잇입니다. 이는 `Vec<T>`와 같은 컬렉션 타입과`&Vec<T>` 및 `&[T]` 와 같은이에 대한 참조에 의해 구현됩니다. 범위도 이를 구현합니다. 이런 이유로 `for i in some_vec { .. }`를 사용하여 벡터를 반복할 수있지만 `some_vec.next()` 는 존재하지 않습니다.

25.2 IntoIterator

The `Iterator` trait tells you how to *iterate* once you have created an iterator. The related trait `IntoIterator` defines how to create an iterator for a type. It is used automatically by the `for` loop.

```
struct Grid {
    x_coords: Vec<u32>,
    y_coords: Vec<u32>,
}

impl IntoIterator for Grid {
    type Item = (u32, u32);
    type IntoIter = GridIter;
    fn into_iter(self) -> GridIter {
        GridIter { grid: self, i: 0, j: 0 }
    }
}

struct GridIter {
    grid: Grid,
    i: usize,
    j: usize,
}

impl Iterator for GridIter {
    type Item = (u32, u32);

    fn next(&mut self) -> Option<(u32, u32)> {
        if self.i >= self.grid.x_coords.len() {
            self.i = 0;
            self.j += 1;
            if self.j >= self.grid.y_coords.len() {
                return None;
            }
        }
        let res = Some((self.grid.x_coords[self.i], self.grid.y_coords[self.j]));
```

```
            self.i += 1;
            res
        }
    }

    fn main() {
        let grid = Grid { x_coords: vec![3, 5, 7, 9], y_coords: vec![10, 20, 30, 40] };
        for (x, y) in grid {
            println!("point = {x}, {y}");
        }
    }
```

Click through to the docs for `IntoIterator`. Every implementation of `IntoIterator` must declare two types:

- `Item`: the type to iterate over, such as `i8`,
- `IntoIter`: `into_iter` 메서드에서 반환되는 `Iterator` 타입.

`IntoIter` 에는 `Item` 이 연결되어 있음을 주목하세요. `IntoIter` 반복자는 `Item` 타입의 데이터를 가리켜야합니다. 즉, 반복자는 `Option<Item>`을리턴합니다

이 예는 x 및 y 좌표의 모든 조합을 순회합니다.

`main` 에서 그리드를 두 번 반복해 보세요. 왜 실패하나요? `IntoIterator::into_iter` 는'self' 의 소유권을 가져옵니다.

`&Grid` 에 `IntoIterator` 를 구현하고 `GridIter` 에 `Grid` 참조를 저장하여 이 문제를 해결하세요.

표준 라이브러리 타입에서 동일한 문제가 발생할 수 있습니다. `for e in some_vector` 는 `some_vector` 의 소유권을가져와 해당 벡터에서 소유한 요소를 반복합니다. `some_vector` 의요소에 대한 참조를 반복하려면 대신 `for e in &some_vector` 를사용하세요.

25.3 FromIterator

어떤 컬렉션이 `FromIterator` 를구현하고 있다면 `Iterator` 로부터그 컬렉션을 만들 수 있습니다.

```
fn main() {
    let primes = vec![2, 3, 5, 7];
    let prime_squares = primes.into_iter().map(|p| p * p).collect::<Vec<_>>();
    println!("prime_squares: {prime_squares:?}");
}
```

`Iterator` implements

```
fn collect<B>(self) -> B
where
    B: FromIterator<Self::Item>,
    Self: Sized
```

이 메서드에 `B` 를 지정하는 방법에는 두 가지가 있습니다.

- With the "turbofish": `some_iterator.collect::<COLLECTION_TYPE>()`, as shown. The _ shorthand used here lets Rust infer the type of the Vec elements.
- 타입 추론 사용: `let prime_squares: Vec<_> = some_iterator.collect()` 를사용합니다. 이 방법으로 예제를 다시 작성해 보세요.

There are basic implementations of `FromIterator` for `Vec`, `HashMap`, etc. There are also more specialized implementations which let you do cool things like convert an `Iterator<Item = Result<V, E>>` into a `Result<Vec<V>, E>`.

25.4 연습문제: 반복자 메서드체이닝

In this exercise, you will need to find and use some of the provided methods in the `Iterator` trait to implement a complex calculation.

Copy the following code to https://play.rust-lang.org/ and make the tests pass. Use an iterator expression and `collect` the result to construct the return value.

```
/// Calculate the differences between elements of `values` offset by `offset`,
/// wrapping around from the end of `values` to the beginning.
///
/// Element `n` of the result is `values[(n+offset)%len] - values[n]`.
fn offset_differences<N>(offset: usize, values: Vec<N>) -> Vec<N>
where
    N: Copy + std::ops::Sub<Output = N>,
{
    unimplemented!()
}

#[test]
fn test_offset_one() {
    assert_eq!(offset_differences(1, vec![1, 3, 5, 7]), vec![2, 2, 2, -6]);
    assert_eq!(offset_differences(1, vec![1, 3, 5]), vec![2, 2, -4]);
    assert_eq!(offset_differences(1, vec![1, 3]), vec![2, -2]);
}

#[test]
fn test_larger_offsets() {
    assert_eq!(offset_differences(2, vec![1, 3, 5, 7]), vec![4, 4, -4, -4]);
    assert_eq!(offset_differences(3, vec![1, 3, 5, 7]), vec![6, -2, -2, -2]);
    assert_eq!(offset_differences(4, vec![1, 3, 5, 7]), vec![0, 0, 0, 0]);
    assert_eq!(offset_differences(5, vec![1, 3, 5, 7]), vec![2, 2, 2, -6]);
}

#[test]
fn test_custom_type() {
    assert_eq!(
        offset_differences(1, vec![1.0, 11.0, 5.0, 0.0]),
        vec![10.0, -6.0, -5.0, 1.0]
    );
}

#[test]
fn test_degenerate_cases() {
    assert_eq!(offset_differences(1, vec![0]), vec![0]);
    assert_eq!(offset_differences(1, vec![1]), vec![0]);
    let empty: Vec<i32> = vec![];
```

```
    assert_eq!(offset_differences(1, empty), vec![]);
}
```

25.4.1 해답

```
/// Calculate the differences between elements of `values` offset by `offset`,
/// wrapping around from the end of `values` to the beginning.
///
/// Element `n` of the result is `values[(n+offset)%len] - values[n]`.
fn offset_differences<N>(offset: usize, values: Vec<N>) -> Vec<N>
where
    N: Copy + std::ops::Sub<Output = N>,
{
    let a = (&values).into_iter();
    let b = (&values).into_iter().cycle().skip(offset);
    a.zip(b).map(|(a, b)| *b - *a).collect()
}

#[test]
fn test_offset_one() {
    assert_eq!(offset_differences(1, vec![1, 3, 5, 7]), vec![2, 2, 2, -6]);
    assert_eq!(offset_differences(1, vec![1, 3, 5]), vec![2, 2, -4]);
    assert_eq!(offset_differences(1, vec![1, 3]), vec![2, -2]);
}

#[test]
fn test_larger_offsets() {
    assert_eq!(offset_differences(2, vec![1, 3, 5, 7]), vec![4, 4, -4, -4]);
    assert_eq!(offset_differences(3, vec![1, 3, 5, 7]), vec![6, -2, -2, -2]);
    assert_eq!(offset_differences(4, vec![1, 3, 5, 7]), vec![0, 0, 0, 0]);
    assert_eq!(offset_differences(5, vec![1, 3, 5, 7]), vec![2, 2, 2, -6]);
}

#[test]
fn test_custom_type() {
    assert_eq!(
        offset_differences(1, vec![1.0, 11.0, 5.0, 0.0]),
        vec![10.0, -6.0, -5.0, 1.0]
    );
}

#[test]
fn test_degenerate_cases() {
    assert_eq!(offset_differences(1, vec![0]), vec![0]);
    assert_eq!(offset_differences(1, vec![1]), vec![0]);
    let empty: Vec<i32> = vec![];
    assert_eq!(offset_differences(1, empty), vec![]);
}

fn main() {}
```

제 26 장

모듈

This segment should take about 40 minutes. It contains:

Slide	Duration
모듈	3 minutes
파일시스템 계층	5 minutes
가시성	5 minutes
use, super, self	10 minutes
연습문제: GUI 라이브러리 모듈	15 minutes

26.1 모듈

`impl` 블록은 해당 타입의 함수들에 대한 네임스페이스를 제공합니다.

마찬가지로, `mod` 는 타입과 함수들에 대해 네임스페이스를제공합니다:

```
mod foo {
    pub fn do_something() {
        println!("foo 모듈 내부");
    }
}

mod bar {
    pub fn do_something() {
        println!("bar 모듈 내부");
    }
}

fn main() {
    foo::do_something();
    bar::do_something();
}
```

- 패키지는 기능을 제공하며 하나의 대표 `Cargo.toml` 파일을포함합니다. 패키지를 구성하는 크레이트들을 빌드하는 방법이 이 파일에기술됩니다.

- 크레이트는 모듈의 트리입니다. 바이너리 크레이트는 실행파일로 빌드되고, 라이브러리 크레이트는 라이브러리로 빌드됩니다.
- 모듈은 코드를 조직화하고 스코프를 정의하는 단위입니다.

26.2 파일시스템계층

모듈의 내용을 기술하지 않으면, 러스트는 다른 파일에서 그 내용을읽습니다:

```
mod garden;
```

위 코드는 러스트로 하여금 `garden` 모듈의 내용을 `src/garden.rs` 에서 찾도록 합니다. 비슷하게, `garden::vegetables` 모듈은 `src/garden/vegetables.rs` 에서찾습니다.

`crate`(크레이트) 의 루트는 아래 경로 입니다:

- `src/lib.rs` (라이브러리 크레이트)
- `src/main.rs` (바이너리 크레이트)

모듈도 "내부 문서 주석"을 사용하여 문서화할 수 있습니다. 이러한 모듈은모듈이 포함된 항목 (이 경우에는 모듈) 을 문서화합니다.

```
//! 이 모듈은 정원을 구현합니다.

// 이 모듈에서 타입을 다시 내보냅니다.
pub use garden::Garden;
pub use seeds::SeedPacket;

/// 주어진 씨앗 패킷을 뿌립니다.
pub fn sow(seeds: Vec<SeedPacket>) {
    todo!()
}

/// 준비된 농작물을 정원에서 수확합니다.
pub fn harvest(garden: &mut Garden) {
    todo!()
}
```

- `module/mod.rs` 를 `module.rs` 로 바꾼다 하더라도 Rust 2018 에서는 하위 모듈을 사용할 수 있습니다.
- `filename.rs` 를 `filename/mod.rs` 대신 사용할 수 있도록하는 주된 이유는, `mod.rs` 라는 이름을 가진 파일이 많을 경우 IDE 에서 이들을 서로 구별하는게 힘들기 때문입니다.
- 폴더를 이용해서 더 깊은 계층 구조를 구현할 수 있습니다. 심지어는 메인모듈이 파일이더라도요:

```
src/
├── main.rs
├── top_module.rs
└── top_module/
    └── sub_module.rs
```

- 러스트가 어디서 모듈들을 찾을지는 컴파일러 디렉티브로 변경 가능합니다:

```
#[path = "some/path.rs"]
mod some_module;
```

이는 Go 언어 에서처럼 어떤 모듈의 테스트를 `some_module_test.rs` 같은 파일에 두고 싶은 경우에유용합니다.

26.3 가시성

모듈의 타입이나 함수는 기본적으로 바깥에 노출되지 않습니다:

- 따라서 모듈의 세부 구현 내용이 감춰집니다.
- 부모와 이웃 항목은 언제나 접근 가능합니다.
- 즉, 모듈 `foo` 에서 접근 가능한 항목이라면 `foo` 아래의모든 모듈에서 접근가능합니다.

```
mod outer {
    fn private() {
        println!("outer::private");
    }

    pub fn public() {
        println!("outer::public");
    }

    mod inner {
        fn private() {
            println!("outer::inner::private");
        }

        pub fn public() {
            println!("outer::inner::public");
            super::private();
        }
    }
}

fn main() {
    outer::public();
}
```

- pub 키워드는 모듈에도 사용할 수 있습니다.

또한, 고급 기능으로 `pub(...)` 지정자를 사용하여 공개 범위를제한할 수 있습니다.

- 공식문서를 참고하세요.
- `pub(crate)` 로 가시성을 지정하는 것이 자주 쓰입니다.
- 자주 쓰이진 않지만 특정 경로에 대해서만 가시성을 부여할 수 있습니다.
- 어떤 경우이든 가시성이 부여되면 해당 모듈을 포함하여 하위의 모든모듈이 적용받습니다.

26.4 use, super, self

모듈은 `use` 를 사용하여 다른 모듈의 심볼을 내 스코프로 가져올 수있습니다. 일반적으로 각 모듈의 상단에 다음과 같은 내용이 옵니다:

```
use std::collections::HashSet;
use std::process::abort;
```

경로

경로는 아래와 같이 구분합니다:

1. 상대 경로:
    - `foo` 또는 `self::foo` 는 현재 모듈 내부의 `foo` 를 가리킵니다,
    - `super::foo` 는 부모 모듈의 `foo` 를 가리킵니다.
2. 절대 경로:
    - `crate::foo` 는 현재 크레이트 루트의 `foo` 를가리킵니다,
    - `bar::foo` 는 `bar` 크레이트의 `foo` 를가리킵니다.

- 더 짧은 경로에서 기호를 ' 다시내보내기' 하는 것이 일반적입니다. 예를 들어 크레이트의최상위 `lib.rs` 는

  ```
  mod storage;

  pub use storage::disk::DiskStorage;
  pub use storage::network::NetworkStorage;
  ```

  편리하고 짧은 경로로 다른 크레이트에서 `DiskStorage` 및 `NetworkStorage` 를 사용할 수 있도록 할 수 있습니다.
- 대부분의 경우 모듈에 나타나는 항목만'use' 처리 되어야 합니다. 그러나트레잇을 구현하는 타입이 이미 범위 내에 있더라도 해당 트레잇에서메서드를 호출하려면 트레잇이 범위 내에 있어야 합니다. 예를 들어 `Read` 트레잇을 구현하는 타입에서 `read_to_string` 메서드를 사용하려면 `use std::io::Read` 를 사용해야 합니다.
- `use` 문에는 와일드 카드 (`use std::io::*`) 를 사용할 수있습니다. 이는 가져오는 항목이 명확하지 않고 시간이 지남에 따라 변경될수 있으므로 권장되지 않습니다.

26.5 연습문제: GUI 라이브러리모듈

In this exercise, you will reorganize a small GUI Library implementation. This library defines a `Widget` trait and a few implementations of that trait, as well as a `main` function.

It is typical to put each type or set of closely-related types into its own module, so each widget type should get its own module.

Cargo Setup

Rust 플레이그라운드는 하나의 파일만 지원하므로 로컬 파일 시스템에 Cargo 프로젝트를 만들어야 합니다.

```
cargo init gui-modules
cd gui-modules
cargo run
```

Edit the resulting `src/main.rs` to add `mod` statements, and add additional files in the `src` directory.

Source

Here's the single-module implementation of the GUI library:

```rust
pub trait Widget {
    /// `self`의 자연 너비
    fn width(&self) -> usize;

    /// 버퍼에 위젯을 그립니다.
    fn draw_into(&self, buffer: &mut dyn std::fmt::Write);

    /// 표준 출력에 위젯을 그립니다.
    fn draw(&self) {
        let mut buffer = String::new();
        self.draw_into(&mut buffer);
        println!("{buffer}");
    }
}

pub struct Label {
    label: String,
}

impl Label {
    fn new(label: &str) -> Label {
        Label { label: label.to_owned() }
    }
}

pub struct Button {
    label: Label,
}

impl Button {
    fn new(label: &str) -> Button {
        Button { label: Label::new(label) }
    }
}

pub struct Window {
    title: String,
    widgets: Vec<Box<dyn Widget>>,
}

impl Window {
    fn new(title: &str) -> Window {
        Window { title: title.to_owned(), widgets: Vec::new() }
    }

    fn add_widget(&mut self, widget: Box<dyn Widget>) {
        self.widgets.push(widget);
    }

    fn inner_width(&self) -> usize {
        std::cmp::max(
```

```
            self.title.chars().count(),
            self.widgets.iter().map(|w| w.width()).max().unwrap_or(0),
        )
    }
}

impl Widget for Window {
    fn width(&self) -> usize {
        // 테두리용 패딩 4개 추가
        self.inner_width() + 4
    }

    fn draw_into(&self, buffer: &mut dyn std::fmt::Write) {
        let mut inner = String::new();
        for widget in &self.widgets {
            widget.draw_into(&mut inner);
        }

        let inner_width = self.inner_width();

        // TODO: Change draw_into to return Result<(), std::fmt::Error>. Then use the
        // ?-operator here instead of .unwrap().
        writeln!(buffer, "+-{:-<inner_width$}-+", "").unwrap();
        writeln!(buffer, "| {:^inner_width$} |", &self.title).unwrap();
        writeln!(buffer, "+={:=<inner_width$}=+", "").unwrap();
        for line in inner.lines() {
            writeln!(buffer, "| {:inner_width$} |", line).unwrap();
        }
        writeln!(buffer, "+-{:-<inner_width$}-+", "").unwrap();
    }
}

impl Widget for Button {
    fn width(&self) -> usize {
        self.label.width() + 8 // 패딩을 약간 추가합니다.
    }

    fn draw_into(&self, buffer: &mut dyn std::fmt::Write) {
        let width = self.width();
        let mut label = String::new();
        self.label.draw_into(&mut label);

        writeln!(buffer, "+{:-<width$}+", "").unwrap();
        for line in label.lines() {
            writeln!(buffer, "|{:^width$}|", &line).unwrap();
        }
        writeln!(buffer, "+{:-<width$}+", "").unwrap();
    }
}

impl Widget for Label {
```

```
    fn width(&self) -> usize {
        self.label.lines().map(|line| line.chars().count()).max().unwrap_or(0)
    }

    fn draw_into(&self, buffer: &mut dyn std::fmt::Write) {
        writeln!(buffer, "{}", &self.label).unwrap();
    }
}

fn main() {
    let mut window = Window::new("Rust GUI 데모 1.23");
    window.add_widget(Box::new(Label::new("작은 텍스트 GUI 데모입니다.")));
    window.add_widget(Box::new(Button::new("클릭해 주세요!")));
    window.draw();
}
```

학생들에게는 본인에게 자연스러운 방식으로 코드를 나누도록 권장하세요. 그리고나서 필요한 mod, use, pub 선언에익숙해지도록 합니다. 그런 다음 어떤 구성이 가장 자연스러운지 논의합니다.

26.5.1 해답

```
src
├── main.rs
├── widgets
│   ├── button.rs
│   ├── label.rs
│   └── window.rs
└── widgets.rs
```

```
// ---- src/widgets.rs ----
mod button;
mod label;
mod window;

pub trait Widget {
    /// `self`의 자연 너비
    fn width(&self) -> usize;

    /// 버퍼에 위젯을 그립니다.
    fn draw_into(&self, buffer: &mut dyn std::fmt::Write);

    /// 표준 출력에 위젯을 그립니다.
    fn draw(&self) {
        let mut buffer = String::new();
        self.draw_into(&mut buffer);
        println!("{buffer}");
    }
}

pub use button::Button;
pub use label::Label;
pub use window::Window;
```

```rust
// ---- src/widgets/label.rs ----
use super::Widget;

pub struct Label {
    label: String,
}

impl Label {
    pub fn new(label: &str) -> Label {
        Label { label: label.to_owned() }
    }
}

impl Widget for Label {
    fn width(&self) -> usize {
        // ANCHOR_END: Label-width
        self.label.lines().map(|line| line.chars().count()).max().unwrap_or(0)
    }

    // ANCHOR: Label-draw_into
    fn draw_into(&self, buffer: &mut dyn std::fmt::Write) {
        // ANCHOR_END: Label-draw_into
        writeln!(buffer, "{}", &self.label).unwrap();
    }
}
// ---- src/widgets/button.rs ----
use super::{Label, Widget};

pub struct Button {
    label: Label,
}

impl Button {
    pub fn new(label: &str) -> Button {
        Button { label: Label::new(label) }
    }
}

impl Widget for Button {
    fn width(&self) -> usize {
        // ANCHOR_END: Button-width
        self.label.width() + 8 // 패딩을 약간 추가합니다.
    }

    // ANCHOR: Button-draw_into
    fn draw_into(&self, buffer: &mut dyn std::fmt::Write) {
        // ANCHOR_END: Button-draw_into
        let width = self.width();
        let mut label = String::new();
        self.label.draw_into(&mut label);
```

```
        writeln!(buffer, "+{:-<width$}+", "").unwrap();
        for line in label.lines() {
            writeln!(buffer, "|{:^width$}|", &line).unwrap();
        }
        writeln!(buffer, "+{:-<width$}+", "").unwrap();
    }
}

// ---- src/widgets/window.rs ----
use super::Widget;

pub struct Window {
    title: String,
    widgets: Vec<Box<dyn Widget>>,
}

impl Window {
    pub fn new(title: &str) -> Window {
        Window { title: title.to_owned(), widgets: Vec::new() }
    }

    pub fn add_widget(&mut self, widget: Box<dyn Widget>) {
        self.widgets.push(widget);
    }

    fn inner_width(&self) -> usize {
        std::cmp::max(
            self.title.chars().count(),
            self.widgets.iter().map(|w| w.width()).max().unwrap_or(0),
        )
    }
}

impl Widget for Window {
    fn width(&self) -> usize {
        // ANCHOR_END: Window-width
        // 테두리에 패딩 4 개 추가
        self.inner_width() + 4
    }

    // ANCHOR: Window-draw_into
    fn draw_into(&self, buffer: &mut dyn std::fmt::Write) {
        // ANCHOR_END: Window-draw_into
        let mut inner = String::new();
        for widget in &self.widgets {
            widget.draw_into(&mut inner);
        }

        let inner_width = self.inner_width();
```

```
        // TODO: 오류 처리에 관해 알아본 후,
        // Result<(), std::fmt::Error>를 반환하도록 draw_into를 변경할 수 있습니다. 그런 다음
        // .unwrap() 대신 ?-연산자를 여기에 사용하세요.
        writeln!(buffer, "+-{:-<inner_width$}-+", "").unwrap();
        writeln!(buffer, "| {:^inner_width$} |", &self.title).unwrap();
        writeln!(buffer, "+={:=<inner_width$}=+", "").unwrap();
        for line in inner.lines() {
            writeln!(buffer, "| {:inner_width$} |", line).unwrap();
        }
        writeln!(buffer, "+-{:-<inner_width$}-+", "").unwrap();
    }
}

// ---- src/main.rs ----
mod widgets;

use widgets::Widget;

fn main() {
    let mut window = widgets::Window::new("Rust GUI 데모 1.23");
    window
        .add_widget(Box::new(widgets::Label::new("작은 텍스트 GUI 데모입니다.")));
    window.add_widget(Box::new(widgets::Button::new("클릭해 주세요!")));
    window.draw();
}
```

제 27 장

테스트

This segment should take about 45 minutes. It contains:

Slide	Duration
테스트 모듈	5 minutes
다른 프로젝트	5 minutes
컴파일러 린트 및 Clippy	3 minutes
룬 알고리즘	30 minutes

27.1 단위테스트

러스트와 카고 (cargo) 는 간단한 단위 테스트 프레임워크와 함께 제공됩니다:

- 단위 테스트는 코드 전반에서 지원됩니다.
- 통합 테스트는 `tests/` 디렉터리를 통해 지원됩니다.

Tests are marked with `#[test]`. Unit tests are often put in a nested `tests` module, using `#[cfg(test)]` to conditionally compile them only when building tests.

```
fn first_word(text: &str) -> &str {
    match text.find(' ') {
        Some(idx) => &text[..idx],
        None => &text,
    }
}

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

    #[test]
    fn test_empty() {
        assert_eq!(first_word(""), "");
    }
```

```
    #[test]
    fn test_single_word() {
        assert_eq!(first_word("안녕하세요"), "안녕하세요");
    }

    #[test]
    fn test_multiple_words() {
        assert_eq!(first_word("Hello World"), "안녕하세요");
    }
}
```

- 이렇게 서브 모듈로 테스트를 만들면 private 한 헬퍼 함수에 대한 단위테스트도 가능합니다.
- `#[cfg(test)]` 어트리뷰트가 추가된 항목은 `cargo test` 를 수행했을 경우에만 동작합니다.

플레이그라운드에서 테스트를 실행하여 결과를 표시합니다.

27.2 다른프로젝트

통합테스트

라이브러리를 사용자 입장에서 테스트 하려면, 통합 테스트를 해야 합니다.

`test/`디렉터리 아래에 `.rs` 파일을 하나 만드세요:

```
// tests/my_library.rs
use my_library::init;

#[test]
fn test_init() {
    assert!(init().is_ok());
}
```

이 테스트는 크레이트의 공개 API 에만 접근할 수 있습니다.

문서화주석테스트

러스트는 문서화주석에 대한 테스트를 내장하여 제공합니다:

```
/// Shortens a string to the given length.
///
/// ```
/// # use playground::shorten_string;
/// assert_eq!(shorten_string("Hello World", 5), "Hello");
/// assert_eq!(shorten_string("Hello World", 20), "Hello World");
/// ```
pub fn shorten_string(s: &str, length: usize) -> &str {
    &s[..std::cmp::min(length, s.len())]
}
```

- `///` 주석안의 코드 블록은 자동으로 러스트 코드로 인식됩니다.
- 이 코드 블록은 `cargo test` 호출하면 자동으로 컴파일되고실행됩니다.
- Adding # in the code will hide it from the docs, but will still compile/run it.
- 위 코드를 Rust Playground 에서 테스트 해 보시기 바랍니다.

27.3 컴파일러 린트 및 Clippy

Rust 컴파일러는 유용한 내장 린트뿐 아니라 멋진 오류 메시지를 생성합니다. Clippy 는 프로젝트별로 사용설정할 수 있는 그룹으로 구성된 더 많은 린트를 제공합니다.

```
#[deny(clippy::cast_possible_truncation)]
fn main() {
    let x = 3;
    while (x < 70000) {
        x *= 2;
    }
    println!("X 는 u16 에 맞지 않을까요? {}", x as u16);
}
```

코드 샘플을 실행하고 오류 메시지를 확인합니다. 여기에도 린트가표시되지만 코드가 컴파일되고 나면 표시되지 않습니다. 플레이그라운드사이트로 전환하여 이러한 린트를 표시합니다.

린트를 해결한 후 플레이그라운드 사이트에서 `clippy` 를 실행하여 clippy 경고를 표시합니다. Clippy 는 린트에 관한 광범위한 문서를 보유하고있으며 항상 새로운 린트 (default-deny 린트 포함) 를 추가합니다.

`help: ...` 가 포함된 오류나 경고는 `cargo fix` 또는편집기를 통해 수정할 수 있습니다.

27.4 룬알고리즘

룬알고리즘

룬 (Luhn) 알고리즘은 신용카드 번호 검증에 사용되는 알고리즘 입니다. 이 알고리즘은신용카드 번호를 문자열로 입력받고, 아래의 순서에 따라 신용카드번호의 유효성을 확인합니다:

- Ignore all spaces. Reject number with fewer than two digits.
- **오른쪽에서 왼쪽으로** 이동하며 2 번째 자리마다 숫자를 2 배증가시킵니다. 예를 들어 1234 에서 3 과 1 에각각 2 를 곱합니다.
- After doubling a digit, sum the digits if the result is greater than 9. So doubling 7 becomes 14 which becomes `1 + 4 = 5`.
- 모든 자리의 숫자를 더합니다.
- 합계의 끝자리가 `0` 인 경우 유효한 신용카드 번호입니다.

제공된 코드는 대부분의 알고리즘이 올바르게 구현되었는지 확인하는 두 가지기본 단위 테스트와 함께 luhn 알고리즘의 버그가 있는 구현을 제공합니다.

Copy the code below to https://play.rust-lang.org/ and write additional tests to uncover bugs in the provided implementation, fixing any bugs you find.

```
pub fn luhn(cc_number: &str) -> bool {
    let mut sum = 0;
    let mut double = false;

    for c in cc_number.chars().rev() {
        if let Some(digit) = c.to_digit(10) {
            if double {
                let double_digit = digit * 2;
```

```
                sum +=
                    if double_digit > 9 { double_digit - 9 } else { double_digit };
            } else {
                sum += digit;
            }
            double = !double;
        } else {
            continue;
        }
    }

    sum % 10 == 0
}

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

    #[test]
    fn test_valid_cc_number() {
        assert!(luhn("4263 9826 4026 9299"));
        assert!(luhn("4539 3195 0343 6467"));
        assert!(luhn("7992 7398 713"));
    }

    #[test]
    fn test_invalid_cc_number() {
        assert!(!luhn("4223 9826 4026 9299"));
        assert!(!luhn("4539 3195 0343 6476"));
        assert!(!luhn("8273 1232 7352 0569"));
    }
}
```

27.4.1 해답

```
// 문제에 표시되는 버그가 있는 버전입니다.
#[cfg(never)]
pub fn luhn(cc_number: &str) -> bool {
    let mut sum = 0;
    let mut double = false;

    for c in cc_number.chars().rev() {
        if let Some(digit) = c.to_digit(10) {
            if double {
                let double_digit = digit * 2;
                sum +=
                    if double_digit > 9 { double_digit - 9 } else { double_digit };
            } else {
                sum += digit;
            }
            double = !double;
```

```
        } else {
            continue;
        }
    }

    sum % 10 == 0
}

// 솔루션이며 아래의 모든 테스트를 통과합니다.
pub fn luhn(cc_number: &str) -> bool {
    let mut sum = 0;
    let mut double = false;
    let mut digits = 0;

    for c in cc_number.chars().rev() {
        if let Some(digit) = c.to_digit(10) {
            digits += 1;
            if double {
                let double_digit = digit * 2;
                sum +=
                    if double_digit > 9 { double_digit - 9 } else { double_digit };
            } else {
                sum += digit;
            }
            double = !double;
        } else if c.is_whitespace() {
            continue;
        } else {
            return false;
        }
    }

    digits >= 2 && sum % 10 == 0
}

fn main() {
    let cc_number = "1234 5678 1234 5670";
    println!(
        "{cc_number}은 (는) 유효한 신용카드 번호인가요? {}",
        if luhn(cc_number) { "예" } else { "아니요" }
    );
}

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

    #[test]
    fn test_valid_cc_number() {
        assert!(luhn("4263 9826 4026 9299"));
        assert!(luhn("4539 3195 0343 6467"));
```

```
        assert!(luhn("7992 7398 713"));
    }

    #[test]
    fn test_invalid_cc_number() {
        assert!(!luhn("4223 9826 4026 9299"));
        assert!(!luhn("4539 3195 0343 6476"));
        assert!(!luhn("8273 1232 7352 0569"));
    }

    #[test]
    fn test_non_digit_cc_number() {
        assert!(!luhn("foo"));
        assert!(!luhn("foo 0 0"));
    }

    #[test]
    fn test_empty_cc_number() {
        assert!(!luhn(""));
        assert!(!luhn(" "));
        assert!(!luhn("  "));
        assert!(!luhn("    "));
    }

    #[test]
    fn test_single_digit_cc_number() {
        assert!(!luhn("0"));
    }

    #[test]
    fn test_two_digit_cc_number() {
        assert!(luhn(" 0 0 "));
    }
}
```

제 VIII 편

1일차 오후

제 28 장

Welcome Back

Including 10 minute breaks, this session should take about 2 hours and 10 minutes. It contains:

Segment	Duration
오류처리	55 minutes
안전하지 않은 러스트	1 hour and 5 minutes

제 29 장

오류처리

This segment should take about 55 minutes. It contains:

Slide	Duration
패닉	3 minutes
Iterator	5 minutes
묵시적 형변환	5 minutes
Error	5 minutes
From 과 Into	5 minutes
Result 를 이용한 구조화된 오류처리	30 minutes

29.1 패닉

Rust 는 ' 패닉' 으로 치명적인 오류를처리합니다.

러스트는 수행 중 치명적인 오류를 만나면 패닉을 발생할 것입니다:

```
fn main() {
    let v = vec![10, 20, 30];
    println!("v[100]: {}", v[100]);
}
```

- 패닉은 복구할 수 없고 예상치 못한 오류입니다.
  - 패닉은 프로그램에 버그가 있다는 것을 나타냅니다.
  - 경계 검사 실패와 같은 런타임 실패로 인해 패닉이 발생할 수 있습니다.
  - 실패 시 어설션 (예: `assert!`) 패닉
  - 목적별 패닉은 `panic!` 매크로를 사용할 수 있습니다.
- 패닉은 스택을 ' 해제' 하여 함수가 반환된것처럼 값을 삭제합니다.
- 충돌 (크래시) 을 허용하지 않아야 하는 경우, 패닉을 유발하지 않는 API(`Vec::get` 등) 를 사용하면 됩니다.

기본적으로, 패닉이 발생하면 스택 되감기를 합니다. 스택 되감기는 다음과같이 캐치가 가능합니다:

```
use std::panic;

fn main() {
```

```
    let result = panic::catch_unwind(|| "괜찮습니다.");
    println!("{result:?}");

    let result = panic::catch_unwind(|| {
        panic!("이런");
    });
    println!("{result:?}");
}
```

- 포착은 흔하지 않습니다. `catch_unwind` 로 예외를 구현하려고시도하지 마세요.
- 이것은 단일 요청이 크래시 되더라도 프로그램이 계속 실행되야 하는서버에 유용합니다.
- 만약 `Cargo.toml` 설정파일에 `panic = abort` 을설정했다면 크래시를 캐치할 수 없습니다.

29.2 Iterator

Runtime errors like connection-refused or file-not-found are handled with the `Result` type, but matching this type on every call can be cumbersome. The try-operator `?` is used to return errors to the caller. It lets you turn the common

```
match some_expression {
    Ok(value) => value,
    Err(err) => return Err(err),
}
```

이렇게 짧게 쓸 수 있습니다

```
some_expression?
```

이제 우리 예제에 적용해 보겠습니다:

```
use std::io::Read;
use std::{fs, io};

fn read_username(path: &str) -> Result<String, io::Error> {
    let username_file_result = fs::File::open(path);
    let mut username_file = match username_file_result {
        Ok(file) => file,
        Err(err) => return Err(err),
    };

    let mut username = String::new();
    match username_file.read_to_string(&mut username) {
        Ok(_) => Ok(username),
        Err(err) => Err(err),
    }
}

fn main() {
    //fs::write("config.dat", "alice").unwrap();
    let username = read_username("config.dat");
    println!("사용자 이름 또는 오류: {username:?}");
}
```

`?`를 사용하도록 `read_username` 함수를 단순화합니다.

키 포인트:

- `username` 변수는 `Ok(string)` 이거나 `Err(error)` 일 수 있습니다.
- `fs::write` 메서드를 사용하여 파일이 없거나, 비었거나, 중복되는경우 등을 테스트해 봅니다.
- Note that `main` can return a `Result<(), E>` as long as it implements `std::process::Termination`. In practice, this means that `E` implements `Debug`. The executable will print the `Err` variant and return a nonzero exit status on error.

29.3 묵시적형변환

실제로 ?가 적용되는 과정은 아까 설명한 것 보다 좀 더복잡합니다:

```
expression?
```

위 표현은 아래와 같습니다

```
match expression {
    Ok(value) => value,
    Err(err)  => return Err(From::from(err)),
}
```

The `From::from` call here means we attempt to convert the error type to the type returned by the function. This makes it easy to encapsulate errors into higher-level errors.

예제

```
use std::error::Error;
use std::fmt::{self, Display, Formatter};
use std::fs::File;
use std::io::{self, Read};

#[derive(Debug)]
enum ReadUsernameError {
    IoError(io::Error),
    EmptyUsername(String),
}

impl Error for ReadUsernameError {}

impl Display for ReadUsernameError {
    fn fmt(&self, f: &mut Formatter) -> fmt::Result {
        match self {
            Self::IoError(e) => write!(f, "IO 오류: {e}"),
            Self::EmptyUsername(path) => write!(f, "Found no username in {path}"),
        }
    }
}

impl From<io::Error> for ReadUsernameError {
    fn from(err: io::Error) -> Self {
        Self::IoError(err)
    }
```

```
}

fn read_username(path: &str) -> Result<String, ReadUsernameError> {
    let mut username = String::with_capacity(100);
    File::open(path)?.read_to_string(&mut username)?;
    if username.is_empty() {
        return Err(ReadUsernameError::EmptyUsername(String::from(path)));
    }
    Ok(username)
}

fn main() {
    //fs::write("config.dat", "").unwrap();
    let username = read_username("config.dat");
    println!("사용자 이름 또는 오류: {username:?}");
}
```

The ? operator must return a value compatible with the return type of the function. For `Result`, it means that the error types have to be compatible. A function that returns `Result<T, ErrorOuter>` can only use ? on a value of type `Result<U, ErrorInner>` if `ErrorOuter` and `ErrorInner` are the same type or if `ErrorOuter` implements `From<ErrorInner>`.

`From` 구현의 일반적인 대안은 특히 변환이 한 곳에서만 발생하는경우 `Result::map_err` 입니다.

There is no compatibility requirement for `Option`. A function returning `Option<T>` can use the ? operator on `Option<U>` for arbitrary `T` and `U` types.

A function that returns `Result` cannot use ? on `Option` and vice versa. However, `Option::ok_or` converts `Option` to `Result` whereas `Result::ok` turns `Result` into `Option`.

29.4 동적인 에러타입

Sometimes we want to allow any type of error to be returned without writing our own enum covering all the different possibilities. The `std::error::Error` trait makes it easy to create a trait object that can contain any error.

```
use std::error::Error;
use std::fs;
use std::io::Read;

fn read_count(path: &str) -> Result<i32, Box<dyn Error>> {
    let mut count_str = String::new();
    fs::File::open(path)?.read_to_string(&mut count_str)?;
    let count: i32 = count_str.parse()?;
    Ok(count)
}

fn main() {
    fs::write("count.dat", "1i3").unwrap();
    match read_count("count.dat") {
        Ok(count) => println!("개수: {count}"),
```

```
        Err(err) => println!("오류: {err}"),
    }
}
```

`read_count` 함수는 `std::io::Error`(파일 작업에서) 또는 `std::num::ParseIntError`(`String::parse`에서) 를 반환할 수있습니다.

Boxing errors saves on code, but gives up the ability to cleanly handle different error cases differently in the program. As such it's generally not a good idea to use `Box<dyn Error>` in the public API of a library, but it can be a good option in a program where you just want to display the error message somewhere.

Make sure to implement the `std::error::Error` trait when defining a custom error type so it can be boxed. But if you need to support the `no_std` attribute, keep in mind that the `std::error::Error` trait is currently compatible with `no_std` in nightly only.

29.5 `thiserror` and `anyhow`

The `thiserror` and `anyhow` crates are widely used to simplify error handling.

- `thiserror` is often used in libraries to create custom error types that implement `From<T>`.
- `anyhow` is often used by applications to help with error handling in functions, including adding contextual information to your errors.

```
use anyhow::{bail, Context, Result};
use std::fs;
use std::io::Read;
use thiserror::Error;

#[derive(Clone, Debug, Eq, Error, PartialEq)]
#[error("{0}에서 사용자 이름을 찾을 수 없습니다.")]
struct EmptyUsernameError(String);

fn read_username(path: &str) -> Result<String> {
    let mut username = String::with_capacity(100);
    fs::File::open(path)
        .with_context(|| format!("{path}을 (를) 열지 못했습니다."))?
        .read_to_string(&mut username)
        .context("읽지 못했습니다.")?;
    if username.is_empty() {
        bail!(EmptyUsernameError(path.to_string()));
    }
    Ok(username)
}

fn main() {
    //fs::write("config.dat", "").unwrap();
    match read_username("config.dat") {
        Ok(username) => println!("사용자 이름: {username}"),
        Err(err) => println!("오류: {err:?}"),
    }
}
```

thiserror

- The `Error` derive macro is provided by `thiserror`, and has lots of useful attributes to help define error types in a compact way.
- The `std::error::Error` trait is derived automatically.
- The message from `#[error]` is used to derive the `Display` trait.

anyhow

- `anyhow::Error`는 `Box<dyn Error>`의 래퍼 타입이라 할 수있습니다. 따라서 라이브러리의 공개 API 로서 사용하기에 부적합하다고 할수 있지만 많은 애플리케이션에 널리 사용되고 있습니다.
- `anyhow::Result<V>`는 `Result<V, anyhow::Error>`의 타입앨리어스 (alias) 입니다.
- 필요하다면 `anyhow::Error`에 저장된 진짜 에러 타입을 꺼내어검사할 수도 있습니다.
- `anyhow::Result<T>`가 제공하는 기능들이 Go 언어 개발자들에게는 익숙할 것입니다. Go 언어에서 반환 값으로사용하는 `(T, error)` 패턴과 비슷하기 때문입니다.
- `anyhow::Context` is a trait implemented for the standard `Result` and `Option` types. `use anyhow::Context` is necessary to enable `.context()` and `.with_context()` on those types.

29.6 Result를 이용한 구조화된오류처리

다음은 표현식 언어의 매우 간단한 파서를 구현합니다. 그러나 패닉을 통해오류를 처리합니다. 대신 관용적인 오류 처리를 사용하고 오류를 `main`의 반환으로 전파하도록 다시 작성합니다. `thiserror` 및 `anyhow`를 얼마든지 사용하세요.

힌트: 먼저 `parse` 함수에서 오류 처리를 수정하세요. 제대로작동하면 `Iterator<Item=Result<Token, TokenizerError>>`를구현하도록 `Tokenizer`를 업데이트하고 파서에서 처리합니다.

```
use std::iter::Peekable;
use std::str::Chars;

/// 산술 연산자입니다.
#[derive(Debug, PartialEq, Clone, Copy)]
enum Op {
    Add,
    Sub,
}

/// 표현식 언어의 토큰입니다.
#[derive(Debug, PartialEq)]
enum Token {
    Number(String),
    Identifier(String),
    Operator(Op),
}

/// 표현식 언어의 표현식입니다.
#[derive(Debug, PartialEq)]
enum Expression {
    /// 변수 참조입니다.
```

```
    Var(String),
    /// 리터럴 숫자입니다.
    Number(u32),
    /// 이진 연산입니다.
    Operation(Box<Expression>, Op, Box<Expression>),
}

fn tokenize(input: &str) -> Tokenizer {
    return Tokenizer(input.chars().peekable());
}

struct Tokenizer<'a>(Peekable<Chars<'a>>);

impl<'a> Iterator for Tokenizer<'a> {
    type Item = Token;

    fn next(&mut self) -> Option<Token> {
        let c = self.0.next()?;
        match c {
            '0'..='9' => {
                let mut num = String::from(c);
                while let Some(c @ '0'..='9') = self.0.peek() {
                    num.push(*c);
                    self.0.next();
                }
                Some(Token::Number(num))
            }
            'a'..='z' => {
                let mut ident = String::from(c);
                while let Some(c @ ('a'..='z' | '_' | '0'..='9')) = self.0.peek() {
                    ident.push(*c);
                    self.0.next();
                }
                Some(Token::Identifier(ident))
            }
            '+' => Some(Token::Operator(Op::Add)),
            '-' => Some(Token::Operator(Op::Sub)),
            _ => panic!("예기치 않은 문자 {c}"),
        }
    }
}

fn parse(input: &str) -> Expression {
    let mut tokens = tokenize(input);

    fn parse_expr<'a>(tokens: &mut Tokenizer<'a>) -> Expression {
        let Some(tok) = tokens.next() else {
            panic!("예기치 않은 입력 종료");
        };
        let expr = match tok {
            Token::Number(num) => {
```

```
                let v = num.parse().expect("잘못된 32 비트 정수입니다.'");
                Expression::Number(v)
            }
            Token::Identifier(ident) => Expression::Var(ident),
            Token::Operator(_) => panic!("예기치 않은 토큰 {tok:?}"),
        };
        // 이진 연산이 있는 경우 이를 파싱합니다.
        match tokens.next() {
            None => expr,
            Some(Token::Operator(op)) => Expression::Operation(
                Box::new(expr),
                op,
                Box::new(parse_expr(tokens)),
            ),
            Some(tok) => panic!("예기치 않은 토큰 {tok:?}"),
        }
    }

    parse_expr(&mut tokens)
}

fn main() {
    let expr = parse("10+foo+20-30");
    println!("{expr:?}");
}
```

29.6.1 해답

```
use thiserror::Error;
use std::iter::Peekable;
use std::str::Chars;

/// 산술 연산자입니다.
#[derive(Debug, PartialEq, Clone, Copy)]
enum Op {
    Add,
    Sub,
}

/// 표현식 언어의 토큰입니다.
#[derive(Debug, PartialEq)]
enum Token {
    Number(String),
    Identifier(String),
    Operator(Op),
}

/// 표현식 언어의 표현식입니다.
#[derive(Debug, PartialEq)]
enum Expression {
    /// 변수 참조입니다.
```

```
    Var(String),
    /// 리터럴 숫자입니다.
    Number(u32),
    /// 이진 연산입니다.
    Operation(Box<Expression>, Op, Box<Expression>),
}

fn tokenize(input: &str) -> Tokenizer {
    return Tokenizer(input.chars().peekable());
}

#[derive(Debug, Error)]
enum TokenizerError {
    #[error("입력에 예상치 못한 문자 '{0}'이(가) 있습니다.")]
    UnexpectedCharacter(char),
}

struct Tokenizer<'a>(Peekable<Chars<'a>>);

impl<'a> Iterator for Tokenizer<'a> {
    type Item = Result<Token, TokenizerError>;

    fn next(&mut self) -> Option<Result<Token, TokenizerError>> {
        let c = self.0.next()?;
        match c {
            '0'..='9' => {
                let mut num = String::from(c);
                while let Some(c @ '0'..='9') = self.0.peek() {
                    num.push(*c);
                    self.0.next();
                }
                Some(Ok(Token::Number(num)))
            }
            'a'..='z' => {
                let mut ident = String::from(c);
                while let Some(c @ ('a'..='z' | '_' | '0'..='9')) = self.0.peek() {
                    ident.push(*c);
                    self.0.next();
                }
                Some(Ok(Token::Identifier(ident)))
            }
            '+' => Some(Ok(Token::Operator(Op::Add))),
            '-' => Some(Ok(Token::Operator(Op::Sub))),
            _ => Some(Err(TokenizerError::UnexpectedCharacter(c))),
        }
    }
}

#[derive(Debug, Error)]
enum ParserError {
    #[error("토큰나이저 오류: {0}")]
```

```
    TokenizerError(#[from] TokenizerError),
    #[error("예기치 않은 입력 종료")]
    UnexpectedEOF,
    #[error("예기치 않은 토큰 {0:?}")]
    UnexpectedToken(Token),
    #[error("잘못된 번호")]
    InvalidNumber(#[from] std::num::ParseIntError),
}

fn parse(input: &str) -> Result<Expression, ParserError> {
    let mut tokens = tokenize(input);

    fn parse_expr<'a>(
        tokens: &mut Tokenizer<'a>,
    ) -> Result<Expression, ParserError> {
        let tok = tokens.next().ok_or(ParserError::UnexpectedEOF)??;
        let expr = match tok {
            Token::Number(num) => {
                let v = num.parse()?;
                Expression::Number(v)
            }
            Token::Identifier(ident) => Expression::Var(ident),
            Token::Operator(_) => return Err(ParserError::UnexpectedToken(tok)),
        };
        // 이진 연산이 있는 경우 이를 파싱합니다.
        Ok(match tokens.next() {
            None => expr,
            Some(Ok(Token::Operator(op))) => Expression::Operation(
                Box::new(expr),
                op,
                Box::new(parse_expr(tokens)?),
            ),
            Some(Err(e)) => return Err(e.into()),
            Some(Ok(tok)) => return Err(ParserError::UnexpectedToken(tok)),
        })
    }

    parse_expr(&mut tokens)
}

fn main() -> anyhow::Result<()> {
    let expr = parse("10+foo+20-30")?;
    println!("{expr:?}");
    Ok(())
}
```

제 30 장

안전하지 않은러스트

This segment should take about 1 hour and 5 minutes. It contains:

Slide	Duration
안전하지 않은 러스트	5 minutes
원시 포인터 역참조 (따라가기)	10 minutes
정적 가변 변수	5 minutes
Unions	5 minutes
안전하지 않은 함수 호출	5 minutes
안전하지 않은 트레잇 구현하기	5 minutes
연습문제: FFI 래퍼 (wrapper)	30 minutes

30.1 안전하지 않은러스트

러스트로 작성된 프로그램은 크게 두 부분으로 나뉩니다:

- **안전한 러스트:** 메모리 관련 오류 발생 불가능, 정의되지 않은동작 없음.
- **안전하지 않은 러스트:** 특별한 조건을 만족하지 않은채로사용되면 정의되지 않은 동작을 유발할 수 있음.

We saw mostly safe Rust in this course, but it's important to know what Unsafe Rust is.

보통, 안전하지 않은 러스트 코드는 크기가 작으며, 독립적으로 존재합니다. 그리고 코드가 왜 잘 작동하는지에 대해 세밀하게 문서화가 되어 있습니다. 그리고, 많은 경우 안전한 러스트 코드를 통해서 추상화를 시킨 후 사용하게됩니다.

안전하지 않은 러스트를 이용하면 다음과 같은 다섯 가지 것들이 가능해집니다:

- 원시 포인터 역참조 (따라가기)
- 정적 가변변수 접근 및 수정.
- `union` 필드 접근.
- `extern` 함수를 포함한 `unsafe` 함수 호출.
- `unsafe` 트레잇 구현.

위 기능들에 대해 간략히 살펴보겠습니다. 자세한 내용은러스트프로그래밍 언어, 19.1 절과 Rustonomicon 를 참조하세요.

Unsafe Rust does not mean the code is incorrect. It means that developers have turned off some compiler safety features and have to write correct code by themselves. It means the compiler no longer enforces Rust's memory-safety rules.

30.2 원시 포인터역참조 (따라가기)

포인터를 만드는 것은 안전합니다. 하지만 역참조 (따라가기) 할 경우 `unsafe` 가 필요합니다:

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

    let r1 = &mut s as *mut String;
    let r2 = r1 as *const String;

    // Safe because r1 and r2 were obtained from references and so are
    // guaranteed to be non-null and properly aligned, the objects underlying
    // the references from which they were obtained are live throughout the
    // whole unsafe block, and they are not accessed either through the
    // references or concurrently through any other pointers.
    unsafe {
        println!("r1은 {}입니다.", *r1);
        *r1 = String::from("이런");
        println!("r2는 {}입니다.", *r2);
    }

    // 안전하지 않음. 이렇게 하지 마세요.
    /*
    let r3: &String = unsafe { &*r1 };
    drop(s);
    println!("r3 is: {}", *r3);
    */
}
```

모든 `unsafe` 블록에 대해 왜 그 코드가 안전한지에 대한 설명을주석으로 다는 것은 좋은 습관입니다 (사실 안드로이드의 러스트 스타일가이드에서는 이게 필수입니다).

포인터 역참조를 할 경우, 포인터가 해야합니다. 예를 들어:

- 포인터는 null 이면 안됩니다.
- 포인터는 따라가기가 가능해야 합니다 (객체의 어느 한 부분을 가리키고있어야 합니다).
- 이미 반환된 객체를 가리키면 안됩니다.
- 같은 위치에 대해 동시적인 접근이 있으면 안됩니다.
- 참조를 캐스팅 해서 포인터를 만들었다면, 그 참조가 가리키는 객체는 살아있어야 하며, 그 객체의 메모리를 접근하는 참조가 하나도 없어야 합니다.

대부분의 경우 포인터는 align 되어 있어야 합니다.

The ”NOT SAFE” section gives an example of a common kind of UB bug: `*r1` has the `'static` lifetime, so `r3` has type `&'static String`, and thus outlives `s`. Creating a reference from a pointer requires *great care*.

30.3 정적 가변변수

불변 정적변수를 읽는 것은 안전합니다:

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

fn main() {
    println!("HELLO_WORLD: {HELLO_WORLD}");
}
```

하지만, 데이터 레이스가 발생할 수 있으므로 정적 가변변수를 읽고 쓰는것은 안전하지 않습니다:

```
static mut COUNTER: u32 = 0;

fn add_to_counter(inc: u32) {
    unsafe {
        COUNTER += inc;
    }
}

fn main() {
    add_to_counter(42);

    unsafe {
        println!("COUNTER: {COUNTER}");
    }
}
```

- 이 프로그램은 단일 스레드이므로 안전합니다. 그러나 Rust 컴파일러는보수적이며 최악의 상황을 가정합니다. unsafe 를 삭제해 보고컴파일러가 여러 스레드에서 static 을 변경하는 것이 정의되지 않은동작이라고 어떻게 설명하는지 확인하세요.
- 일반적으로 이야기 해서, 정적 가변 변수를 쓰는 것은 좋은 아이디어가아닙니다. 그러나 no_std 와 같은 저수준 코딩을 할 경우에는필요하기도 합니다. 예를 들어 힙 할당기를 구현하거나, C API 를 사용하는게 그런 경우입니다.

30.4 Unions

유니온 타입은 열거형 (enum) 과 비슷하지만, 어떤 필드에 해당하는 값을가지고 있는지 여부를 프로그래머가 수동으로 추적해야 합니다:

```
#[repr(C)]
union MyUnion {
    i: u8,
    b: bool,
}

fn main() {
    let u = MyUnion { i: 42 };
    println!("int: {}", unsafe { u.i });
    println!("부울: {}", unsafe { u.b }); // Undefined behavior!
}
```

러스트에는 열거형이 있기 때문에 유니온이 필요한 경우는 극히 드뭅니다. 유니온은 C 라이브러리 API를 사용할 때 가끔 필요합니다.

바이트들을 특정 타입으로 재해석 하고 싶다면 `std::mem::transmute` 나좀 더 안전한 `zerocopy` 크레이트를 사용하세요.

30.5 안전하지 않은 함수호출

안전하지 않은 함수호출

함수나 메서드가 정의되지 않은 동작으로 빠지지 않게 하기 위해서 만족해야하는 전제 조건이 있는 경우, 그 함수나 메서드를 `unsafe` 로 표시할수 있습니다:

```
extern "C" {
    fn abs(input: i32) -> i32;
}

fn main() {
    let emojis = "🗻∈🌏";

    // 색인이 올바른 순서이고 문자열 슬라이스의 경계 내에
    // 있으며 UTF-8 시퀀스 경계에 있으므로 안전합니다.
    unsafe {
        println!("이모티콘: {}", emojis.get_unchecked(0..4));
        println!("이모티콘: {}", emojis.get_unchecked(4..7));
        println!("이모티콘: {}", emojis.get_unchecked(7..11));
    }

    println!("문자 수: {}", count_chars(unsafe { emojis.get_unchecked(0..7) }));

    unsafe {
        // Undefined behavior if abs misbehaves.
        println!("C 에 따른 절댓값 -3: {}", abs(-3));
    }

    // Not upholding the UTF-8 encoding requirement breaks memory safety!
    // println!("emoji: {}", unsafe { emojis.get_unchecked(0..3) });
    // println!("char count: {}", count_chars(unsafe {
    // emojis.get_unchecked(0..3) }));
}

fn count_chars(s: &str) -> usize {
    s.chars().count()
}
```

안전하지 않은 함수작성하기

여러분이 작성한 함수를 사용할 때 어떤 특별한 조건을 만족해야 한다면, `unsafe` 로 마킹할 수 있습니다.

```
/// 지정된 포인터가 가리키는 값을 바꿉니다.
///
```

```rust
/// # Safety
///
/// 포인터는 유효하고 올바르게 정렬되어야 합니다.
unsafe fn swap(a: *mut u8, b: *mut u8) {
    let temp = *a;
    *a = *b;
    *b = temp;
}

fn main() {
    let mut a = 42;
    let mut b = 66;

    // 다음과 같은 이유로 안전합니다.
    unsafe {
        swap(&mut a, &mut b);
    }

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

안전하지 않은 함수호출

get_unchecked, like most _unchecked functions, is unsafe, because it can create UB if the range is incorrect. abs is incorrect for a different reason: it is an external function (FFI). Calling external functions is usually only a problem when those functions do things with pointers which might violate Rust's memory model, but in general any C function might have undefined behaviour under any arbitrary circumstances.

위 예제 코드에서 "C"는 ABI 를 의미합니다. 다른 ABI 도 있습니다.

안전하지 않은 함수작성하기

We wouldn't actually use pointers for a swap function - it can be done safely with references.

Note that unsafe code is allowed within an unsafe function without an unsafe block. We can prohibit this with #[deny(unsafe_op_in_unsafe_fn)]. Try adding it and see what happens. This will likely change in a future Rust edition.

30.6 안전하지 않은 트레잇구현하기

함수에서와 마찬가지로 트레잇도 unsafe 로 마킹 가능합니다. 만약그 트레잇을 구현할 때 정의되지 않은 동작을 피하기 위해 특별한 조건이필요하다면 말이지요.

예를 들어 zerocopy 크레이트에는안전하지않은 트레잇이 있습니다:

```rust
use std::mem::size_of_val;
use std::slice;

/// ...
/// # Safety
/// 타입에는 정의된 표현이 있어야 하며 패딩은 없어야 합니다.
```

```
pub unsafe trait AsBytes {
    fn as_bytes(&self) -> &[u8] {
        unsafe {
            slice::from_raw_parts(
                self as *const Self as *const u8,
                size_of_val(self),
            )
        }
    }
}

// u32에 정의된 표현이 있고 패딩이 없으므로 안전합니다.
unsafe impl AsBytes for u32 {}
```

안전하지 않은 트레잇을 만들 때에는 주석에 `# Safety` 항목이있어서 이 트레잇을 안전하게 구현하려면 어떤 요구사항들을 만족해야하는지를 설명해야 합니다.

`AsBytes` 에서 지켜야 할 안전성에 대한 실제 설명은 좀 더 길고복잡합니다.

빌트인 트레잇인 `Send` 와 `Sync` 는 안전하지 않은 트레잇입니다.

30.7 FFI 래퍼

Rust has great support for calling functions through a *foreign function interface* (FFI). We will use this to build a safe wrapper for the `libc` functions you would use from C to read the names of files in a directory.

아래 리눅스 메뉴얼 문서들을 참조하시기 바랍니다:

- `opendir(3)`
- `readdir(3)`
- `closedir(3)`

아마 `std::ffi` 모듈을참조할 필요가 있을 것입니다. 거기에는 이번 예제를 수행하는데 필요한다양한 종류의 문자열 타입들이 소개되어 있습니다:

타입	인코딩	사용
`str` 과 `String`	UTF-8	러스트에서의 문자열 처리
`CStr` 과 `CString`	널 (NUL) 로 끝남	C 함수와 연동하기
`OsStr` 와 `OsString`	OS 가 정의함	OS 와 연동하기 위한 문자열

이 타입들 간의 변환은 다음과 같습니다:

- `&str` 에서 `CString` 으로의 변환: 맨 마지막의`\0` 문자를 저장하기 위한 공간을 할당해야합니다,
- `CString` 에서 `*const i8` 로의 변환: C 함수를 호출하기위해서는 포인터가 필요합니다,
- `*const i8` 에서 `&CStr` 로의 변환: 주어진 바이트시퀀스가 `\0` 로 끝나는지 확인하고 싶은 경우,
- `&CStr` to `&[u8]`: a slice of bytes is the universal interface for "some unknown data",
- `&[u8]` 에서 `&OsStr` 로의 변환: `&OsStr` 는 `OsString` 으로 가기 위한 중간 단계 입니다. `OsStrExt` 를사용해서 `OsStr` 를 생성하세요,
- `&OsStr` 에서 `OsString` 으로의 변환: `&OsStr` 이가리키고 있는 데이터를 복사함으로써, 이 데이터를 리턴하고, `readdir` 함수를 호출할 때 사용할 수 있게 해 줍니다.

Nomicon 에 FFI 와관련한 아주 유용한 챕터가 있습니다.

아래 코드를 https://play.rust-lang.org/ 에 복사하고 빠진 함수와메서드를 채워봅니다:

```
// TODO: 구현이 완료되면 이를 삭제합니다.
#![allow(unused_imports, unused_variables, dead_code)]

mod ffi {
    use std::os::raw::{c_char, c_int};
    #[cfg(not(target_os = "매크로"))]
    use std::os::raw::{c_long, c_uchar, c_ulong, c_ushort};

    // 불투명 타입입니다. https://doc.rust-lang.org/nomicon/ffi.html 을 참고하세요.
    #[repr(C)]
    pub struct DIR {
        _data: [u8; 0],
        _marker: core::marker::PhantomData<(*mut u8, core::marker::PhantomPinned)>,
    }

    // readdir(3) 의 Linux man 페이지에 따른 레이아웃입니다.
    // 여기서 ino_t 및 off_t 는
    // /usr/include/x86_64-linux-gnu/{sys/types.h, bits/typesizes.h}의 정의에 따라 확인됩니
    #[cfg(not(target_os = "매크로"))]
    #[repr(C)]
    pub struct dirent {
        pub d_ino: c_ulong,
        pub d_off: c_long,
        pub d_reclen: c_ushort,
        pub d_type: c_uchar,
        pub d_name: [c_char; 256],
    }

    // dir(5) 의 macOS man 페이지에 따른 레이아웃입니다.
    #[cfg(all(target_os = "매크로"))]
    #[repr(C)]
    pub struct dirent {
        pub d_fileno: u64,
        pub d_seekoff: u64,
        pub d_reclen: u16,
        pub d_namlen: u16,
        pub d_type: u8,
        pub d_name: [c_char; 1024],
    }

    extern "C" {
        pub fn opendir(s: *const c_char) -> *mut DIR;

        #[cfg(not(all(target_os = "매크로", target_arch = "x86_64")))]
        pub fn readdir(s: *mut DIR) -> *const dirent;

        // https://github.com/rust-lang/libc/issues/414 및
// stat(2) 에 관한 macOS man 페이지의 _DARWIN_FEATURE_64_BIT_INODE 섹션을 참고하세요.
```

```
 //
 // ' 이 업데이트가 제공되기 전에 존재했던 플랫폼은'
 // Intel 및 PowerPC의 macOS (iOS/wearOS 등이 아님)를 의미합니다.
        #[cfg(all(target_os = "매크로", target_arch = "x86_64"))]
        #[link_name = "readdir$INODE64"]
        pub fn readdir(s: *mut DIR) -> *const dirent;

        pub fn closedir(s: *mut DIR) -> c_int;
    }
}

use std::ffi::{CStr, CString, OsStr, OsString};
use std::os::unix::ffi::OsStrExt;

#[derive(Debug)]
struct DirectoryIterator {
    path: CString,
    dir: *mut ffi::DIR,
}

impl DirectoryIterator {
    fn new(path: &str) -> Result<DirectoryIterator, String> {
        // opendir을 호출하고 제대로 작동하면 Ok 값을 반환하고
        // 그렇지 않으면 메시지와 함께 Err을 반환합니다.
        unimplemented!()
    }
}

impl Iterator for DirectoryIterator {
    type Item = OsString;
    fn next(&mut self) -> Option<OsString> {
        // NULL 포인터를 다시 가져올 때까지 readdir을 계속 호출합니다.
        unimplemented!()
    }
}

impl Drop for DirectoryIterator {
    fn drop(&mut self) {
        // 필요에 따라 closedir을 호출합니다.
        unimplemented!()
    }
}

fn main() -> Result<(), String> {
    let iter = DirectoryIterator::new(".")?;
    println!("파일: {:#?}", iter.collect::<Vec<_>>());
    Ok(())
}
```

30.7.1 해답

```
mod ffi {
    use std::os::raw::{c_char, c_int};
    #[cfg(not(target_os = "매크로"))]
    use std::os::raw::{c_long, c_uchar, c_ulong, c_ushort};

    // 불투명 타입입니다. https://doc.rust-lang.org/nomicon/ffi.html을 참고하세요.
    #[repr(C)]
    pub struct DIR {
        _data: [u8; 0],
        _marker: core::marker::PhantomData<(*mut u8, core::marker::PhantomPinned)>,
    }

    // readdir(3)의 Linux man 페이지에 따른 레이아웃입니다.
    // 여기서 ino_t 및 off_t는
    // /usr/include/x86_64-linux-gnu/{sys/types.h, bits/typesizes.h}의 정의에 따라 확인됩니
    #[cfg(not(target_os = "매크로"))]
    #[repr(C)]
    pub struct dirent {
        pub d_ino: c_ulong,
        pub d_off: c_long,
        pub d_reclen: c_ushort,
        pub d_type: c_uchar,
        pub d_name: [c_char; 256],
    }

    // dir(5)의 macOS man 페이지에 따른 레이아웃입니다.
    #[cfg(all(target_os = "매크로"))]
    #[repr(C)]
    pub struct dirent {
        pub d_fileno: u64,
        pub d_seekoff: u64,
        pub d_reclen: u16,
        pub d_namlen: u16,
        pub d_type: u8,
        pub d_name: [c_char; 1024],
    }

    extern "C" {
        pub fn opendir(s: *const c_char) -> *mut DIR;

        #[cfg(not(all(target_os = "매크로", target_arch = "x86_64")))]
        pub fn readdir(s: *mut DIR) -> *const dirent;

        // https://github.com/rust-lang/libc/issues/414 및
// stat(2)에 관한 macOS man 페이지의 _DARWIN_FEATURE_64_BIT_INODE 섹션을 참고하세요.
//
// ' 이 업데이트가 제공되기 전에 존재했던 플랫폼은'
// Intel 및 PowerPC의 macOS (iOS/wearOS 등이 아님)를 의미합니다.
        #[cfg(all(target_os = "매크로", target_arch = "x86_64"))]
```

```
        #[link_name = "readdir$INODE64"]
        pub fn readdir(s: *mut DIR) -> *const dirent;

        pub fn closedir(s: *mut DIR) -> c_int;
    }
}

use std::ffi::{CStr, CString, OsStr, OsString};
use std::os::unix::ffi::OsStrExt;

#[derive(Debug)]
struct DirectoryIterator {
    path: CString,
    dir: *mut ffi::DIR,
}

impl DirectoryIterator {
    fn new(path: &str) -> Result<DirectoryIterator, String> {
        // opendir을 호출하고 제대로 작동하면 Ok 값을 반환하고
        // 그렇지 않으면 메시지와 함께 Err을 반환합니다.
        let path =
            CString::new(path).map_err(|err| format!("잘못된 경로: {err}"))?;
        // SAFETY: path.as_ptr()은 NULL일 수 없습니다.
        let dir = unsafe { ffi::opendir(path.as_ptr()) };
        if dir.is_null() {
            Err(format!("{:?}을(를) 열 수 없습니다.", path))
        } else {
            Ok(DirectoryIterator { path, dir })
        }
    }
}

impl Iterator for DirectoryIterator {
    type Item = OsString;
    fn next(&mut self) -> Option<OsString> {
        // NULL 포인터를 다시 얻을 때까지 readdir을 계속 호출합니다.
        // SAFETY: self.dir은 NULL이 아닙니다.
        let dirent = unsafe { ffi::readdir(self.dir) };
        if dirent.is_null() {
            // 디렉터리의 끝에 도달했습니다.
            return None;
        }
        // SAFETY: dirent는 NULL이 아니며 dirent.d_name은 NUL
        // 종료됩니다.
        let d_name = unsafe { CStr::from_ptr((*dirent).d_name.as_ptr()) };
        let os_str = OsStr::from_bytes(d_name.to_bytes());
        Some(os_str.to_owned())
    }
}

impl Drop for DirectoryIterator {
```

```
    fn drop(&mut self) {
        // 필요에 따라 closedir을 호출합니다.
        if !self.dir.is_null() {
            // SAFETY: self.dir은 NULL이 아닙니다.
            if unsafe { ffi::closedir(self.dir) } != 0 {
                panic!("{:?}을(를) 닫을 수 없습니다.", self.path);
            }
        }
    }
}

fn main() -> Result<(), String> {
    let iter = DirectoryIterator::new(".")?;
    println!("파일: {:#?}", iter.collect::<Vec<_>>());
    Ok(())
}

#[cfg(test)]
mod tests {
    use super::*;
    use std::error::Error;

    #[test]
    fn test_nonexisting_directory() {
        let iter = DirectoryIterator::new("no-such-directory");
        assert!(iter.is_err());
    }

    #[test]
    fn test_empty_directory() -> Result<(), Box<dyn Error>> {
        let tmp = tempfile::TempDir::new()?;
        let iter = DirectoryIterator::new(
            tmp.path().to_str().ok_or("경로에 UTF-8이 아닌 문자가 있음")?,
        )?;
        let mut entries = iter.collect::<Vec<_>>();
        entries.sort();
        assert_eq!(entries, &[".", ".."]);
        Ok(())
    }

    #[test]
    fn test_nonempty_directory() -> Result<(), Box<dyn Error>> {
        let tmp = tempfile::TempDir::new()?;
        std::fs::write(tmp.path().join("foo.txt"), "Foo 다이어리\n")?;
        std::fs::write(tmp.path().join("bar.png"), "<PNG>\n")?;
        std::fs::write(tmp.path().join("crab.rs"), "//! Crab\n")?;
        let iter = DirectoryIterator::new(
            tmp.path().to_str().ok_or("경로에 UTF-8이 아닌 문자가 있음")?,
        )?;
        let mut entries = iter.collect::<Vec<_>>();
        entries.sort();
```

```
        assert_eq!(entries, &[".", "..", "bar.png", "crab.rs", "foo.txt"]);
        Ok(())
    }
}
```

제 IX 편

안드로이드

제 31 장

Welcome to Rust in Android

Rust is supported for system software on Android. This means that you can write new services, libraries, drivers or even firmware in Rust (or improve existing code as needed).

> 우리는 오늘 여러분의 프로젝트에서 러스트 코드를 호출해볼 것입니다. 그프로젝트에서 러스트로 옮길만 한 작은 부분을 정하세요. 의존성이 적고"특이한" 타입이 적을 수록 좋습니다. 바이트 몇 개를 파싱하는 코드라면완벽합니다.

Android 에서 Rust 사용이 늘어난 점을 감안할 때 발표자는 다음 내용을언급할 수 있습니다.

- 서비스 예: DNS over HTTP
- 라이브러리: Rutabaga 가상 그래픽 인터페이스
- 커널 드라이버: 바인더
- 펌웨어: pKVM 펌웨어

제 32 장

설치

We will be using a Cuttlefish Android Virtual Device to test our code. Make sure you have access to one or create a new one with:

```
source build/envsetup.sh
lunch aosp_cf_x86_64_phone-trunk_staging-userdebug
acloud create
```

자세한 내용은 Android Developer Codelab 을 참조하십시오.

키 포인트:

- Cuttlefish is a reference Android device designed to work on generic Linux desktops. MacOS support is also planned.
- Cuttlefish 는 실제 하드웨어를 매우 충실히 재현하고 있으며, Rust 를사용해 보기에이상적인 에뮬레이터입니다.

제 33 장

빌드규칙

안드로이드 빌드 시스템 (Soong) 은 다음과 같은 여러 모듈을 통해 러스트를지원합니다:

Module Type	Description
`rust_binary`	러스트 바이너리를 생성합니다.
`rust_library`	러스트 라이브러리 (rlib 혹은 dylib) 를생성합니다.
`rust_ffi`	cc 모듈에서 사용할 수 있는 C library (정적 혹은 동적) 를 생성합니다.
`rust_proc_macro`	`proc-macro` 를 구현하는 러스트라이브러리를 생성합니다. 컴파일러의 플러그인으로 생각해도 좋습니다.
`rust_test`	표준 러스트 테스트 러너를 사용하는 테스트바이너리를 생성합니다.
`rust_fuzz`	`libfuzzer` 를 사용하여 fuzz 바이너리를생성합니다.
`rust_protobuf`	프로토버프 (protobuf) 인터페이스를 제공하는러스트 라이브러리를 생성합니다.
`rust_bindgen`	C 라이브러리에 대한 러스트 바인딩을 제공하는러스트 라이브러리를 생성합니다.

다음은 `rust_binary` 와 `rust_library` 를 살펴봅니다.

발표자가 언급할 수 있는 추가 항목:

- Cargo 는 다국어 저장소에 최적화되지 않았으며 인터넷에서 패키지를다운로드합니다.
- Android 에서는 규정상, 그리고 빌드 속도를 위해, 크레이트들이 Android 소스코드 트리 안에 미리 포함되어 있어야 합니다. 빌드 시 다운로드 받을수 없습니다. 또한 C/C++/Java 코드와 상호 운용되어야 합니다. Android 빌드 시스템인 Soong 이 이 공백을 메웁니다.
- Soong 은 Blaze(google3 에서 사용됨) 의 오픈소스 변형인 Bazel 과 많이비슷합니다.
- Android, ChromeOS, Fuchsia 를 Bazel 로 전환할 계획입니다.
- Bazel 과 유사한 빌드 규칙을 배우는 것은 모든 Rust OS 개발자에게유용합니다.
- 재미있는 사실: 스타트렉의 캐릭터 중 한명인 데이터 (Data) 는 사실 Soong 타입의 안드로이드 (Android) 입니다.

33.1 러스트바이너리

간단한 응용 프로그램으로 시작해 보겠습니다. AOSP 체크아웃의 루트에서다음 파일을 생성합니다:

hello_rust/Android.bp:

```
rust_binary {
    name: "hello_rust",
    crate_name: "hello_rust",
    srcs: ["src/main.rs"],
}
```

hello_rust/src/main.rs:

```
//! Rust 데모입니다.

/// 인사말을 표준 출력으로 인쇄합니다.
fn main() {
    println!("Hello from Rust!");
}
```

그런 다음, 이 바이너리를 빌드하고, 가상 디바이스에 넣고, 실행합니다:

```
m hello_rust
adb push "$ANDROID_PRODUCT_OUT/system/bin/hello_rust" /data/local/tmp
adb shell /data/local/tmp/hello_rust
```

```
Hello from Rust!
```

33.2 러스트라이브러리

`rust_library`를 사용하여 안드로이드용 새 러스트 라이브러리를만듭니다.

여기서 두 개의 라이브러리에 대한 의존성을 선언합니다:

- 아래에 정의한 `libgreeting`.
- `external/rust/crates/`에존재하는 `libtextwrap`.

hello_rust/Android.bp:

```
rust_binary {
    name: "hello_rust_with_dep",
    crate_name: "hello_rust_with_dep",
    srcs: ["src/main.rs"],
    rustlibs: [
        "libgreetings",
        "libtextwrap",
    ],
    prefer_rlib: true, // Need this to avoid dynamic link error.
}

rust_library {
    name: "libgreetings",
    crate_name: "인사말",
    srcs: ["src/lib.rs"],
}
```

hello_rust/src/main.rs:

```
//! Rust 데모입니다.

use greetings::greeting;
use textwrap::fill;

/// 인사말을 표준 출력으로 인쇄합니다.
fn main() {
    println!("{}", fill(&greeting("Bob"), 24));
}
```

hello_rust/src/lib.rs:

```
//! 인사말 라이브러리입니다.

/// `name`에게 인사합니다.
pub fn greeting(name: &str) -> String {
    format!("{name}님, 안녕하세요. 만나서 반갑습니다.")
}
```

이전처럼, 빌드하고, 가상 디바이스로 넣고, 실행합니다:

```
m hello_rust_with_dep
adb push "$ANDROID_PRODUCT_OUT/system/bin/hello_rust_with_dep" /data/local/tmp
adb shell /data/local/tmp/hello_rust_with_dep
```

```
Hello Bob, it is very
nice to meet you!
```

제 34 장

AIDL

러스트는안드로이드인터페이스 정의 언어 (AIDL) 를 지원합니다:

- 러스트 코드에서 기존 AIDL 서버를 호출 할 수 있습니다.
- 러스트에서 새로운 AIDL 서버를 생성할 수 있습니다.

34.1 /** 생일 서비스 인터페이스입니다. */

To illustrate how to use Rust with Binder, we're going to walk through the process of creating a Binder interface. We're then going to both implement the described service and write client code that talks to that service.

34.1.1 AIDL 인터페이스

AIDL 인터페이스를 이용해서 서비스의 API 를 선언합니다:

birthday_service/aidl/com/example/birthdayservice/IBirthdayService.aidl:

```
/** 생일 서비스 인터페이스입니다. */
interface IBirthdayService {
    /** 생일 축하 메시지를 생성합니다. */
    String wishHappyBirthday(String name, int years);
}
```

birthday_service/aidl/Android.bp:

```
aidl_interface {
    name: "com.example.birthdayservice",
    srcs: ["com/example/birthdayservice/*.aidl"],
    unstable: true,
    backend: {
        rust: { // Rust는 기본적으로 사용 설정되지 않습니다.
            enabled: true,
        },
    },
}
```

- Note that the directory structure under the `aidl/` directory needs to match the package name used in the AIDL file, i.e. the package is `com.example.birthdayservice` and the file is at `aidl/com/example/IBirthdayService.aidl`.

34.1.2 Generated Service API

Binder generates a trait corresponding to the interface definition. trait to talk to the service.

birthday_service/aidl/com/example/birthdayservice/IBirthdayService.aidl:

```
/** 생일 서비스 인터페이스입니다. */
interface IBirthdayService {
    /** 생일 축하 메시지를 생성합니다. */
    String wishHappyBirthday(String name, int years);
}
```

Generated trait:

```
trait IBirthdayService {
    fn wishHappyBirthday(&self, name: &str, years: i32) -> binder::Result<String>;
}
```

Your service will need to implement this trait, and your client will use this trait to talk to the service.

- The generated bindings can be found at `out/soong/.intermediates/<path to module>/`.
- Point out how the generated function signature, specifically the argument and return types, correspond the interface definition.
  - `String` for an argument results in a different Rust type than `String` as a return type.

34.1.3 서비스구현

이제 AIDL 서비스를 구현할 수 있습니다:

birthday_service/src/lib.rs:

```
use com_example_birthdayservice::aidl::com::example::birthdayservice::IBirthdayService::
use com_example_birthdayservice::binder;

/// The `IBirthdayService` implementation.
pub struct BirthdayService;

impl binder::Interface for BirthdayService {}

impl IBirthdayService for BirthdayService {
    fn wishHappyBirthday(&self, name: &str, years: i32) -> binder::Result<String> {
        Ok(format!("{name}님의 생일을 축하합니다. {years}주년을 축하합니다."))
    }
}
```

birthday_service/Android.bp:

```
rust_library {
    name: "libbirthdayservice",
```

```
    srcs: ["src/lib.rs"],
    crate_name: "birthdayservice",
    rustlibs: [
        "com.example.birthdayservice-rust",
        "libbinder_rs",
    ],
}
```

- Point out the path to the generated `IBirthdayService` trait, and explain why each of the segments is necessary.
- TODO: What does the `binder::Interface` trait do? Are there methods to override? Where source?

34.1.4 AIDL 서버

마지막으로 서비스를 제공하는 서버를 만들 수 있습니다:

birthday_service/src/server.rs:

```
//! Birthday service.
use birthdayservice::BirthdayService;
use com_example_birthdayservice::aidl::com::example::birthdayservice::IBirthdayService::
use com_example_birthdayservice::binder;

const SERVICE_IDENTIFIER: &str = "birthdayservice";

/// 생일 서비스의 진입점입니다.
fn main() {
    let birthday_service = BirthdayService;
    let birthday_service_binder = BnBirthdayService::new_binder(
        birthday_service,
        binder::BinderFeatures::default(),
    );
    binder::add_service(SERVICE_IDENTIFIER, birthday_service_binder.as_binder())
        .expect("서비스 등록 실패");
    binder::ProcessState::join_thread_pool()
}
```

birthday_service/Android.bp:

```
rust_binary {
    name: "birthday_server",
    crate_name: "birthday_server",
    srcs: ["src/server.rs"],
    rustlibs: [
        "com.example.birthdayservice-rust",
        "libbinder_rs",
        "libbirthdayservice",
    ],
    prefer_rlib: true, // To avoid dynamic link error.
}
```

The process for taking a user-defined service implementation (in this case the `BirthdayService` type, which implements the `IBirthdayService`) and starting it as a Binder service has

multiple steps, and may appear more complicated than students are used to if they've used Binder from C++ or another language. Explain to students why each step is necessary.

1. Create an instance of your service type (`BirthdayService`).
2. Wrap the service object in corresponding `Bn*` type (`BnBirthdayService` in this case). This type is generated by Binder and provides the common Binder functionality that would be provided by the `BnBinder` base class in C++. We don't have inheritance in Rust, so instead we use composition, putting our `BirthdayService` within the generated `BnBinderService`.
3. Call `add_service`, giving it a service identifier and your service object (the `BnBirthdayService` object in the example).
4. Call `join_thread_pool` to add the current thread to Binder's thread pool and start listening for connections.

34.1.5 배포

서비스를 빌드하고, 가상 디바이스에 넣고, 시작 할 수 있습니다:

```
m birthday_server
adb push "$ANDROID_PRODUCT_OUT/system/bin/birthday_server" /data/local/tmp
adb root
adb shell /data/local/tmp/birthday_server
```

다른 터미널을 띄워서 서비스가 잘 수행되고 있는지 확인합니다:

```
adb shell service check birthdayservice
```

```
Service birthdayservice: found
```

`service call` 명령어로 서비스를 호출할 수도 있습니다:

```
adb shell service call birthdayservice 1 s16 Bob i32 24
```

```
Result: Parcel(
  0x00000000: 00000000 00000036 00610048 00700070 '....6...H.a.p.p.'
  0x00000010: 00200079 00690042 00740072 00640068 'y. .B.i.r.t.h.d.'
  0x00000020: 00790061 00420020 0062006f 0020002c 'a.y. .B.o.b.,. .'
  0x00000030: 006f0063 0067006e 00610072 00750074 'c.o.n.g.r.a.t.u.'
  0x00000040: 0061006c 00690074 006e006f 00200073 'l.a.t.i.o.n.s. .'
  0x00000050: 00690077 00680074 00740020 00650068 'w.i.t.h. .t.h.e.'
  0x00000060: 00320020 00200034 00650079 00720061 ' .2.4. .y.e.a.r.'
  0x00000070: 00210073 00000000                   's.!.....        ')
```

34.1.6 AIDL 클라이언트

마지막으로, 아까 추가한 서비스에 대한 클라이언트를 러스트로만들겠습니다.

birthday_service/src/client.rs:

```
use com_example_birthdayservice::aidl::com::example::birthdayservice::IBirthdayService:
use com_example_birthdayservice::binder;

const SERVICE_IDENTIFIER: &str = "birthdayservice";

/// 생일 서비스를 호출합니다.
fn main() -> Result<(), Box<dyn Error>> {
```

```
    let name = std::env::args().nth(1).unwrap_or_else(|| String::from("Bob"));
    let years = std::env::args()
        .nth(2)
        .and_then(|arg| arg.parse::<i32>().ok())
        .unwrap_or(42);

    binder::ProcessState::start_thread_pool();
    let service = binder::get_interface::<dyn IBirthdayService>(SERVICE_IDENTIFIER)
        .map_err(|_| "BirthdayService에 연결할 수 없습니다.")?;

    // Call the service.
    let msg = service.wishHappyBirthday(&name, years)?;
    println!("{msg}");
}
```

birthday_service/Android.bp:

```
rust_binary {
    name: "birthday_client",
    crate_name: "birthday_client",
    srcs: ["src/client.rs"],
    rustlibs: [
        "com.example.birthdayservice-rust",
        "libbinder_rs",
    ],
    prefer_rlib: true, // To avoid dynamic link error.
}
```

클라이언트는 `libbirthdayservice`에 의존하지 않음에 주목하세요.

빌드하고, 가상 디바이스로 넣고, 실행합니다:

```
m birthday_client
adb push "$ANDROID_PRODUCT_OUT/system/bin/birthday_client" /data/local/tmp
adb shell /data/local/tmp/birthday_client Charlie 60
```

```
Happy Birthday Charlie, congratulations with the 60 years!
```

- `Strong<dyn IBirthdayService>` is the trait object representing the service that the client has connected to.
  - `Strong` is a custom smart pointer type for Binder. It handles both an in-process ref count for the service trait object, and the global Binder ref count that tracks how many processes have a reference to the object.
  - Note that the trait object that the client uses to talk to the service uses the exact same trait that the server implements. For a given Binder interface, there is a single Rust trait generated that both client and server use.
- Use the same service identifier used when registering the service. This should ideally be defined in a common crate that both the client and server can depend on.

34.1.7 API 수정

API를 확장하여 더 많은 기능을 제공해 봅시다. 클라이언트가 생일 카드에담길 내용을 지정할 수 있도록 하겠습니다:

```
package com.example.birthdayservice;

/** 생일 서비스 인터페이스입니다. */
interface IBirthdayService {
    /** 생일 축하 메시지를 생성합니다. */
    String wishHappyBirthday(String name, int years, in String[] text);
}
```

This results in an updated trait definition for `IBirthdayService`:

```
trait IBirthdayService {
    fn wishHappyBirthday(
        &self,
        name: &str,
        years: i32,
        text: &[String],
    ) -> binder::Result<String>;
}
```

- Note how the `String[]` in the AIDL definition is translated as a `&[String]` in Rust, i.e. that idiomatic Rust types are used in the generated bindings wherever possible:
  - `in` array arguments are translated to slices.
  - `out` and `inout` args are translated to `&mut Vec<T>`.
  - Return values are translated to returning a `Vec<T>`.

34.1.8 Updating Client and Service

Update the client and server code to account for the new API.

birthday_service/src/lib.rs:

```
impl IBirthdayService for BirthdayService {
    fn wishHappyBirthday(
        &self,
        name: &str,
        years: i32,
        text: &[String],
    ) -> binder::Result<String> {
        let mut msg = format!(
            "{name}님의 생일을 축하합니다. {years}주년을 축하합니다.",
        );

        for line in text {
            msg.push('\n');
            msg.push_str(line);
        }

        Ok(msg)
    }
}
```

birthday_service/src/client.rs:

```
let msg = service.wishHappyBirthday(
    &name,
```

```
    years,
    &[
        String::from("Habby birfday to yuuuuu"),
        String::from("And also: many more"),
    ],
)?;
```

- TODO: Move code snippets into project files where they'll actually be built?

34.2 Working With AIDL Types

AIDL types translate into the appropriate idiomatic Rust type:

- Primitive types map (mostly) to idiomatic Rust types.
- Collection types like slices, `Vec`s and string types are supported.
- References to AIDL objects and file handles can be sent between clients and services.
- File handles and parcelables are fully supported.

34.2.1 Primitive Types

Primitive types map (mostly) idiomatically:

AIDL Type	Rust Type	Note
`boolean`	`bool`	
`byte`	`i8`	Note that bytes are signed.
`char`	`u16`	Note the usage of u16, NOT u32.
`int`	`i32`	
`long`	`i64`	
`float`	`f32`	
`double`	`f64`	
`String`	`String`	

34.2.2 배열

The array types (`T[]`, `byte[]`, and `List<T>`) get translated to the appropriate Rust array type depending on how they are used in the function signature:

Position	Rust Type
`in` argument	`&[T]`
`out`/`inout` argument	`&mut Vec<T>`
Return	`Vec<T>`

- In Android 13 or higher, fixed-size arrays are supported, i.e. `T[N]` becomes `[T; N]`. Fixed-size arrays can have multiple dimensions (e.g. int[3][4]). In the Java backend, fixed-size arrays are represented as array types.
- Arrays in parcelable fields always get translated to `Vec<T>`.

34.2.3 트레잇객체

AIDL objects can be sent either as a concrete AIDL type or as the type-erased `IBinder` interface:

birthday_service/aidl/com/example/birthdayservice/IBirthdayInfoProvider.aidl:

```
package com.example.birthdayservice;

interface IBirthdayInfoProvider {
    String name();
    int years();
}
```

birthday_service/aidl/com/example/birthdayservice/IBirthdayService.aidl:

```
import com.example.birthdayservice.IBirthdayInfoProvider;

interface IBirthdayService {
    /** The same thing, but using a binder object. */
    String wishWithProvider(IBirthdayInfoProvider provider);

    /** The same thing, but using `IBinder`. */
    String wishWithErasedProvider(IBinder provider);
}
```

birthday_service/src/client.rs:

```
/// Rust struct implementing the `IBirthdayInfoProvider` interface.
struct InfoProvider {
    name: String,
    age: u8,
}

impl binder::Interface for InfoProvider {}

impl IBirthdayInfoProvider for InfoProvider {
    fn name(&self) -> binder::Result<String> {
        Ok(self.name.clone())
    }

    fn years(&self) -> binder::Result<i32> {
        Ok(self.age as i32)
    }
}

fn main() {
    binder::ProcessState::start_thread_pool();
    let service = connect().expect("BirthdayService에 연결할 수 없습니다.");

    // Create a binder object for the `IBirthdayInfoProvider` interface.
    let provider = BnBirthdayInfoProvider::new_binder(
        InfoProvider { name: name.clone(), age: years as u8 },
        BinderFeatures::default(),
    );
```

```
    // Send the binder object to the service.
    service.wishWithProvider(&provider)?;

    // Perform the same operation but passing the provider as an `SpIBinder`.
    service.wishWithErasedProvider(&provider.as_binder())?;
}
```

- Note the usage of `BnBirthdayInfoProvider`. This serves the same purpose as `BnBirthdayService` that we saw previously.

34.2.4 변수

Binder for Rust supports sending parcelables directly:

birthday_service/aidl/com/example/birthdayservice/BirthdayInfo.aidl:

```
package com.example.birthdayservice;

parcelable BirthdayInfo {
    String name;
    int years;
}
```

birthday_service/aidl/com/example/birthdayservice/IBirthdayService.aidl:

```
import com.example.birthdayservice.BirthdayInfo;

interface IBirthdayService {
    /** The same thing, but with a parcelable. */
    String wishWithInfo(in BirthdayInfo info);
}
```

birthday_service/src/client.rs:

```
fn main() {
    binder::ProcessState::start_thread_pool();
    let service = connect().expect("BirthdayService에 연결할 수 없습니다.");

    service.wishWithInfo(&BirthdayInfo { name: name.clone(), years })?;
}
```

34.2.5 Sending Files

Files can be sent between Binder clients/servers using the `ParcelFileDescriptor` type:

birthday_service/aidl/com/example/birthdayservice/IBirthdayService.aidl:

```
interface IBirthdayService {
    /** The same thing, but loads info from a file. */
    String wishFromFile(in ParcelFileDescriptor infoFile);
}
```

birthday_service/src/client.rs:

```
fn main() {
    binder::ProcessState::start_thread_pool();
```

```
    let service = connect().expect("BirthdayService에 연결할 수 없습니다.");

    // Open a file and put the birthday info in it.
    let mut file = File::create("/data/local/tmp/birthday.info").unwrap();
    writeln!(file, "{name}")?;
    writeln!(file, "{years}")?;

    // Create a `ParcelFileDescriptor` from the file and send it.
    let file = ParcelFileDescriptor::new(file);
    service.wishFromFile(&file)?;
}
```

birthday_service/src/lib.rs:

```
impl IBirthdayService for BirthdayService {
    fn wishFromFile(
        &self,
        info_file: &ParcelFileDescriptor,
    ) -> binder::Result<String> {
        // Convert the file descriptor to a `File`. `ParcelFileDescriptor` wraps
        // an `OwnedFd`, which can be cloned and then used to create a `File`
        // object.
        let mut info_file = info_file
            .as_ref()
            .try_clone()
            .map(File::from)
            .expect("Invalid file handle");

        let mut contents = String::new();
        info_file.read_to_string(&mut contents).unwrap();

        let mut lines = contents.lines();
        let name = lines.next().unwrap();
        let years: i32 = lines.next().unwrap().parse().unwrap();

        Ok(format!("{name}님의 생일을 축하합니다. {years}주년을 축하합니다."))
    }
}
```

- `ParcelFileDescriptor` wraps an `OwnedFd`, and so can be created from a `File` (or any other type that wraps an `OwnedFd`), and can be used to create a new `File` handle on the other side.
- Other types of file descriptors can be wrapped and sent, e.g. TCP, UDP, and UNIX sockets.

제 35 장

Testing in Android

Building on Testing, we will now look at how unit tests work in AOSP. Use the `rust_test` module for your unit tests:

testing/Android.bp:

```
rust_library {
    name: "libleftpad",
    crate_name: "leftpad",
    srcs: ["src/lib.rs"],
}

rust_test {
    name: "libleftpad_test",
    crate_name: "leftpad_test",
    srcs: ["src/lib.rs"],
    host_supported: true,
    test_suites: ["general-tests"],
}
```

testing/src/lib.rs:

```
//! Left-padding library.

/// Left-pad `s` to `width`.
pub fn leftpad(s: &str, width: usize) -> String {
    format!("{s:>width$}")
}

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

    #[test]
    fn short_string() {
        assert_eq!(leftpad("foo", 5), "  foo");
    }
```

```
    #[test]
    fn long_string() {
        assert_eq!(leftpad("foobar", 6), "foobar");
    }
}
```

You can now run the test with

```
atest --host libleftpad_test
```

The output looks like this:

```
INFO: Elapsed time: 2.666s, Critical Path: 2.40s
INFO: 3 processes: 2 internal, 1 linux-sandbox.
INFO: Build completed successfully, 3 total actions
//comprehensive-rust-android/testing:libleftpad_test_host            PASSED in 2.3s
    PASSED  libleftpad_test.tests::long_string (0.0s)
    PASSED  libleftpad_test.tests::short_string (0.0s)
Test cases: finished with 2 passing and 0 failing out of 2 test cases
```

Notice how you only mention the root of the library crate. Tests are found recursively in nested modules.

35.1 GoogleTest

The GoogleTest crate allows for flexible test assertions using *matchers*:

```
use googletest::prelude::*;

#[googletest::test]
fn test_elements_are() {
    let value = vec!["foo", "bar", "baz"];
    expect_that!(value, elements_are!(eq("foo"), lt("xyz"), starts_with("b")));
}
```

마지막 요소를 '!' 로 변경하면테스트가 실패하고 오류를 정확히 가리키는 구조화된 오류 메시지가표시됩니다.

```
---- test_elements_are stdout ----
Value of: value
Expected: has elements:
  0. is equal to "foo"
  1. is less than "xyz"
  2. starts with prefix "!"
Actual: ["foo", "bar", "baz"],
  where element #2 is "baz", which does not start with "!"
  at src/testing/googletest.rs:6:5
Error: See failure output above
```

- GoogleTest 는 Rust 플레이그라운드의 일부가 아니므로 로컬 환경에서 이예를 실행해야 합니다. `cargo add googletest` 를 사용하여 기존 Cargo 프로젝트에 빠르게 추가하세요.
- `use googletest::prelude::*;` 줄은일반적으로사용되는 매크로 및 타입을 여러 개 가져옵니다.
- 이는 일부일 뿐이며 내장된 매처가 많이 있습니다.

- A particularly nice feature is that mismatches in multi-line strings are shown as a diff:

```
#[test]
fn test_multiline_string_diff() {
    let haiku = "Memory safete found,\n\
                 Rust's strong typing guides the way,\n\
                 Secure code you'll write.";
    assert_that!(
        haiku,
        eq("Memory safety found,\n\
            Rust's silly humor guides the way,\n\
            Secure code you'll write.")
    );
}
```

색상으로 구분된 diff 를 표시합니다 (여기에서는 색상이 표시되지 않습니다).

```
    Value of: haiku
Expected: is equal to "Memory safety found,\nRust's silly humor guides the way,\nSecure
Actual: "Memory safety found,\nRust's strong typing guides the way,\nSecure code you'll
  which isn't equal to "Memory safety found,\nRust's silly humor guides the way,\nSecure
Difference(-actual / +expected):
 Memory safety found,
-Rust's strong typing guides the way,
+Rust's silly humor guides the way,
 Secure code you'll write.
  at src/testing/googletest.rs:17:5
```

- 크레이트는 C++용 GoogleTest 의 Rust 포트입니다.

35.2 모의처리

모의 처리의 경우 널리 사용되는 라이브러리인 Mockall 이 있습니다. 트레잇을 사용하도록코드를 리팩터링해야 합니다. 그런 다음 빠르게 모의 처리할 수 있습니다.

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

#[mockall::automock]
pub trait Pet {
    fn is_hungry(&self, since_last_meal: Duration) -> bool;
}

#[test]
fn test_robot_dog() {
    let mut mock_dog = MockPet::new();
    mock_dog.expect_is_hungry().return_const(true);
    assert_eq!(mock_dog.is_hungry(Duration::from_secs(10)), true);
}
```

- Mockall is the recommended mocking library in Android (AOSP). There are other mocking libraries available on crates.io, in particular in the area of mocking HTTP services. The other mocking libraries work in a similar fashion as Mockall, meaning that they make it easy to get a mock implementation of a given trait.

- 모의 처리는 다소 . 모의를 사용하면테스트를 종속 항목에서 완전히 분리할 수 있습니다. 그 결과 테스트실행이 더욱 빠르고 안정적으로 이루어집니다. 반면에 모의는 잘못구성되어 실제 종속 항목이 실행하는 것과 다른 출력을 반환할 수있습니다.

  가능하면 실제 종속 항목을 사용하는 것이 좋습니다. 예를 들어 많은데이터베이스에서는 메모리 내 백엔드를 구성할 수 있습니다. 즉, 테스트에서 올바른 동작을 가져올 뿐만 아니라 속도가 빠르고 자동으로정리됩니다.

  마찬가지로 많은 웹 프레임워크에서도 `localhost` 의 임의 포트에바인딩되는 프로세스 내 서버를 시작할 수 있습니다. 프레임워크를 모의처리하는 것보다는 항상 이 방법을 사용하는 것이 좋습니다. 실제 환경에서코드를 테스트하는 데 도움이 됩니다.

- Mockall 은 Rust 플레이그라운드의 일부가 아니므로 로컬 환경에서 이 예를실행해야 합니다. Mockall 을 기존 Cargo 프로젝트에 빠르게 추가하려면 `cargo add mockall` 을 사용합니다.
- Mockall 에는 더 많은 기능이 있습니다. 특히 전달된 인수에 따라 기대치를설정할 수 있습니다. 여기서는 마지막으로 먹이를 먹고 3 시간이 지나면배고파지는 고양이를 모의하는 데 이를 사용합니다.

```
#[test]
fn test_robot_cat() {
    let mut mock_cat = MockPet::new();
    mock_cat
        .expect_is_hungry()
        .with(mockall::predicate::gt(Duration::from_secs(3 * 3600)))
        .return_const(true);
    mock_cat.expect_is_hungry().return_const(false);
    assert_eq!(mock_cat.is_hungry(Duration::from_secs(1 * 3600)), false);
    assert_eq!(mock_cat.is_hungry(Duration::from_secs(5 * 3600)), true);
}
```

- `.times(n)` 를 사용하여 모의 메서드가 호출될 수 있는 횟수를 `n` 으로 제한할 수 있습니다. 이 조건이 충족되지 않으면 모의메서드가 삭제될 때 자동으로 패닉 상태가 됩니다.

제 36 장

로깅

log 크레이트를 사용하면 안드로이드 디바이스 안에서 수행될때에는 logcat 으로, 호스트에서 수행될 때에는 stdout 으로로그가 자동으로 출력이 되도록 할 수 있습니다:

hello_rust_logs/Android.bp:

```
rust_binary {
    name: "hello_rust_logs",
    crate_name: "hello_rust_logs",
    srcs: ["src/main.rs"],
    rustlibs: [
        "liblog_rust",
        "liblogger",
    ],
    host_supported: true,
}
```

hello_rust_logs/src/main.rs:

```
//! Rust 로깅 데모입니다.

use log::{debug, error, info};

/// 인사말을 기록합니다.
fn main() {
    logger::init(
        logger::Config::default()
            .with_tag_on_device("rust")
            .with_min_level(log::Level::Trace),
    );
    debug!("프로그램을 시작하는 중입니다.");
    info!("잘 진행되고 있습니다.");
    error!("문제가 발생했습니다!");
}
```

빌드하고, 가상 디바이스에 넣고, 실행합니다:

```
m hello_rust_logs
adb push "$ANDROID_PRODUCT_OUT/system/bin/hello_rust_logs" /data/local/tmp
```

```
adb shell /data/local/tmp/hello_rust_logs
```

`adb logcat` 커맨드로 로그를 확인합니다:

```
adb logcat -s rust
```

```
09-08 08:38:32.454  2420  2420 D rust: hello_rust_logs: Starting program.
09-08 08:38:32.454  2420  2420 I rust: hello_rust_logs: Things are going fine.
09-08 08:38:32.454  2420  2420 E rust: hello_rust_logs: Something went wrong!
```

제 37 장

상호운용성

러스트는 다음과 같이 다른 언어와의 상호운용성을 훌륭히 지원합니다:

- 타 언어에서 러스트 함수를 호출합니다.
- 타 언어의 함수를 러스트에서 호출합니다.

타 언어의 함수를 호출해서 사용하는 것을 FFI(foreign function interface) 라고 합니다.

37.1 C 와의상호운용성

러스트는 C 호출규약을 따르는 오브젝트 파일과 링킹할 수 있습니다. 또한, 반대로 러스트 함수를 내보내서 C 에서 호출 할 수 도 있습니다.

원한다면 아래와 같이 수동으로 코딩할 수 있습니다:

```
extern "C" {
    fn abs(x: i32) -> i32;
}

fn main() {
    let x = -42;
    let abs_x = unsafe { abs(x) };
    println!("{x}, {abs_x}");
}
```

우리는 이미 Safe FFI 래퍼 연습문제에서 이를 다루었습니다.

> 이러한 방법은 타겟 플랫폼의 모든 부분을 사전에 알고 있다는 전제를 깔고있습니다. 상용 프로젝트에서는 권장하지 않습니다.

좀 더 나은 옵션을 살펴보겠습니다.

37.1.1 Bindgen 사용하기

bindgen 는 C 헤더파일에서 러스트 바인딩을 자동으로 생성하는 도구입니다.

먼저 작은 C 라이브러리를 만들어 보겠습니다:

interoperability/bindgen/libbirthday.h:

```
typedef struct card {
  const char* name;
  int years;
} card;

void print_card(const card* card);
```

interoperability/bindgen/libbirthday.c:

```
#include <stdio.h>
#include "libbirthday.h"

void print_card(const card* card) {
  printf("+--------------\n");
  printf("| %s 님, 생일 축하합니다.\n", card->name);
  printf("| %i 주년을 축하합니다!\n", card->years);
  printf("+--------------\n");
}
```

`Android.bp` 파일에 아래를 추가합니다:

interoperability/bindgen/Android.bp:

```
cc_library {
    name: "libbirthday",
    srcs: ["libbirthday.c"],
}
```

라이브러리에 대한 헤더 파일을 만듭니다 (이 예시에서는 반드시 필요한 것은아닙니다.):

interoperability/bindgen/libbirthday_wrapper.h:

```
#include "libbirthday.h"
```

이제 바인딩을 자동으로 생성할 수 있습니다:

interoperability/bindgen/Android.bp:

```
rust_bindgen {
    name: "libbirthday_bindgen",
    crate_name: "birthday_bindgen",
    wrapper_src: "libbirthday_wrapper.h",
    source_stem: "bindings",
    static_libs: ["libbirthday"],
}
```

마침내, 러스트 프로그램에서 바인딩을 사용할 수 있습니다:

interoperability/bindgen/Android.bp:

```
rust_binary {
    name: "print_birthday_card",
    srcs: ["main.rs"],
    rustlibs: ["libbirthday_bindgen"],
}
```

interoperability/bindgen/main.rs:

```
//! Bindgen 데모입니다.
```

```rust
use birthday_bindgen::{card, print_card};

fn main() {
    let name = std::ffi::CString::new("피터").unwrap();
    let card = card { name: name.as_ptr(), years: 42 };
    // SAFETY: `print_card` is safe to call with a valid `card` pointer.
    unsafe {
        print_card(&card as *const card);
    }
}
```

빌드하고, 가상 디바이스에 넣고, 실행합니다:

```
m print_birthday_card
adb push "$ANDROID_PRODUCT_OUT/system/bin/print_birthday_card" /data/local/tmp
adb shell /data/local/tmp/print_birthday_card
```

마지막으로, 바인딩이 잘 작동하는지 확인하기 위해, 자동 생성된 테스트를실행해 보겠습니다:

interoperability/bindgen/Android.bp:

```
rust_test {
    name: "libbirthday_bindgen_test",
    srcs: [":libbirthday_bindgen"],
    crate_name: "libbirthday_bindgen_test",
    test_suites: ["general-tests"],
    auto_gen_config: true,
    clippy_lints: "none", // 생성된 파일, 린트 작업 건너뛰기
    lints: "none",
}
```

```
atest libbirthday_bindgen_test
```

37.1.2 C 에서 러스트호출

러스트에서 타입과 함수를 C 로 내보내는 것은 간단합니다:

interoperability/rust/libanalyze/analyze.rs

```rust
//! Rust FFI 데모입니다.
#![deny(improper_ctypes_definitions)]

use std::os::raw::c_int;

/// 수치를 분석합니다.
#[no_mangle]
pub extern "C" fn analyze_numbers(x: c_int, y: c_int) {
    if x < y {
        println!("x({x}) 가 가장 작습니다.");
    } else {
        println!("y({y}) 는 x({x}) 보다 클 수 있습니다.");
    }
}
```

interoperability/rust/libanalyze/analyze.h

```
#ifndef ANALYSE_H
#define ANALYSE_H

extern "C" {
void analyze_numbers(int x, int y);
}

#endif
```

interoperability/rust/libanalyze/Android.bp

```
rust_ffi {
    name: "libanalyze_ffi",
    crate_name: "analyze_ffi",
    srcs: ["analyze.rs"],
    include_dirs: ["."],
}
```

이제 이 러스트 함수를 C 바이너리에서 호출할 수 있습니다:

interoperability/rust/analyze/main.c

```
#include "analyze.h"

int main() {
  analyze_numbers(10, 20);
  analyze_numbers(123, 123);
  return 0;
}
```

interoperability/rust/analyze/Android.bp

```
cc_binary {
    name: "analyze_numbers",
    srcs: ["main.c"],
    static_libs: ["libanalyze_ffi"],
}
```

빌드하고, 가상 디바이스에 넣고, 실행합니다:

```
m analyze_numbers
adb push "$ANDROID_PRODUCT_OUT/system/bin/analyze_numbers" /data/local/tmp
adb shell /data/local/tmp/analyze_numbers
```

`#[no_mangle]` 은 러스트의 네임 맹글링 (name mangling) 을비활성화하므로 외부로 노출되는 심볼의 이름은 함수의 이름 그대로가됩니다. 심볼 이름을 바꾸고 싶다면`#[export_name = "some_name"]` 을 사용합니다.

37.2 C++와의상호운용성

CXX 크레이트는 러스트와 C++ 사이의 안전한상호운용성을 가능하게 해줍니다.

전체적인 접근 방법은 다음과 같습니다:

37.2.1 테스트모듈

CXX 는 각 언어에서 다른 언어로 노출되는 함수 서명에 관한 설명을사용합니다. `#[cxx::bridge]` 속성 매크로로 주석이 달린 Rust 모듈에서 extern 블록을 사용하여 이 설명을 제공합니다.

```
#[allow(unsafe_op_in_unsafe_fn)]
#[cxx::bridge(namespace = "org::blobstore")]
mod ffi {
    // 두 언어 모두에 표시되는 필드가 있는 공유 구조체입니다.
    struct BlobMetadata {
        size: usize,
        tags: Vec<String>,
    }

    // C++에 노출된 Rust 타입 및 메서드 시그니쳐입니다.
    extern "Rust" {
        type MultiBuf;

        fn next_chunk(buf: &mut MultiBuf) -> &[u8];
    }

    // Rust에 노출된 C++ 타입 및 함수 시그니쳐입니다.
    unsafe extern "C++" {
        include!("include/blobstore.h");

        type BlobstoreClient;

        fn new_blobstore_client() -> UniquePtr<BlobstoreClient>;
        fn put(self: Pin<&mut BlobstoreClient>, parts: &mut MultiBuf) -> u64;
        fn tag(self: Pin<&mut BlobstoreClient>, blobid: u64, tag: &str);
        fn metadata(&self, blobid: u64) -> BlobMetadata;
    }
}
```

- 브리지는 일반적으로 크레이트 내의 `ffi` 모듈에 선언됩니다.
- 브리지 모듈에서 이루어진 선언으로부터 CXX 는 일치하는 Rust 및 C++ 타입/함수 정의를 생성하여 이러한 항목을 두 언어 모두에 노출합니다.
- 생성된 Rust 코드를 보려면 cargo-expand 를사용하여 확장된 proc 매크로를 확인하세요. 대부분의 예에서는 `cargo expand ::ffi` 를 사용하여 `ffi` 모듈만확장합니다 (Android 프로젝트에는 적용되지 않음).
- 생성된 C++ 코드를 보려면 `target/cxxbridge` 를 확인하세요.

37.2.2 Rust Bridge Declarations

```
#[cxx::bridge]
mod ffi {
    extern "Rust" {
        type MyType; // 불투명 타입입니다.
        fn foo(&self); // `MyType`의 메서드입니다.
        fn bar() -> Box<MyType>; // Free function
    }
}
```

```
struct MyType(i32);

impl MyType {
    fn foo(&self) {
        println!("{}", self.0);
    }
}

fn bar() -> Box<MyType> {
    Box::new(MyType(123))
}
```

- extern "Rust"에 선언된 항목은 상위 모듈의 범위 내에 있는항목을 참조합니다.
- CXX 코드 생성기는 extern "Rust" 섹션을 사용하여 상응하는 C++ 선언이 포함된 C++ 헤더 파일을 생성합니다. 생성된 헤더는 파일 확장자가.rs.h 인 경우를 제외하고 브리지가 포함된 Rust 소스 파일과 동일한 경로를갖습니다.

37.2.3 생성된 C++

```
#[cxx::bridge]
mod ffi {
    // C++에 노출된 Rust 타입 및 메서드 시그니쳐입니다.
    extern "Rust" {
        type MultiBuf;

        fn next_chunk(buf: &mut MultiBuf) -> &[u8];
    }
}
```

그 결과는 대략 다음과 같은 C++입니다.

```
struct MultiBuf final : public ::rust::Opaque {
  ~MultiBuf() = delete;

private:
  friend ::rust::layout;
  struct layout {
    static ::std::size_t size() noexcept;
    static ::std::size_t align() noexcept;
  };
};

::rust::Slice<::std::uint8_t const> next_chunk(::org::blobstore::MultiBuf &buf) noexcept
```

37.2.4 C++ 브리지선언

```
#[cxx::bridge]
mod ffi {
    // Rust에 노출된 C++ 타입 및 함수 시그니쳐입니다.
    unsafe extern "C++" {
        include!("include/blobstore.h");
```

```
        type BlobstoreClient;

        fn new_blobstore_client() -> UniquePtr<BlobstoreClient>;
        fn put(self: Pin<&mut BlobstoreClient>, parts: &mut MultiBuf) -> u64;
        fn tag(self: Pin<&mut BlobstoreClient>, blobid: u64, tag: &str);
        fn metadata(&self, blobid: u64) -> BlobMetadata;
    }
}
```

결과는 대략 다음과 같은 Rust 입니다.

```
#[repr(C)]
pub struct BlobstoreClient {
    _private: ::cxx::private::Opaque,
}

pub fn new_blobstore_client() -> ::cxx::UniquePtr<BlobstoreClient> {
    extern "C" {
        #[link_name = "org$blobstore$cxxbridge1$new_blobstore_client"]
        fn __new_blobstore_client() -> *mut BlobstoreClient;
    }
    unsafe { ::cxx::UniquePtr::from_raw(__new_blobstore_client()) }
}

impl BlobstoreClient {
    pub fn put(&self, parts: &mut MultiBuf) -> u64 {
        extern "C" {
            #[link_name = "org$blobstore$cxxbridge1$BlobstoreClient$put"]
            fn __put(
                _: &BlobstoreClient,
                parts: *mut ::cxx::core::ffi::c_void,
            ) -> u64;
        }
        unsafe {
            __put(self, parts as *mut MultiBuf as *mut ::cxx::core::ffi::c_void)
        }
    }
}

// ...
```

- 프로그래머는 자신이 입력한 시그니쳐가 정확하다고 보장할 필요가없습니다. CXX 는 시그니쳐가 C++에서 선언된 것과 정확히 일치하는지를체크하기 위해 정적으로 assertion 을 실행합니다.
- `unsafe extern` 블록을 사용하면 Rust 에서 안전하게 호출할 수있는 C++ 함수를 선언할 수 있습니다.

37.2.5 공유타입

```
#[cxx::bridge]
mod ffi {
    #[derive(Clone, Debug, Hash)]
```

```
    struct PlayingCard {
        suit: Suit,
        value: u8,  // A=1, J=11, Q=12, K=13
    }

    enum Suit {
        Clubs,
        Diamonds,
        Hearts,
        Spades,
    }
}
```

- C 와 같은 (단위) enum 만 지원됩니다.
- 공유 타입의 `#[derive()]` 에는 제한된 수의 트레잇이지원됩니다. C++ 코드에 대해서도 상응하는 기능이 생성됩니다. 예를 들어 `Hash` 를 파생하면 해당 C++ 타입에 대한 `std::hash` 구현도 생성됩니다.

37.2.6 공유 Enum

```
#[cxx::bridge]
mod ffi {
    enum Suit {
        Clubs,
        Diamonds,
        Hearts,
        Spades,
    }
}
```

생성된 러스트:

```
#[derive(Copy, Clone, PartialEq, Eq)]
#[repr(transparent)]
pub struct Suit {
    pub repr: u8,
}

#[allow(non_upper_case_globals)]
impl Suit {
    pub const Clubs: Self = Suit { repr: 0 };
    pub const Diamonds: Self = Suit { repr: 1 };
    pub const Hearts: Self = Suit { repr: 2 };
    pub const Spades: Self = Suit { repr: 3 };
}
```

Generated C++:

```
enum class Suit : uint8_t {
  Clubs = 0,
  Diamonds = 1,
  Hearts = 2,
  Spades = 3,
};
```

- Rust 측에서, 공유된 enum 에 관해 생성된 코드는 실제로 숫자 값을래핑하는 구조체입니다. 이는 enum 클래스가 나열된 모든 변형과 다른 값을보유하는 것이 C++에서 UB 가 아니고 Rust 표현도 동일한 동작을 가져야하기 때문입니다.

37.2.7 오류처리

```
#[cxx::bridge]
mod ffi {
    extern "Rust" {
        fn fallible(depth: usize) -> Result<String>;
    }
}

fn fallible(depth: usize) -> anyhow::Result<String> {
    if depth == 0 {
        return Err(anyhow::Error::msg("fallible1 에 깊이 > 0 필요"));
    }

    Ok("완료!".into())
}
```

- `Result` 를 반환하는 Rust 함수는 C++ 측에서 예외로 변환됩니다.
- 발생하는 예외는 항상 `rust::Error` 타입이며 주로 오류 메시지문자열을 가져오는 방법을 노출합니다. 오류 메시지는 오류 타입의 `Display` impl 에서 가져옵니다.
- Rust 에서 C++로 패닉이 해제되면 프로세스가 즉시 종료됩니다.

37.2.8 오류처리

```
#[cxx::bridge]
mod ffi {
    unsafe extern "C++" {
        include!("example/include/example.h");
        fn fallible(depth: usize) -> Result<String>;
    }
}

fn main() {
    if let Err(err) = ffi::fallible(99) {
        eprintln!("오류: {}", err);
        process::exit(1);
    }
}
```

- `Result` 를 반환하도록 선언된 C++ 함수는 C++ 측에서 발생한예외를 포착하고 이를 호출 Rust 함수에 `Err` 값으로 반환합니다.
- `Result` 를 반환하도록 CXX 브리지에서 선언하지 않은 extern 'C++' 함수에서 예외가 발생하면 프로그램은 C++의 `std::terminate` 를 호출합니다. 이 동작은 `noexcept` C++ 함수를 통해 발생하는 동일한 예외와 같습니다.

37.2.9 추가타입

Rust Type	C++ Type
`String`	`rust::String`
`&str`	`rust::Str`
`CxxString`	`std::string`
`&[T]/&mut [T]`	`rust::Slice`
`Box<T>`	`rust::Box<T>`
`UniquePtr<T>`	`std::unique_ptr<T>`
`Vec<T>`	`rust::Vec<T>`
`CxxVector<T>`	`std::vector<T>`

- 이러한 타입은 공유 구조체의 필드와 extern 함수의 인수 및 반환에서사용할 수 있습니다.
- Rust 의 `String` 은 `std::string` 에 직접 매핑되지않습니다. 여기에는 다음과 같은 몇 가지 이유가 있습니다.
  - `std::string` 은 `String` 에 필요한 UTF-8 불변값을유지하지 않습니다.
  - 두 타입은 메모리에 서로 다른 레이아웃을 가지고 있으므로 언어 간에직접 전달될 수 없습니다.
  - `std::string` 에는 Rust 의 이동 의미 체계와 일치하지 않는 이동생성자가 필요하므로 `std::string` 을 값으로 Rust 에 전달할 수없습니다.

37.2.10 Building in Android

`cc_library_static` 을 만들어 CXX 에서 생성된 헤더와 소스 파일을비롯하여 C++ 라이브러리를 빌드합니다.

```
cc_library_static {
    name: "libcxx_test_cpp",
    srcs: ["cxx_test.cpp"],
    generated_headers: [
        "cxx-bridge-header",
        "libcxx_test_bridge_header"
    ],
    generated_sources: ["libcxx_test_bridge_code"],
}
```

- `libcxx_test_bridge_header` 및 `libcxx_test_bridge_code` 가 CXX 에서 생성된 C++ 바인딩의 종속항목이라는 점을 지적합니다. 다음 슬라이드에서 설정 방법을 알아봅니다.
- 일반적인 CXX 정의를 가져오려면 `cxx-bridge-header` 라이브러리도사용해야 합니다.
- Android 에서 CXX 를 사용하는 방법에 관한 전체 문서는 Android 문서에서 확인할 수 있습니다. 학생들이 나중에 이 안내를 어디에서 다시찾을 수 있는지 알 수 있도록 이 링크를 수업에 공유하는 것이 좋습니다.

37.2.11 Building in Android

두 개의 genrule 을 만듭니다. 하나는 CXX 헤더를 생성하고 다른 하나는 CXX 소스 파일을 생성합니다. 그런 다음 `cc_library_static` 에 대한입력으로 사용됩니다.

```
// lib.rs 의 Rust 내보내기 함수에 대한
// C++ 바인딩이 포함된 C++ 헤더를 생성합니다.
genrule {
    name: "libcxx_test_bridge_header",
    tools: ["cxxbridge"],
```

```
    cmd: "$(location cxxbridge) $(in) --header > $(out)",
    srcs: ["lib.rs"],
    out: ["lib.rs.h"],
}

// Rust가 호출하는 C++ 코드를 생성합니다.
genrule {
    name: "libcxx_test_bridge_code",
    tools: ["cxxbridge"],
    cmd: "$(location cxxbridge) $(in) > $(out)",
    srcs: ["lib.rs"],
    out: ["lib.rs.cc"],
}
```

- cxxbridge 도구는 C++ 측의 브리지 모듈을 생성하는 독립형도구입니다. 이 도구는 Android 에 포함되어 있으며 Soong 도구로 사용할 수있습니다.
- 일반적으로 Rust 소스 파일이 `lib.rs` 인 경우 헤더 파일의 이름은 `lib.rs.h` 이고 소스 파일의 이름은 `lib.rs.cc` 가 됩니다. 그러나 이 이름 지정 규칙은 강제되지 않습니다.

37.2.12 Building in Android

`libcxx` 및 `cc_library_static` 에 종속되는 `rust_binary` 를 생성합니다.

```
rust_binary {
    name: "cxx_test",
    srcs: ["lib.rs"],
    rustlibs: ["libcxx"],
    static_libs: ["libcxx_test_cpp"],
}
```

37.3 Java 와의상호운용성

자바는 Java Native Interface(JNI) 를 통해 공유 라이브러리를 로드할 수 있습니다. `jni` 크레이트를 사용하여 JNI 라이브러리를 만들 수 있습니다.

먼저, 자바로 내보낼 러스트 함수를 생성합니다:

interoperability/java/src/lib.rs:

```
//! Rust <-> Java FFI 데모입니다.

use jni::objects::{JClass, JString};
use jni::sys::jstring;
use jni::JNIEnv;

/// HelloWorld::hello method implementation.
#[no_mangle]
pub extern "system" fn Java_HelloWorld_hello(
    env: JNIEnv,
    _class: JClass,
    name: JString,
) -> jstring {
```

```
    let input: String = env.get_string(name).unwrap().into();
    let greeting = format!("Hello, {input}!");
    let output = env.new_string(greeting).unwrap();
    output.into_inner()
}
```

interoperability/java/Android.bp:

```
rust_ffi_shared {
    name: "libhello_jni",
    crate_name: "hello_jni",
    srcs: ["src/lib.rs"],
    rustlibs: ["libjni"],
}
```

We then call this function from Java:

interoperability/java/HelloWorld.java:

```
class HelloWorld {
    private static native String hello(String name);

    static {
        System.loadLibrary("hello_jni");
    }

    public static void main(String[] args) {
        String output = HelloWorld.hello("Alice");
        System.out.println(output);
    }
}
```

interoperability/java/Android.bp:

```
java_binary {
    name: "helloworld_jni",
    srcs: ["HelloWorld.java"],
    main_class: "HelloWorld",
    required: ["libhello_jni"],
}
```

마지막으로 바이너리를 빌드, 싱크, 실행합니다:

```
m helloworld_jni
adb sync  # requires adb root && adb remount
adb shell /system/bin/helloworld_jni
```

제 38 장

연습문제

This is a group exercise: We will look at one of the projects you work with and try to integrate some Rust into it. Some suggestions:

- 당신의 AIDL 서비스를 러스트 클라이언트에서 호출해봅니다.
- 당신의 프로젝트의 함수를 러스트로 옮기고 호출해봅니다.

이 연습문제는 열려있기 때문에 해답이 제공되지 않습니다. 클래스에서제출된 코드에 의존합니다.

제 X 편

Chromium

제 39 장

Welcome to Rust in Chromium

Rust 는 Chromium 의 서드 파티 라이브러리에 대해 지원되며, Rust 와 기존 Chromium C++ 코드 간에 연결하는 퍼스트 파티 글루 코드를 포함합니다.

> 오늘은 Rust 를 호출하여 문자열로 우스꽝스러운 작업을 해 보겠습니다. 사용자에게 UTF8 문자열을 표시하는 코드 부분이 있는 경우, 언급되는 정확한부분 대신 코드베이스에서 이 레시피를 따르시면 됩니다.

제 40 장

설치

Chromium 을 빌드하고 실행할 수 있는지 확인합니다. 코드가 비교적최신이라면 (커밋 위치 1223636 이후, 2023 년 11 월에 해당) 어떤 플랫폼이나빌드 플래그 집합도 괜찮습니다.

```
gn gen out/Debug
autoninja -C out/Debug chrome
out/Debug/chrome # or on Mac, out/Debug/Chromium.app/Contents/MacOS/Chromium
```

개발 속도를 빠르게 하고 싶다면 디버그 빌드를 사용하는 것이 좋습니다. 이게 기본값입니다.

아직 빌드하지 않았다면 Chromium 빌드 방법을 참고하세요. 주의: Chromium 빌드 환경 셋업은 시간 이걸립니다.

또한 Visual Studio Code 가 설치되어 있는 것이 좋습니다.

About the exercises

과정의 이 부분에는 서로 연계되는 일련의 연습문제가 있습니다. 마지막에한꺼번에 하지 않고 과정 전반에 연습문제가 흩어져 있습니다. 특정 부분을완료할 시간이 없더라도 걱정하지 마세요. 다음번에 따라잡을 수 있습니다.

제 41 장

Chromium 및 Cargo 생태계비교

Rust 커뮤니티는 일반적으로 crates.io 의 `cargo` 및 라이브러리를 사용합니다. Chromium 은 세심하게 선별된의존성들과 함께 gn, `ninja` 을 이용하여 빌드됩니다.

Rust 로 코드를 작성할 때 선택할 수 있는 옵션은 다음과 같습니다.

- `//build/rust/*.gni` 의 템플릿 (예: 나중에 다룰 `rust_static_library`) 을 통해 gn 및 `ninja` 를사용합니다. 이는 Chromium 의 감사 도구 모음 및 크레이트를 사용합니다.
- `cargo` 를 사용하되 [Chromium 의 감사 도구 모음 및 크레이트로제한] 합니다 (https://chromium.googlesource.com/+/refs/heads/main/docs/rust.md#Using-cargo).
- `cargo` 를 사용하여 도구 모음 또는인터넷에서 다운로드한 크레이트를신뢰합니다.

여기서부터는 gn 과 `ninja` 에 집중할 겁니다. 그 도구들을써야만 Chromium 브라우저에 Rust 코드를 빌드해서 넣을 수 있기 때문입니다. 이와 동시에, Cargo 는 Rust 생태계에서 중요한 부분이므로 Cargo 에도익숙해야 합니다.

Mini exercise

소규모 그룹으로 나눈 다음:

- 'cargo' 가 유리할 수 있는 시나리오를브레인스토밍하고 이러한 시나리오의 위험 프로필을 평가합니다.
- gn 및 `ninja`, 오프라인 `cargo` 등을 사용할 때신뢰해야 하는 도구, 라이브러리, 사용자 그룹을 논의합니다.

학생들에게 연습문제를 완료하기 전에 발표자 노트를 엿보지 말라고 합니다. 과정을 수강하는 학생들이 실제로 함께 있다고 가정하고 3~4 명으로 구성된 소그룹으로 토론하도록 합니다.

연습문제의 첫 부분과 관련된 참고사항/힌트 ('Cargo 가이점을 제공할 수 있는 시나리오'):

- 도구를 작성하거나 Chromium 일부의 프로토타입을 제작할 때 crates.io 라이브러리의 풍부한 생태계에 액세스할 수 있다는 것은 멋진 일입니다. 거의 모든 것을 위해 크레이트가 있으며 일반적으로 사용하기 매우좋습니다. 명령줄 파싱을 위한 `clap`, 다양한 형식 간직렬화/역직렬화를 위한 `serde`, 반복자를 사용하는 경우의 `itertools` 등이 있습니다.
  - `cargo` 를 사용하면 라이브러리를 쉽게 사용해 볼 수 있습니다. `Cargo.toml` 에 한 줄을 추가하고 코드 작성을 시작하면 됩니다.
  - `perl` 이 인기를 얻게 된 데 CPAN 이 어떤 도움이 되었는지 비교해보는 것도 좋습니다. `python` + `pip` 와 비교해도됩니다.

- 핵심 Rust 도구 (예: 나이틀리, 최신 안정화 버전, 이전 안정화 버전에서작동해야 하는 크레이트를 테스트할 때 `rustup` 을 사용하여 다른 `rustc` 버전으로 전환) 뿐만 아니라 서드 파티 도구의 생태계 (예: Mozilla 는 보안 감사 간소화 및 공유를 위해 `cargo vet` 제공, `criterion` 크레이트는 간소화된 벤치마크 실행 방법을 제공함) 를통해서도 개발 환경이 개선됩니다.
  - `cargo` 를 사용하면 `cargo install --locked cargo-vet` 를 통해 도구를 쉽게추가할 수 있습니다.
  - Chrome 확장 프로그램 또는 VScode 확장 프로그램과 비교해 보는 것이좋습니다.
- `cargo` 가 적합할 수 있는 프로젝트의 광범위하고 일반적인 예:
  - 놀랍게도 업계에서 Rust 가 명령줄 도구 작성으로 점점 인기를 얻고있습니다. 라이브러리의 다양함과 사용편의성은 Python 과 유사하지만풍부한 타입 시스템 덕분에 더 안전하고 더 빠르게실행됩니다 (인터프리트 언어가 아닌 컴파일된 언어로).
  - Rust 생태계에 참여하려면 Cargo 와 같은 표준 Rust 도구를 사용해야합니다. 외부 기여를 원하고 Chromium 외부 (예: Bazel 또는 Android/Soong 빌드 환경) 에서 사용하고자 하는 라이브러리는 Cargo 를사용해야 할 수 있습니다.
- `cargo\` 기반 Chromium 관련 프로젝트의 예는다음과 같습니다.
  - `serde_json_lenient`(Google 의 다른 부분에서 실험하여 성능이개선된 PR 이 나옴)
  - `font-types` 와 같은 글꼴 라이브러리
  - `gnrt` 도구 (과정 후반부에서 설명) 는 명령줄 파싱의 경우 `clap`, 구성 파일의 경우 `toml` 에 종속됩니다.
    * 주의: 여기서 `cargo` 를 사용하는 유일한 이유는 Rust 도구모음을 빌드할 때 Rust 표준 라이브러리를 빌드하고 부트스트랩하는경우 `gn` 을 사용할 수 없기 때문입니다.
    * `run_gnrt.py` 는 Chromium 의 `cargo` 및 `rustc` 사본을 사용합니다. `gnrt` 는 인터넷에서다운로드한 서드 파티 라이브러리에 종속되며 `run_gnrt.py` 는 `cargo` 에 `Cargo.lock` 을 통해 `--locked` 콘텐츠만 허용된다고 합니다.

학생은 다음 항목을 암시적 또는 명시적으로 신뢰할 수 있는 것으로 식별할수 있습니다.

- `rustc`(Rust 컴파일러) 는 차례로 LLVM 라이브러리, Clang 컴파일러, `rustc` 소스 (GitHub 에서 가져옴, Rust 컴파일러팀에서검토), 부트스트랩을 위해 다운로드한 바이너리 Rust 컴파일러에 종속됩니다.
- `rustup`(`rustup` 은 https://github.com/rust-lang/ 조직산하에서 개발되었으며 `rustc` 와 동일함)
- `cargo`, `rustfmt` 등
- 다양한 내부 인프라 ('rustc' 를 빌드하는봇, 사전 빌드된 도구 모음을 Chromium 엔지니어에게 배포하기 위한 시스템등)
- `cargo audit`, `cargo vet` 등과 같은 Cargo 도구
- `//third_party/rust` 에 공급되는 Rust 라이브러리 (security@chromium.org 에서 감사)
- 기타 Rust 라이브러리 (일부는 틈새시장용, 일부는 매우 인기 있으며 흔히사용됨)

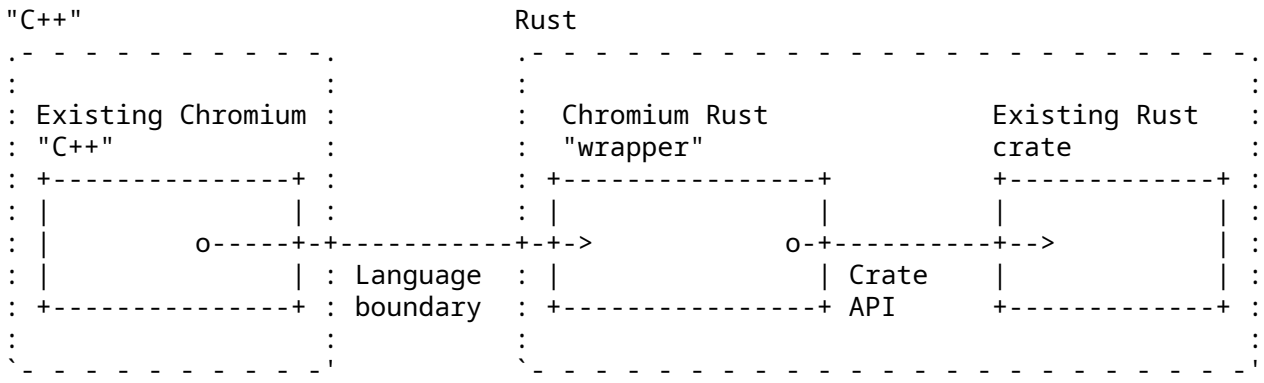
제 42 장

Chromium Rust 정책

Chromium 에서는 아직 퍼스트 파티 Rust 를 허용하지 않습니다. 단, Chromium 의 Area Tech Leads 에서 승인한 드문 경우는 예외입니다.

서드 파티 라이브러리에 관한 Chromium 의 정책은여기에설명되어 있습니다. Rust 는 성능이나 보안을 위해 최상의 옵션인 경우 등다양한 상황에서 서드 파티 라이브러리에 허용됩니다.

C/C++ API 를 직접 노출하는 Rust 라이브러리는 극소수이므로 이러한 거의모든 라이브러리에는 소량의 퍼스트 파티 글루 코드가 필요합니다.



특정 서드 파티 크레이트의 퍼스트 파티 Rust 글루 코드는 일반적으로 `third_party/rust/<crate>/<version>/w` 에보관해야 합니다.

따라서 오늘 과정에서는 다음과 같은 내용을 중점적으로 다룹니다.

- 서드 파티 Rust 라이브러리 ('crates') 가져오기
- Chromium C++에서 이러한 크레이트를 사용할 수 있도록 글루 코드를작성합니다.

시간이 지나면서 이 정책이 변경되면 교육 과정도 이에 맞게 변경됩니다.

제 43 장

빌드규칙

Rust 코드는 일반적으로 `cargo` 를 사용하여 빌드됩니다. Chromium 은효율성을 위해 `gn` 및 `ninja` 로 빌드됩니다. 정적 규칙은최대 동시 로드를 허용합니다. Rust 도 예외는 아닙니다.

Chromium 에 Rust 코드추가

일부 기존 Chromium `BUILD.gn` 파일에서 `rust_static_library` 를 선언합니다.

```
import("//build/rust/rust_static_library.gni")

rust_static_library("my_rust_lib") {
  crate_root = "lib.rs"
  sources = [ "lib.rs" ]
}
```

다른 Rust 타겟에도 `deps` 를 추가할 수 있습니다. 나중에 서드 파티코드에 의존하기 위해 이를 사용합니다.

크레이트 루트 소스 전체 목록 _둘 다_를 지정해야 합니다. `crate_root` 는 컴파일 단위의 루트 파일 (일반적으로 `lib.rs`) 을 나타내는 Rust 컴파일러에 제공되는 파일입니다. `sources` 는 재빌드가 필요한 시점을 결정하기 위해 `ninja` 에필요한 모든 소스 파일의 전체 목록입니다.

(Rust 에서는 크레이트 전체가 컴파일 단위이므로 Rust `source_set` 와 같은 것은 없습니다. `static_library` 가최소 단위입니다.)

학생들은 gn 의내장 Rust 정적 라이브러리 지원을 사용하는 대신 gn 템플릿이 필요한이유를 궁금해할 수 있습니다. 대답은 이 템플릿이 CXX 상호 운용성, Rust 기능, 단위 테스트를 지원한다는 것입니다. 이 중 일부는 나중에 사용하게됩니다.

43.1 `unsafe` Rust 코드포함

`rust_static_library` 에는 안전하지 않은 Rust 코드가 기본적으로금지되어 있으므로 컴파일되지 않습니다. 안전하지 않은 Rust 코드가필요하면 gn 타겟에 `allow_unsafe = true` 를 추가하세요. (이과정의 후반부에 이것이 필요한 상황을 살펴봅니다.)

```
import("//build/rust/rust_static_library.gni")
```

```
rust_static_library("my_rust_lib") {
  crate_root = "lib.rs"
  sources = [
    "lib.rs",
    "hippopotamus.rs"
  ]
  allow_unsafe = true
}
```

43.2 Chromium C++의 Rust 코드에의존

위의 타겟을 일부 Chromium C++ 타겟의 deps 에 추가하기만 하면됩니다.

```
import("//build/rust/rust_static_library.gni")

rust_static_library("my_rust_lib") {
  crate_root = "lib.rs"
  sources = [ "lib.rs" ]
}

# or source_set, static_library etc.
component("preexisting_cpp") {
  deps = [ ":my_rust_lib" ]
}
```

43.3 Visual Studio Code

Rust 코드에서는 타입이 생략되므로 우수한 IDE 가 C++보다 훨씬 더유용해집니다. Visual Studio Code 는 Chromium 의 Rust 에서 잘 작동합니다. 사용하려면 다음을 실행합니다.

- VSCode 에 이전 형태의 Rust 지원이 아닌 `rust-analyzer` 확장프로그램이 있는지 확인하세요.
- `gn gen out/Debug --export-rust-project`(또는 출력디렉터리에 상응)
- `ln -s out/Debug/rust-project.json rust-project.json`

누군가가 IDE 에 대해 회의적인 경우 rust-analyzer 의 코드 주석 및 탐색 기능중 일부를 시연해 보면 생각을 바꾸는데 유용할 수 있습니다.

다음 단계는 데모에 도움이 될 수 있습니다. 하지만 가장 익숙한 Chromium 관련 Rust 를 대신 사용해도 됩니다.

- `components/qr_code_generator/qr_code_generator_ffi_glue.rs` 를엽니다.
- 커서를 `qr_code_generator_ffi_glue.rs` 의 `\QrCode::new‘` 호출 (26 번 줄 부근) 위로 이동합니다.
- 데모 **문서 표시** (일반적인 바인딩: vscode = ctrl k i; vim/CoC = K)
- 데모 **정의로 이동** (일반적인 바인딩: vscode = F12; vim/CoC = g d) 그러면 `//third_party/rust/.../qr_code-.../src/lib.rs` 로이동합니다.
- **개요** 데모를 실행하고 `QrCode::with_bits` 메서드 (164 번줄 근처, 개요는 vscode 의 파일 탐색기 창에 있음, 일반적인 vim/CoC 바인딩 = space o) 로 이동합니다.
- 데모 **유형 주석** (`QrCode::with_bits` 메서드에 몇 가지좋은 예가 있음)

BUILD.gn 파일을 수정한 후 gn gen ... --export-rust-project 를 다시 실행해야 한다는점은 주목할 필요가 있습니다. 이 작업은 이 세션의 연습 전반에 걸쳐 몇 번반복하게 됩니다.

43.4 빌드규칙

Chromium 빌드에서 다음을 포함하는 //ui/base/BUILD.gn 에 새 Rust 타겟을 추가합니다.

```
#[no_mangle]
pub extern "C" fn hello_from_rust() {
    println!("Hello from Rust!")
}
```

중요: 여기서 no_mangle 은 Rust 컴파일러에 의해안전하지 않은 유형으로 간주되므로 gn 타겟에서 안전하지 않은코드를 허용해야 합니다.

이 새로운 Rust 타겟을 //ui/base:base 의 종속 항목으로추가합니다. 이 함수를 ui/base/resource/resource_bund 의 맨위에서 선언합니다. 나중에 바인딩 생성 도구로 자동화하는 방법을살펴봅니다.

```
extern "C" void hello_from_rust();
```

ui/base/resource/resource_bundle.cc 의 어딘가에서 이 함수를호출합니다. ResourceBundle::MaybeMangle 의 상단이좋습니다. Chromium 을 빌드하고 실행하여 'Hello from Rust!' 가 여러 번 출력되는지 확인합니다.

VSCode 를 사용하는 경우 이제 VSCode 에서 잘 작동하도록 Rust 를 설정합니다. 이후 연습에서 유용합니다. 성공하면 println! 에서' 정의로 이동' 을 마우스 오른쪽 버튼으로클릭할 수 있습니다.

도움을 받을 수 있는곳

- rust_static_library gn 템플릿에서 사용 가능한 옵션
- #[no_mangle] 에관한 정보
- extern "C"에관한 정보
- gn 의--export-rust-project 전환 정보
- VSCode 에서 rust-analyzer 를 설치하는 방법

이 예는 최소 공통분모 상호 운용성 언어인 C 로 귀결되기 때문에 일반적이지않습니다. C++와 Rust 모두 기본적으로 C ABI 함수를 선언하고 호출할 수있습니다. 이 과정의 후반부에서 C++를 Rust 에 직접 연결합니다.

여기서 allow_unsafe = true 가 필요한 이유는#[no_mangle] 이 Rust 가 이름이 같은 함수 두 개를 생성할 수있도록 할 수 있고 Rust 는 더 이상 올바른 함수가 호출된다고 보장할 수 없기때문입니다.

순수한 Rust 실행 파일이 필요하면 rust_executable gn 템플릿을사용하면 됩니다.

제 44 장

테스트

Rust community typically authors unit tests in a module placed in the same source file as the code being tested. This was covered earlier in the course and looks like this:

```
#[cfg(test)]
mod tests {
    #[test]
    fn my_test() {
        todo!()
    }
}
```

In Chromium we place unit tests in a separate source file and we continue to follow this practice for Rust — this makes tests consistently discoverable and helps to avoid rebuilding `.rs` files a second time (in the `test` configuration).

This results in the following options for testing Rust code in Chromium:

- Native Rust tests (i.e. `#[test]`). Discouraged outside of `//third_party/rust`.
- `gtest` tests authored in C++ and exercising Rust via FFI calls. Sufficient when Rust code is just a thin FFI layer and the existing unit tests provide sufficient coverage for the feature.
- `gtest` tests authored in Rust and using the crate under test through its public API (using `pub mod for_testing { ... }` if needed). This is the subject of the next few slides.

Mention that native Rust tests of third-party crates should eventually be exercised by Chromium bots. (Such testing is needed rarely — only after adding or updating third-party crates.)

Some examples may help illustrate when C++ `gtest` vs Rust `gtest` should be used:

- QR has very little functionality in the first-party Rust layer (it's just a thin FFI glue) and therefore uses the existing C++ unit tests for testing both the C++ and the Rust implementation (parameterizing the tests so they enable or disable Rust using a `ScopedFeatureList`).
- Hypothetical/WIP PNG integration may need to implement memory-safe implementation of pixel transformations that are provided by `libpng` but missing in the png crate - e.g. RGBA => BGRA, or gamma correction. Such functionality may benefit from separate tests authored in Rust.

44.1 `rust_gtest_interop` Library

The `rust_gtest_interop` library provides a way to:

- Use a Rust function as a `gtest` testcase (using the `#[gtest(...)]` attribute)
- Use `expect_eq!` and similar macros (similar to `assert_eq!` but not panicking and not terminating the test when the assertion fails).

Example:

```
use rust_gtest_interop::prelude::*;

#[gtest(MyRustTestSuite, MyAdditionTest)]
fn test_addition() {
    expect_eq!(2 + 2, 4);
}
```

44.2 GN Rules for Rust Tests

The simplest way to build Rust `gtest` tests is to add them to an existing test binary that already contains tests authored in C++. For example:

```
test("ui_base_unittests") {
  ...
  sources += [ "my_rust_lib_unittest.rs" ]
  deps += [ ":my_rust_lib" ]
}
```

Authoring Rust tests in a separate `static_library` also works, but requires manually declaring the dependency on the support libraries:

```
rust_static_library("my_rust_lib_unittests") {
  testonly = true
  is_gtest_unittests = true
  crate_root = "my_rust_lib_unittest.rs"
  sources = [ "my_rust_lib_unittest.rs" ]
  deps = [
    ":my_rust_lib",
    "//testing/rust_gtest_interop",
  ]
}

test("ui_base_unittests") {
  ...
  deps += [ ":my_rust_lib_unittests" ]
}
```

44.3 `chromium::import!` Macro

After adding `:my_rust_lib` to GN deps, we still need to learn how to import and use `my_rust_lib` from `my_rust_lib_unittest.rs`. We haven't provided an explicit `crate_name` for `my_rust_lib` so its crate name is computed based on the full target path

and name. Fortunately we can avoid working with such an unwieldy name by using the `chromium::import!` macro from the automatically-imported `chromium` crate:

```
chromium::import! {
    "//ui/base:my_rust_lib";
}

use my_rust_lib::my_function_under_test;
```

Under the covers the macro expands to something similar to:

```
extern crate ui_sbase_cmy_urust_ulib as my_rust_lib;

use my_rust_lib::my_function_under_test;
```

More information can be found in the doc comment of the `chromium::import` macro.

`rust_static_library` supports specifying an explicit name via `crate_name` property, but doing this is discouraged. And it is discouraged because the crate name has to be globally unique. crates.io guarantees uniqueness of its crate names so `cargo_crate` GN targets (generated by the `gnrt` tool covered in a later section) use short crate names.

44.4 Testing exercise

새로운 연습문제를 풀어봅시다!

In your Chromium build:

- Add a testable function next to `hello_from_rust`. Some suggestions: adding two integers received as arguments, computing the nth Fibonacci number, summing integers in a slice, etc.
- Add a separate `..._unittest.rs` file with a test for the new function.
- Add the new tests to `BUILD.gn`.
- Build the tests, run them, and verify that the new test works.

제 45 장

C와의상호운용성

Rust 커뮤니티는 C++/Rust 상호 운용성을 위한 여러 옵션을 제공하며, 새로운도구가 계속 개발되고 있습니다. 현재 Chromium 은 CXX 라는 도구를사용합니다.

인터페이스 정의 언어 (Rust 와 매우 유사함) 에서 전체 언어 경계를 설명하면 CXX 도구가 Rust 및 C++ 모두에서 함수와 유형에 관한 선언을 생성합니다.

See the CXX tutorial for a full example of using this.

다이어그램을 통해 설명합니다. 내부적으로는 이전과 동일한 작업을실행한다고 설명합니다. 프로세스를 자동화하면 다음과 같은 이점이있습니다.

- 이 도구는 C++와 Rust 측의 일치를 보장합니다. 예를 들어`#[cxx::bridge]` 가 실제 C++ 또는 Rust 정의와 일치하지 않는경우 컴파일 오류가 발생하지만 동기화되지 않은 수동 바인딩을 사용하면정의되지 않은 동작이 발생합니다.
- 이 도구는 비 C 기능의 FFI thunk(소형, C-ABI 호환, 무료 함수) 생성을자동화합니다 (예: Rust 또는 C++ 메서드에 대한 FFI 호출 사용 설정, 수동바인딩의 경우 이러한 최상위 무료 함수를 수동으로 작성해야 함).
- 도구와 라이브러리는 핵심 유형 집합을 처리할 수 있습니다. 예를 들면다음과 같습니다.
  - `&[T]` 는 특정 ABI 나 메모리 레이아웃을 보장하지 않더라도 FFI 경계를 넘어 전달될 수 있습니다. 수동 바인딩을 사용하면 `std::span<T>` / `&[T]` 는수동으로 디스트럭처링해야 하고 포인터와 길이로 다시 빌드해야 합니다. 이는 각 언어가 빈 슬라이스를 약간 다르게 표현하는 점을 고려할 때오류가 발생하기 쉽습니다.
  - `std::unique_ptr<T>`, `std::shared_ptr<T>`, `Box` 등의 스마트 포인터가 기본적으로 지원됩니다. 수동 바인딩을 사용하면 C-ABI 호환 원시 포인터를 전달해야 하므로 전체 기간 및 메모리 안전위험이 증가합니다.
  - `rust::String` 및 `CxxString` 유형은 언어 간 문자열표현의 차이를 이해하고 유지합니다. 예를 들어 `rust::String::lossy` 는 UTF8 이 아닌 입력에서 Rust 문자열을빌드할 수 있고 `rust::String::c_str` 은 문자열을 NUL 종료할수 있습니다.

45.1 예제

CXX 에서는 전체 C++/Rust 경계가 `.rs` 소스 코드 내의 `cxx::bridge` 'modules' 에선언되어야 합니다.

```
#[cxx::bridge]
mod ffi {
    extern "Rust" {
```

```
        type MultiBuf;

        fn next_chunk(buf: &mut MultiBuf) -> &[u8];
    }

    unsafe extern "C++" {
        include!("example/include/blobstore.h");

        type BlobstoreClient;

        fn new_blobstore_client() -> UniquePtr<BlobstoreClient>;
        fn put(self: &BlobstoreClient, buf: &mut MultiBuf) -> Result<u64>;
    }
}

// Rust 유형 및 함수의 정의는 여기를 참고하세요.
```

참고:

- Although this looks like a regular Rust `mod`, the `#[cxx::bridge]` procedural macro does complex things to it. The generated code is quite a bit more sophisticated - though this does still result in a `mod` called `ffi` in your code.
- Rust 에서 C++' 의 `std::unique_ptr` 기본 지원
- Native support for Rust slices in C++
- C++에서 Rust 호출 및 Rust 유형 (상단)
- Rust 에서 C++ 호출 및 C++ 유형 (하단)

일반적인 오해: C++ 헤더가 Rust 에서 파싱되는 오해의 소지가 있습니다. 이 헤더는 Rust 에서 해석되지 않으며 C++ 컴파일러의 이점을 위해 생성된 C++ 코드에 단순히`#include` 됩니다.

CXX 제한사항

CXX 를 사용할 때 단연 가장 유용한 페이지는유형 참조입니다.

CXX 는 기본적으로 다음과 같은 사례에 적합합니다.

- Rust-C++ 인터페이스는 매우 단순하여 모두 선언할 수 있습니다.
- 이미 CXX 에서 기본적으로 지원하는 유형만 사용하고 있습니다 (예: `std::unique_ptr`, `std::string`, `&[u8]` 등).

많은 제한이 있습니다. 예를 들어 Rust 의 `Option` 유형은 지원되지않습니다.

이러한 제한사항으로 인해 임의의 Rust-C++ 상호 운용성이 아닌 잘 격리된' 리프 노드' 의 경우에만 Chromium 에서 Rust 를 사용할 수 있습니다. Chromium 에서 Rust 사용 사례를 고려할 때 좋은출발점은 언어 경계의 CXX 바인딩 초안을 작성하여 충분히 단순하게표시되는지 확인하는 것입니다.

CXX 의 다른 어려운 문제도 논의해야 합니다. 예를 들면 다음과 같습니다.

- 오류 처리는 C++ 예외를 기반으로 합니다 (다음 슬라이드에 나와 있음).
- 함수 포인터는 사용하기 어색합니다.

45.2 오류처리

CXX's support for `Result<T,E>` relies on C++ exceptions, so we can't use that in Chromium. Alternatives:

- The `T` part of `Result<T, E>` can be:
  - Returned via out parameters (e.g. via `&mut T`). This requires that `T` can be passed across the FFI boundary - for example `T` has to be:
    * A primitive type (like `u32` or `usize`)
    * A type natively supported by `cxx` (like `UniquePtr<T>`) that has a suitable default value to use in a failure case (*unlike* `Box<T>`).
  - Retained on the Rust side, and exposed via reference. This may be needed when `T` is a Rust type, which cannot be passed across the FFI boundary, and cannot be stored in `UniquePtr<T>`.
- The `E` part of `Result<T, E>` can be:
  - Returned as a boolean (e.g. `true` representing success, and `false` representing failure)
  - Preserving error details is in theory possible, but so far hasn't been needed in practice.

45.2.1 CXX Error Handling: QR Example

QR 코드 생성기에서와 같이 간단한 부울로 성공을 나타낼 수 있는 경우: 성공을나타내는 부울을 반환하고 out 매개변수를 사용하여 결과를 기록합니다.

```
#[cxx::bridge(namespace = "qr_code_generator")]
mod ffi {
    extern "Rust" {
        fn generate_qr_code_using_rust(
            data: &[u8],
            min_version: i16,
            out_pixels: Pin<&mut CxxVector<u8>>,
            out_qr_size: &mut usize,
        ) -> bool;
    }
}
```

Students may be curious about the semantics of the `out_qr_size` output. This is not the size of the vector, but the size of the QR code (and admittedly it is a bit redundant - this is the square root of the size of the vector).

It may be worth pointing out the importance of initializing `out_qr_size` before calling into the Rust function. Creation of a Rust reference that points to uninitialized memory results in Undefined Behavior (unlike in C++, when only the act of dereferencing such memory results in UB).

If students ask about `Pin`, then explain why CXX needs it for mutable references to C++ data: the answer is that C++ data can't be moved around like Rust data, because it may contain self-referential pointers.

45.2.2 CXX Error Handling: PNG Example

A prototype of a PNG decoder illustrates what can be done when the successful result cannot be passed across the FFI boundary:

```
#[cxx::bridge(namespace = "gfx::rust_bindings")]
mod ffi {
```

```
    extern "Rust" {
        /// This returns an FFI-friendly equivalent of `Result<PngReader<'a>,
        /// ()>`.
        fn new_png_reader<'a>(input: &'a [u8]) -> Box<ResultOfPngReader<'a>>;

        /// C++ bindings for the `crate::png::ResultOfPngReader` type.
        type ResultOfPngReader<'a>;
        fn is_err(self: &ResultOfPngReader) -> bool;
        fn unwrap_as_mut<'a, 'b>(
            self: &'b mut ResultOfPngReader<'a>,
        ) -> &'b mut PngReader<'a>;

        /// C++ bindings for the `crate::png::PngReader` type.
        type PngReader<'a>;
        fn height(self: &PngReader) -> u32;
        fn width(self: &PngReader) -> u32;
        fn read_rgba8(self: &mut PngReader, output: &mut [u8]) -> bool;
    }
}
```

PngReader and ResultOfPngReader are Rust types — objects of these types cannot cross the FFI boundary without indirection of a Box<T>. We can't have an out_parameter: &mut PngReader, because CXX doesn't allow C++ to store Rust objects by value.

This example illustrates that even though CXX doesn't support arbitrary generics nor templates, we can still pass them across the FFI boundary by manually specializing / monomorphizing them into a non-generic type. In the example ResultOfPngReader is a non-generic type that forwards into appropriate methods of Result<T, E> (e.g. into is_err, unwrap, and/or as_mut).

Chromium 에서 cxx 사용

Chromium 에서는 Rust 를 사용하려는 리프 노드마다 독립적인#[cxx::bridge] mod 를 정의합니다. 일반적으로 rust_static_library 마다 하나씩 있습니다. 추가하기만 하면됩니다.

```
cxx_bindings = [ "my_rust_file.rs" ]
   # 모든 소스 파일이 아닌 #[cxx::bridge]가 포함된 파일 목록
allow_unsafe = true
```

crate_root 및 sources 와 함께 기존 rust_static_library 타겟에 추가합니다.

C++ 헤더는 적절한 위치에 생성됩니다. 따라서

```
#include 'ui/base/my_rust_file.rs.h'
```

//base 에서 Chromium C++ 유형으로 변환하는 또는 Chromium C++ 유형에서 CXX Rust 유형으로 변환하는 유틸리티 함수를 확인할 수 있습니다. 예: SpanToRustSlice

allow_unsafe = true 가 계속 필요한 이유가 궁금할 수 있습니다.

광범위한 의미의 답변은 일반적인 Rust 표준에서는 C/C++ 코드가' 안전하지' 않다는 것입니다. Rust 에서 C/C++를 여기저기 호출하면 메모리에 임의적인 작업을 할 수 있으며 Rust 자체 데이터 레이아웃의 안전성이 손상될 수 있습니다. C/C++ 상호 운용성에 unsafe 키워드가 이러한 키워드의 신호대잡음비에 해를 끼칠 수 있으며논란의 소지가있습니다. 그러나 엄격하게는 외부 코드를 Rust 바이너리로 가져오면 Rust 의관점에서 예기치 않은 동작이 발생할 수 있습니다.

The narrow answer lies in the diagram at the top of this page — behind the scenes, CXX generates Rust `unsafe` and `extern "C"` functions just like we did manually in the previous section.

45.3 Exercise: Interoperability with C++

1 부

- 이전에 만든 Rust 파일에서, C++에서 호출할 `hello_from_rust` 라는 단일 함수를 지정하는 `#[cxx::bridge]` 를 추가합니다. 이 함수는 매개변수를사용하지 않고 값을 반환하지 않습니다.
- 이전의 `hello_from_rust` 함수를 수정하여 `extern "C"` 및 `#[no_mangle]` 을 삭제합니다. 이 함수는 이제 표준 Rust 함수입니다.
- gn 타겟을 수정하여 이러한 바인딩을 빌드합니다.
- C++ 코드에서 `hello_from_rust` 의 정방향 선언을 삭제합니다. 대신 생성된 헤더 파일을 포함합니다.
- 빌드 및 실행

2 부

CXX 를 사용해 보는 것도 좋은 방법입니다. Chromium 의 Rust 가 실제로 얼마나유연한지 생각해 보는 데 도움이 됩니다.

Some things to try:

- Rust 에서 C++로 다시 호출 필요한 사항은 다음과 같습니다.
  - `cxx::bridge` 에서 `include!`할 수 있는 추가 헤더파일입니다. 새 헤더 파일에서 C++ 함수를 선언해야 합니다.
  - 이러한 함수를 호출하거나여기에설명된 대로 `#[cxx::bridge]` 에서 `unsafe` 키워드를 지정하는 `unsafe` 블록입니다.
  - `#include "third_party/rust/cxx/v1/crate/include/cxx.h"`가필요할 수도 있습니다.
- C++에서 Rust 로 C++ 문자열을 전달합니다.
- C++ 객체 참조를 Rust 로 전달합니다.
- 의도적으로 `#[cxx::bridge]` 에서 일치하지 않는 Rust 함수서명을 가져와서 표시되는 오류에 익숙해집니다.
- 의도적으로 `#[cxx::bridge]` 에서 일치하지 않는 C++ 함수서명을 가져와서 표시되는 오류에 익숙해집니다.
- Rust 가 C++ 객체를 소유할 수 있도록 C++에서 일부 유형의 `std::unique_ptr` 을 Rust 로 전달합니다.
- Rust 객체를 만들어 C++로 전달하여 C++에서 소유하도록 합니다. 힌트: `Box` 필요
- C++ 유형에 일부 메서드를 선언합니다. Rust 에서 이를 호출하세요.
- Rust 유형에 일부 메서드를 선언합니다. C++에서 이를 호출하세요.

3 부

지금까지 CXX 상호 운용성의 강점과 한계를 이해했으니, 인터페이스가 충분히간단한 Chromium 의 Rust 사용 사례를 생각해 보세요. 해당 인터페이스를정의하는 방법을 스케치합니다.

도움을 받을 수 있는곳

- `cxx` 바인딩 참조

- `rust_static_library` gn 템플릿

다음과 같은 문제가 있을 수 있습니다.

- 유형 Y 로 유형 X 의 변수를 초기화하는 데 문제가 있습니다. 여기서 X 와 Y 는모두 함수 유형입니다. 이는 C++ 함수가 `cxx::bridge` 의 선언과일치하지 않기 때문입니다.
- C++ 참조를 Rust 참조로 자유롭게 변환할 수 있는 것 같습니다. 이렇게하면 UB 가 발생하지 않을까요? CXX 의 유형의 경우 발생하지않습니다. 크기가 0 이기 때문입니다. CXX 사소한 유형의 경우 UB 를유발하는 것이 _가능_하지만 CXX 의 설계상 이러한 예를 만들기가 상당히어렵습니다.

제 46 장

서드 파티 크레이트추가

Rust 라이브러리는 ' 크레이트' 라고 하며 crates.io 에서 찾을 수 있습니다. Rust 크레이트가 서로 종속되는 것은 . 따라서 서로종속됩니다.

속성	C++ library	Rust crate
Build system	1 억+	일관성: `Cargo.toml`
일반적인 라이브러리 크기	큰 편	작게
모든 종속성들	적음	1 억+

Chromium 엔지니어에게는 다음과 같은 장단점이 있습니다.

- 모든 크레이트는 공통 빌드 시스템을 사용하므로 Chromium 에 자동으로포함할 수 있습니다.
- 그러나 크레이트에는 일반적으로 전이 종속 항목이 있으므로 여러라이브러리를 가져와야 할 수 있습니다.

다룰 내용은 다음과 같습니다.

- Chromium 소스 코드 트리에 크레이트를 추가하는 방법
- 이를 위해 gn 빌드 규칙을 만드는 방법
- 충분한 안전성을 위해 소스 코드를 감사하는 방법

46.1 크레이트를 추가하도록 `Cargo.toml` 파일구성

Chromium 에는 중앙에서 관리되는 직접 크레이트 종속 항목의 단일 세트가있습니다. 이는 단일 `Cargo.toml` 을통해 관리됩니다.

```
[dependencies]
bitflags = "1"
cfg-if = "1"
cxx = "1"
# lots more...
```

다른 `Cargo.toml` 과 마찬가지로종속항목에 관한 자세한 내용을 지정할 수 있습니다. 가장 흔하게는 크레이트에서 사용 설정하려는 `features` 를 지정하는 것이 좋습니다.

Chromium 에 크레이트를 추가할 때는 다음 단계에서 다룰 추가 파일 `gnrt_config.toml` 에 몇 가지 정보를 추가로 제공해야 하는 경우가많습니다.

46.2 Configuring gnrt_config.toml

Cargo.toml 과 함께 gnrt_config.toml 이있습니다. 여기에는 크레이트 처리를 위한 Chromium 전용 확장 프로그램이포함되어 있습니다.

새 크레이트를 추가하는 경우 적어도 group 을 지정해야 합니다. 다음 중 하나입니다.

```
#   'safe': The library satisfies the rule-of-2 and can be used in any process.
#   'sandbox': The library does not satisfy the rule-of-2 and must be used in
#              a sandboxed process such as the renderer or a utility process.
#   'test': The library is only used in tests.
```

예를 들면 다음과 같습니다.

```
[crate.my-new-crate]
group = 'test' # only used in test code
```

크레이트 소스 코드 레이아웃에 따라 이 파일을 사용하여 LICENSE 파일을 찾을 수 있는 위치를 지정해야 할 수도 있습니다.

나중에 문제를 해결하기 위해 이 파일에서 구성해야 하는 몇 가지 사항을살펴봅니다.

46.3 크레이트다운로드

gnrt 라는 도구는 크레이트를 다운로드하는 방법과 BUILD.gn 규칙을 생성하는 방법을 알고 있습니다.

시작하려면 다음과 같이 원하는 크레이트를 다운로드합니다.

```
cd chromium/src
vpython3 tools/crates/run_gnrt.py -- vendor
```

> gnrt 도구는 Chromium 소스 코드의 일부이지만 이 명령어를실행하면 crates.io 에서 종속 항목을 다운로드하고 실행하게됩니다. 이 보안 관련 결정에 관해서는이전 섹션을참고하세요.

이 vendor 명령어는 다음을 다운로드할 수 있습니다.

- Your crate
- 직접 및 임시 종속 항목
- Chromium 에서 필요한 전체 크레이트 세트를 해결하기 위해 cargo 에서 요구하는 다른 크레이트의 새 버전입니다.

Chromium 은 //third_party/rust/chromium_crates_io/patches 에보관되는 일부 크레이트의 패치를 유지관리합니다. 이는 자동으로 다시적용되지만 패치에 실패하면 직접 조치를 취해야 할 수도 있습니다.

46.4 Generating gn Build Rules

크레이트를 다운로드한 후에는 다음과 같이 BUILD.gn 파일을생성합니다.

```
vpython3 tools/crates/run_gnrt.py -- gen
```

이제 git status 를 실행합니다. 다음을 확인할 수 있습니다.

- third_party/rust/chromium_crates_io/vendor 에 하나 이상의새 크레이트 소스 코드가 있습니다.

- `third_party/rust/<crate name>/v<major semver version>`에 새 `BUILD.gn` 이 하나 이상 있습니다.
- 적절한 `README.chromium`

The "major semver version" is a Rust "semver" version number.

특히 `third_party/rust` 에서 생성된 항목을 자세히 살펴보세요.

semver 에 관해 좀 더 이야기합니다. 특히 Chromium 에서는 호환되지 않는크레이트 버전을 여러 개 허용하는 방식이 있는데 이는 권장되지 않지만 Cargo 생태계에서는 때때로 필요합니다.

46.5 문제해결

빌드가 실패하는 경우, 이는 빌드 시간에 임의의 작업을 실행하는 프로그램인 `build.rs` 때문일 수 있습니다. 이는 빌드의 병렬성과 재현성을최대화하기 위해 정적인 빌드 규칙을 목표로 하는 gn 및 `ninja` 의 설계와 근본적으로 상충됩니다.

일부 `build.rs` 작업은 자동으로 지원됩니다. 그 외는 조치가필요합니다.

빌드 스크립트 효과	Google 의 gn 템플릿에서 지원	필요한 작업
기능을 사용 및 사용 중지로 구성하기 위해 rustc 버전 확인	예	없음
기능을 사용 및 사용 중지로 구성하기 위해 플랫폼 또는 CPU 확인	예	없음
Generating code	예	예 - `gnrt_config.toml` 에 지정
C/C++ 빌드	아니오	주변에 패치를 적용합니다.
임의의 기타 작업	아니오	주변에 패치를 적용합니다.

다행히 대부분의 크레이트에는 빌드 스크립트가 포함되어 있지 않으며, 다행히 대부분의 빌드 스크립트는 상위 두 가지 작업만 실행합니다.

46.5.1 코드를 생성하는 스크립트빌드

`ninja` 가 파일 누락에 관한 불만을 제기하는 경우 `build.rs` 에서 소스 코드 파일을 작성하는지 확인합니다.

그렇다면 `gnrt_config.toml` 을수정하여 `build-script-outputs` 를 크레이트에 추가합니다. 이것이전이 종속 항목, 즉 Chromium 코드가 직접 종속되면 안 되는 종속 항목인경우 `allow-first-party-usage=false` 도 추가합니다. 이 파일에는이미 다음과 같은 몇 가지 예가 있습니다.

```
[crate.unicode-linebreak]
allow-first-party-usage = false
build-script-outputs = [ "tables.rs" ]
```

이제 `gnrt.py -- gen` 을다시 실행하여 `BUILD.gn` 파일을 다시 생성하고 이 특정 출력 파일이후속 빌드 단계의 입력이라고 ninja 에 알립니다.

46.5.2 C++를 빌드하거나 임의의 작업을 실행하는 스크립트빌드

일부 크레이트는 cc 크레이트를 사용하여 C/C++ 라이브러리를 빌드하고 연결합니다. 다른크레이트는 빌드 스크립트 내에서 bindgen 을 사용하여 C/C++를 파싱합니다. 이러한 작업은 Chromium 컨텍스트에서는 지원되지않습니다. — Google 의 gn, ninja, LLVM 빌드 시스템은 빌드 작업 간의관계를 매우 구체적으로 표현합니다.

옵션은 다음과 같습니다.

- 이 크레이트는 피하세요.
- 크레이트에 패치를 적용합니다.

패치는 third_party/rust/chromium_crates_io/patches/<crate>에보관해야 합니다. 예는 cxx 크레이트에 대한 패치를 참고하세요. 그리고 크레이트를 업그레이드할때마다 gnrt 에 의해 자동으로 적용됩니다.

46.6 크레이트에 따라다름

서드 파티 크레이트를 추가하고 빌드 규칙을 생성하고 나면 크레이트에 따른작업은 간단합니다. rust_static_library 타겟을 찾고 크레이트내의 :lib 타겟에 dep 를 추가합니다.

Specifically,

```
                      +------------+      +----------------------+
"//서드 파티/rust" | 크레이트 이름 | "/v" | 메이저 semver 버전 | "/:lib"
                      +------------+      +----------------------+
```

예를 들면 다음과 같습니다.

```
rust_static_library("my_rust_lib") {
  crate_root = "lib.rs"
  sources = [ "lib.rs" ]
  deps = [ "//third_party/rust/example_rust_crate/v1:lib" ]
}
```

46.7 서드 파티 크레이트감사

새 라이브러리 추가에는 Chromium 의 표준정책이적용되지만 또한 보안 검토도 적용됩니다. 단일 크레이트뿐만 아니라 전이종속 항목도 가져올 수 있으므로 검토할 코드가 많을 수 있습니다. 반면에안전한 Rust 코드에서는 부정적인 부작용이 제한될 수 있습니다. 어떻게검토해야 할까요?

Chromium 은 시간이 지남에 따라 cargo vet 를 기반으로 한프로세스로 전환하는 것을 목표로 합니다.

한편 새로운 크레이트가 추가될 때마다 다음 사항을 확인하고 있습니다.

- 각 크레이트가 사용되는 이유를 이해합니다. 크레이트 간 관계는어떠한가요? 각 크레이트의 빌드 시스템에 build.rs 또는 절차매크로가 포함된 경우 그 용도를 파악해야 합니다. Chromium 이 일반적으로빌드되는 방식과 호환되나요?
- 각 크레이트가 적절히 잘 유지관리되는지 확인합니다.
- cd third-party/rust/chromium_crates_io; cargo audit 을사용하여 알려진 취약점을 확인합니다. 먼저 cargo install cargo-audit 을 실행해야 합니다. 여기에는인터넷 2 에서여러 종속 항목을 다운로드하는 작업이 포함되어 있습니다.
- unsafe 코드가 2 의법칙에 적합한지 확인합니다.
- fs 또는 net API 의 사용 확인

- 악의적으로 삽입되었을 수 있는 잘못된 코드를 찾을 수 있는 충분한수준으로 모든 코드를 읽습니다. 현실적으로 100% 완벽을 추구할 수는없습니다. 코드가 너무 많을 때가 많습니다.

이 내용은 가이드라인에 불과합니다. `security@chromium.org` 의검토자와 협력하여 크레이트에 관한 확신을 가질 수 있는 올바른 방법을찾아보세요.

46.8 Chromium 소스 코드로 크레이트확인

`git status` 는 다음을 표시해야 합니다.

- `//third_party/rust/chromium_crates_io` 의 크레이트 코드
- `//third_party/rust/<crate>/<version>`의메타데이터(`BUILD.gn` 및 `README.chromium`)

후자의 위치에 `OWNERS` 파일도 추가하세요.

이 모든 것을 `Cargo.toml` 및 `gnrt_config.toml` 변경사항과 함께 Chromium 저장소에 배치해야 합니다.

중요: `git add -f` 를 사용해야 합니다. 그러지 않으면 `.gitignore` 파일로 인해 일부 파일을 건너뛸 수 있습니다.

이렇게 하면 포용적이지 않은 언어로 인해 사전 제출 검사가 실패할 수도있습니다. 이는 Rust 크레이트 데이터에는 git 브랜치 이름이 포함되는경향이 있고, 많은 프로젝트에서는 여전히 포용적이지 않은 용어를 거기서사용하기 때문입니다. 따라서 다음을 실행해야 할 수 있습니다.

```
infra/update_inclusive_language_presubmit_exempt_dirs.sh > infra/inclusive_language_pres
git add -p infra/inclusive_language_presubmit_exempt_dirs.txt # add whatever changes are
```

46.9 크레이트를 최신 상태로유지

서드 파티 Chromium 종속 항목의 OWNER 이므로모든보안 수정사항을 통해 최신 상태로 유지해야 합니다. 곧 Rust 크레이트의경우 이 작업을 자동화할 수 있기를 바라지만, 당분간은 다른 서드 파티 종속 항목과 마찬가지로 개발자의 책임입니다.

46.10 연습문제

Chromium 에 uwuify 를 추가하여크레이트의기본기능을 사용 중지합니다. 크레이트가 Chromium 배송에 사용되지만 신뢰할 수없는 입력을 처리하는 데는 사용되지 않는다고 가정합니다.

다음 연습에서는 Chromium 의 uwuify 를 사용합니다. 원한다면 건너뛰고 지금해 봐도 됩니다. 또는 `uwuify` 를 사용하는 새로운 `rust_executable` 타겟을 만들 수 있습니다.

직접 및 임시 종속 항목

The total crates needed are:

- `instant`,
- `lock_api`,
- `parking_lot`,
- `parking_lot_core`,
- `redox_syscall`,
- `scopeguard`,
- `smallvec`, and
- `uwuify`.

If students are downloading even more than that, they probably forgot to turn off the default features.

이 크레이트를 제공해 주신다니엘 리우에게감사드립니다.

제 47 장

Bringing It Together — Exercise

이 연습에서는 이미 학습한 모든 내용을 종합하는 완전히 새로운 Chromium 기능을 추가합니다.

The Brief from Product Management

외딴 열대 우림에 사는 pixie 커뮤니티가 발견되었습니다. Pixie 용 Chromium 을 최대한 빨리 배송하는 것이 중요합니다.

Chromium 의 모든 UI 문자열을 Pixie 언어로 번역해야 합니다.

제대로 번역될 때까지 기다릴 시간은 없지만, 다행히 pixie 언어는 영어와매우 유사한 데다 번역을 하는 Rust 크레이트가 있습니다.

사실 이전 연습에서 이 크레이트를이미 가져왔습니다.

물론 Chrome 의 실제 번역 작업을 위해서는 상당한 주의와 노력이 필요합니다. 배송하지 마세요.

걸음수

표시하기 전에 모든 문자열을 uwuify 하도록 `ResourceBundle::MaybeMangleLocalizedString` 을 수정합니다. 이특수 Chromium 빌드에서는 `mangle_localized_strings_`의 설정과관계없이 항상 이 작업을 실행해야 합니다.

이 모든 연습에서 모든 작업을 제대로 완료했다면 축하합니다. pixie 용 Chrome 을 만드셨을 것입니다.

- UTF16 과 UTF8. 학생들은 Rust 문자열이 항상 UTF8 이라는 점을 알고 있어야하며, `base::UTF16ToUTF8` 을 사용하여 C++ 측에서 변환을 실행하고다시 그 반대로 변환하는 것이 더 낫다고 결정할 수 있습니다.
- Rust 측에서 변환하기로 결정한 학생들은 `String::from_utf16` 을고려하고 오류 처리를 고려하고많은 u16 을 전송할 수 있는 CXX 지원 유형을 고려해야 합니다.
- 학생들은 여러 가지 방법으로 C++/Rust 경계를 설계할 수 있습니다. 예를들어 값으로 문자열을 가져와 반환하거나 문자열에 대한 변경 가능한참조를 사용할 수 있습니다. 변경 가능한 참조를 사용하면 CXX 에서학생에게 `Pin` 을사용해야 한다고 알릴 가능성이 높습니다. `Pin` 의 역할을 설명하고이것이 C++ 데이터에 대한 변경 가능한 참조를 위해 CXX 에 필요한 이유를설명해야 할 수도 있습니다. 대답은 C++ 데이터는 Rust 데이터처럼 이동할수 없다는 것입니다. 자체 참조 포인터가 포함되어 있을 수 있기때문입니다.

- `ResourceBundle::MaybeMangleLocalizedString`이 포함된 C++ 타겟은 `rust_static_library` 타겟에 종속되어야 합니다. 학생은이미 이를 실행했을 것입니다.
- `rust_static_library` 타겟은`//third_party/rust/uwuify/v0_2:lib`에 종속되어야 합니다.

제 48 장

연습문제해답

Solutions to the Chromium exercises can be found in this series of CLs.

제 XI 편

Bare Metal 오전

제 49 장

Welcome to Bare Metal Rust

이 과정은 Rust 에 대해 어느정도 경험이 있고 (아마도 Comprehensive Rust 과정을 통해) C 와 같은 다른 언어로 bare-metal 프로그래밍을 해 본 사용자를대상으로 하는 bare-metal Rust 에 관한 독립적인 1 일 과정입니다.

오늘은 OS 를 사용하지 않고 Rust 코드를 실행하는'bare-metal' Rust 에 관해 알아봅니다. 본강의의 구성은 다음과 같습니다.

- no_std Rust 란 무엇인가요?
- 마이크로컨트롤러용 펌웨어 작성
- 애플리케이션 프로세서를 위한 부트로더 / 커널 코드 작성
- bare-metal Rust 개발을 위한 유용한 크레이트

이 강의에서는 BBC micro:bit v2 마이크로컨트롤러를 사용합니다. 이 마이크로컨트롤러는 Nordic nRF51822 마이크로컨트롤러에 기반한개발 보드로써, LED 와 버튼, I2C 연결 가속도계 및 나침반, 온보드 SWD 디버거를 포함하고 있습니다.

시작하기전에, 앞으로 사용할 도구를 설치해야 합니다. gLinux 또는 Debian 를사용하고 있다면 아래와 같이 하세요.

```
sudo apt install gcc-aarch64-linux-gnu gdb-multiarch libudev-dev picocom pkg-config qemu
rustup update
rustup target add aarch64-unknown-none thumbv7em-none-eabihf
rustup component add llvm-tools-preview
cargo install cargo-binutils cargo-embed
```

plugdev 그룹의 사용자에게 micro:bit 프로그래머 장치에 대한액세스 권한을 부여합니다.

```
echo 'SUBSYSTEM=="usb", ATTR{idVendor}=="0d28", MODE="0664", GROUP="plugdev"' |\
  sudo tee /etc/udev/rules.d/50-microbit.rules
sudo udevadm control --reload-rules
```

MacOS 에서:

```
xcode-select --install
brew install gdb picocom qemu
brew install --cask gcc-aarch64-embedded
rustup update
rustup target add aarch64-unknown-none thumbv7em-none-eabihf
rustup component add llvm-tools-preview
cargo install cargo-binutils cargo-embed
```

제 50 장

no_std

core

alloc

std

- Slices, &str, CStr
- NonZeroU8...
- Option, Result
- Display, Debug, write!...
- Iterator
- panic!, assert_eq!...
- NonNull 및 모든 일반적인 포인터 관련 함수
- Future and async/await
- fence, AtomicBool, AtomicPtr, AtomicU32...
- Duration

- Box, Cow, Arc, Rc
- Vec, BinaryHeap, BtreeMap, LinkedList, VecDeque
- String, CString, format!

- Error
- HashMap
- Mutex, Condvar, Barrier, Once, RwLock, mpsc
- File 및 나머지 fs
- println!, Read, Write, Stdin, Stdout 및 나머지 io
- Path, OsString
- net
- Command, Child, ExitCode
- spawn, sleep 및 나머지 thread
- SystemTime, Instant

- HashMap 은 RNG 에 의존합니다.
- std 는 core 및 alloc 를 포함합니다.

50.1 최소한의 no_std 프로그램

```
#![no_main]
#![no_std]

use core::panic::PanicInfo;

#[panic_handler]
fn panic(_panic: &PanicInfo) -> ! {
    loop {}
}
```

- 이 코드는 빈 바이너리로 컴파일됩니다.
- std 는 패닉 핸들러를 제공하지만, 우리는 자체적으로 핸들러를만들어야합니다.
- 패닉 핸들러는 panic-halt 와 같은 크레이트를 통해서 만들수도있습니다.
- 타겟에 따라 panic = "abort"로 컴파일해야 할 수 있습니다. 이는 eh_personality 에 관한 오류를 방지하기 위함입니다.
- main 과 같은 프로그램 진입점이 없습니다. 개발자가 자체적으로진입점을 정의해야 합니다. 진입점을 정의하는 작업은, 일반적으로 링커스크립트와 어셈블리 코드를 필요로 합니다.

50.2 alloc

alloc 을 사용하려면전역 (힙) 할당자를 구현해야 합니다.

```
#![no_main]
#![no_std]

extern crate alloc;
extern crate panic_halt as _;

use alloc::string::ToString;
use alloc::vec::Vec;
use buddy_system_allocator::LockedHeap;

#[global_allocator]
static HEAP_ALLOCATOR: LockedHeap<32> = LockedHeap::<32>::new();

static mut HEAP: [u8; 65536] = [0; 65536];

pub fn entry() {
    // `HEAP`이 여기서만 사용되고 `entry`가 한 번만 호출되므로 안전합니다.
    unsafe {
        // 할당자에게 할당할 메모리를 제공합니다.
        HEAP_ALLOCATOR.lock().init(HEAP.as_mut_ptr() as usize, HEAP.len());
    }

    // 이제 힙 할당이 필요한 작업을 할 수 있습니다.
    let mut v = Vec::new();
    v.push("A string".to_string());
}
```

- `buddy_system_allocator` 는 간단한 버디 시스템 할당자를구현하는 서드 파티 크레이트입니다. 이 외에도, 다른 크레이트를사용하거나, 직접 할당자를 만들거나, 이미 존재하는 다른 할당자에 후킹할수 있습니다.
- `LockHeap` 타입의 const 매개변수는 할당자의 최대 크기를 2 진수로표현했을 때의 자릿수입니다. 즉, 이 경우처럼 32 인 경우 최대 2**32 바이트크기의 영역을 다룰 수 있습니다.
- 한 바이너리에서 `alloc` 에 의존하는 크레이트가 하나라도 있다면바이너리 전체에서 전역 할당자가 반드시 하나 존재해야 합니다. 일반적으로 전역 할당자를 선언하는 작업은 최상위 바이너리 크레이트에서이루어집니다.
- `panic_halt` 크레이트가 연결되어 패닉 핸들러를 가져오도록하려면 `extern crate panic_halt as _`가 필요합니다.
- 이 예시 코드는 빌드는 되지만, 진입점이 없기 때문에 실행되지는않습니다.

제 51 장

마이크로컨트롤러

cortex_m_rt 크레이트는 Cortex M 마이크로컨트롤러를 초기화하는 핸들러를 제공합니다.

```
#![no_main]
#![no_std]

extern crate panic_halt as _;

mod interrupts;

use cortex_m_rt::entry;

#[entry]
fn main() -> ! {
    loop {}
}
```

이제, 주변장치에 액세스하는 방법을 알아보겠습니다. 가장 기계에 가까운낮은 단계에서 시작해서 점점 추상화 수준을 올리겠습니다.

- `cortex_m_rt::entry` 매크로는 진입점으로 사용되는함수가 `fn() -> !` 타입 (즉, 리턴하지 않는) 임을요구합니다. 만약, 리턴하게 되면, 프로그램 수행 후 리셋 핸들러로돌아가게 되는 것인데 이는 말이 되지 않기 때문입니다.
- `cargo embed --bin minimal` 을 사용하여 예시를 실행합니다.

51.1 원시 MMIO

대부분의 마이크로컨트롤러는 메모리 매핑 IO 를 통해 주변기기에액세스합니다. micro:bit 에서 LED 를 켜보겠습니다.

```
#![no_main]
#![no_std]

extern crate panic_halt as _;

mod interrupts;
```

```rust
use core::mem::size_of;
use cortex_m_rt::entry;

/// GPIO 포트 0 주변기기 주소
const GPIO_P0: usize = 0x5000_0000;

// GPIO 주변기기 오프셋
const PIN_CNF: usize = 0x700;
const OUTSET: usize = 0x508;
const OUTCLR: usize = 0x50c;

// PIN_CNF 필드
const DIR_OUTPUT: u32 = 0x1;
const INPUT_DISCONNECT: u32 = 0x1 << 1;
const PULL_DISABLED: u32 = 0x0 << 2;
const DRIVE_S0S1: u32 = 0x0 << 8;
const SENSE_DISABLED: u32 = 0x0 << 16;

#[entry]
fn main() -> ! {
    // GPIO 0 핀 21 및 28을 푸시-풀 출력으로 구성합니다.
    let pin_cnf_21 = (GPIO_P0 + PIN_CNF + 21 * size_of::<u32>()) as *mut u32;
    let pin_cnf_28 = (GPIO_P0 + PIN_CNF + 28 * size_of::<u32>()) as *mut u32;
    // 유효한 주변기기 제어 레지스터에 대한 포인터이고
    // 별칭이 없으므로 안전합니다.
    unsafe {
        pin_cnf_21.write_volatile(
            DIR_OUTPUT
                | INPUT_DISCONNECT
                | PULL_DISABLED
                | DRIVE_S0S1
                | SENSE_DISABLED,
        );
        pin_cnf_28.write_volatile(
            DIR_OUTPUT
                | INPUT_DISCONNECT
                | PULL_DISABLED
                | DRIVE_S0S1
                | SENSE_DISABLED,
        );
    }

    // 핀 28을 낮게, 핀 21을 높게 설정하여 LED를 켭니다.
    let gpio0_outset = (GPIO_P0 + OUTSET) as *mut u32;
    let gpio0_outclr = (GPIO_P0 + OUTCLR) as *mut u32;
    // 유효한 주변기기 제어 레지스터에 대한 포인터이고
    // 별칭이 없으므로 안전합니다.
    unsafe {
        gpio0_outclr.write_volatile(1 << 28);
        gpio0_outset.write_volatile(1 << 21);
    }
```

```
    loop {}
}
```

- GPIO 0 핀 21 은 LED 매트릭스의 첫 번째 열에 연결되고 핀 28 은 첫 번째행에 연결됩니다.

아래 명령어로 예제 코드를 실행하세요.

```
cargo embed --bin mmio
```

51.2 주변기기 액세스크레이트

svd2rust 크레이트를이용하면 메모리 매핑된 주변장치를 기술하는 CMSIS-SVD 파일로부터 Rust 래퍼를 생성할 수 있습니다.

```
#![no_main]
#![no_std]

extern crate panic_halt as _;

use cortex_m_rt::entry;
use nrf52833_pac::Peripherals;

#[entry]
fn main() -> ! {
    let p = Peripherals::take().unwrap();
    let gpio0 = p.P0;

    // GPIO 0 핀 21 및 28을 푸시-풀 출력으로 구성합니다.
    gpio0.pin_cnf[21].write(|w| {
        w.dir().output();
        w.input().disconnect();
        w.pull().disabled();
        w.drive().s0s1();
        w.sense().disabled();
        w
    });
    gpio0.pin_cnf[28].write(|w| {
        w.dir().output();
        w.input().disconnect();
        w.pull().disabled();
        w.drive().s0s1();
        w.sense().disabled();
        w
    });

    // 핀 28을 낮게, 핀 21을 높게 설정하여 LED를 켭니다.
    gpio0.outclr.write(|w| w.pin28().clear());
    gpio0.outset.write(|w| w.pin21().set());

    loop {}
}
```

- SVD(System View Description) 파일은 일반적으로 실리콘 공급업체에서제공하는 XML 파일로, 기기의 메모리 맵을 기술합니다.
  - 주변기기, 레지스터, 필드, 값으로 구성되며 이름, 설명, 주소 등이포함됩니다.
  - SVD 파일에는 버그가 있을 수 있고 불완전하기 때문에, 이러한 문제들을패치하는 다양한 프로젝트들이 있습니다.
- `cortex-m-rt` 는 무엇보다도 벡터 테이블을 제공합니다.
- `cargo install cargo-binutils` 을 실행한 후, `cargo objdump --bin pac -- -d --no-show-raw-insn` 을실행하여 생성된 바이너리의 내용을 확인할 수 있습니다.

아래 명령어로 예제 코드를 실행하세요.

```
cargo embed --bin pac
```

51.3 HAL 크레이트들

다양한 주변 장치에 대한 래퍼를 제공하는 HAL 크레이트들이 있습니다. 이 크레이트들은 일반적으로 `embedded-hal` 의트레잇을 구현합니다.

```
#![no_main]
#![no_std]

extern crate panic_halt as _;

use cortex_m_rt::entry;
use nrf52833_hal::gpio::{p0, Level};
use nrf52833_hal::pac::Peripherals;
use nrf52833_hal::prelude::*;

#[entry]
fn main() -> ! {
    let p = Peripherals::take().unwrap();

    // GPIO 포트 0의 HAL 래퍼를 만듭니다.
    let gpio0 = p0::Parts::new(p.P0);

    // GPIO 0 핀 21 및 28을 푸시-풀 출력으로 구성합니다.
    let mut col1 = gpio0.p0_28.into_push_pull_output(Level::High);
    let mut row1 = gpio0.p0_21.into_push_pull_output(Level::Low);

    // 핀 28을 낮게, 핀 21을 높게 설정하여 LED를 켭니다.
    col1.set_low().unwrap();
    row1.set_high().unwrap();

    loop {}
}
```

- `set_low` 및 `set_high` 는 `embedded_hal` `OutputPin` 트레잇의 메서드입니다.
- 다양한 STM32, GD32, nRF, NXP, MSP430, AVR, PIC 마이크로컨트롤러를비롯한 많은 Cortex-M 및 RISC-V 기기를 위한 HAL 크레이트가 있습니다

아래 명령어로 예제 코드를 실행하세요.

```
cargo embed --bin hal
```

51.4 Board support crates

보드 지원 크레이트들은, 특정 보드를 더 손쉽게 사용할 수 있게 해 주는 더높은 수준의 추상화를 제공합니다.

```
#![no_main]
#![no_std]

extern crate panic_halt as _;

use cortex_m_rt::entry;
use microbit::hal::prelude::*;
use microbit::Board;

#[entry]
fn main() -> ! {
    let mut board = Board::take().unwrap();

    board.display_pins.col1.set_low().unwrap();
    board.display_pins.row1.set_high().unwrap();

    loop {}
}
```

- 이 경우 보드 지원 크레이트는 좀 더 직관적인 이름들과 적당한 수준의초기화를 제공합니다.
- 이 크레이트는 마이크로컨트롤 밖에 있는 (즉, 보드에 설치된) 장치에 대한드라이버도 포함할 수 있습니다.
  - microbit-v2 에는 LED 매트릭스를 위한 간단한 드라이버가포함되어 있습니다.

아래 명령어로 예제 코드를 실행하세요.

```
cargo embed --bin board_support
```

51.5 타입 상태패턴

```
#[entry]
fn main() -> ! {
    let p = Peripherals::take().unwrap();
    let gpio0 = p0::Parts::new(p.P0);

    let pin: P0_01<Disconnected> = gpio0.p0_01;

    // let gpio0_01_again = gpio0.p0_01; // 오류가 발생하여 이동했습니다.
    let pin_input: P0_01<Input<Floating>> = pin.into_floating_input();
    if pin_input.is_high().unwrap() {
        // ...
    }
    let mut pin_output: P0_01<Output<OpenDrain>> = pin_input
        .into_open_drain_output(OpenDrainConfig::Disconnect0Standard1, Level::Low);
    pin_output.set_high().unwrap();
    // pin_input.is_high(); // 오류가 발생하여 이동했습니다.
```

```
    let _pin2: P0_02<Output<OpenDrain>> = gpio0
        .p0_02
        .into_open_drain_output(OpenDrainConfig::Disconnect0Standard1, Level::Low);
    let _pin3: P0_03<Output<PushPull>> =
        gpio0.p0_03.into_push_pull_output(Level::Low);

    loop {}
}
```

- 핀은 Copy 또는 Clone 을 구현하지 않으므로 각각 하나의인스턴스만 존재할 수 있습니다. 핀을 포트 구조체 밖으로 이동하면 아무도사용할 수 없습니다.
- 핀 구성을 변경하면 이전 핀 인스턴스가 사용되므로 이후에 이전인스턴스를 계속 사용할 수 없습니다.
- 값 타입은 현재 상태를 나타냅니다. 예를 들어 이 경우에는 GPIO 핀의 구성상태입니다. 이렇게 하면 상태 머신이 타입 시스템으로 인코딩되며, 먼저올바르게 구성한 후 특정 방식으로 핀을 사용하도록 합니다. 잘못된 상태전환은 컴파일 시간에 포착됩니다.
- 입력 핀에서 `is_high` 를 호출하고 출력 핀에서 `set_high` 를 호출할 수 있지만 그 반대로는 안됩니다.
- 많은 HAL 크레이트들이 이 패턴을 따릅니다.

51.6 `embedded-hal`

embedded-hal 크레이트는 다양한 마이크로컨트롤러에서 공통적으로 찾아볼 수 있는주변기기를 추상회 하는 다양한 트레잇을 제공합니다.

- GPIO
- ADC
- I2C, SPI, UART, CAN
- RNG
- 타이머
- 워치독

그러면 다른 크레이트는 이 트레잇들을 활용하여드라이버를구현합니다. 예를 들어 가속도계 드라이버를 구현할 때 I2C 또는 SPI 버스구현을 사용할 수 있습니다.

- 라스베리 파이에서 돌아가는 리눅스 같은 플랫폼 뿐만 아니라 다른 여러마이크로컨트롤러에 대한 구현이 있습니다.
- `embedded-hal` 의 'async' 버전에 관한 작업이 진행 중이지만 아직 안정적이지 않습니다.

51.7 `probe-rs` and `cargo-embed`

probe-rs 는 임베디드 시스템을 위한 도구모임입니다. OpenOCD 와 비슷하지만, Rust 에 더 잘 통합되어 있습니다.

- SWD (Serial Wire Debug) and JTAG via CMSIS-DAP, ST-Link and J-Link probes
- GDB 스텁 및 Microsoft DAP(디버그 어댑터 프로토콜) 서버
- Cargo 에 통합됨

`cargo-embed` is a cargo subcommand to build and flash binaries, log RTT (Real Time Transfers) output and connect GDB. It's configured by an `Embed.toml` file in your project directory.

- CMSIS-DAP 는 Arm 에서 정의한 프로토콜로, USB 를 통해 Arm Cortex 프로세서의 CoreSight 디버그 액세스 포트에 접근할 수 있게 해 줍니다. BBC micro:bit 에 있는온보드 디버거도 이 프로토콜을 지원합니다.
- ST-Link 는 ST Microelectronics 사에서 만든 in-circuit 디버거들이며, J-Link 는 SEGGER 사의 in-circuit 디버거들입니다.
- 디버그 액세스 포트의 물리적인 구성은 일반적으로 5 핀 JTAG 인터페이스혹은, 2 핀 Serial Wire Debug 인터페이스 입니다.
- probe-rs 는 라이브러리로 구현되어 있어서, 다른 도구들에 통합되기가쉽습니다.
- Microsoft 디버그 어댑터 프로토콜을 사용하면 VSCode 나 다른 IDE 상에서마이크로컨트롤러에서 수행중인 코드를 디버깅 할 수 있습니다.
- cargo-embed 는 probe-rs 라이브러리를 사용하여 빌드된 바이너리입니다.
- RTT(Real Time Transfers) 는 여러 링 버퍼를 통해 디버그 호스트와 타겟간에 데이터를 전송하는 메커니즘입니다.

51.7.1 디버깅

Embed.toml:

```
[default.general]
chip = "nrf52833_xxAA"

[debug.gdb]
enabled = true
```

`src/bare-metal/microcontrollers/example/` 에 터미널을 열고:

```
cargo embed --bin board_support debug
```

In another terminal in the same directory:

gLinux 또는 Debian 에서:

```
gdb-multiarch target/thumbv7em-none-eabihf/debug/board_support --eval-command="target re
```

MacOS 에서:

```
arm-none-eabi-gdb target/thumbv7em-none-eabihf/debug/board_support --eval-command="targe
```

GDB 에서 다음을 실행해 보세요.

```
b src/bin/board_support.rs:29
b src/bin/board_support.rs:30
b src/bin/board_support.rs:32
c
c
c
```

51.8 다른프로젝트

- RTIC
  - "실시간 인터럽트 기반 동시 실행 (Real-Time Interrupt-driven Concurrency)"
  - 공유 리소스 관리, 메시지 전달, 태스크 스케줄링, 타이머 대기열 지원
- Embassy
  - 우선순위, 타이머, 네트워킹, USB 가 포함된 `async` 실행자
- TockOS

  - 선점형 스케줄링 및 MMU 를 지원하는, 보안에 중점을 둔 실시간 운영체제
- Hubris
  - Oxide Computer Company 에서 만든 마이크로커널 기반 실시간 운영체제로, 메모리 보호, 권한이 없이 수행되는 드라이버 등을 지원함.
- FreeRTOS 용 바인딩
- `std` 가 구현된 플랫폼도 있습니다 (예: esp-idf).

- RTIC 는 실시간 운영체제로 볼 수도 있고, 동시성 지원을 위한 프레임워크로볼 수도 있습니다.
  - HAL 을 포함하지는 않습니다.
  - 스케줄링은 커널이 아니라 Cortex-M NVIC(Nested Virtual Interrupt Controller) 로 구현이 됩니다.
  - Cortex-M 전용입니다.
- Google 에서는 Titan 보안 키에 사용되는 Haven 마이크로컨트롤러에서 TockOS 를 사용합니다.
- FreeRTOS 는 대부분 C 로 작성되지만, 애플리케이션을 Rust 로 작성할 수있도록 해 주는 Rust 바인딩이 제공됩니다.

제 52 장

연습문제

I2C 나침반에서 방향을 읽고, 읽은 값을 직렬 포트에 기록하세요.

After looking at the exercises, you can look at the solutions provided.

52.1 나침반

I2C 나침반에서 방향을 읽고 판독값을 직렬 포트에 기록하세요. 시간이있으면 어떻게든 LED 에 표시하거나 버튼을 사용하세요.

힌트:

- `lsm303agr` 및 `microbit-v2` 크레이트 및 micro:bit 하드웨어의 문서를 확인하세요.
- LSM303AGR 관성 측정 장치는 내부 I2C 버스에 연결됩니다.
- TWI 는 I2C 의 또 다른 이름이므로 I2C 마스터 주변기기는 TWIM 이라고합니다.
- LSM303AGR 드라이버에는 `embedded_hal::blocking::i2c::WriteRead` 트레잇을 구현하는것이 필요합니다. `microbit::hal::Twim` 구조체가 이를 구현합니다.
- 다양한 핀과 주변기기에 관한 필드가 있는 `microbit::Board` 구조체가 있습니다.
- 원하는 경우 nRF52833 데이터시트를 확인할 수도 있지만 이 연습에서 반드시 필요한 것은아닙니다.

연습템플릿을 다운로드하고 `compass` 디렉터리에서 다음 파일을찾습니다.

src/main.rs:

```
#![no_main]
#![no_std]

extern crate panic_halt as _;

use core::fmt::Write;
use cortex_m_rt::entry;
use microbit::{hal::uarte::{Baudrate, Parity, Uarte}, Board};

#[entry]
fn main() -> ! {
    let board = Board::take().unwrap();
```

```
    // Configure serial port.
    let mut serial = Uarte::new(
        board.UARTE0,
        board.uart.into(),
        Parity::EXCLUDED,
        Baudrate::BAUD115200,
    );

    // Use the system timer as a delay provider.
    let mut delay = Delay::new(board.SYST);

    // Set up the I2C controller and Inertial Measurement Unit.
    // TODO

    writeln!(serial, "Ready.").unwrap();

    loop {
        // Read compass data and log it to the serial port.
        // TODO
    }
}
```

Cargo.toml (변경할 필요가 없음):

```
[workspace]

[package]
name = "compass"
version = "0.1.0"
edition = "2021"
publish = false

[dependencies]
cortex-m-rt = "0.7.3"
embedded-hal = "1.0.0"
lsm303agr = "0.3.0"
microbit-v2 = "0.13.0"
panic-halt = "0.2.0"
```

Embed.toml (변경할 필요가 없음):

```
[default.general]
chip = "nrf52833_xxAA"

[debug.gdb]
enabled = true

[debug.reset]
halt_afterwards = true
```

.cargo/config.toml (변경할 필요가 없음):

```
[build]
target = "thumbv7em-none-eabihf" # Cortex-M4F
```

```
[target.'cfg(all(target_arch = "arm", target_os = "none"))']
rustflags = ["-C", "link-arg=-Tlink.x"]
```

다음을 사용하여 Linux 에서 직렬 출력을 확인하세요.

```
picocom --baud 115200 --imap lfcrlf /dev/ttyACM0
```

또는 다음과 같이 Mac OS(기기 이름은 약간 다를 수 있음) 에서 확인하세요.

```
picocom --baud 115200 --imap lfcrlf /dev/tty.usbmodem14502
```

Ctrl+A Ctrl+Q 를 사용하여 picocom 을 종료합니다.

52.2 Bare Metal Rust Morning Exercise

나침반

(연습문제로돌아가기)

```
#![no_main]
#![no_std]

extern crate panic_halt as _;

use core::fmt::Write;
use cortex_m_rt::entry;
use core::cmp::{max, min};
use lsm303agr::{
    AccelMode, AccelOutputDataRate, Lsm303agr, MagMode, MagOutputDataRate,
};
use microbit::display::blocking::Display;
use microbit::hal::prelude::*;
use microbit::hal::twim::Twim;
use microbit::hal::uarte::{Baudrate, Parity, Uarte};
use microbit::hal::{Delay, Timer};
use microbit::pac::twim0::frequency::FREQUENCY_A;
use microbit::Board;

const COMPASS_SCALE: i32 = 30000;
const ACCELEROMETER_SCALE: i32 = 700;

#[entry]
fn main() -> ! {
    let board = Board::take().unwrap();

    // 직렬 포트를 설정하세요.
    let mut serial = Uarte::new(
        board.UARTE0,
        board.uart.into(),
        Parity::EXCLUDED,
        Baudrate::BAUD115200,
    );
```

```
// Use the system timer as a delay provider.
let mut delay = Delay::new(board.SYST);

// I2C 컨트롤러와 관성 측정 장치를 설정합니다.
writeln!(serial, "IMU 설정 중...").unwrap();
let i2c = Twim::new(board.TWIM0, board.i2c_internal.into(), FREQUENCY_A::K100);
let mut imu = Lsm303agr::new_with_i2c(i2c);
imu.init().unwrap();
imu.set_mag_mode_and_odr(
    &mut delay,
    MagMode::HighResolution,
    MagOutputDataRate::Hz50,
)
.unwrap();
imu.set_accel_mode_and_odr(
    &mut delay,
    AccelMode::Normal,
    AccelOutputDataRate::Hz50,
)
.unwrap();
let mut imu = imu.into_mag_continuous().ok().unwrap();

// 디스플레이 및 타이머를 설정합니다.
let mut timer = Timer::new(board.TIMER0);
let mut display = Display::new(board.display_pins);

let mut mode = Mode::Compass;
let mut button_pressed = false;

writeln!(serial, "Ready").unwrap();

loop {
    // 나침반 데이터를 읽고 직렬 포트에 기록합니다.
    while !(imu.mag_status().unwrap().xyz_new_data()
        && imu.accel_status().unwrap().xyz_new_data())
    {}
    let compass_reading = imu.magnetic_field().unwrap();
    let accelerometer_reading = imu.acceleration().unwrap();
    writeln!(
        serial,
        "{},{},{}\t{},{},{}",
        compass_reading.x_nt(),
        compass_reading.y_nt(),
        compass_reading.z_nt(),
        accelerometer_reading.x_mg(),
        accelerometer_reading.y_mg(),
        accelerometer_reading.z_mg(),
    )
    .unwrap();
```

```
        let mut image = [[0; 5]; 5];
        let (x, y) = match mode {
            Mode::Compass => (
                scale(-compass_reading.x_nt(), -COMPASS_SCALE, COMPASS_SCALE, 0, 4)
                    as usize,
                scale(compass_reading.y_nt(), -COMPASS_SCALE, COMPASS_SCALE, 0, 4)
                    as usize,
            ),
            Mode::Accelerometer => (
                scale(
                    accelerometer_reading.x_mg(),
                    -ACCELEROMETER_SCALE,
                    ACCELEROMETER_SCALE,
                    0,
                    4,
                ) as usize,
                scale(
                    -accelerometer_reading.y_mg(),
                    -ACCELEROMETER_SCALE,
                    ACCELEROMETER_SCALE,
                    0,
                    4,
                ) as usize,
            ),
        };
        image[y][x] = 255;
        display.show(&mut timer, image, 100);

        // 버튼 A를 누르면 다음 모드로 전환되고 모든 LED가 잠시 깜박입니다.
        if board.buttons.button_a.is_low().unwrap() {
            if !button_pressed {
                mode = mode.next();
                display.show(&mut timer, [[255; 5]; 5], 200);
            }
            button_pressed = true;
        } else {
            button_pressed = false;
        }
    }
}

#[derive(Copy, Clone, Debug, Eq, PartialEq)]
enum Mode {
    Compass,
    Accelerometer,
}

impl Mode {
    fn next(self) -> Self {
        match self {
            Self::Compass => Self::Accelerometer,
```

```
            Self::Accelerometer => Self::Compass,
        }
    }
}

fn scale(value: i32, min_in: i32, max_in: i32, min_out: i32, max_out: i32) -> i32 {
    let range_in = max_in - min_in;
    let range_out = max_out - min_out;
    cap(min_out + range_out * (value - min_in) / range_in, min_out, max_out)
}

fn cap(value: i32, min_value: i32, max_value: i32) -> i32 {
    max(min_value, min(value, max_value))
}
```

제 XII 편

Bare Metal 오후

제 53 장

애플리케이션프로세서

지금까지 Arm Cortex-M 시리즈와 같은 마이크로컨트롤러에 관해알아봤습니다. 이제 애플리케이션 프로세서인 Cortex-A 를 위한 코드를 작성해보겠습니다. 편의상 QEMU 의 aarch64 'virt' 보드를 사용합니다.

- 일반적으로 마이크로컨트롤러에는 MMU 또는 다중 레벨 권한 (Arm CPU 에서는익셉션 레벨 (exception level), x86 에서는 링 (ring)) 이 없습니다. 애플리케이션 프로세서는 이들을 가지고 있습니다.
- QEMU 는 아키텍처별로 다양한 머신 또는 보드 모델을 에뮬레이션할 수있습니다. 'virt' 보드는 특정 실제하드웨어를 에뮬레이션 하지 않으며, 가상 머신용으로만 설계되었습니다.

53.1 Rust 수행준비

Rust 코드 실행을 시작하기 전에 몇 가지 초기화를 실행해야 합니다.

```
.section .init.entry, "ax"
.global entry
entry:
    /*
     * 메모리 관리 구성을 로드하고 적용합니다. MMU 와 캐시를
     * 사용 설정할 준비가 되었습니다. .
     */
 adrp x30, idmap
    msr ttbr0_el1, x30

    mov_i x30, .Lmairval
    msr mair_el1, x30

    mov_i x30, .Ltcrval
    /* 지원되는 PA 범위를 TCR_EL1.IPS 에 복사합니다. */
    mrs x29, id_aa64mmfr0_el1
    bfi x30, x29, #32, #4

    msr tcr_el1, x30

    mov_i x30, .Lsctlrval
```

```
    /*
     * 이 지점 앞에 오는 모든 내용이 완료되었는지 확인하고
     * 오래된 로컬 TLB 항목을 사용하기 전에 무효화합니다.
     */
    isb
    tlbi vmalle1
    ic iallu
    dsb nsh
    isb

    /*
     * MMU와 캐시를 사용하도록 sctlr_el1을 구성하고 이 작업이
     * 완료될 때까지 진행하지 않습니다.
     */
    msr sctlr_el1, x30
    isb

    /* EL1에서 트래핑 부동 소수점 액세스를 사용 중지합니다. */
    mrs x30, cpacr_el1
    orr x30, x30, #(0x3 << 20)
    msr cpacr_el1, x30
    isb

    /* bss 섹션을 0으로 만듭니다. */
    adr_l x29, bss_begin
    adr_l x30, bss_end
0:  cmp x29, x30
    b.hs 1f
    stp xzr, xzr, [x29], #16
    b 0b

1:  /* 스택을 준비합니다. */
    adr_l x30, boot_stack_end
    mov sp, x30

    /* 예외 벡터를 설정합니다. */
    adr x30, vector_table_el1
    msr vbar_el1, x30

    /* Rust 코드를 호출합니다. */
    bl main

    /* 루프가 인터럽트를 영원히 기다립니다. */
2:  wfi
    b 2b
```

- 이는 C 의 경우와 동일합니다. 프로세서 상태를 초기화하고 BSS 를 0 으로설정하고 스택 포인터를 설정합니다.
  - BSS(역사적인 이유로, 블록 시작 기호 block starting symbol 이라고불림) 는 오브젝트 파일에서 0 으로 초기화된 정적으로 할당된 변수들을담고 있습니다. BSS 는 실제로 이미지에

포함되지는 않습니다. 어차피 0 으로 초기화 될 것이기 때문입니다. 컴파일러는 로더 프로그램이 BSS 에속하는 변수들을 0 으로 초기화 할 것으로기대하고 오브젝트 파일을만듭니다.

- 메모리가 초기화되고 이미지가 로드되는 방식에 따라 BSS 가 이미 0 으로설정되었을 수도 있지만 확실히 하기 위해 0 으로 설정합니다.
- 메모리를 읽거나 쓰기 전에 MMU 와 캐시를 사용 설정해야 합니다. 그러지않으면 다음과 같은 상황이 발생할 수 있습니다.
  - 정렬되지 않은 액세스에 오류가 발생합니다. 컴파일러가 정렬되지 않은액세스를 생성하지 않도록 `+strict-align` 을 설정하는 `aarch64-unknown-none` 타겟의 Rust 코드를 빌드하므로 이경우에는 문제가 없지만, 일반적으로 반드시 그런 것은 아닙니다.
  - VM 에서 실행한다면 캐시 일관성 문제가 발생할 수 있습니다. 문제는 VM 은캐시가 사용 중지된 상태로 메모리에 직접 액세스하는 반면 호스트에는동일한 메모리에 대해 캐시할 수 있는 별칭이 있다는 점입니다. 호스트가메모리에 명시적으로 액세스하지 않더라도 추측 액세스는 캐시 채우기로이어질 수 있으며, 캐시가 정리되거나 VM 이 캐시를 사용 설정하면 둘 중 하나의 변경사항이 손실됩니다. 캐시는 VA 또는 IPA 가 아닌 실제 주소로키가 지정됩니다.
- 편의상 기기용 주소 공간의 처음 1GiB, DRAM 용 다음 1GiB, 더 많은 기기를위한 또 다른 1GiB 를 ID 매핑하는 하드코딩된 페이지테이블 (`idmap.S` 참고) 을 사용합니다. 이는 QEMU 에서 사용하는메모리 레이아웃과 일치합니다.
- 예외 벡터 (`vbar_el1`) 도 설정합니다. 이에 관해서는 나중에자세히 알아봅니다.
- 오늘 오후의 모든 예에서는 예외 수준 1(EL1) 에서 실행한다고 가정합니다. 다른 예외 수준에서 실행해야 하는 경우 이에 따라 `entry.S` 를수정해야 합니다.

53.2 인라인어셈블리

Sometimes we need to use assembly to do things that aren't possible with Rust code. For example, to make an HVC (hypervisor call) to tell the firmware to power off the system:

```
#![no_main]
#![no_std]

use core::arch::asm;
use core::panic::PanicInfo;

mod exceptions;

const PSCI_SYSTEM_OFF: u32 = 0x84000008;

#[no_mangle]
extern "C" fn main(_x0: u64, _x1: u64, _x2: u64, _x3: u64) {
    // 선언된 레지스터만 사용하고 메모리로는
    // 아무것도 하지 않으므로 안전합니다.
    unsafe {
        asm!("hvc #0",
            inout("w0") PSCI_SYSTEM_OFF => _,
            inout("w1") 0 => _,
            inout("w2") 0 => _,
            inout("w3") 0 => _,
            inout("w4") 0 => _,
            inout("w5") 0 => _,
            inout("w6") 0 => _,
            inout("w7") 0 => _,
```

```
            options(nomem, nostack)
        );
    }

    loop {}
}
```

실제로 이를 실행하려면 이러한 모든 함수를 위한 래퍼가 포함된 smccc 크레이트를사용하세요.

- PSCI (Power State Coordination Interface) 는 시스템 및 CPU 전원 상태를관리하는 Arm 의 표준 인터페이스입니다. 이 인터페이스는 EL3 펌웨어와하이퍼바이저에 의해 구현됩니다.
- `0 => _` 문법은 인라인 어셈블리 코드를 실행하기전에 레지스터를 0 으로 초기화하고 그 후에는 그 레지스터의 값을 무시함을의미합니다. 호출 시 레지스터의 값이 덮어 써질 수 있으므로 `in` 대신 `inout` 을 사용해야 합니다.
- 이 `main` 함수는 `#[no_mangle]` 및 `extern "C"`여야 합니다. 왜냐하면 이 함수는 Rust 코드가 아닌, 어셈블러로 작성된 `entry.S` 에서 호출되기 때문입니다.
- `_x0` –`_x3` 는 `x0` 에서 `x3` 레지스터들의값입니다. 이 레지스터들은 일반적으로 부트로더에서 디바이스 트리에 대한포인터 등을 전달할 때 사용됩니다. 표준 aarch64 호출규약 (`extern "C"`에서 사용하도록 지정) 에 따라 레지스터 `x0` 에서 `x7` 이 함수에 전달된 처음 8 개 인수에 사용되므로 `entry.S` 는 이러한 레지스터를 변경하지 않는지 확인하는 것외에는 특별히 할 작업이 없습니다.
- `src/bare-metal/aps/examples` 에서 `make qemu_psci` 를입력하면 예제 코드가 QEMU 에서 수행됩니다.

53.3 MMIO 를 위한 휘발성 (volatile) 메모리액세스

- `pointer::read_volatile` 및 `pointer::write_volatile` 을 사용하세요.
- 참조를 유지하지 마세요.
- `addr_of!` 를 사용하면 임시 용도의 참조를 만들지 않고도구조체의 필드를 읽을 수 있습니다.
- 휘발성 (volatile) 액세스: 읽기 또는 쓰기 작업이 부수 효과 (side effect) 를 동반할 수 있기 때문에 컴파일러나 하드웨어가 임의로 이를 읽기쓰기 작업의 순서를 바꾸거나, 중복해서 수행하거나 또는 제거하지 못하게합니다.
  - 일반적으로 쓰고 난 후 읽으면 (예: 변경 가능한 참조를 통해) 컴파일러는읽은 값이 방금 쓴 값과 동일하다고 가정하고 실제로 메모리를 읽지 않을수 있습니다.
- 하드웨어에 대한 휘발성 액세스를 위한 일부 기존 크레이트는 참조를유지하지만 이는 올바른 것이 아닙니다. 참조가 있을 때마다 컴파일러는이를 역참조하도록 선택할 수 있습니다.
- `addr_of!` 매크로를 사용하여 구조체 포인터에서 구조체 필드포인터를 가져옵니다.

53.4 UART 드라이버작성

QEMU 의 'virt' 보드에는 PL011 UART 가있으므로 이를 위한 드라이버를 작성해 보겠습니다.

```
const FLAG_REGISTER_OFFSET: usize = 0x18;
const FR_BUSY: u8 = 1 << 3;
const FR_TXFF: u8 = 1 << 5;

/// PL011 UART 의 최소 드라이버입니다.
#[derive(Debug)]
pub struct Uart {
    base_address: *mut u8,
}
```

```
impl Uart {
    /// 지정된 기본 주소에 PL011 기기에 대한 UART 드라이버의 새 인스턴스를
 /// 생성합니다.
 ///
 /// # 안전
 ///
 /// 지정된 기본 주소는 PL011 기기의
 /// MMIO 제어 레지스터 8 개를 가리켜야 하며,
 /// 이는 프로세스의 주소 공간에 기기 메모리로
 /// 매핑되어야 하며 다른 별칭은 없어야 합니다.
    pub unsafe fn new(base_address: *mut u8) -> Self {
        Self { base_address }
    }

    /// UART 에 단일 바이트를 씁니다.
    pub fn write_byte(&self, byte: u8) {
        // TX 버퍼에 공간이 확보될 때까지 기다립니다.
        while self.read_flag_register() & FR_TXFF != 0 {}

        // 기본 주소가 적절하게 매핑된 PL011 기기의 제어 레지스터를
        // 가리키고 있으므로 안전합니다.
        unsafe {
            // TX 버퍼에 씁니다.
            self.base_address.write_volatile(byte);
        }

        // UART 가 더 이상 사용 중이 아닐 때까지 기다립니다.
        while self.read_flag_register() & FR_BUSY != 0 {}
    }

    fn read_flag_register(&self) -> u8 {
        // 기본 주소가 적절하게 매핑된 PL011 기기의 제어 레지스터를
        // 가리키고 있으므로 안전합니다.
        unsafe { self.base_address.add(FLAG_REGISTER_OFFSET).read_volatile() }
    }
}
```

- `Uart::new` 는 안전하지 않지만 (usafe), 그 외 다른 메서드들은안전한 (safe) 점에 주목하세요. 다른 메서드들이 안전할 수 있는 이유는, `Uart::new` 의 안전 요구사항 (즉, 지정된 UART 의 드라이버인스턴스가 하나만 있으며 주소 공간에 별칭을 지정하는 다른 항목이 없음) 이 만족되기만 하면 `write_byte` 와 같은 함수를 안전하게호출하는데 있어서 필요한 모든 전제조건이 만족되기 때문입니다.
- 반대 방법으로도 실행할 수 있지만 (`new` 를 안전하게 만들고 `write_byte` 를 안전하지 않게 만듦) 이는 `write_byte` 를호출하는 모든 위치에서 안전성에 관해 추론해야 하므로 사용 편의성이 훨씬 떨어집니다.
- 이는 안전하지 않은 코드의 안전한 래퍼를 작성하는 일반적인 패턴입니다. 안전에 관한 증명 부담을 여러 많은 위치에서 소수의 위치로 옮기는것입니다.

53.4.1 More traits

Debug 트레잇을 상속했습니다. 몇 가지 트레잇을 추가로 더구현하는 것도 유용할 수 있습니다.

```
use core::fmt::{self, Write};

impl Write for Uart {
    fn write_str(&mut self, s: &str) -> fmt::Result {
        for c in s.as_bytes() {
            self.write_byte(*c);
        }
        Ok(())
    }
}

// 모든 컨텍스트에서 액세스할 수 있는 기기 메모리에 대한
// 포인터만 포함하므로 안전합니다.
unsafe impl Send for Uart {}
```

- `Write` 를 구현하면 `write!` 및 `writeln!` 매크로를 `Uart` 타입과 함께 사용할 수 있습니다.
- `src/bare-metal/aps/examples` 에서 `make qemu_minimal` 을 사용하여 QEMU 에서 예를 실행합니다.

53.5 더 나은 UART 드라이버

PL011 에는 실제로훨씬더 많은 레지스터가 있으며, 이에 액세스할 포인터를 구성하기 위해오프셋을 추가하면 오류가 발생하기 쉽고 읽기 어렵습니다. 또한 그중 일부는구조화된 방식으로 액세스할 수 있는 비트 필드입니다.

오프셋	레지스터 이름	너비
0x00	DR	12
0x04	RSR	4
0x18	FR	9
0x20	ILPR	8
0x24	IBRD	16
0x28	FBRD	6
0x2c	LCR_H	8
0x30	CR	16
0x34	IFLS	6
0x38	IMSC	11
0x3c	RIS	11
0x40	MIS	11
0x44	ICR	11
0x48	DMACR	3

- 간결성을 위해 일부 ID 레지스터는 생략되었습니다.

53.5.1 비트플래그

`bitflags` 크레이트는비트플래그를 사용하는 데 유용합니다.

```
use bitflags::bitflags;

bitflags! {
    /// UART 플래그 레지스터의 플래그입니다.
    #[repr(transparent)]
    #[derive(Copy, Clone, Debug, Eq, PartialEq)]
    struct Flags: u16 {
        /// 보내려면 지웁니다.
        const CTS = 1 << 0;
        /// 데이터 세트가 준비되었습니다.
        const DSR = 1 << 1;
        /// 데이터 이동통신사 감지
        const DCD = 1 << 2;
        /// UART 에서 데이터를 전송 중입니다.
        const BUSY = 1 << 3;
        /// 수신 FIFO 가 비어 있습니다.
        const RXFE = 1 << 4;
        /// 전송 FIFO 가 가득 찼습니다.
        const TXFF = 1 << 5;
        /// 수신 FIFO 가 가득 찼습니다.
        const RXFF = 1 << 6;
        /// 전송 FIFO 가 비어 있습니다.
        const TXFE = 1 << 7;
        /// 벨소리 표시기입니다.
        const RI = 1 << 8;
    }
}
```

- `bitflags!` 매크로는 플래그를 가져오고 설정하기 위한 여러메서드 구현과 함께 `Flags(u16)` 와 같은 새로운 타입을생성합니다.

53.5.2 Multiple registers

구조체를 사용하여 UART 레지스터의 메모리 레이아웃을 나타낼 수 있습니다.

```
#[repr(C, align(4))]
struct Registers {
    dr: u16,
    _reserved0: [u8; 2],
    rsr: ReceiveStatus,
    _reserved1: [u8; 19],
    fr: Flags,
    _reserved2: [u8; 6],
    ilpr: u8,
    _reserved3: [u8; 3],
    ibrd: u16,
    _reserved4: [u8; 2],
    fbrd: u8,
    _reserved5: [u8; 3],
    lcr_h: u8,
    _reserved6: [u8; 3],
    cr: u16,
```

```
    _reserved7: [u8; 3],
    ifls: u8,
    _reserved8: [u8; 3],
    imsc: u16,
    _reserved9: [u8; 2],
    ris: u16,
    _reserved10: [u8; 2],
    mis: u16,
    _reserved11: [u8; 2],
    icr: u16,
    _reserved12: [u8; 2],
    dmacr: u8,
    _reserved13: [u8; 3],
}
```

- `#[repr(C)]` 는 C 와 동일한 규칙에 따라 구조체 필드를 순서대로 배치하도록 컴파일러에지시합니다. 기본 Rust 표현을 사용하면 컴파일러가 원하는 대로 필드의순서를 변경할 수 있으므로 구조체에서 예측 가능한 레이아웃을 사용하는데 필요합니다.

53.5.3 드라이버

이제 드라이버에서 새로운 `Registers` 구조체를 사용해 보겠습니다.

```
/// PL011 UART용 드라이버
#[derive(Debug)]
pub struct Uart {
    registers: *mut Registers,
}

impl Uart {
    /// 지정된 기본 주소에 PL011 기기에 대한 UART 드라이버의 새 인스턴스를
 /// 생성합니다.
 ///
 /// # 안전
 ///
 /// 지정된 기본 주소는 PL011 기기의
 /// MMIO 제어 레지스터 8개를 가리켜야 하며,
 /// 이는 프로세스의 주소 공간에 기기 메모리로
 /// 매핑되어야 하며 다른 별칭은 없어야 합니다.
    pub unsafe fn new(base_address: *mut u32) -> Self {
        Self { registers: base_address as *mut Registers }
    }

    /// UART에 단일 바이트를 씁니다.
    pub fn write_byte(&self, byte: u8) {
        // TX 버퍼에 공간이 확보될 때까지 기다립니다.
        while self.read_flag_register().contains(Flags::TXFF) {}

        // self.registers가 적절하게 매핑된 PL011 기기의 제어 레지스터를
        // 가리키고 있으므로 안전합니다.
        unsafe {
            // TX 버퍼에 씁니다.
```

```
            addr_of_mut!((*self.registers).dr).write_volatile(byte.into());
        }

        // UART가 더 이상 사용 중이 아닐 때까지 기다립니다.
        while self.read_flag_register().contains(Flags::BUSY) {}
    }

    /// Reads and returns a pending byte, or `None` if nothing has been
    /// received.
    pub fn read_byte(&self) -> Option<u8> {
        if self.read_flag_register().contains(Flags::RXFE) {
            None
        } else {
            let data = unsafe { addr_of!((*self.registers).dr).read_volatile() };
            // TODO: 비트 8~11에서 오류 상태를 확인합니다.
            Some(data as u8)
        }
    }

    fn read_flag_register(&self) -> Flags {
        // self.registers가 적절하게 매핑된 PL011 기기의 제어 레지스터를
        // 가리키고 있으므로 안전합니다.
        unsafe { addr_of!((*self.registers).fr).read_volatile() }
    }
}
```

- `addr_of!` / `addr_of_mut!`를 사용하여 중간 참조를만들지 않고 개별 필드 포인터를 가져오는 것은 불안정할 수 있습니다.

53.5.4 Using it

드라이버를 사용하여 직렬 콘솔에 쓰고 수신되는 바이트를 에코하는 간단한프로그램을 작성해 보겠습니다.

```
#![no_main]
#![no_std]

mod exceptions;
mod pl011;

use crate::pl011::Uart;
use core::fmt::Write;
use core::panic::PanicInfo;
use log::error;
use smccc::psci::system_off;
use smccc::Hvc;

/// 기본 PL011 UART의 기본 주소입니다.
const PL011_BASE_ADDRESS: *mut u32 = 0x900_0000 as _;

#[no_mangle]
extern "C" fn main(x0: u64, x1: u64, x2: u64, x3: u64) {
```

```
    // `PL011_BASE_ADDRESS`가 PL011 기기의 기본 주소이고
    // 이 주소 범위에 액세스하는 다른 항목이 없으므로 안전합니다.
    let mut uart = unsafe { Uart::new(PL011_BASE_ADDRESS) };

    writeln!(uart, "main({x0:#x}, {x1:#x}, {x2:#x}, {x3:#x})").unwrap();

    loop {
        if let Some(byte) = uart.read_byte() {
            uart.write_byte(byte);
            match byte {
                b'\r' => {
                    uart.write_byte(b'\n');
                }
                b'q' => break,
                _ => {}
            }
        }
    }

    writeln!(uart, "안녕!").unwrap();
    system_off::<Hvc>().unwrap();
}
```

- 인라인어셈블리 예에서와 같이 이 main 함수는 entry.S 의진입점 코드에서 호출됩니다. 자세한 내용은 해당 발표자 노트를참고하세요.
- src/bare-metal/aps/examples 에서 make qemu 를사용하여 QEMU 에서 예를 실행합니다.

53.6 로깅

log 크레이트의 로깅매크로를 사용할 수 있으면 좋습니다. 이는 Log 트레잇을 구현하면됩니다.

```
use crate::pl011::Uart;
use core::fmt::Write;
use log::{LevelFilter, Log, Metadata, Record, SetLoggerError};
use spin::mutex::SpinMutex;

static LOGGER: Logger = Logger { uart: SpinMutex::new(None) };

struct Logger {
    uart: SpinMutex<Option<Uart>>,
}

impl Log for Logger {
    fn enabled(&self, _metadata: &Metadata) -> bool {
        true
    }

    fn log(&self, record: &Record) {
        writeln!(
            self.uart.lock().as_mut().unwrap(),
```

```
            "[{}] {}",
            record.level(),
            record.args()
        )
        .unwrap();
    }

    fn flush(&self) {}
}

/// UART 로거를 초기화합니다.
pub fn init(uart: Uart, max_level: LevelFilter) -> Result<(), SetLoggerError> {
    LOGGER.uart.lock().replace(uart);

    log::set_logger(&LOGGER)?;
    log::set_max_level(max_level);
    Ok(())
}
```

- log 함수 안에서 unwrap 하는 것은 괜찮습니다.. 왜냐하면 set_logger 를 호출하기 전에 LOGGER 를 초기화하기때문입니다.

53.6.1 Using it

로거를 사용하려면 먼저 초기화해야 합니다.

```
#![no_main]
#![no_std]

mod exceptions;
mod logger;
mod pl011;

use crate::pl011::Uart;
use core::panic::PanicInfo;
use log::{error, info, LevelFilter};
use smccc::psci::system_off;
use smccc::Hvc;

/// 기본 PL011 UART의 기본 주소입니다.
const PL011_BASE_ADDRESS: *mut u32 = 0x900_0000 as _;

#[no_mangle]
extern "C" fn main(x0: u64, x1: u64, x2: u64, x3: u64) {
    // `PL011_BASE_ADDRESS`가 PL011 기기의 기본 주소이고
    // 이 주소 범위에 액세스하는 다른 항목이 없으므로 안전합니다.
    let uart = unsafe { Uart::new(PL011_BASE_ADDRESS) };
    logger::init(uart, LevelFilter::Trace).unwrap();

    info!("main({x0:#x}, {x1:#x}, {x2:#x}, {x3:#x})");

    assert_eq!(x1, 42);
```

```
    system_off::<Hvc>().unwrap();
}

#[panic_handler]
fn panic(info: &PanicInfo) -> ! {
    error!("{info}");
    system_off::<Hvc>().unwrap();
    loop {}
}
```

- 패닉 핸들러가 이제 패닉의 세부정보를 기록할 수 있습니다.
- `src/bare-metal/aps/examples` 에서 `make qemu_logger` 를사용하여 QEMU 에서 예를 실행합니다.

53.7 예외

AArch64 는 4 개 상태 (SP0 을 사용하는 현재 EL, SPx 를 사용하는 현재 EL, AArch64 를 사용하는 하위 EL, AArch32 를 사용하는 하위 EL) 의 4 가지 예외타입 (동기, IRQ, FIQ, SError) 에 대해 16 개 항목이 있는 예외 벡터 테이블을정의합니다. Rust 코드를 호출하기 전에 휘발성 레지스터를 스택에 저장하기위해 어셈블리에서 이를 구현합니다.

```
use log::error;
use smccc::psci::system_off;
use smccc::Hvc;

#[no_mangle]
extern "C" fn sync_exception_current(_elr: u64, _spsr: u64) {
    error!("sync_exception_current");
    system_off::<Hvc>().unwrap();
}

#[no_mangle]
extern "C" fn irq_current(_elr: u64, _spsr: u64) {
    error!("irq_current");
    system_off::<Hvc>().unwrap();
}

#[no_mangle]
extern "C" fn fiq_current(_elr: u64, _spsr: u64) {
    error!("fiq_current");
    system_off::<Hvc>().unwrap();
}

#[no_mangle]
extern "C" fn serr_current(_elr: u64, _spsr: u64) {
    error!("serr_current");
    system_off::<Hvc>().unwrap();
}

#[no_mangle]
```

```
extern "C" fn sync_lower(_elr: u64, _spsr: u64) {
    error!("sync_lower");
    system_off::<Hvc>().unwrap();
}

#[no_mangle]
extern "C" fn irq_lower(_elr: u64, _spsr: u64) {
    error!("irq_lower");
    system_off::<Hvc>().unwrap();
}

#[no_mangle]
extern "C" fn fiq_lower(_elr: u64, _spsr: u64) {
    error!("fiq_lower");
    system_off::<Hvc>().unwrap();
}

#[no_mangle]
extern "C" fn serr_lower(_elr: u64, _spsr: u64) {
    error!("serr_lower");
    system_off::<Hvc>().unwrap();
}
```

- EL 은 예외 수준입니다. 오늘 오후의 모든 예는 EL1 에서 실행됩니다.
- 편의상 현재 EL 예외의 SP0 과 SPx 를 구별하거나 하위 EL 예외의 AArch32 와 AArch64 를 구별하지 않습니다.
- 이 예에서는 예외가 실제로 발생할 것으로 예상되지 않으므로 예외를기록하고 전원을 끕니다.
- 예외 핸들러와 기본 실행 컨텍스트는 서로 다른 스레드와 거의 비슷하다고생각할 수 있습니다. Send 및 Sync 는 스레드와 마찬가지로 이들 간에 공유할 수 있는항목을 제어합니다. 예를 들어 예외 핸들러와 프로그램의 나머지 부분 간에값을 공유하려고 하는데 Send 이지만 Sync 는 아닌 경우, Mutex 와 같은 것으로 래핑하고 정적인 것으로 배치해야 합니다.

53.8 다른프로젝트

- oreboot
  - ”coreboot without the C”
  - Supports x86, aarch64 and RISC-V.
  - Relies on LinuxBoot rather than having many drivers itself.
- Rust RaspberryPi OS tutorial
  - Initialisation, UART driver, simple bootloader, JTAG, exception levels, exception handling, page tables
  - Some dodginess around cache maintenance and initialisation in Rust, not necessarily a good example to copy for production code.
- `cargo-call-stack`
  - Static analysis to determine maximum stack usage.
- RaspberryPi OS 튜토리얼에서는 MMU 와 캐시가 사용 설정되기 전에 Rust 코드를 실행합니다. 이렇게 하면 메모리 (예: 스택) 를 읽고 쓸 수 있습니다. 하지만 다음과 같은 예외가 적용됩니다.
  - MMU 와 캐시가 없으면 정렬되지 않은 액세스에 오류가 발생합니다. 컴파일러가 정렬되지 않은 액세스를 생성하지 않도록`+strict-align` 을 설정하는 `aarch64-unknown-none` 으로빌드되므로 문제가 없지만, 일반적으로 반드시 그런 것은 아닙니다.

- VM 에서 실행한다면 캐시 일관성 문제가 발생할 수 있습니다. 문제는 VM 은캐시가 사용 중지된 상태로 메모리에 직접 액세스하는 반면 호스트에는동일한 메모리에 대해 캐시할 수 있는 별칭이 있다는 점입니다. 호스트가메모리에 명시적으로 액세스하지 않더라도 추측 액세스는 캐시 채우기로이어질 수 있으며 둘 중 하나의 변경사항이 손실됩니다. 이번에도하이퍼바이저 없이 하드웨어에서 직접 실행되는 이 특정 경우에는 문제가없지만, 일반적으로 좋은 패턴은 아닙니다.

제 54 장

유용한크레이트

bare-metal 프로그래밍의 몇 가지 일반적인 문제를 해결하는 크레이트를살펴봅니다.

54.1 zerocopy

Fuchsia 팀이 만든 zerocopy 크레이트는 바이트 시퀀스를 다른 타입으로 안전하게 변환하기 위한 트레잇및 매크로를 제공합니다.

```
use zerocopy::AsBytes;

#[repr(u32)]
#[derive(AsBytes, Debug, Default)]
enum RequestType {
    #[default]
    In = 0,
    Out = 1,
    Flush = 4,
}

#[repr(C)]
#[derive(AsBytes, Debug, Default)]
struct VirtioBlockRequest {
    request_type: RequestType,
    reserved: u32,
    sector: u64,
}

fn main() {
    let request = VirtioBlockRequest {
        request_type: RequestType::Flush,
        sector: 42,
        ..Default::default()
    };

    assert_eq!(
```

```
        request.as_bytes(),
        &[4, 0, 0, 0, 0, 0, 0, 0, 42, 0, 0, 0, 0, 0, 0, 0]
    );
}
```

이 크레이트는 휘발성 (volatile) 읽기 및 쓰기를 사용하지 않으므로 MMIO 에적합하지 않지만, 하드웨어와 공유되거나 (예: DMA 에서) 외장 인터페이스를통해 전송되는 구조체를 다루는 데에는 유용할 수 있습니다.

- 어떤 타입이 가능한 모든 바이트 패턴들에 대해 올바른 값을 가질 때에만 , 그 타입이 `FromBytes` 를 구현할 수 있습니다. 그렇게 해서 신뢰할수 없는 바이트 시퀀스를 안전하게 해당 타입으로 변환할 수 있습니다.
- 위 코드에서 정의한 타입에 대해 `FromBytes` 를 구현하려고 하면에러가 발생합니다. `RequestType` 은 가능한 모든 u32 값을식별자로 받아들이지 않기 때문입니다. 즉 모든 바이트 패턴이 유효한 `RequestType` 값은 아닙니다.
- `zerocopy::byteorder` 에는 바이트 오더에 따른 서로 다른 표현방식을 지원하는 숫자 타입을 제공합니다.
- `src/bare-metal/useful-crates/zerocopy-example/`에서 `cargo run` 을 사용하여 예시를 실행합니다 (종속성 문제로 인해플레이그라운드에서는 실행되지 않습니다).

54.2 aarch64-paging

aarch64-paging 크레이트를 사용하면 AArch64 가상 메모리 시스템 아키텍처에 따라 페이지테이블을 만들 수 있습니다.

```
use aarch64_paging::{
    idmap::IdMap,
    paging::{Attributes, MemoryRegion},
};

const ASID: usize = 1;
const ROOT_LEVEL: usize = 1;

// ID 매핑을 사용하여 새 페이지 테이블을 만듭니다.
let mut idmap = IdMap::new(ASID, ROOT_LEVEL);
// 2MiB 메모리 영역을 읽기 전용으로 매핑합니다.
idmap.map_range(
    &MemoryRegion::new(0x80200000, 0x80400000),
    Attributes::NORMAL | Attributes::NON_GLOBAL | Attributes::READ_ONLY,
).unwrap();
// `TTBR0_EL1`을 설정하여 페이지 테이블을 활성화합니다.
idmap.activate();
```

- 현재는 EL1 만 지원하지만 다른 익셉션 레벨 (Exception Level: EL) 도 어렵지않게 추가할 수 있습니다.
- Android 에서보호된 VM 펌웨어에 사용됩니다.
- 이 예시를 간단하게 실행하는 방법은 없습니다. 실제 하드웨어 또는 QEMU 에서 실행해야 하기 때문입니다.

54.3 buddy_system_allocator

buddy_system_allocator 는버디 시스템 할당자를 구현하는 서드 파티 크레이트입니다. 이 크레이트의 LockedHeap 은 GlobalAlloc 를구현합니다. 따라서 여러분은 버디 시스템 할당자를'alloc' 크레이트를 통해서 사용할 수있습니다 (이전에확인함). 또는 다른 주소 공간을 할당하는 데 사용할 수 있습니다. 예를 들어 PCI BAR 에 MMIO 공간을 할당할 수 있습니다.

```
use buddy_system_allocator::FrameAllocator;
use core::alloc::Layout;

fn main() {
    let mut allocator = FrameAllocator::<32>::new();
    allocator.add_frame(0x200_0000, 0x400_0000);

    let layout = Layout::from_size_align(0x100, 0x100).unwrap();
    let bar = allocator
        .alloc_aligned(layout)
        .expect("Failed to allocate 0x100 byte MMIO region");
    println!("Allocated 0x100 byte MMIO region at {:#x}", bar);
}
```

- PCI BAR 는 BAR 영역의 크기에 맞추어 정렬됩니다.
- src/bare-metal/useful-crates/allocator-example/에서 cargo run 을 사용하여 예시를 실행합니다 (종속성 문제로 인해플레이그라운드에서는 실행되지 않습니다).

54.4 tinyvec

힙에 메모리 할당하지 않고 크기 조절이 가능한 컨테이너 (예: Vec 같은) 가 필요할 때가 있습니다. tinyvec 을 사용하면됩니다. tinyvec 에서 벡터는 배열 또는 슬라이스로부터 생성이되며, 이들은 정적으로 할당되었거나 스택에 할당되어 있을 수있습니다.tinyvec 은 현재 벡터 안에 얼마나 많은 엘리먼트들이존재하는 지를 추적하고 있으며, 할당된 양보다 더 많이 사용하려고 하면패닉을 발생시킵니다.

```
use tinyvec::{array_vec, ArrayVec};

fn main() {
    let mut numbers: ArrayVec<[u32; 5]> = array_vec!(42, 66);
    println!("{numbers:?}");
    numbers.push(7);
    println!("{numbers:?}");
    numbers.remove(1);
    println!("{numbers:?}");
}
```

- tinyvec 를 사용하려면 엘리먼트의 타입이 Default 를통해 초기화 될 수 있어야 합니다.
- Rust 플레이그라운드에는 tinyvec 가 포함되어 있으므로 이 예시는인라인으로 실행됩니다.

54.5 spin

std::sync::Mutex 및 std::sync 의 기타 동기화프리미티브는 core 또는 alloc 에서 사용할 수 없습니다. 그러면 어떻게 동기화 또는 interior mutability 와 같은 기능이 필요할 경우어떻게 해야 할까요?

spin 크레이트는 이러한동기화 프리미티브들을 스핀록으로 구현하고 있습니다.

```
use spin::mutex::SpinMutex;

static counter: SpinMutex<u32> = SpinMutex::new(0);

fn main() {
    println!("count: {}", counter.lock());
    *counter.lock() += 2;
    println!("count: {}", counter.lock());
}
```

- 인터럽트 핸들러에서 락을 걸 경우, 교착 상태가 발생하지 않도록주의하세요.
- spin also has a ticket lock mutex implementation; equivalents of RwLock, Barrier and Once from std::sync; and Lazy for lazy initialisation.
- once_cell 크레이트에는 지연된 초기화를 위한 몇 가지 유용한 타입이 있는데 spin::once::Once 와는 약간 다른 접근 방식을 사용합니다.
- Rust 플레이그라운드에는 spin 이 포함되어 있으므로 이 예시는인라인으로 실행됩니다.

제 55 장

안드로이드

AOSP 에서 bare-metal Rust 바이너리를 빌드하려면 `rust_ffi_static` 을 사용하여 Rust 코드를 빌드하고, 링커스크립트가 포함된 `cc_binary` 를 사용하여 ELF 바이너리를생성하고, `raw_binary` 를 사용해 ELF 를 곧바로 수행될 수 있는원시 (raw) 바이너리로 변환합니다.

```
rust_ffi_static {
    name: "libvmbase_example",
    defaults: ["vmbase_ffi_defaults"],
    crate_name: "vmbase_example",
    srcs: ["src/main.rs"],
    rustlibs: [
        "libvmbase",
    ],
}

cc_binary {
    name: "vmbase_example",
    defaults: ["vmbase_elf_defaults"],
    srcs: [
        "idmap.S",
    ],
    static_libs: [
        "libvmbase_example",
    ],
    linker_scripts: [
        "image.ld",
        ":vmbase_sections",
    ],
}

raw_binary {
    name: "vmbase_example_bin",
    stem: "vmbase_example.bin",
    src: ":vmbase_example",
    enabled: false,
    target: {
```

```
        android_arm64: {
            enabled: true,
        },
    },
}
```

55.1 vmbase

vmbase 라이브러리는, aarch64 의 crosvm 에서 실행되는 VM 을 타겟하여, 진입점, UART 콘솔 로깅, 링커 스크립트, 빌드 룰 등에 대한 기본 구현들을 제공합니다.

```rust
#![no_main]
#![no_std]

use vmbase::{main, println};

main!(main);

pub fn main(arg0: u64, arg1: u64, arg2: u64, arg3: u64) {
    println!("Hello world");
}
```

- `main!` 매크로는 `vmbase` 진입점에서 호출될 main 함수를표시합니다.
- `vmbase` 가 제공하는 진입점은 콘솔을 초기화 하며, main 함수가리턴하면 PSCI_SYSTEM_OFF 메시지를 PSCI 를 통해 보내어서 VM 을종료시킵니다.

제 56 장

연습문제

PL031 실시간 시계 기기용 드라이버를 작성합니다.

After looking at the exercises, you can look at the solutions provided.

56.1 RTC driver

The QEMU aarch64 virt machine has a PL031 real-time clock at 0x9010000. For this exercise, you should write a driver for it.

1. 직렬 콘솔에 현재 시간을 출력하는 데 사용합니다. 날짜/시간 형식지정에는 `chrono` 크레이트를 사용할 수 있습니다.
2. 일치 레지스터와 원시 인터럽트 상태를 사용하여 특정 시간 (예: 향후 3 초) 까지 바쁜 대기합니다. 루프 내에서 `core::hint::spin_loop` 를호출합니다.
3. *:* RTC 일치로 생성된 인터럽트를 사용설정하고 처리합니다. `arm-gic` 크레이트에 제공된 드라이버를 사용하여 Arm 일반 인터럽트 컨트롤러를구성할 수 있습니다.
   - GIC 에 `IntId::spi(2)` 로 연결된 RTC 인터럽트를 사용합니다.
   - 인터럽트를 사용 설정한 후에는 `arm_gic::wfi()` 를 통해 코어를절전 모드로 전환할 수 있습니다. 이 경우 인터럽트를 수신할 때까지코어가 절전 모드로 전환됩니다.

연습템플릿을 다운로드하고 `rtc` 디렉터리에서 다음 파일을 찾습니다.

src/main.rs:

```
#![no_main]
#![no_std]

mod exceptions;
mod logger;
mod pl011;

use crate::pl011::Uart;
use arm_gic::gicv3::GicV3;
use core::panic::PanicInfo;
use log::{error, info, trace, LevelFilter};
use smccc::psci::system_off;
use smccc::Hvc;
```

```
/// Base addresses of the GICv3.
const GICD_BASE_ADDRESS: *mut u64 = 0x800_0000 as _;
const GICR_BASE_ADDRESS: *mut u64 = 0x80A_0000 as _;

/// Base address of the primary PL011 UART.
const PL011_BASE_ADDRESS: *mut u32 = 0x900_0000 as _;

#[no_mangle]
extern "C" fn main(x0: u64, x1: u64, x2: u64, x3: u64) {
    // Safe because `PL011_BASE_ADDRESS` is the base address of a PL011 device,
    // and nothing else accesses that address range.
    let uart = unsafe { Uart::new(PL011_BASE_ADDRESS) };
    logger::init(uart, LevelFilter::Trace).unwrap();

    info!("main({:#x}, {:#x}, {:#x}, {:#x})", x0, x1, x2, x3);

    // Safe because `GICD_BASE_ADDRESS` and `GICR_BASE_ADDRESS` are the base
    // addresses of a GICv3 distributor and redistributor respectively, and
    // nothing else accesses those address ranges.
    let mut gic = unsafe { GicV3::new(GICD_BASE_ADDRESS, GICR_BASE_ADDRESS) };
    gic.setup();

    // TODO: Create instance of RTC driver and print current time.

    // TODO: Wait for 3 seconds.

    system_off::<Hvc>().unwrap();
}

#[panic_handler]
fn panic(info: &PanicInfo) -> ! {
    error!("{info}");
    system_off::<Hvc>().unwrap();
    loop {}
}
```

_src/exceptions.rs_는 연습의 세 번째 부분에서만 변경하면 됩니다.

```
// Copyright 2023 Google LLC
//
// Licensed under the Apache License, Version 2.0 (the "License");
// you may not use this file except in compliance with the License.
// You may obtain a copy of the License at
//
//      http://www.apache.org/licenses/LICENSE-2.0
//
// Unless required by applicable law or agreed to in writing, software
// distributed under the License is distributed on an "AS IS" BASIS,
// WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
// See the License for the specific language governing permissions and
// limitations under the License.
```

```rust
use arm_gic::gicv3::GicV3;
use log::{error, info, trace};
use smccc::psci::system_off;
use smccc::Hvc;

#[no_mangle]
extern "C" fn sync_exception_current(_elr: u64, _spsr: u64) {
    error!("sync_exception_current");
    system_off::<Hvc>().unwrap();
}

#[no_mangle]
extern "C" fn irq_current(_elr: u64, _spsr: u64) {
    trace!("irq_current");
    let intid =
        GicV3::get_and_acknowledge_interrupt().expect("No pending interrupt");
    info!("IRQ {intid:?}");
}

#[no_mangle]
extern "C" fn fiq_current(_elr: u64, _spsr: u64) {
    error!("fiq_current");
    system_off::<Hvc>().unwrap();
}

#[no_mangle]
extern "C" fn serr_current(_elr: u64, _spsr: u64) {
    error!("serr_current");
    system_off::<Hvc>().unwrap();
}

#[no_mangle]
extern "C" fn sync_lower(_elr: u64, _spsr: u64) {
    error!("sync_lower");
    system_off::<Hvc>().unwrap();
}

#[no_mangle]
extern "C" fn irq_lower(_elr: u64, _spsr: u64) {
    error!("irq_lower");
    system_off::<Hvc>().unwrap();
}

#[no_mangle]
extern "C" fn fiq_lower(_elr: u64, _spsr: u64) {
    error!("fiq_lower");
    system_off::<Hvc>().unwrap();
}

#[no_mangle]
extern "C" fn serr_lower(_elr: u64, _spsr: u64) {
```

```
    error!("serr_lower");
    system_off::<Hvc>().unwrap();
}
```

src/logger.rs(변경할 필요가 없음):

```
// Copyright 2023 Google LLC
//
// Licensed under the Apache License, Version 2.0 (the "License");
// you may not use this file except in compliance with the License.
// You may obtain a copy of the License at
//
//      http://www.apache.org/licenses/LICENSE-2.0
//
// Unless required by applicable law or agreed to in writing, software
// distributed under the License is distributed on an "AS IS" BASIS,
// WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
// See the License for the specific language governing permissions and
// limitations under the License.

// ANCHOR: main
use crate::pl011::Uart;
use core::fmt::Write;
use log::{LevelFilter, Log, Metadata, Record, SetLoggerError};
use spin::mutex::SpinMutex;

static LOGGER: Logger = Logger { uart: SpinMutex::new(None) };

struct Logger {
    uart: SpinMutex<Option<Uart>>,
}

impl Log for Logger {
    fn enabled(&self, _metadata: &Metadata) -> bool {
        true
    }

    fn log(&self, record: &Record) {
        writeln!(
            self.uart.lock().as_mut().unwrap(),
            "[{}] {}",
            record.level(),
            record.args()
        )
        .unwrap();
    }

    fn flush(&self) {}
}

/// Initialises UART logger.
pub fn init(uart: Uart, max_level: LevelFilter) -> Result<(), SetLoggerError> {
```

```
    LOGGER.uart.lock().replace(uart);

    log::set_logger(&LOGGER)?;
    log::set_max_level(max_level);
    Ok(())
}
```

src/pl011.rs (변경할 필요가 없음):

```
// Copyright 2023 Google LLC
//
// Licensed under the Apache License, Version 2.0 (the "License");
// you may not use this file except in compliance with the License.
// You may obtain a copy of the License at
//
//      http://www.apache.org/licenses/LICENSE-2.0
//
// Unless required by applicable law or agreed to in writing, software
// distributed under the License is distributed on an "AS IS" BASIS,
// WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
// See the License for the specific language governing permissions and
// limitations under the License.

#![allow(unused)]

use core::fmt::{self, Write};
use core::ptr::{addr_of, addr_of_mut};

// ANCHOR: Flags
use bitflags::bitflags;

bitflags! {
    /// Flags from the UART flag register.
    #[repr(transparent)]
    #[derive(Copy, Clone, Debug, Eq, PartialEq)]
    struct Flags: u16 {
        /// Clear to send.
        const CTS = 1 << 0;
        /// Data set ready.
        const DSR = 1 << 1;
        /// Data carrier detect.
        const DCD = 1 << 2;
        /// UART busy transmitting data.
        const BUSY = 1 << 3;
        /// Receive FIFO is empty.
        const RXFE = 1 << 4;
        /// Transmit FIFO is full.
        const TXFF = 1 << 5;
        /// Receive FIFO is full.
        const RXFF = 1 << 6;
        /// Transmit FIFO is empty.
        const TXFE = 1 << 7;
```

```
        /// Ring indicator.
        const RI = 1 << 8;
    }
}
// ANCHOR_END: Flags

bitflags! {
    /// Flags from the UART Receive Status Register / Error Clear Register.
    #[repr(transparent)]
    #[derive(Copy, Clone, Debug, Eq, PartialEq)]
    struct ReceiveStatus: u16 {
        /// Framing error.
        const FE = 1 << 0;
        /// Parity error.
        const PE = 1 << 1;
        /// Break error.
        const BE = 1 << 2;
        /// Overrun error.
        const OE = 1 << 3;
    }
}

// ANCHOR: Registers
#[repr(C, align(4))]
struct Registers {
    dr: u16,
    _reserved0: [u8; 2],
    rsr: ReceiveStatus,
    _reserved1: [u8; 19],
    fr: Flags,
    _reserved2: [u8; 6],
    ilpr: u8,
    _reserved3: [u8; 3],
    ibrd: u16,
    _reserved4: [u8; 2],
    fbrd: u8,
    _reserved5: [u8; 3],
    lcr_h: u8,
    _reserved6: [u8; 3],
    cr: u16,
    _reserved7: [u8; 3],
    ifls: u8,
    _reserved8: [u8; 3],
    imsc: u16,
    _reserved9: [u8; 2],
    ris: u16,
    _reserved10: [u8; 2],
    mis: u16,
    _reserved11: [u8; 2],
    icr: u16,
    _reserved12: [u8; 2],
```

```
    dmacr: u8,
    _reserved13: [u8; 3],
}
// ANCHOR_END: Registers

// ANCHOR: Uart
/// Driver for a PL011 UART.
#[derive(Debug)]
pub struct Uart {
    registers: *mut Registers,
}

impl Uart {
    /// Constructs a new instance of the UART driver for a PL011 device at the
    /// given base address.
    ///
    /// # Safety
    ///
    /// The given base address must point to the MMIO control registers of a
    /// PL011 device, which must be mapped into the address space of the process
    /// as device memory and not have any other aliases.
    pub unsafe fn new(base_address: *mut u32) -> Self {
        Self { registers: base_address as *mut Registers }
    }

    /// Writes a single byte to the UART.
    pub fn write_byte(&self, byte: u8) {
        // Wait until there is room in the TX buffer.
        while self.read_flag_register().contains(Flags::TXFF) {}

        // Safe because we know that self.registers points to the control
        // registers of a PL011 device which is appropriately mapped.
        unsafe {
            // Write to the TX buffer.
            addr_of_mut!((*self.registers).dr).write_volatile(byte.into());
        }

        // Wait until the UART is no longer busy.
        while self.read_flag_register().contains(Flags::BUSY) {}
    }

    /// Reads and returns a pending byte, or `None` if nothing has been
    /// received.
    pub fn read_byte(&self) -> Option<u8> {
        if self.read_flag_register().contains(Flags::RXFE) {
            None
        } else {
            let data = unsafe { addr_of!((*self.registers).dr).read_volatile() };
            // TODO: Check for error conditions in bits 8-11.
            Some(data as u8)
        }
```

```
    }

    fn read_flag_register(&self) -> Flags {
        // Safe because we know that self.registers points to the control
        // registers of a PL011 device which is appropriately mapped.
        unsafe { addr_of!((*self.registers).fr).read_volatile() }
    }
}
// ANCHOR_END: Uart

impl Write for Uart {
    fn write_str(&mut self, s: &str) -> fmt::Result {
        for c in s.as_bytes() {
            self.write_byte(*c);
        }
        Ok(())
    }
}

// Safe because it just contains a pointer to device memory, which can be
// accessed from any context.
unsafe impl Send for Uart {}
```

Cargo.toml (변경할 필요가 없음):

```
[workspace]

[package]
name = "rtc"
version = "0.1.0"
edition = "2021"
publish = false

[dependencies]
arm-gic = "0.1.0"
bitflags = "2.4.2"
chrono = { version = "0.4.35", default-features = false }
log = "0.4.21"
smccc = "0.1.1"
spin = "0.9.8"

[build-dependencies]
cc = "1.0.90"
```

build.rs(변경할 필요가 없음):

```
// Copyright 2023 Google LLC
//
// Licensed under the Apache License, Version 2.0 (the "License");
// you may not use this file except in compliance with the License.
// You may obtain a copy of the License at
//
//      http://www.apache.org/licenses/LICENSE-2.0
```

```
//
// Unless required by applicable law or agreed to in writing, software
// distributed under the License is distributed on an "AS IS" BASIS,
// WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
// See the License for the specific language governing permissions and
// limitations under the License.

use cc::Build;
use std::env;

fn main() {
    #[cfg(target_os = "linux")]
    env::set_var("CROSS_COMPILE", "aarch64-linux-gnu");
    #[cfg(not(target_os = "linux"))]
    env::set_var("CROSS_COMPILE", "aarch64-none-elf");

    Build::new()
        .file("entry.S")
        .file("exceptions.S")
        .file("idmap.S")
        .compile("empty")
}
```

entry.S(변경할 필요가 없음):

```
/*
 * Copyright 2023 Google LLC
 *
 * Licensed under the Apache License, Version 2.0 (the "License");
 * you may not use this file except in compliance with the License.
 * You may obtain a copy of the License at
 *
 *     https://www.apache.org/licenses/LICENSE-2.0
 *
 * Unless required by applicable law or agreed to in writing, software
 * distributed under the License is distributed on an "AS IS" BASIS,
 * WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
 * See the License for the specific language governing permissions and
 * limitations under the License.
 */

.macro adr_l, reg:req, sym:req
    adrp \reg, \sym
    add \reg, \reg, :lo12:\sym
.endm

.macro mov_i, reg:req, imm:req
    movz \reg, :abs_g3:\imm
    movk \reg, :abs_g2_nc:\imm
    movk \reg, :abs_g1_nc:\imm
    movk \reg, :abs_g0_nc:\imm
.endm
```

```
.set .L_MAIR_DEV_nGnRE, 0x04
.set .L_MAIR_MEM_WBWA,  0xff
.set .Lmairval, .L_MAIR_DEV_nGnRE | (.L_MAIR_MEM_WBWA << 8)

/* 4 KiB granule size for TTBR0_EL1. */
.set .L_TCR_TG0_4KB, 0x0 << 14
/* 4 KiB granule size for TTBR1_EL1. */
.set .L_TCR_TG1_4KB, 0x2 << 30
/* Disable translation table walk for TTBR1_EL1, generating a translation fault instead
.set .L_TCR_EPD1, 0x1 << 23
/* Translation table walks for TTBR0_EL1 are inner sharable. */
.set .L_TCR_SH_INNER, 0x3 << 12
/*
 * Translation table walks for TTBR0_EL1 are outer write-back read-allocate write-alloca
 * cacheable.
 */
.set .L_TCR_RGN_OWB, 0x1 << 10
/*
 * Translation table walks for TTBR0_EL1 are inner write-back read-allocate write-alloca
 * cacheable.
 */
.set .L_TCR_RGN_IWB, 0x1 << 8
/* Size offset for TTBR0_EL1 is 2**39 bytes (512 GiB). */
.set .L_TCR_T0SZ_512, 64 - 39
.set .Ltcrval, .L_TCR_TG0_4KB | .L_TCR_TG1_4KB | .L_TCR_EPD1 | .L_TCR_RGN_OWB
.set .Ltcrval, .Ltcrval | .L_TCR_RGN_IWB | .L_TCR_SH_INNER | .L_TCR_T0SZ_512

/* Stage 1 instruction access cacheability is unaffected. */
.set .L_SCTLR_ELx_I, 0x1 << 12
/* SP alignment fault if SP is not aligned to a 16 byte boundary. */
.set .L_SCTLR_ELx_SA, 0x1 << 3
/* Stage 1 data access cacheability is unaffected. */
.set .L_SCTLR_ELx_C, 0x1 << 2
/* EL0 and EL1 stage 1 MMU enabled. */
.set .L_SCTLR_ELx_M, 0x1 << 0
/* Privileged Access Never is unchanged on taking an exception to EL1. */
.set .L_SCTLR_EL1_SPAN, 0x1 << 23
/* SETEND instruction disabled at EL0 in aarch32 mode. */
.set .L_SCTLR_EL1_SED, 0x1 << 8
/* Various IT instructions are disabled at EL0 in aarch32 mode. */
.set .L_SCTLR_EL1_ITD, 0x1 << 7
.set .L_SCTLR_EL1_RES1, (0x1 << 11) | (0x1 << 20) | (0x1 << 22) | (0x1 << 28) | (0x1 <<
.set .Lsctlrval, .L_SCTLR_ELx_M | .L_SCTLR_ELx_C | .L_SCTLR_ELx_SA | .L_SCTLR_EL1_ITD |
.set .Lsctlrval, .Lsctlrval | .L_SCTLR_ELx_I | .L_SCTLR_EL1_SPAN | .L_SCTLR_EL1_RES1

/**
 * This is a generic entry point for an image. It carries out the operations required to
 * loaded image to be run. Specifically, it zeroes the bss section using registers x25 a
 * prepares the stack, enables floating point, and sets up the exception vector. It pres
 * for the Rust entry point, as these may contain boot parameters.
```

```
 */
.section .init.entry, "ax"
.global entry
entry:
    /* Load and apply the memory management configuration, ready to enable MMU and cache
    adrp x30, idmap
    msr ttbr0_el1, x30

    mov_i x30, .Lmairval
    msr mair_el1, x30

    mov_i x30, .Ltcrval
    /* Copy the supported PA range into TCR_EL1.IPS. */
    mrs x29, id_aa64mmfr0_el1
    bfi x30, x29, #32, #4

    msr tcr_el1, x30

    mov_i x30, .Lsctlrval

    /*
     * Ensure everything before this point has completed, then invalidate any potentiall
     * local TLB entries before they start being used.
     */
    isb
    tlbi vmalle1
    ic iallu
    dsb nsh
    isb

    /*
     * Configure sctlr_el1 to enable MMU and cache and don't proceed until this has comp
     */
    msr sctlr_el1, x30
    isb

    /* Disable trapping floating point access in EL1. */
    mrs x30, cpacr_el1
    orr x30, x30, #(0x3 << 20)
    msr cpacr_el1, x30
    isb

    /* Zero out the bss section. */
    adr_l x29, bss_begin
    adr_l x30, bss_end
0:  cmp x29, x30
    b.hs 1f
    stp xzr, xzr, [x29], #16
    b 0b

1:  /* Prepare the stack. */
```

```
    adr_l x30, boot_stack_end
    mov sp, x30

    /* Set up exception vector. */
    adr x30, vector_table_el1
    msr vbar_el1, x30

    /* Call into Rust code. */
    bl main

    /* Loop forever waiting for interrupts. */
2:  wfi
    b 2b
```

exceptions.S(변경할 필요가 없음):

```
/*
 * Copyright 2023 Google LLC
 *
 * Licensed under the Apache License, Version 2.0 (the "License");
 * you may not use this file except in compliance with the License.
 * You may obtain a copy of the License at
 *
 *     https://www.apache.org/licenses/LICENSE-2.0
 *
 * Unless required by applicable law or agreed to in writing, software
 * distributed under the License is distributed on an "AS IS" BASIS,
 * WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
 * See the License for the specific language governing permissions and
 * limitations under the License.
 */

/**
 * Saves the volatile registers onto the stack. This currently takes 14
 * instructions, so it can be used in exception handlers with 18 instructions
 * left.
 *
 * On return, x0 and x1 are initialised to elr_el2 and spsr_el2 respectively,
 * which can be used as the first and second arguments of a subsequent call.
 */
.macro save_volatile_to_stack
    /* Reserve stack space and save registers x0-x18, x29 & x30. */
    stp x0, x1, [sp, #-(8 * 24)]!
    stp x2, x3, [sp, #8 * 2]
    stp x4, x5, [sp, #8 * 4]
    stp x6, x7, [sp, #8 * 6]
    stp x8, x9, [sp, #8 * 8]
    stp x10, x11, [sp, #8 * 10]
    stp x12, x13, [sp, #8 * 12]
    stp x14, x15, [sp, #8 * 14]
    stp x16, x17, [sp, #8 * 16]
    str x18, [sp, #8 * 18]
```

```
    stp x29, x30, [sp, #8 * 20]

    /*
     * Save elr_el1 & spsr_el1. This such that we can take nested exception
     * and still be able to unwind.
     */
    mrs x0, elr_el1
    mrs x1, spsr_el1
    stp x0, x1, [sp, #8 * 22]
.endm

/**
 * Restores the volatile registers from the stack. This currently takes 14
 * instructions, so it can be used in exception handlers while still leaving 18
 * instructions left; if paired with save_volatile_to_stack, there are 4
 * instructions to spare.
 */
.macro restore_volatile_from_stack
    /* Restore registers x2-x18, x29 & x30. */
    ldp x2, x3, [sp, #8 * 2]
    ldp x4, x5, [sp, #8 * 4]
    ldp x6, x7, [sp, #8 * 6]
    ldp x8, x9, [sp, #8 * 8]
    ldp x10, x11, [sp, #8 * 10]
    ldp x12, x13, [sp, #8 * 12]
    ldp x14, x15, [sp, #8 * 14]
    ldp x16, x17, [sp, #8 * 16]
    ldr x18, [sp, #8 * 18]
    ldp x29, x30, [sp, #8 * 20]

    /* Restore registers elr_el1 & spsr_el1, using x0 & x1 as scratch. */
    ldp x0, x1, [sp, #8 * 22]
    msr elr_el1, x0
    msr spsr_el1, x1

    /* Restore x0 & x1, and release stack space. */
    ldp x0, x1, [sp], #8 * 24
.endm

/**
 * This is a generic handler for exceptions taken at the current EL while using
 * SP0. It behaves similarly to the SPx case by first switching to SPx, doing
 * the work, then switching back to SP0 before returning.
 *
 * Switching to SPx and calling the Rust handler takes 16 instructions. To
 * restore and return we need an additional 16 instructions, so we can implement
 * the whole handler within the allotted 32 instructions.
 */
.macro current_exception_sp0 handler:req
    msr spsel, #1
    save_volatile_to_stack
```

```
    bl \handler
    restore_volatile_from_stack
    msr spsel, #0
    eret
.endm

/**
 * This is a generic handler for exceptions taken at the current EL while using
 * SPx. It saves volatile registers, calls the Rust handler, restores volatile
 * registers, then returns.
 *
 * This also works for exceptions taken from EL0, if we don't care about
 * non-volatile registers.
 *
 * Saving state and jumping to the Rust handler takes 15 instructions, and
 * restoring and returning also takes 15 instructions, so we can fit the whole
 * handler in 30 instructions, under the limit of 32.
 */
.macro current_exception_spx handler:req
    save_volatile_to_stack
    bl \handler
    restore_volatile_from_stack
    eret
.endm

.section .text.vector_table_el1, "ax"
.global vector_table_el1
.balign 0x800
vector_table_el1:
sync_cur_sp0:
    current_exception_sp0 sync_exception_current

.balign 0x80
irq_cur_sp0:
    current_exception_sp0 irq_current

.balign 0x80
fiq_cur_sp0:
    current_exception_sp0 fiq_current

.balign 0x80
serr_cur_sp0:
    current_exception_sp0 serr_current

.balign 0x80
sync_cur_spx:
    current_exception_spx sync_exception_current

.balign 0x80
irq_cur_spx:
    current_exception_spx irq_current
```

```
.balign 0x80
fiq_cur_spx:
    current_exception_spx fiq_current

.balign 0x80
serr_cur_spx:
    current_exception_spx serr_current

.balign 0x80
sync_lower_64:
    current_exception_spx sync_lower

.balign 0x80
irq_lower_64:
    current_exception_spx irq_lower

.balign 0x80
fiq_lower_64:
    current_exception_spx fiq_lower

.balign 0x80
serr_lower_64:
    current_exception_spx serr_lower

.balign 0x80
sync_lower_32:
    current_exception_spx sync_lower

.balign 0x80
irq_lower_32:
    current_exception_spx irq_lower

.balign 0x80
fiq_lower_32:
    current_exception_spx fiq_lower

.balign 0x80
serr_lower_32:
    current_exception_spx serr_lower
```

idmap.S(변경할 필요가 없음):

```
/*
 * Copyright 2023 Google LLC
 *
 * Licensed under the Apache License, Version 2.0 (the "License");
 * you may not use this file except in compliance with the License.
 * You may obtain a copy of the License at
 *
 *     https://www.apache.org/licenses/LICENSE-2.0
 *
```

```
 * Unless required by applicable law or agreed to in writing, software
 * distributed under the License is distributed on an "AS IS" BASIS,
 * WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
 * See the License for the specific language governing permissions and
 * limitations under the License.
 */

.set .L_TT_TYPE_BLOCK, 0x1
.set .L_TT_TYPE_PAGE,  0x3
.set .L_TT_TYPE_TABLE, 0x3

/* Access flag. */
.set .L_TT_AF, 0x1 << 10
/* Not global. */
.set .L_TT_NG, 0x1 << 11
.set .L_TT_XN, 0x3 << 53

.set .L_TT_MT_DEV, 0x0 << 2         // MAIR #0 (DEV_nGnRE)
.set .L_TT_MT_MEM, (0x1 << 2) | (0x3 << 8)  // MAIR #1 (MEM_WBWA), inner shareable

.set .L_BLOCK_DEV, .L_TT_TYPE_BLOCK | .L_TT_MT_DEV | .L_TT_AF | .L_TT_XN
.set .L_BLOCK_MEM, .L_TT_TYPE_BLOCK | .L_TT_MT_MEM | .L_TT_AF | .L_TT_NG

.section ".rodata.idmap", "a", %progbits
.global idmap
.align 12
idmap:
    /* level 1 */
    .quad       .L_BLOCK_DEV | 0x0          // 1 GiB of device mappings
    .quad       .L_BLOCK_MEM | 0x40000000   // 1 GiB of DRAM
    .fill       254, 8, 0x0         // 254 GiB of unmapped VA space
    .quad       .L_BLOCK_DEV | 0x4000000000 // 1 GiB of device mappings
    .fill       255, 8, 0x0         // 255 GiB of remaining VA space
```

image.ld(변경할 필요가 없음):

```
/*
 * Copyright 2023 Google LLC
 *
 * Licensed under the Apache License, Version 2.0 (the "License");
 * you may not use this file except in compliance with the License.
 * You may obtain a copy of the License at
 *
 *     https://www.apache.org/licenses/LICENSE-2.0
 *
 * Unless required by applicable law or agreed to in writing, software
 * distributed under the License is distributed on an "AS IS" BASIS,
 * WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
 * See the License for the specific language governing permissions and
 * limitations under the License.
 */
```

```
/*
 * Code will start running at this symbol which is placed at the start of the
 * image.
 */
ENTRY(entry)

MEMORY
{
    image : ORIGIN = 0x40080000, LENGTH = 2M
}

SECTIONS
{
    /*
     * Collect together the code.
     */
    .init : ALIGN(4096) {
        text_begin = .;
        *(.init.entry)
        *(.init.*)
    } >image
    .text : {
        *(.text.*)
    } >image
    text_end = .;

    /*
     * Collect together read-only data.
     */
    .rodata : ALIGN(4096) {
        rodata_begin = .;
        *(.rodata.*)
    } >image
    .got : {
        *(.got)
    } >image
    rodata_end = .;

    /*
     * Collect together the read-write data including .bss at the end which
     * will be zero'd by the entry code.
     */
    .data : ALIGN(4096) {
        data_begin = .;
        *(.data.*)
        /*
         * The entry point code assumes that .data is a multiple of 32
         * bytes long.
         */
        . = ALIGN(32);
        data_end = .;
```

```
    } >image

    /* Everything beyond this point will not be included in the binary. */
    bin_end = .;

    /* The entry point code assumes that .bss is 16-byte aligned. */
    .bss : ALIGN(16)  {
        bss_begin = .;
        *(.bss.*)
        *(COMMON)
        . = ALIGN(16);
        bss_end = .;
    } >image

    .stack (NOLOAD) : ALIGN(4096) {
        boot_stack_begin = .;
        . += 40 * 4096;
        . = ALIGN(4096);
        boot_stack_end = .;
    } >image

    . = ALIGN(4K);
    PROVIDE(dma_region = .);

    /*
     * Remove unused sections from the image.
     */
    /DISCARD/ : {
        /* The image loads itself so doesn't need these sections. */
        *(.gnu.hash)
        *(.hash)
        *(.interp)
        *(.eh_frame_hdr)
        *(.eh_frame)
        *(.note.gnu.build-id)
    }
}
```

Makefile(변경할 필요가 없음):

```
# Copyright 2023 Google LLC
#
# Licensed under the Apache License, Version 2.0 (the "License");
# you may not use this file except in compliance with the License.
# You may obtain a copy of the License at
#
#      http://www.apache.org/licenses/LICENSE-2.0
#
# Unless required by applicable law or agreed to in writing, software
# distributed under the License is distributed on an "AS IS" BASIS,
# WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
# See the License for the specific language governing permissions and
```

```
# limitations under the License.

UNAME := $(shell uname -s)
ifeq ($(UNAME),Linux)
    TARGET = aarch64-linux-gnu
else
    TARGET = aarch64-none-elf
endif
OBJCOPY = $(TARGET)-objcopy

.PHONY: build qemu_minimal qemu qemu_logger

all: rtc.bin

build:
    cargo build

rtc.bin: build
    $(OBJCOPY) -O binary target/aarch64-unknown-none/debug/rtc $@

qemu: rtc.bin
    qemu-system-aarch64 -machine virt,gic-version=3 -cpu max -serial mon:stdio -display

clean:
    cargo clean
    rm -f *.bin
```

.cargo/config.toml (변경할 필요가 없음):

```
[build]
target = "aarch64-unknown-none"
rustflags = ["-C", "link-arg=-Timage.ld"]
```

`make qemu`를 사용하여 QEMU에서 코드를 실행합니다.

56.2 전 Bare Metal 오후

RTC driver

(연습문제로돌아가기)

main.rs:

```
#![no_main]
#![no_std]

mod exceptions;
mod logger;
mod pl011;
mod pl031;

use crate::pl031::Rtc;
use arm_gic::gicv3::{IntId, Trigger};
```

```rust
use arm_gic::{irq_enable, wfi};
use chrono::{TimeZone, Utc};
use core::hint::spin_loop;
use crate::pl011::Uart;
use arm_gic::gicv3::GicV3;
use core::panic::PanicInfo;
use log::{error, info, trace, LevelFilter};
use smccc::psci::system_off;
use smccc::Hvc;

/// GICv3의 기본 주소입니다.
const GICD_BASE_ADDRESS: *mut u64 = 0x800_0000 as _;
const GICR_BASE_ADDRESS: *mut u64 = 0x80A_0000 as _;

/// 기본 PL011 UART의 기본 주소입니다.
const PL011_BASE_ADDRESS: *mut u32 = 0x900_0000 as _;

/// PL031 RTC의 기본 주소입니다.
const PL031_BASE_ADDRESS: *mut u32 = 0x901_0000 as _;
/// PL031 RTC에서 사용하는 IRQ입니다.
const PL031_IRQ: IntId = IntId::spi(2);

#[no_mangle]
extern "C" fn main(x0: u64, x1: u64, x2: u64, x3: u64) {
    // `PL011_BASE_ADDRESS`가 PL011 기기의 기본 주소이고
    // 이 주소 범위에 액세스하는 다른 항목이 없으므로 안전합니다.
    let uart = unsafe { Uart::new(PL011_BASE_ADDRESS) };
    logger::init(uart, LevelFilter::Trace).unwrap();

    info!("main({:#x}, {:#x}, {:#x}, {:#x})", x0, x1, x2, x3);

    // `GICD_BASE_ADDRESS` 및 `GICR_BASE_ADDRESS`가 각각 GICv3 배포자 및 재배포자의
 // 기본 주소이이고
    // 이러한 주소 범위에 액세스하는 다른 항목이 없으므로 안전합니다.
    let mut gic = unsafe { GicV3::new(GICD_BASE_ADDRESS, GICR_BASE_ADDRESS) };
    gic.setup();

    // `PL031_BASE_ADDRESS`가 PL031 기기의 기본 주소이고
    // 이 주소 범위에 액세스하는 다른 항목이 없으므로 안전합니다.
    let mut rtc = unsafe { Rtc::new(PL031_BASE_ADDRESS) };
    let timestamp = rtc.read();
    let time = Utc.timestamp_opt(timestamp.into(), 0).unwrap();
    info!("RTC: {time}");

    GicV3::set_priority_mask(0xff);
    gic.set_interrupt_priority(PL031_IRQ, 0x80);
    gic.set_trigger(PL031_IRQ, Trigger::Level);
    irq_enable();
    gic.enable_interrupt(PL031_IRQ, true);

    // 인터럽트 없이 3초간 기다립니다.
```

```
    let target = timestamp + 3;
    rtc.set_match(target);
    info!("{}을(를) 기다리는 중", Utc.timestamp_opt(target.into(), 0).unwrap());
    trace!(
        "matched={}, interrupt_pending={}",
        rtc.matched(),
        rtc.interrupt_pending()
    );
    while !rtc.matched() {
        spin_loop();
    }
    trace!(
        "matched={}, interrupt_pending={}",
        rtc.matched(),
        rtc.interrupt_pending()
    );
    info!("대기 완료됨");

    // 인터럽트를 위해 3초 더 기다립니다.
    let target = timestamp + 6;
    info!("{}을(를) 기다리는 중", Utc.timestamp_opt(target.into(), 0).unwrap());
    rtc.set_match(target);
    rtc.clear_interrupt();
    rtc.enable_interrupt(true);
    trace!(
        "matched={}, interrupt_pending={}",
        rtc.matched(),
        rtc.interrupt_pending()
    );
    while !rtc.interrupt_pending() {
        wfi();
    }
    trace!(
        "matched={}, interrupt_pending={}",
        rtc.matched(),
        rtc.interrupt_pending()
    );
    info!("대기 완료됨");

    system_off::<Hvc>().unwrap();
}

#[panic_handler]
fn panic(info: &PanicInfo) -> ! {
    error!("{info}");
    system_off::<Hvc>().unwrap();
    loop {}
}
```

pl031.rs:

```
use core::ptr::{addr_of, addr_of_mut};
```

```rust
#[repr(C, align(4))]
struct Registers {
    /// 데이터 레지스터
    dr: u32,
    /// 일치 레지스터
    mr: u32,
    /// 로드 레지스터
    lr: u32,
    /// 제어 레지스터
    cr: u8,
    _reserved0: [u8; 3],
    /// 인터럽트 마스크 세트 또는 정리 레지스터
    imsc: u8,
    _reserved1: [u8; 3],
    /// 원시 인터럽트 상태
    ris: u8,
    _reserved2: [u8; 3],
    /// 마스킹된 인터럽트 상태
    mis: u8,
    _reserved3: [u8; 3],
    /// 인터럽트 정리 레지스터
    icr: u8,
    _reserved4: [u8; 3],
}

/// PL031 실시간 시계용 드라이버
#[derive(Debug)]
pub struct Rtc {
    registers: *mut Registers,
}

impl Rtc {
    /// 지정된 기본 주소에 PL031 기기에 대한 RTC 드라이버의 새 인스턴스를
 /// 생성합니다.
 ///
 /// # 안전
 ///
 /// 지정된 기본 주소는 PL031 기기의
 /// MMIO 제어 레지스터를 가리켜야 하며,
 /// 이는 프로세스의 주소 공간에 기기 메모리로
 /// 매핑되어야 하며 다른 별칭은 없어야 합니다.
    pub unsafe fn new(base_address: *mut u32) -> Self {
        Self { registers: base_address as *mut Registers }
    }

    /// 현재 RTC 값을 읽습니다.
    pub fn read(&self) -> u32 {
        // self.registers가 적절하게 매핑된 PL031 기기의 제어 레지스터를
        // 가리키고 있으므로 안전합니다.
        unsafe { addr_of!((*self.registers).dr).read_volatile() }
```

```
    }

    /// 일치 값을 작성합니다. RTC 값이 이 값과 일치하면 인터럽트가
    /// 생성됩니다 (사용 설정된 경우).
    pub fn set_match(&mut self, value: u32) {
        // self.registers 가 적절하게 매핑된 PL031 기기의 제어 레지스터를
        // 가리키고 있으므로 안전합니다.
        unsafe { addr_of_mut!((*self.registers).mr).write_volatile(value) }
    }

    /// 인터럽트가 활성화되었는지 여부와 관계없이 일치 레지스터가 RTC 값과 일치하는지 여부를
    /// 반환합니다.
    pub fn matched(&self) -> bool {
        // self.registers 가 적절하게 매핑된 PL031 기기의 제어 레지스터를
        // 가리키고 있으므로 안전합니다.
        let ris = unsafe { addr_of!((*self.registers).ris).read_volatile() };
        (ris & 0x01) != 0
    }

    /// 현재 대기 중인 인터럽트가 있는지 여부를 반환합니다.
    ///
    /// 이는 'matched' 가 true 를 반환하고 인터럽트가 마스킹된 경우에만
    /// true 여야 합니다.
    pub fn interrupt_pending(&self) -> bool {
        // self.registers 가 적절하게 매핑된 PL031 기기의 제어 레지스터를
        // 가리키고 있으므로 안전합니다.
        let ris = unsafe { addr_of!((*self.registers).mis).read_volatile() };
        (ris & 0x01) != 0
    }

    /// 인터럽트 마스크를 설정하거나 지웁니다.
    ///
    /// 마스크가 true 인 경우 인터럽트가 사용 설정됩니다. false 이면
    /// 인터럽트가 사용 중지됩니다.
    pub fn enable_interrupt(&mut self, mask: bool) {
        let imsc = if mask { 0x01 } else { 0x00 };
        // self.registers 가 적절하게 매핑된 PL031 기기의 제어 레지스터를
        // 가리키고 있으므로 안전합니다.
        unsafe { addr_of_mut!((*self.registers).imsc).write_volatile(imsc) }
    }

    /// 대기 중인 인터럽트가 있는 경우 이를 지웁니다.
    pub fn clear_interrupt(&mut self) {
        // self.registers 가 적절하게 매핑된 PL031 기기의 제어 레지스터를
        // 가리키고 있으므로 안전합니다.
        unsafe { addr_of_mut!((*self.registers).icr).write_volatile(0x01) }
    }
}

// 모든 컨텍스트에서 액세스할 수 있는 기기 메모리에 대한
// 포인터만 포함하므로 안전합니다.
```

```
unsafe impl Send for Rtc {}
```

제 XIII 편

동시성 오전

제 57 장

Welcome to Concurrency in Rust

러스트는 동시성 지원이 막강합니다. 운영체제 레벨의 스레드를 사용하며, 뮤택스와 채널도 지원합니다.

러스트의 타입 시스템은 프로그램에 동시성 버그가 있을 경우, 컴파일 시에에러가 발생하도록 해 줍니다. 컴파일러를 이용해서 프로그램이 수행시에정확히 동작함을 미리 보장해 주기 때문에, 사람들은 이를 종종 이라고 합니다.

- Rust lets us access OS concurrency toolkit: threads, sync. primitives, etc.
- The type system gives us safety for concurrency without any special features.
- The same tools that help with ”concurrent” access in a single thread (e.g., a called function that might mutate an argument or save references to it to read later) save us from multi-threading issues.

제 58 장

스레드

러스트의 스레드는 다른 언어의 스레드와 유사하게 동작합니다:

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

fn main() {
    thread::spawn(|| {
        for i in 1..10 {
            println!("스레드의 개수: {i}!");
            thread::sleep(Duration::from_millis(5));
        }
    });

    for i in 1..5 {
        println!("기본 스레드: {i}");
        thread::sleep(Duration::from_millis(5));
    }
}
```

- 스레드는 모두 데몬 스레드입니다. 따라서 메인 스레드는 이 스레드들이끝나기를 기다리지 않습니다.
- 한 스레드에서 발생한 패닉은 다른 스레드에게 영향을 끼치지 않습니다.
  - 패닉은 추가정보 (페이로드) 를 포함할 수 있으며, 이는 `downcast_ref` 로 풀어볼 수 있습니다.
- Rust thread APIs look not too different from e.g. C++ ones.
- Run the example.
  - 5ms timing is loose enough that main and spawned threads stay mostly in lockstep.
  - Notice that the program ends before the spawned thread reaches 10!
  - This is because main ends the program and spawned threads do not make it persist.
    - Compare to pthreads/C++ std::thread/boost::thread if desired.
- How do we wait around for the spawned thread to complete?
- `thread::spawn` returns a `JoinHandle`. Look at the docs.
  - `JoinHandle` has a `.join()` method that blocks.

- Use `let handle = thread::spawn(...)` and later `handle.join()` to wait for the thread to finish and have the program count all the way to 10.
- Now what if we want to return a value?
- Look at docs again:
  - `thread::spawn`'s closure returns `T`
  - `JoinHandle .join()` returns `thread::Result<T>`
- Use the `Result` return value from `handle.join()` to get access to the returned value.
- Ok, what about the other case?
  - Trigger a panic in the thread. Note that this doesn't panic `main`.
  - Access the panic payload. This is a good time to talk about `Any`.
- Now we can return values from threads! What about taking inputs?
  - Capture something by reference in the thread closure.
  - An error message indicates we must move it.
  - Move it in, see we can compute and then return a derived value.
- If we want to borrow?
  - Main kills child threads when it returns, but another function would just return and leave them running.
  - That would be stack use-after-return, which violates memory safety!
  - How do we avoid this? see next slide.

58.1 범위 스레드 (Scoped Threads)

보통, 스레드는 스레드 밖에서 데이터를 빌릴 수 없습니다:

```
use std::thread;

fn foo() {
    let s = String::from("안녕하세요");
    thread::spawn(|| {
        println!("길이: {}", s.len());
    });
}

fn main() {
    foo();
}
```

하지만, scoped thread 에서는 가능합니다:

```
use std::thread;

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

    thread::scope(|scope| {
        scope.spawn(|| {
            println!("길이: {}", s.len());
```

```
        });
    });
}
```

- `thread::scope` 함수가 완료되면 그 안에서 생성된 모든스레드들이 종료했음이 보장되기 때문에, 그 때 빌렸던 데이터들을 다시반환할 수 있기 때문입니다.
- 일반적인 러스트의 빌림 규칙이 적용됩니다: 한 스레드에 의한 가변 빌림또는 여러 스레드에 대한 불변 빌림중 하나만 가능합니다.

제 59 장

채널

러스트의 채널은 Sender<T> 와 Receiver<T> 두 부분으로 구성됩니다. 이둘은 채널을 통해 서로 연결되어 있지만, 우리는 채널을 볼 수는 없고 이 양끝단만을 사용하게 됩니다.

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

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

    tx.send(10).unwrap();
    tx.send(20).unwrap();

    println!("수신됨: {:?}", rx.recv());
    println!("수신됨: {:?}", rx.recv());

    let tx2 = tx.clone();
    tx2.send(30).unwrap();
    println!("수신됨: {:?}", rx.recv());
}
```

- mpsc 는 "Multi-Produce, Single-Consumer"를 의미합니다. Sender 와 SyncSender 는 Clone 을 구현하지만 (즉, 여러개의 producer 를 만들수 있습니다) Receiver 는 Clone 을 구현하지 않습니다.
- send() 와 recv() 는 Result 를 반환합니다. 만일 Err 가 반환된다면, 상대방의 Sender 또는 Receiver 가 삭제되었고 채널이 닫혔다는 뜻입니다.

59.1 무경계채널

mpsc::channel() 함수는 경계가 없는 비동기 채널을 생성합니다:

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

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

```
    thread::spawn(move || {
        let thread_id = thread::current().id();
        for i in 1..10 {
            tx.send(format!("메시지 {i}")).unwrap();
            println!("{thread_id:?}: 보낸 메시지 {i}");
        }
        println!("{thread_id:?}: 완료");
    });
    thread::sleep(Duration::from_millis(100));

    for msg in rx.iter() {
        println!("기본: {msg} 받음");
    }
}
```

59.2 경계채널

With bounded (synchronous) channels, `send` can block the current thread:

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

fn main() {
    let (tx, rx) = mpsc::sync_channel(3);

    thread::spawn(move || {
        let thread_id = thread::current().id();
        for i in 1..10 {
            tx.send(format!("메시지 {i}")).unwrap();
            println!("{thread_id:?}: 보낸 메시지 {i}");
        }
        println!("{thread_id:?}: 완료");
    });
    thread::sleep(Duration::from_millis(100));

    for msg in rx.iter() {
        println!("기본: {msg} 받음");
    }
}
```

- `send` 를 호출하면 채널에 새 메시지를 위한 공간이 확보될 때까지현재 스레드가 차단됩니다. 채널에서 읽는 사람이 없는 경우 스레드가무기한 차단될 수 있습니다.
- `send` 호출은 오류와 함께 중단됩니다. 따라서 채널이 닫히면 `Result` 를 반환합니다. 수신자를 삭제하면 채널이 닫힙니다.
- A bounded channel with a size of zero is called a ”rendezvous channel”. Every send will block the current thread until another thread calls `recv`.

제 60 장

Send와 Sync

How does Rust know to forbid shared access across threads? The answer is in two traits:

- Send: T 가 스레드 간 이동이 안전하다면, T 의 타입은 Send 입니다.
- Sync: &T 가 스레드 간 이동이 안전하다면, &T 의 타입은 Sync 입니다.

Send 와 Sync 트레잇은안전하지않은 트레잇입니다. 컴파일러는 타입의 요소들이 모두 Send 와 Sync 타입이면 자동으로 이 트레잇들을 적용시켜 줍니다. 물론여러분 스스로 맞다고 알고 있다면 직접 구현해도 됩니다.

- Sync 와 Send 는 어떤 타입이 특정한 스레드-안전 속성을가짐을 나타내는 마커로 생각할 수 있습니다.
- 이 두 트레이트는 제너릭에서 제약 조건을 나타내는 트레이트로 사용될수도 있습니다.

60.1 Send

> T 가 스레드 간에 안전하게 이동될 수 있다면, T 의 타입은 Send 입니다.

소유권을 다른 스레드로 이동하면 소멸자가 해당 스레드에서 실행됩니다. 여기서 의문은 ”언제 한 스레드에서 값을 할당하고 다른 스레드에서 값을할당 해제할 수 있는가” 입니다.

예를 들어 SQLite 라이브러리 연결은 단일 스레드에서만 액세스해야 합니다.

60.2 Sync

> &T 가 여러 스레드에서 안전하게 접근될 수 있다면, &T 의타입은 Sync 입니다.

좀 더 정확한 정의는 다음과 같습니다:

> &T 가 Send 인 경우에만 T 의 타입이 Sync 가 됩니다

위 문장을 풀어서 이야기 하면, 어떤 타입이 스레드 간에 공유되어서사용되기에 안전하다면 그 타입의 참조 타입은 스레드 간에 이동 가능하다는것입니다.

이는 다음과 같이 증명할 수 있습니다: 어떤 타입이 Sync 라는 말은곧 그 타입이 여러 스레드들 사이에서 데이터 레이스나 여타 동기화 문제없이 공유 가능하다는 말입니다. 스레드 간 공유가 안전하다면, 스레드간이동도 안전할 수 밖에 없습니다. 어떤 타입의 스레드간 이동이 안전하다면, 그 타입의 참조 또한 스레드간 이동이 안전할 수 밖에 없습니다.

60.3 예제

Send + Sync

여러분이 다루게 될 대부분의 타입은 Send + Sync 입니다:

- i8, f32, bool, char, &str, ...
- (T1, T2), [T; N], &[T], struct { x: T }, ...
- String, Option<T>, Vec<T>, Box<T>, ...
- Arc<T>: 참조 카운트 조작을 아토믹 하기때문에 스레드 안전함.
- Mutex<T>: 값을 접근하기 위해 뮤택스를잠궈야 하기 때문에 스레드 안전함.
- AtomicBool, AtomicU8, ...: 값을 접근할 때 특별한아토믹 명령어들을 사용합니다.

제네릭 타입은 일반적으로 타입 파라메터가 Send + Sync 이면 Send + Sync 입니다.

Send + !Sync

아래 타입들은 다른 스레드로 이동될 수 있지만 내부적으로 값이 변경될 수있기 때문에 스레드 안전하지 않습니다:

- mpsc::Sender<T>
- mpsc::Receiver<T>
- Cell<T>
- RefCell<T>

!Send + Sync

아래 타입들은 스레드 안전하지만 다른 스레드로 이동될 수 없습니다:

- MutexGuard<T: Sync>: Uses OS level primitives which must be deallocated on the thread which created them.

!Send + !Sync

아래 타입들은 스레드 안전하지도 않고 다른 스레드로 이동될 수도 없습니다:

- Rc<T>: Rc<T> 는 아토믹하지 않은 방식으로 참조카운트를 조작하는 RcBox<T>를참조합니다.
- *const T, *mut T: 러스트는 포인터가 스레드안전하지 않다고 가정합니다.

제 61 장

상태공유

러스트는 주로 아래 두 가지 타입을 이용해서 공유 데이터 동기화를수행합니다:

- Arc<T>, T 에 대한 아토믹 참조 카운트: 이 참조는 다수의 스레드 사이에서공유될 수 있고, 참조하던 마지막 스레드가 종료할 경우 T 를반환합니다.
- Mutex<T>: T 값에 대한 상호 배제 엑세스를 보장합니다.

61.1 Arc

Arc<T> allows shared read-only access via Arc::clone:

```
use std::sync::Arc;
use std::thread;

fn main() {
    let v = Arc::new(vec![10, 20, 30]);
    let mut handles = Vec::new();
    for _ in 1..5 {
        let v = Arc::clone(&v);
        handles.push(thread::spawn(move || {
            let thread_id = thread::current().id();
            println!("{thread_id:?}: {v:?}");
        }));
    }

    handles.into_iter().for_each(|h| h.join().unwrap());
    println!("v: {v:?}");
}
```

- Arc stands for "Atomic Reference Counted", a thread safe version of Rc that uses atomic operations.
- Arc<T> implements Clone whether or not T does. It implements Send and Sync if and only if T implements them both.
- Arc::clone() has the cost of atomic operations that get executed, but after that the use of the T is free.
- Beware of reference cycles, Arc does not use a garbage collector to detect them.
  - std::sync::Weak can help.

61.2 Mutex

Mutex<T> ensures mutual exclusion *and* allows mutable access to T behind a read-only interface (another form of interior mutability):

```
use std::sync::Mutex;

fn main() {
    let v = Mutex::new(vec![10, 20, 30]);
    println!("v: {:?}", v.lock().unwrap());

    {
        let mut guard = v.lock().unwrap();
        guard.push(40);
    }

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

모든 Mutex<T>는 impl<T: Send> Sync for Mutex<T>를자동으로 구현함을 참조하세요.

- Mutex in Rust looks like a collection with just one element — the protected data.
  - It is not possible to forget to acquire the mutex before accessing the protected data.
- You can get an &mut T from an &Mutex<T> by taking the lock. The MutexGuard ensures that the &mut T doesn't outlive the lock being held.
- Mutex<T> implements both Send and Sync iff (if and only if) T implements Send.
- A read-write lock counterpart: RwLock.
- Why does lock() return a Result?
  - If the thread that held the Mutex panicked, the Mutex becomes "poisoned" to signal that the data it protected might be in an inconsistent state. Calling lock() on a poisoned mutex fails with a PoisonError. You can call into_inner() on the error to recover the data regardless.

61.3 예제

Arc 와 Mutex 의 동작을 살펴봅시다:

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

fn main() {
    let v = vec![10, 20, 30];
    let handle = thread::spawn(|| {
        v.push(10);
    });
    v.push(1000);

    handle.join().unwrap();
    println!("v: {v:?}");
}
```

가능한 해결책:

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

fn main() {
    let v = Arc::new(Mutex::new(vec![10, 20, 30]));

    let v2 = Arc::clone(&v);
    let handle = thread::spawn(move || {
        let mut v2 = v2.lock().unwrap();
        v2.push(10);
    });

    {
        let mut v = v.lock().unwrap();
        v.push(1000);
    }

    handle.join().unwrap();

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

눈여겨 볼 부분:

- v 는 Arc 와 Mutex 모두에 포함되어 있습니다. 이는 Arc 와 Mutex 가 서로 완전히 다른 문제를 위한도구이기 때문입니다.
  - Mutex 를 Arc 로 래핑하는 것은 가변 상태를 스레드들간에 공유할 때 흔히 사용하는 패턴입니다.
- v: Arc<_>를 다른 스레드에서사용하려면, 먼저 v2 로 복사를 하고 이를 그 스레드로 이동 해야 합니다. 그래서 람다의 시그니처에 move 가 있는 것입니다.
- 블록은 LockGuard 의 범위를 최대한 좁히기 위해 사용되었습니다.

제 62 장

연습문제

동시성 기법들을 연습해 봅시다

- 식사하는 철학자 문제: 고적적인 동시성 문제입니다.
- 멀티 스레드 링크 검사기: 병렬적으로 웹페이지의 링크들을 체크합니다. 카고를 통해 몇 가지 의존성들을 다운도르 받아야 하는 큰 프로젝트입니다.

After looking at the exercises, you can look at the solutions provided.

62.1 식사하는철학자들

식사하는 철학자 문제는 동시성에 있어서 고전적인 문제입니다:

> 5 명의 철학자가 원탁에서 식사를 하고 있습니다. 철학자는 원탁에서 자신의자리에 앉아있습니다. 포크는 각 접시 사이에 있습니다. 제공되는 요리를먹기 위해서는 두 개의 포크를 모두 사용해야합니다. 철학자는 생각을 하다가배가 고프면 자신의 좌, 우의 포크를 들어 요리를 먹습니다. 철학자는 요리를먹은 후에는 포크를 다시 자리에 내려놓습니다. 철학자는 자신의 좌, 우에포크가 있을때만 요리를 먹을 수 있습니다. 따라서 두 개의 포크는 오직자신의 좌, 우 철학자가 생각을 할 때만 사용할 수 있습니다.

You will need a local Cargo installation for this exercise. Copy the code below to a file called `src/main.rs`, fill out the blanks, and test that `cargo run` does not deadlock:

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

struct Fork;

struct Philosopher {
    name: String,
    // left_fork: ...
    // right_fork: ...
    // thoughts: ...
}

impl Philosopher {
```

```
    fn think(&self) {
        self.thoughts
            .send(format!("유레카! {}에 새로운 아이디어가 있습니다.", &self.name))
            .unwrap();
    }

    fn eat(&self) {
        // 포크를 드세요...
        println!("{}님이 먹고 있습니다...", &self.name);
        thread::sleep(Duration::from_millis(10));
    }
}

static PHILOSOPHERS: &[&str] =
    &["Socrates", "히파티아", "플라톤", "아리스토텔레스", "피타고라스"];

fn main() {
    // 포크 만들기

    // 철학자 만들기

    // 각각 100 번 생각하고 먹도록 합니다.

    // 생각을 출력합니다.
}
```

다음과 같은 `Cargo.toml` 을 사용할 수 있습니다.

```
[package]
name = "dining-philosophers"
version = "0.1.0"
edition = "2021"
```

62.2 멀티스레드 링크검사기

새로 배운것들을 활용해서 멀티 스레드 링크 검사기를 만듭니다. 이 검사기는웹페이지 안에 있는 링크들이 유효한지 확인합니다. 그리고 재귀적으로 동일도메인의 다른 모든 페이지가 유효한지 확인합니다.

이를 위해서 `reqwest` 와 같은 HTTP 클라이언트가 필요합니다. 새로운 로컬 프로젝트를 만들고 `reqwest` 를 의존성에추가하십시요:

```
cargo new link-checker
cd link-checker
cargo add --features blocking,rustls-tls reqwest
```

> 만일 `cargo add` 커맨드가 `error: no such subcommand` 로 실패한다면 `Cargo.toml` 파일을 직접 수정해도 됩니다. 아래에전체 의존성 내용이 있습니다.

링크를 찾기 위해서 `scraper` 도추가합니다:

```
cargo add scraper
```

마지막으로 오류 처리하는 방법으로 `thiserror` 도 추가합니다:

```
cargo add thiserror
```

모든 `cargo add` 가 끝나면 `Cargo.toml` 에 아래 내용이추가됩니다:

```
[package]
name = "link-checker"
version = "0.1.0"
edition = "2021"
publish = false

[dependencies]
reqwest = { version = "0.11.12", features = ["blocking", "rustls-tls"] }
scraper = "0.13.0"
thiserror = "1.0.37"
```

이제 `https://www.google.org/` 같은 웹 페이지를 탐색할 수있습니다.

`rc/main.rs` 파일은 아래와 같습니다:

```
use reqwest::blocking::Client;
use reqwest::Url;
use scraper::{Html, Selector};
use thiserror::Error;

#[derive(Error, Debug)]
enum Error {
    #[error("요청 오류: {0}")]
    ReqwestError(#[from] reqwest::Error),
    #[error("잘못된 http 응답: {0}")]
    BadResponse(String),
}

#[derive(Debug)]
struct CrawlCommand {
    url: Url,
    extract_links: bool,
}

fn visit_page(client: &Client, command: &CrawlCommand) -> Result<Vec<Url>, Error> {
    println!("{:#} 확인 중", command.url);
    let response = client.get(command.url.clone()).send()?;
    if !response.status().is_success() {
        return Err(Error::BadResponse(response.status().to_string()));
    }

    let mut link_urls = Vec::new();
    if !command.extract_links {
        return Ok(link_urls);
    }

    let base_url = response.url().to_owned();
    let body_text = response.text()?;
    let document = Html::parse_document(&body_text);
```

```
    let selector = Selector::parse("a").unwrap();
    let href_values = document
        .select(&selector)
        .filter_map(|element| element.value().attr("href"));
    for href in href_values {
        match base_url.join(href) {
            Ok(link_url) => {
                link_urls.push(link_url);
            }
            Err(err) => {
                println!("{base_url:#}에서: 파싱할 수 없는 {href:?}: {err}을(를) 무시함");
            }
        }
    }
    Ok(link_urls)
}

fn main() {
    let client = Client::new();
    let start_url = Url::parse("https://www.google.org").unwrap();
    let crawl_command = CrawlCommand{ url: start_url, extract_links: true };
    match visit_page(&client, &crawl_command) {
        Ok(links) => println!("링크: {links:#?}"),
        Err(err) => println!("링크를 추출할 수 없습니다: {err:#}"),
    }
}
```

아래 커맨드로 소스를 실행합니다

```
cargo run
```

태스크

- 스레드를 사용하여 링크를 병렬로 확인합니다: URL 을 채널로 보내서 몇개의 스레드가 URL 을 병렬로 체크하도록 합니다.
- `www.google.org` 도메인의 모든 페이지를 재귀적으로 확인하기 위해코드를 확장해서 작성합니다: 차단당하지 않도록 100 페이지 정도로 제한을두시기 바랍니다.

62.3 Concurrency Morning Exercise

식사하는철학자들

(연습문제로돌아가기

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

struct Fork;

struct Philosopher {
```

```
    name: String,
    left_fork: Arc<Mutex<Fork>>,
    right_fork: Arc<Mutex<Fork>>,
    thoughts: mpsc::SyncSender<String>,
}

impl Philosopher {
    fn think(&self) {
        self.thoughts
            .send(format!("유레카! {}에 새로운 아이디어가 있습니다.", &self.name))
            .unwrap();
    }

    fn eat(&self) {
        println!("{}님이 식사를 하려고 합니다.", &self.name);
        let _left = self.left_fork.lock().unwrap();
        let _right = self.right_fork.lock().unwrap();

        println!("{}님이 먹고 있습니다...", &self.name);
        thread::sleep(Duration::from_millis(10));
    }
}

static PHILOSOPHERS: &[&str] =
    &["Socrates", "히파티아", "플라톤", "아리스토텔레스", "피타고라스"];

fn main() {
    let (tx, rx) = mpsc::sync_channel(10);

    let forks = (0..PHILOSOPHERS.len())
        .map(|_| Arc::new(Mutex::new(Fork)))
        .collect::<Vec<_>>();

    for i in 0..forks.len() {
        let tx = tx.clone();
        let mut left_fork = Arc::clone(&forks[i]);
        let mut right_fork = Arc::clone(&forks[(i + 1) % forks.len()]);

        // 교착 상태를 방지하려면 어딘가에서
        // 대칭을 중단시켜야 합니다. 이렇게 하면 둘 중 어느 하나라도 초기화하지 않고
        // 포크가 교체됩니다.
        if i == forks.len() - 1 {
            std::mem::swap(&mut left_fork, &mut right_fork);
        }

        let philosopher = Philosopher {
            name: PHILOSOPHERS[i].to_string(),
            thoughts: tx,
            left_fork,
            right_fork,
        };
```

```
        thread::spawn(move || {
            for _ in 0..100 {
                philosopher.eat();
                philosopher.think();
            }
        });
    }

    drop(tx);
    for thought in rx {
        println!("{thought}");
    }
}
```

Link Checker

(연습문제로돌아가기)

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

use reqwest::blocking::Client;
use reqwest::Url;
use scraper::{Html, Selector};
use thiserror::Error;

#[derive(Error, Debug)]
enum Error {
    #[error("요청 오류: {0}")]
    ReqwestError(#[from] reqwest::Error),
    #[error("잘못된 http 응답: {0}")]
    BadResponse(String),
}

#[derive(Debug)]
struct CrawlCommand {
    url: Url,
    extract_links: bool,
}

fn visit_page(client: &Client, command: &CrawlCommand) -> Result<Vec<Url>, Error> {
    println!("{:#} 확인 중", command.url);
    let response = client.get(command.url.clone()).send()?;
    if !response.status().is_success() {
        return Err(Error::BadResponse(response.status().to_string()));
    }

    let mut link_urls = Vec::new();
    if !command.extract_links {
        return Ok(link_urls);
```

```
    }

    let base_url = response.url().to_owned();
    let body_text = response.text()?;
    let document = Html::parse_document(&body_text);

    let selector = Selector::parse("a").unwrap();
    let href_values = document
        .select(&selector)
        .filter_map(|element| element.value().attr("href"));
    for href in href_values {
        match base_url.join(href) {
            Ok(link_url) => {
                link_urls.push(link_url);
            }
            Err(err) => {
                println!("{base_url:#}에서: 파싱할 수 없는 {href:?}: {err}을(를) 무시함");
            }
        }
    }
    Ok(link_urls)
}

struct CrawlState {
    domain: String,
    visited_pages: std::collections::HashSet<String>,
}

impl CrawlState {
    fn new(start_url: &Url) -> CrawlState {
        let mut visited_pages = std::collections::HashSet::new();
        visited_pages.insert(start_url.as_str().to_string());
        CrawlState { domain: start_url.domain().unwrap().to_string(), visited_pages }
    }

    /// 주어진 페이지 내의 링크를 추출해야 하는지 여부를 결정합니다.
    fn should_extract_links(&self, url: &Url) -> bool {
        let Some(url_domain) = url.domain() else {
            return false;
        };
        url_domain == self.domain
    }

    /// 지정된 페이지를 방문한 것으로 표시하고 이미 방문한 경우
    /// false를 반환합니다.
    fn mark_visited(&mut self, url: &Url) -> bool {
        self.visited_pages.insert(url.as_str().to_string())
    }
}

type CrawlResult = Result<Vec<Url>, (Url, Error)>;
```

```
fn spawn_crawler_threads(
    command_receiver: mpsc::Receiver<CrawlCommand>,
    result_sender: mpsc::Sender<CrawlResult>,
    thread_count: u32,
) {
    let command_receiver = Arc::new(Mutex::new(command_receiver));

    for _ in 0..thread_count {
        let result_sender = result_sender.clone();
        let command_receiver = command_receiver.clone();
        thread::spawn(move || {
            let client = Client::new();
            loop {
                let command_result = {
                    let receiver_guard = command_receiver.lock().unwrap();
                    receiver_guard.recv()
                };
                let Ok(crawl_command) = command_result else {
                    // 발신자가 삭제되었습니다. 더 이상 명령어가 수신되지 않습니다.
                    break;
                };
                let crawl_result = match visit_page(&client, &crawl_command) {
                    Ok(link_urls) => Ok(link_urls),
                    Err(error) => Err((crawl_command.url, error)),
                };
                result_sender.send(crawl_result).unwrap();
            }
        });
    }
}

fn control_crawl(
    start_url: Url,
    command_sender: mpsc::Sender<CrawlCommand>,
    result_receiver: mpsc::Receiver<CrawlResult>,
) -> Vec<Url> {
    let mut crawl_state = CrawlState::new(&start_url);
    let start_command = CrawlCommand { url: start_url, extract_links: true };
    command_sender.send(start_command).unwrap();
    let mut pending_urls = 1;

    let mut bad_urls = Vec::new();
    while pending_urls > 0 {
        let crawl_result = result_receiver.recv().unwrap();
        pending_urls -= 1;

        match crawl_result {
            Ok(link_urls) => {
                for url in link_urls {
                    if crawl_state.mark_visited(&url) {
                        let extract_links = crawl_state.should_extract_links(&url);
```

```
                        let crawl_command = CrawlCommand { url, extract_links };
                        command_sender.send(crawl_command).unwrap();
                        pending_urls += 1;
                    }
                }
            }
            Err((url, error)) => {
                bad_urls.push(url);
                println!("크롤링 오류 발생: {:#}", error);
                continue;
            }
        }
    }
    bad_urls
}

fn check_links(start_url: Url) -> Vec<Url> {
    let (result_sender, result_receiver) = mpsc::channel::<CrawlResult>();
    let (command_sender, command_receiver) = mpsc::channel::<CrawlCommand>();
    spawn_crawler_threads(command_receiver, result_sender, 16);
    control_crawl(start_url, command_sender, result_receiver)
}

fn main() {
    let start_url = reqwest::Url::parse("https://www.google.org").unwrap();
    let bad_urls = check_links(start_url);
    println!("잘못된 URL: {:#?}", bad_urls);
}
```

제 XIV 편

동시성 오후

제 63 장

Async Rust

”Async”는 블럭될 (더 이상 진행할 수 없을) 때까지 각 작업을 실행한 다음진행할 준비가 된 다른 작업으로 전환하여 여러 작업을 동시에 실행하는 동시실행 모델입니다. 이 모델을 사용하면 제한된 수의 스레드에서 더 많은작업을 실행할 수 있습니다. 이는, 한 작업을 유지하고 수행하는데 필요한오버헤드가 (스레드에 비해) 매우 낮고 운영체제가 여러 I/O 들에서 현재 진행가능한 I/O 들을 효과적으로 식별해 주는 프리미티브를 제공하기 때문입니다.

Rust 의 비동기 작업은 ”futures”를 기반으로 하며 이는 미래에 완료될 수있는 작업을 나타냅니다. Futures 는 완료되었다는 신호를 보낼 때까지”폴링”됩니다.

Futures 는 비동기 런타임에 의해 폴링되며, 비동기 런타임에는 여러 다양한종류가 있습니다.

비교

- 파이썬에도 `asyncio` 라는 유사한 모델이 있습니다. 그러나파이썬의 `Future` 타입은 콜백 기반이며 폴링되지 않습니다. 파이썬으로 비동기 프로그래밍을 할 때에는, Rust 에서 런타임이 내부적으로해 주는 것과 유사한, ”루프”를 명시적으로 사용해야 합니다.
- 자바스크립트의 `Promise` 도 비슷하지만 역시 콜백 기반입니다. 자바스크립트에서는 이벤트 루프가런타임 엔진에서 구현되므로 `Promise` 가 처리되는 세부 과정이 숨겨집니다.

63.1 `async`/`await`

겉에서 보았을 때, 비동기 Rust 코드는 일반적인 절차적 코드와 매우유사합니다.

```
use futures::executor::block_on;

async fn count_to(count: i32) {
    for i in 1..=count {
        println!("수: {i}개!");
    }
}

async fn async_main(count: i32) {
    count_to(count).await;
}
```

```
fn main() {
    block_on(async_main(10));
}
```

키 포인트:

- Rust 비동기 문법을 보여주는 간단한 예시입니다. 여기에는 오래 실행되는작업이나, 실제로 동시에 수행되는 것들은 없습니다.
- `async` 함수의 리턴 타입은 무엇인가요?
  - `main` 에서 'let future: () = async_main(10); 을 사용하여타입을 확인하세요.
- The "async" keyword is syntactic sugar. The compiler replaces the return type with a future.
- `main` 을 비동기 함수로 만들수는 없습니다. 만약 그렇게 할 경우컴파일러는 리턴 타입인 future 를 어떻게 사용할 지 모르기 때문입니다.
- You need an executor to run async code. `block_on` blocks the current thread until the provided future has run to completion.
- `.await` 는 다른 작업이 완료될 때까지 비동기적으로 대기합니다. `block_on` 과 달리 `.await` 는 현재 스레드를 블록하지않습니다.
- `.await` can only be used inside an `async` function (or block; these are introduced later).

63.2 Future

`Future` 는트레잇입니다. 이 트레잇은 아직 완료되지 않았을 수도 있는 작업을나타냅니다. Future 는 `poll` 함수를 통해 폴링될 수 있으며, 이함수는 `Poll` 을반환합니다.

```
use std::pin::Pin;
use std::task::Context;

pub trait Future {
    type Output;
    fn poll(self: Pin<&mut Self>, cx: &mut Context<'_>) -> Poll<Self::Output>;
}

pub enum Poll<T> {
    Ready(T),
    Pending,
}
```

비동기 함수는 `impl Future` 를 반환합니다. 새로운 타입을 만들고이 타입이 `Future` 를 구현하게 할 수도 있지만 일반적이지는않습니다. 예를 들어 `tokio::spawn` 가 리턴하는 `JoinHandle` 은 `Future` 를 구현하며, 이를 통해 생성된스레드에 join 할 수 있습니다.

Future 에 `.await` 를 호출하면, 해당 Future 가 준비될 때까지 현재비동기 함수가 일시 중지됩니다. 그런 다음 Future 가 준비가 되면, 그 값이 `.await` 구문의 값이 됩니다.

- `Future` 와 `Poll` 타입의 실제 정의는 위에 보이는 그대로입니다. 링크를 클릭하면 Rust 문서에서 한 번 더 확인할 수 있습니다.
- 본 강의의 목적은 비동기 코드를 작성하는데 있기 때문에, 새로운 비동기프리미티브를 만드는데 필요한 `Pin` 과 `Context` 는 다루지않습니다. 이들에 대해 간단히 설명하자면:

  - `Context` 를 사용하면 Future 가 이벤트가 발생할 때 다시폴링되도록 예약할 수 있습니다.
  - `Pin` 을 사용하면 메모리에서 Future 의 위치가 고정되기 때문에해당 future 의 포인터가 항상 유효하게 유지됩니다. 이는 `.await` 후에 참조를 유효한 상태로 유지하기 위해 필요합니다.

63.3 비동기런타임들

비동기 _런타임_은 (비동기식 작업 실행을 지원) 와 (futures 를 실행) 의 두 가지 역할을 합니다. Rust 언어자체에서 기본 제공하는 비동기 런타임은 없습니다. 그러나 다음과 같은비동기 런타임 크레잇들이 있습니다.

- Tokio: performant, with a well-developed ecosystem of functionality like Hyper for HTTP or Tonic for gRPC.
- async-std: aims to be a "std for async", and includes a basic runtime in `async::task`.
- smol: simple and lightweight

여러 대규모 애플리케이션에는 자체 런타임이 있는 경우도 있습니다. 예들들어 Fuchsia 가있습니다.

- Rust 플레이그라운드에서는 위에 나열된 비동기 런타임 중에서 Tokio 만사용할 수 있습니다. 또한 Rust 플레이그라운드는 I/O 를 허용하지 않으므로 async 를 가지고 할 수 있는 많은 흥미로운 작업들이 불가능 합니다.
- Futures 는 실행자가 폴링하지 않는 한 아무것도 하지 않는다는 점에서 (I/O 작업조차 시작하지 않음) "비활성" 상태입니다. 이는 사용되지 않는경우에도 완료될 때 까지 실행되는, 자바 스크립트의 promise 와 다릅니다.

63.3.1 Tokio

Tokio provides:

- 비동기 코드 실행을 위한 멀티스레드 런타임
- 표준 라이브러리의 비동기 버전
- 대규모 라이브러리 생태계

```
use tokio::time;

async fn count_to(count: i32) {
    for i in 1..=count {
        println!("작업 개수: {i}개!");
        time::sleep(time::Duration::from_millis(5)).await;
    }
}

#[tokio::main]
async fn main() {
    tokio::spawn(count_to(10));

    for i in 1..5 {
        println!("기본 작업: {i}");
        time::sleep(time::Duration::from_millis(5)).await;
    }
}
```

- 이제 `tokio::main` 매크로를 사용하면 `main` 을 비동기로만들 수 있습니다.
- `spawn` 함수는 동시 실행되는 새로운 "작업"을 만듭니다.
- 참고: `spawn` 은 `Future` 를 인자로 받습니다. 때문에 `count_to` 에 `.await` 를 호출하지 않는 점을 주목하세요.

심화 학습:

- `count_to` 가 10 에 도달하지 않는 경우가 많은데 그 이유는무엇일까요? 이는 비동기적인 취소를 보여주는 예입니다. `tokio::spawn` 이 리턴하는 것은 완료될 때까지 기다리도록대기하는데 사용되는 핸들입니다.
- `tokio::spawn` 대신 `count_to(10).await` 를 사용해보세요.
- `tokio::spawn` 에서 반환된 작업을 `await` 해 보세요.

63.4 태스크

Rust 의 태스크 (작업) 시스템은 경량 스레딩의 한 종류로 볼 수 있습니다.

하나의 작업에는, 실행자가 이 작업을 진행하기 위해 계속 폴링하는, 최상위 future 가 한 개 있습니다. 이 future 에는 `poll` 메서드가 폴링하는중첩된 future 가 한 개 이상 있을 수 있습니다. 이러한 중첩된 future 는일반적인 함수 호출 스택하고 비슷한 역할을 합니다. 한 작업 안에서 여러자식 future 들을 폴링하면, 타이머를 켜는 것과 어떤 I/O 작업을 동시에수행시킨 후 타이머와 I/O 중 먼저 끝나는 것을 기다리는 것과 같은동시성도구현할 수 있습니다.

```
use tokio::io::{self, AsyncReadExt, AsyncWriteExt};
use tokio::net::TcpListener;

#[tokio::main]
async fn main() -> io::Result<()> {
    let listener = TcpListener::bind("127.0.0.1:0").await?;
    println!("포트 {}에서 수신 대기", listener.local_addr()?.port());

    loop {
        let (mut socket, addr) = listener.accept().await?;

        println!("{addr:?}에서 연결");

        tokio::spawn(async move {
            socket.write_all(b"누구세요?\n").await.expect("소켓 오류");

            let mut buf = vec![0; 1024];
            let name_size = socket.read(&mut buf).await.expect("소켓 오류");
            let name = std::str::from_utf8(&buf[..name_size]).unwrap().trim();
            let reply = format!("{name}님, 전화해 주셔서 감사합니다.\n");
            socket.write_all(reply.as_bytes()).await.expect("소켓 오류");
        });
    }
}
```

이 예제를, 로컬 컴퓨터에 만들어 둔 `src/main.rs` 에 복사하고거기에서 실행하세요.

nc 또는 telnet 과 같은 TCP 연결 도구를 사용하여 연결해 보세요.

- 수강생들에게 이 서버에 몇 개의 클라이언트가 연결되면 이 서버의 상태가어떻게 변할지 그림을 그려보도록 하세요. 어떤 태스크들이 있는지, 이태스크들의 Future 는 어떤 상태에 있는지 물어봅니다.
- This is the first time we've seen an `async` block. This is similar to a closure, but does not take any arguments. Its return value is a Future, similar to an `async fn`.
- Async 블록을 함수로 리팩터링하고 ?를 사용하여 오류 처리를개선해 봅시다.

63.5 비동기채널

여러 크레이트에서 비동기 채널을 지원합니다. 예를 들어 `tokio` 에서는 아래와 같이합니다.

```
use tokio::sync::mpsc::{self, Receiver};

async fn ping_handler(mut input: Receiver<()>) {
    let mut count: usize = 0;

    while let Some(_) = input.recv().await {
        count += 1;
        println!("지금까지 핑 {count}개를 받았습니다.");
    }

    println!("ping_handler 완료");
}

#[tokio::main]
async fn main() {
    let (sender, receiver) = mpsc::channel(32);
    let ping_handler_task = tokio::spawn(ping_handler(receiver));
    for i in 0..10 {
        sender.send(()).await.expect("핑을 보내지 못했습니다.");
        println!("지금까지 핑 {}개를 전송했습니다.", i + 1);
    }

    drop(sender);
    ping_handler_task.await.expect("핑 핸들러 작업에 문제가 발생했습니다.");
}
```

- 채널 크기를 3 으로 변경하고 동작이 어떻게 바뀌는지 확인하세요.
- 비동기 채널을 사용하기 위한 인터페이스는오전과정에서 배운 `sync` 채널과 비슷합니다.
- `std::mem::drop` 호출하는 줄을 삭제해 보세요. 어떤 결과가나타나나요? 이유가 무엇인가요?
- Flume 크레이트에는 `sync` 와 `async`, `send` 와 `recv` 를 모두구현하는 채널이 있습니다. 이것은 IO 와 CPU 처리 작업이 많은 복잡한애플리케이션을 구현할 때 매우 유용합니다.
- `async` 채널을 사용하는 것이 더 좋은 이유는 이를 다른 `future` 와 결합하여 복잡한 제어 흐름을 만들 수 있기 때문입니다.

제 64 장

Futures Control Flow

Future 들을 결합하여 계산 과정을 동시성이 있는 플로우 그래프 형태로모델링 할 수 있습니다. 앞서 배운, 각 태스크가 독립적으로 수행되도록 하는것도 Future 들을 결합하는 한 방법으로 볼 수 있습니다.

- Join
- Select

64.1 Join

Join 연산은 모든 future 가 준비될 때까지 기다린 후, 각 future 의 결과값을담은 컬렉션을 리턴합니다. 이는 자바스크립트의 `Promise.all` 이나파이썬의 `asyncio.gather` 와 유사합니다.

```
use anyhow::Result;
use futures::future;
use reqwest;
use std::collections::HashMap;

async fn size_of_page(url: &str) -> Result<usize> {
    let resp = reqwest::get(url).await?;
    Ok(resp.text().await?.len())
}

#[tokio::main]
async fn main() {
    let urls: [&str; 4] = [
        "https://google.com",
        "https://httpbin.org/ip",
        "https://play.rust-lang.org/",
        "BAD_URL",
    ];
    let futures_iter = urls.into_iter().map(size_of_page);
    let results = future::join_all(futures_iter).await;
    let page_sizes_dict: HashMap<&str, Result<usize>> =
        urls.into_iter().zip(results.into_iter()).collect();
    println!("{:?}", page_sizes_dict);
}
```

이 예제를, 로컬 컴퓨터에 만들어 둔 `src/main.rs` 에 복사하고거기에서 실행하세요.

- 서로 다른 타입을 가지는 여러 여러 futures 들을 join 하고자 할 경우 `std::future::join!` 을 사용할 수 있습니다. 이 매크로를사용하려면 futures 가 몇 개나 있을지 컴파일 할 때 알아야 한다는 점을주의하세요. 이 매크로는 지금은 `futures` 크레이트에 있으며 곧안정화 되어 `std::future` 에 포함될 예정입니다.
- The risk of `join` is that one of the futures may never resolve, this would cause your program to stall.
- `join_all` 을 `join!`과 결합하여 http 서비스와데이터베이스에 대한 모든 요청들을 한꺼번에 진행시킬 수도 있습니다. `futures::join!`을 사용하여 `tokio::time::sleep` 을 future 에 추가해 보세요. 이건 타임아웃을 구현하는 것이 아님을주의하세요. 실제로, 타임아웃은 다음 장에서 설명하는 `select!`를사용해서 구현해야 합니다. 여기서는 `tokio::time::sleep` 을사용한 것은 단순히 `join!`의 동작을 설명하기 위함입니다.

64.2 Select

Select 연산은 여러 future 들 모두에 대해서 준비될 때 까지 기다리다가, 그중 어떤 한 future 가 최초로 준비 상태가 되면 해당 future 의 결과값을리턴합니다. 이것은 자바스크립트에서의 `Promise.race` 와 비슷합니다. 파이썬에서라면 `asyncio.wait(task_set, return_when=asyncio.FIRST_COMPLETED)` 가하는 동작과 비슷합니다.

Similar to a match statement, the body of `select!` has a number of arms, each of the form `pattern = future => statement`. When a `future` is ready, its return value is destructured by the `pattern`. The `statement` is then run with the resulting variables. The `statement` result becomes the result of the `select!` macro.

```
use tokio::sync::mpsc::{self, Receiver};
use tokio::time::{sleep, Duration};

#[derive(Debug, PartialEq)]
enum Animal {
    Cat { name: String },
    Dog { name: String },
}

async fn first_animal_to_finish_race(
    mut cat_rcv: Receiver<String>,
    mut dog_rcv: Receiver<String>,
) -> Option<Animal> {
    tokio::select! {
        cat_name = cat_rcv.recv() => Some(Animal::Cat { name: cat_name? }),
        dog_name = dog_rcv.recv() => Some(Animal::Dog { name: dog_name? })
    }
}

#[tokio::main]
async fn main() {
    let (cat_sender, cat_receiver) = mpsc::channel(32);
    let (dog_sender, dog_receiver) = mpsc::channel(32);
    tokio::spawn(async move {
        sleep(Duration::from_millis(500)).await;
```

```
        cat_sender.send(String::from("펠릭스")).await.expect("고양이를 보내지 못했습니다.");
    });
    tokio::spawn(async move {
        sleep(Duration::from_millis(50)).await;
        dog_sender.send(String::from("렉스")).await.expect("개를 보내지 못했습니다.");
    });

    let winner = first_animal_to_finish_race(cat_receiver, dog_receiver)
        .await
        .expect("우승자를 수신하지 못했습니다.");

    println!("우승자: {winner:?}");
}
```

- In this example, we have a race between a cat and a dog. `first_animal_to_finish_race` listens to both channels and will pick whichever arrives first. Since the dog takes 50ms, it wins against the cat that take 500ms.
- You can use `oneshot` channels in this example as the channels are supposed to receive only one `send`.
- Try adding a deadline to the race, demonstrating selecting different sorts of futures.
- Note that `select!` drops unmatched branches, which cancels their futures. It is easiest to use when every execution of `select!` creates new futures.
  - 대안은 future 자체 대신 `&mut future` 를 전달하는 것입니다. 하지만 이렇게 하면 문제가 발생할 수 있습니다 (Pinning 을 다룰 때자세히 설명할 예정임).

제 65 장

async/await에서 주의해야할함정

Async 와 await 는 동시 비동기 프로그래밍을 위한 편리하고 효율적인 추상화를제공합니다. 하지만 Rust 의 async/await 모델에도 문제는 있습니다. 이장에서 몇 가지 예를 살펴보겠습니다.

- 실행자차단
- Pin
- 비동기트레잇
- 취소

65.1 실행자 (executor) 를블록시킴

대부분의 비동기 런타임은 IO 작업만 동시에 실행되도록 허용합니다. 즉, CPU 를 블럭하는 태스크가 있는 경우, 이는 실행자 (executor) 를 블럭하게되며, 그 결과로 다른 태스크가 실행되지 않습니다. 이 문제를 해결하는간단한 방법은, 항상 async 를 지원하는 메서드를 사용하는 것입니다.

```
use futures::future::join_all;
use std::time::Instant;

async fn sleep_ms(start: &Instant, id: u64, duration_ms: u64) {
    std::thread::sleep(std::time::Duration::from_millis(duration_ms));
    println!(
        "future {id}은 (는) {duration_ms}밀리초 동안 절전 모드였고 {}밀리초 후에 완료됨",
        start.elapsed().as_millis()
    );
}

#[tokio::main(flavor = "current_thread")]
async fn main() {
    let start = Instant::now();
    let sleep_futures = (1..=10).map(|t| sleep_ms(&start, t, t * 10));
    join_all(sleep_futures).await;
}
```

- 코드를 실행하여 sleep 들이 동시에 진행되지 않고 순차적으로으로진행되는지 확인하세요.
- `flavor` 를 `"current_thread"` 로 설정하면 모든 태스크가하나의 스레드에서 수행됩니다. 이렇게 하면 문제 상황이 더 분명히드러납니다. 그러나 이 버그는 멀티스레드인 경우에도 여전히 존

재합니다.

- `std::thread::sleep` 을 `tokio::time::sleep` 으로 바꾸고그 결과를 `await` 해 보세요.
- 또 다른 해결 방법은 `tokio::task::spawn_blocking` 입니다. 이는실제 스레드를 생성하고, 그 스레드에 대한 핸들을 future 로 변환함으로써실행자가 블록되는 것을 막습니다.
- 태스크를 OS 스레드라고 생각하면 안 됩니다. 태스크와 OS 스레드는 일대일매핑 관계에 있지 않습니다. 대부분의 실행자는 하나의 OS 스레드에서최대한 많은 태스크를 수행하도록 설계되어 있습니다. 이점은 FFI 를 통해다른 라이브러리와 상호작용할 때 특히 문제가 됩니다. 예를 들어, 해당라이브러리가 스레드 로컬 저장소를 이용하거나 특정 OS 스레드에 매핑될수 있습니다 (예: CUDA). 이러한 상황에서는 `tokio::task::spawn_blocking` 을 사용하는 것이 좋습니다.
- 동기화 뮤텍스를 주의해서 사용하세요. `.await` 위에 뮤텍스를적용하면 다른 작업이 차단될 수 있으며 해당 작업은 동일한 스레드에서실행 중일 수 있습니다.

65.2 Pin

Async blocks and functions return types implementing the `Future` trait. The type returned is the result of a compiler transformation which turns local variables into data stored inside the future.

Some of those variables can hold pointers to other local variables. Because of that, the future should never be moved to a different memory location, as it would invalidate those pointers.

To prevent moving the future type in memory, it can only be polled through a pinned pointer. `Pin` is a wrapper around a reference that disallows all operations that would move the instance it points to into a different memory location.

```
use tokio::sync::{mpsc, oneshot};
use tokio::task::spawn;
use tokio::time::{sleep, Duration};

// 작업 항목. 이 경우 지정된 시간 동안 절전 모드이고
// `respond_on` 채널의 메시지로 응답합니다.
#[derive(Debug)]
struct Work {
    input: u32,
    respond_on: oneshot::Sender<u32>,
}

// 큐에서 작업을 수신 대기하고 실행하는 worker 입니다.
async fn worker(mut work_queue: mpsc::Receiver<Work>) {
    let mut iterations = 0;
    loop {
        tokio::select! {
            Some(work) = work_queue.recv() => {
                sleep(Duration::from_millis(10)).await; // 작업하는 척합니다.
                work.respond_on
                    .send(work.input * 1000)
                    .expect("응답을 보내지 못했습니다.");
                iterations += 1;
            }
            // TODO: 100 밀리초마다 반복 횟수를 보고합니다.
```

```
        }
    }
}

// 작업을 요청하고 작업이 완료되기를 기다리는 요청자입니다.
async fn do_work(work_queue: &mpsc::Sender<Work>, input: u32) -> u32 {
    let (tx, rx) = oneshot::channel();
    work_queue
        .send(Work { input, respond_on: tx })
        .await
        .expect("작업 큐에서 전송하지 못했습니다.");
    rx.await.expect("응답 대기 실패")
}

#[tokio::main]
async fn main() {
    let (tx, rx) = mpsc::channel(10);
    spawn(worker(rx));
    for i in 0..100 {
        let resp = do_work(&tx, i).await;
        println!("반복 작업 결과 {i}: {resp}");
    }
}
```

- 위에서 소개한 것은 액터 (actor) 패턴의 한 예라고 봐도 무방합니다. 액터는 일반적으로 루프 안에서 `select!`를 호출합니다.
- 이전 강의 몇 개의 내용을 요약한 것이기 때문에 천천히 살펴보세요.
  - `_ = sleep(Duration::from_millis(100)) => { println! (..) }`을 `select!`에 추가해 보세요. 이 작업은 실행되지 않습니다. 왜그럴까요?
  - 대신, 해당 future 가 포함된 `timeout_fut` 를 loop 외부에 추가해 보세요.

    ```
    let mut timeout_fut = sleep(Duration::from_millis(100));
    loop {
        select! {
            ..,
            _ = timeout_fut => { println!(..); },
        }
    }
    ```

  - 여전히 작동하지 않습니다. 컴파일러 오류를 따라 `select!`의 `timeout_fut` 에 `&mut` 를 추가하여 Move 시멘틱 관련한문제를 해결하고 `Box::pin` 을 사용하세요.

    ```
    let mut timeout_fut = Box::pin(sleep(Duration::from_millis(100)));
    loop {
        select! {
            ..,
            _ = &mut timeout_fut => { println!(..); },
        }
    }
    ```

  - 이는 컴파일은 되지만 타임 아웃이 되면 매번 반복할 때 마다 `Poll::Ready` 가 됩니다 (융합된 future 가 도움이 될 수 있음). 타임 아웃 될 때마다 `timeout_fut` 를 리셋하도록 수정하세요.

- Box 는 힙에 할당합니다. 경우에 따라 `std::pin::pin!`(최근에야안정화되었으며 이전 코드는 `tokio::pin!`을 사용하는 경우가많음) 도 사용할 수 있지만 이는 재할당된 future 에 사용하기가 어렵습니다.
- 또 다른 방법은 `pin` 을 아예 사용하지 않고 100ms 마다 `oneshot` 채널에 전송할 다른 작업을 생성하는 것입니다.
- Data that contains pointers to itself is called self-referential. Normally, the Rust borrow checker would prevent self-referential data from being moved, as the references cannot outlive the data they point to. However, the code transformation for async blocks and functions is not verified by the borrow checker.
- `Pin` is a wrapper around a reference. An object cannot be moved from its place using a pinned pointer. However, it can still be moved through an unpinned pointer.
- The `poll` method of the `Future` trait uses `Pin<&mut Self>` instead of `&mut Self` to refer to the instance. That's why it can only be called on a pinned pointer.

65.3 비동기트레잇

Async methods in traits are were stabilized only recently, in the 1.75 release. This required support for using return-position `impl Trait` (RPIT) in traits, as the desugaring for `async fn` includes `-> impl Future<Output = ...>`.

However, even with the native support today there are some pitfalls around `async fn` and RPIT in traits:

- Return-position impl Trait captures all in-scope lifetimes (so some patterns of borrowing cannot be expressed)
- Traits whose methods use return-position `impl trait` or `async` are not dyn compatible.

If we do need dyn support, the crate async_trait provides a workaround through a macro, with some caveats:

```
use async_trait::async_trait;
use std::time::Instant;
use tokio::time::{sleep, Duration};

#[async_trait]
trait Sleeper {
    async fn sleep(&self);
}

struct FixedSleeper {
    sleep_ms: u64,
}

#[async_trait]
impl Sleeper for FixedSleeper {
    async fn sleep(&self) {
        sleep(Duration::from_millis(self.sleep_ms)).await;
    }
}
```

```
async fn run_all_sleepers_multiple_times(
    sleepers: Vec<Box<dyn Sleeper>>,
    n_times: usize,
) {
    for _ in 0..n_times {
        println!("모든 수면자를 실행");
        for sleeper in &sleepers {
            let start = Instant::now();
            sleeper.sleep().await;
            println!("{}밀리초 동안 절전 모드", start.elapsed().as_millis());
        }
    }
}

#[tokio::main]
async fn main() {
    let sleepers: Vec<Box<dyn Sleeper>> = vec![
        Box::new(FixedSleeper { sleep_ms: 50 }),
        Box::new(FixedSleeper { sleep_ms: 100 }),
    ];
    run_all_sleepers_multiple_times(sleepers, 5).await;
}
```

- async_trait 은 사용하기 쉽지만 이를 위해 힙에 메모리를할당한다는 점에 유의하세요. 이 힙 할당에는 성능 오버헤드가 있습니다.
- async trait 를 언어 차원에서 지원하는 것과 관련된 문제는매우 전문적인 토픽이며 따라서 이 강의에서 다룰 내용은 아닙니다. 이게시물에 이에 관한 니코 마사키스의 좋은 설명이 있으므로 관심이 있다면참고하세요.
- 임의의 시간 동안 sleep 하는 새로운 sleeper 구조체를 만들어 Vec 에추가해 보세요.

65.4 취소

future 가 누락되면 다시 폴링할 수 없다는 의미입니다. 이를 _취소_라고하며, await 지점에서 발생할 수 있습니다. future 가 취소되더라도시스템이 올바르게 작동할 수 있도록 주의를 기울여야 합니다. 예를 들어교착 상태가 되거나 데이터가 손실되면 안 됩니다.

```
use std::io::{self, ErrorKind};
use std::time::Duration;
use tokio::io::{AsyncReadExt, AsyncWriteExt, DuplexStream};

struct LinesReader {
    stream: DuplexStream,
}

impl LinesReader {
    fn new(stream: DuplexStream) -> Self {
        Self { stream }
    }

    async fn next(&mut self) -> io::Result<Option<String>> {
```

```
        let mut bytes = Vec::new();
        let mut buf = [0];
        while self.stream.read(&mut buf[..]).await? != 0 {
            bytes.push(buf[0]);
            if buf[0] == b'\n' {
                break;
            }
        }
        if bytes.is_empty() {
            return Ok(None);
        }
        let s = String::from_utf8(bytes)
            .map_err(|_| io::Error::new(ErrorKind::InvalidData, "not UTF-8"))?;
        Ok(Some(s))
    }
}

async fn slow_copy(source: String, mut dest: DuplexStream) -> std::io::Result<()> {
    for b in source.bytes() {
        dest.write_u8(b).await?;
        tokio::time::sleep(Duration::from_millis(10)).await
    }
    Ok(())
}

#[tokio::main]
async fn main() -> std::io::Result<()> {
    let (client, server) = tokio::io::duplex(5);
    let handle = tokio::spawn(slow_copy("hi\nthere\n".to_owned(), client));

    let mut lines = LinesReader::new(server);
    let mut interval = tokio::time::interval(Duration::from_millis(60));
    loop {
        tokio::select! {
            _ = interval.tick() => println!("틱!"),
            line = lines.next() => if let Some(l) = line? {
                print!("{}", l)
            } else {
                break
            },
        }
    }
    handle.await.unwrap()?;
    Ok(())
}
```

- 컴파일러는 취소 안전에 도움이 되지 않습니다. API 문서를 읽고 `async fn` 의 상태를 고려해야 합니다.
- `panic` 및 `?`와 달리 취소는 오류 처리가 아닌 일반적인제어 흐름의 일부입니다.
- 이 예에서는 문자열의 일부가 손실됩니다.

  - `tick()` 브랜치가 먼저 완료될 때마다 `next()` 및 `buf` 가 삭제됩니다.
  - 다음과 같이 `buf` 를 구조체의 일부로 만들어 `LinesReader` 가 취소되지 않도록 할 수 있습니다.

    ```
    struct LinesReader {
        stream: DuplexStream,
        bytes: Vec<u8>,
        buf: [u8; 1],
    }

    impl LinesReader {
        fn new(stream: DuplexStream) -> Self {
            Self { stream, bytes: Vec::new(), buf: [0] }
        }
        async fn next(&mut self) -> io::Result<Option<String>> {
            // buf 및 bytes 접두사를 self로 지정합니다.
            // ...
            let raw = std::mem::take(&mut self.bytes);
            let s = String::from_utf8(raw)
            // ...
        }
    }
    ```

- `Interval::tick` 은틱이 'delivered' 됐는지 추적하므로취소에 안전합니다.
- `AsyncReadExt::read` 는데이터를 반환하거나 읽지 않으므로 취소에 안전합니다.
- `AsyncBufReadExt::read_line` 은예와 유사하며 취소에 안전하지 . 자세한 내용과 대안은관련 문서를 참고하세요.

제 66 장

연습문제

Async Rust 기술을 연습할 수 있도록 두 가지 연습문제를 준비했습니다.

- 식사하는 철학자: 이 문제는 이미 오전에 확인했습니다. 이번에는 Async Rust 로 구현합니다.
- 브로드캐스트 채팅 애플리케이션: 더 많은 고급 Async Rust 기능을 실험할수 있는 대규모 프로젝트입니다.

After looking at the exercises, you can look at the solutions provided.

66.1 Dining Philosophers — Async

문제에 관한 설명은식사하는철학자를 참고하세요.

As before, you will need a local Cargo installation for this exercise. Copy the code below to a file called `src/main.rs`, fill out the blanks, and test that `cargo run` does not deadlock:

```
use std::sync::Arc;
use tokio::sync::mpsc::{self, Sender};
use tokio::sync::Mutex;
use tokio::time;

struct Fork;

struct Philosopher {
    name: String,
    // left_fork: ...
    // right_fork: ...
    // thoughts: ...
}

impl Philosopher {
    async fn think(&self) {
        self.thoughts
            .send(format!("유레카! {}에 새로운 아이디어가 있습니다.", &self.name))
            .await
            .unwrap();
    }
```

```
    async fn eat(&self) {
        // Keep trying until we have both forks
        println!("{}님이 먹고 있습니다...", &self.name);
        time::sleep(time::Duration::from_millis(5)).await;
    }
}

static PHILOSOPHERS: &[&str] =
    &["Socrates", "히파티아", "플라톤", "아리스토텔레스", "피타고라스"];

#[tokio::main]
async fn main() {
    // 포크 만들기

    // 철학자 만들기

    // 생각하고 먹게 합니다.

    // 생각을 출력합니다.
}
```

이번에는 Async Rust 를 사용하므로 `tokio` 종속 항목이 필요합니다. 다음 `Cargo.toml` 을 사용할 수 있습니다.

```
[package]
name = "dining-philosophers-async-dine"
version = "0.1.0"
edition = "2021"

[dependencies]
tokio = { version = "1.26.0", features = ["sync", "time", "macros", "rt-multi-thread"] }
```

또한 이번에는 `tokio` 크레이트의 `Mutex` 와 `mpsc` 모듈을 사용해야 합니다.

- Can you make your implementation single-threaded?

66.2 채팅애플리케이션

이 연습에서는 새로운 지식을 사용하여 브로드캐스트 채팅 애플리케이션을구현해 보겠습니다. 클라이언트가 연결하고 메시지를 게시하는 채팅 서버가있습니다. 클라이언트는 표준 입력에서 사용자 메시지를 읽고 서버로전송합니다. 채팅 서버는 수신하는 각 메시지를 모든 클라이언트에브로드캐스트합니다.

For this, we use a broadcast channel on the server, and `tokio_websockets` for the communication between the client and the server.

새 Cargo 프로젝트를 만들고 다음 종속 항목을 추가합니다.

Cargo.toml:

```
[package]
name = "chat-async"
version = "0.1.0"
edition = "2021"
```

```
[dependencies]
futures-util = { version = "0.3.30", features = ["sink"] }
http = "1.1.0"
tokio = { version = "1.36.0", features = ["full"] }
tokio-websockets = { version = "0.7.0", features = ["client", "fastrand", "server", "sha
```

필수 API

You are going to need the following functions from `tokio` and `tokio_websockets`. Spend a few minutes to familiarize yourself with the API.

- StreamExt::next() implemented by `WebSocketStream`: for asynchronously reading messages from a Websocket Stream.
- SinkExt::send() implemented by `WebSocketStream`: for asynchronously sending messages on a Websocket Stream.
- Lines::next_line() 은표준 입력에서 사용자 메시지를 비동기식으로 읽는 데 사용됩니다.
- Sender::subscribe() 는브로드캐스트 채널 구독에 사용됩니다.

Two binaries

Normally in a Cargo project, you can have only one binary, and one `src/main.rs` file. In this project, we need two binaries. One for the client, and one for the server. You could potentially make them two separate Cargo projects, but we are going to put them in a single Cargo project with two binaries. For this to work, the client and the server code should go under `src/bin` (see the documentation).

Copy the following server and client code into `src/bin/server.rs` and `src/bin/client.rs`, respectively. Your task is to complete these files as described below.

src/bin/server.rs:

```
use futures_util::sink::SinkExt;
use futures_util::stream::StreamExt;
use std::error::Error;
use std::net::SocketAddr;
use tokio::net::{TcpListener, TcpStream};
use tokio::sync::broadcast::{channel, Sender};
use tokio_websockets::{Message, ServerBuilder, WebSocketStream};

async fn handle_connection(
    addr: SocketAddr,
    mut ws_stream: WebSocketStream<TcpStream>,
    bcast_tx: Sender<String>,
) -> Result<(), Box<dyn Error + Send + Sync>> {

    // TODO: 힌트는 아래의 작업 설명을 참고하세요.

}

#[tokio::main]
async fn main() -> Result<(), Box<dyn Error + Send + Sync>> {
    let (bcast_tx, _) = channel(16);
```

```
    let listener = TcpListener::bind("127.0.0.1:2000").await?;
    println!("포트 2000 에서 수신 대기");

    loop {
        let (socket, addr) = listener.accept().await?;
        println!("{addr:?}의 새 연결");
        let bcast_tx = bcast_tx.clone();
        tokio::spawn(async move {
            // 원시 TCP 스트림을 websocket 에 래핑합니다.
            let ws_stream = ServerBuilder::new().accept(socket).await?;

            handle_connection(addr, ws_stream, bcast_tx).await
        });
    }
}
```

src/bin/client.rs:

```
use futures_util::stream::StreamExt;
use futures_util::SinkExt;
use http::Uri;
use tokio::io::{AsyncBufReadExt, BufReader};
use tokio_websockets::{ClientBuilder, Message};

#[tokio::main]
async fn main() -> Result<(), tokio_websockets::Error> {
    let (mut ws_stream, _) =
        ClientBuilder::from_uri(Uri::from_static("ws://127.0.0.1:2000"))
            .connect()
            .await?;

    let stdin = tokio::io::stdin();
    let mut stdin = BufReader::new(stdin).lines();


    // TODO: 힌트는 아래의 작업 설명을 참고하세요.

}
```

Running the binaries

Run the server with:

`cargo run --bin server`

and the client with:

`cargo run --bin client`

태스크

- `src/bin/server.rs` 에서 `handle_connection` 함수를구현합니다.

  - 힌트: 연속 루프에서 두 작업을 동시에 실행하는 경우 `tokio::select!`를 사용하세요. 한 작업은 클라이언트에서메시지를 수신하여 브로드캐스트합니다. 다른 하나는 서버가 수신한 메시지를 클라이언트로 보냅니다.
- `src/bin/client.rs` 에서 main 함수를 완료합니다.
  - 힌트: 이전과 마찬가지로 연속 루프에서 두 작업을 동시에 실행하는 경우 `tokio::select!`를 사용하세요. (1) 표준 입력에서 사용자메시지를 읽고 서버로 보냅니다. (2) 서버에서 메시지를 수신하고사용자에게 표시합니다.
- 선택사항: 작업을 완료하면 메시지 발신자를 제외한 모든 클라이언트에게메시지를 브로드캐스트하도록 코드를 변경합니다.

66.3 Concurrency Afternoon Exercise

Dining Philosophers — Async

(연습문제로돌아가기

```
use std::sync::Arc;
use tokio::sync::mpsc::{self, Sender};
use tokio::sync::Mutex;
use tokio::time;

struct Fork;

struct Philosopher {
    name: String,
    left_fork: Arc<Mutex<Fork>>,
    right_fork: Arc<Mutex<Fork>>,
    thoughts: Sender<String>,
}

impl Philosopher {
    async fn think(&self) {
        self.thoughts
            .send(format!("유레카! {}에 새로운 아이디어가 있습니다.", &self.name))
            .await
            .unwrap();
    }

    async fn eat(&self) {
        // Keep trying until we have both forks
        let (_left_fork, _right_fork) = loop {
            // 포크를 드세요...
            let left_fork = self.left_fork.try_lock();
            let right_fork = self.right_fork.try_lock();
            let Ok(left_fork) = left_fork else {
                // If we didn't get the left fork, drop the right fork if we
                // have it and let other tasks make progress.
                drop(right_fork);
                time::sleep(time::Duration::from_millis(1)).await;
                continue;
            };
```

```
            let Ok(right_fork) = right_fork else {
                // If we didn't get the right fork, drop the left fork and let
                // other tasks make progress.
                drop(left_fork);
                time::sleep(time::Duration::from_millis(1)).await;
                continue;
            };
            break (left_fork, right_fork);
        };

        println!("{}님이 먹고 있습니다...", &self.name);
        time::sleep(time::Duration::from_millis(5)).await;

        // 여기에 잠금이 해제됩니다.
    }
}

static PHILOSOPHERS: &[&str] =
    &["Socrates", "히파티아", "플라톤", "아리스토텔레스", "피타고라스"];

#[tokio::main]
async fn main() {
    // 포크 만들기
    let mut forks = vec![];
    (0..PHILOSOPHERS.len()).for_each(|_| forks.push(Arc::new(Mutex::new(Fork))));

    // 철학자 만들기
    let (philosophers, mut rx) = {
        let mut philosophers = vec![];
        let (tx, rx) = mpsc::channel(10);
        for (i, name) in PHILOSOPHERS.iter().enumerate() {
            let left_fork = Arc::clone(&forks[i]);
            let right_fork = Arc::clone(&forks[(i + 1) % PHILOSOPHERS.len()]);
            philosophers.push(Philosopher {
                name: name.to_string(),
                left_fork,
                right_fork,
                thoughts: tx.clone(),
            });
        }
        (philosophers, rx)
        // tx가 여기에서 삭제되므로 나중에 명시적으로 삭제할 필요가 없습니다.
    };

    // 생각하고 먹게 합니다.
    for phil in philosophers {
        tokio::spawn(async move {
            for _ in 0..100 {
                phil.think().await;
                phil.eat().await;
            }
```

```
        });
    }

    // 생각을 출력합니다.
    while let Some(thought) = rx.recv().await {
        println!("의견 보내기: {thought}");
    }
}
```

채팅애플리케이션

(연습문제로돌아가기)

src/bin/server.rs:

```
use futures_util::sink::SinkExt;
use futures_util::stream::StreamExt;
use std::error::Error;
use std::net::SocketAddr;
use tokio::net::{TcpListener, TcpStream};
use tokio::sync::broadcast::{channel, Sender};
use tokio_websockets::{Message, ServerBuilder, WebSocketStream};

async fn handle_connection(
    addr: SocketAddr,
    mut ws_stream: WebSocketStream<TcpStream>,
    bcast_tx: Sender<String>,
) -> Result<(), Box<dyn Error + Send + Sync>> {

    ws_stream
        .send(Message::text("채팅에 오신 것을 환영합니다. 메시지를 입력하세요.".to_string()))
        .await?;
    let mut bcast_rx = bcast_tx.subscribe();

    // 동시에 두 작업을 실행하는 연속 루프: (1) `ws_stream`에서 메시지를 수신하여
    // 브로드캐스팅하고 (2) `bcast_rx`에서 메시지를 수신하여
    // 클라이언트로 전송합니다.
    loop {
        tokio::select! {
            incoming = ws_stream.next() => {
                match incoming {
                    Some(Ok(msg)) => {
                        if let Some(text) = msg.as_text() {
                            println!("클라이언트에서: {addr:?} {text:?}");
                            bcast_tx.send(text.into())?;
                        }
                    }
                    Some(Err(err)) => return Err(err.into()),
                    None => return Ok(()),
                }
            }
            msg = bcast_rx.recv() => {
```

```
                ws_stream.send(Message::text(msg?)).await?;
            }
        }
    }
}

#[tokio::main]
async fn main() -> Result<(), Box<dyn Error + Send + Sync>> {
    let (bcast_tx, _) = channel(16);

    let listener = TcpListener::bind("127.0.0.1:2000").await?;
    println!("포트 2000 에서 수신 대기");

    loop {
        let (socket, addr) = listener.accept().await?;
        println!("{addr:?}의 새 연결");
        let bcast_tx = bcast_tx.clone();
        tokio::spawn(async move {
            // 원시 TCP 스트림을 websocket 에 래핑합니다.
            let ws_stream = ServerBuilder::new().accept(socket).await?;

            handle_connection(addr, ws_stream, bcast_tx).await
        });
    }
}
```

src/bin/client.rs:

```
use futures_util::stream::StreamExt;
use futures_util::SinkExt;
use http::Uri;
use tokio::io::{AsyncBufReadExt, BufReader};
use tokio_websockets::{ClientBuilder, Message};

#[tokio::main]
async fn main() -> Result<(), tokio_websockets::Error> {
    let (mut ws_stream, _) =
        ClientBuilder::from_uri(Uri::from_static("ws://127.0.0.1:2000"))
            .connect()
            .await?;

    let stdin = tokio::io::stdin();
    let mut stdin = BufReader::new(stdin).lines();

    // 동시에 메시지를 보내고 받는 연속 루프
    loop {
        tokio::select! {
            incoming = ws_stream.next() => {
                match incoming {
                    Some(Ok(msg)) => {
                        if let Some(text) = msg.as_text() {
                            println!("서버에서: {}", text);
```

```
                        }
                    },
                    Some(Err(err)) => return Err(err.into()),
                    None => return Ok(()),
                }
            }
            res = stdin.next_line() => {
                match res {
                    Ok(None) => return Ok(()),
                    Ok(Some(line)) => ws_stream.send(Message::text(line.to_string())).au
                    Err(err) => return Err(err.into()),
                }
            }

        }
    }
}
```

제 XV 편

끝으로...

제 67 장

감사인사

Comprehensive Rust 🦀 를 이용해 주셔서 감사합니다. 즐겁고 유익한시간이었기를 바랍니다.
강의가 완벽하진 않으니 실수나 개선점이 있다면 언제든지깃허브로연락주세요.

제 68 장

용어집

다음은 여러 Rust 용어의 간단한 정의를 제공하는 용어집입니다. 번역의 경우용어를 다시 영어 원본에 연결하는 역할도 합니다.

- 할당:
  힙에대한 동적 메모리 할당입니다.
- 인수:
  함수나 메서드에 전달되는 정보입니다.
- Bare-metal Rust:
  낮은 수준의 Rust 개발로, 운영체제가 없는 시스템에 배포되는 경우가많습니다. Bare-metal Rust 를참고하세요.
- 블록:
  블록및 _범위_를 참고하세요.
- 빌림:
  빌림을참고하세요.
- 빌림 검사기:
  모든 빌림이 유효한지 확인하는 Rust 컴파일러의 부분입니다.
- 괄호:
  { and }. _중괄호_라고도 하며 _블록_을구분합니다.
- 빌드:
  소스 코드를 실행 가능한 코드 또는 사용 가능한 프로그램으로 변환하는프로세스입니다.
- 호출:
  함수 또는 메서드를 호출하거나 실행합니다.
- 채널:
  스레드간에 메시지를 안전하게 전달하는 데 사용됩니다.
- Comprehensive Rust 🦀:
  이 과정은 Comprehensive Rust 🦀 로 통칭됩니다.
- 동시 실행:
  여러 작업 또는 프로세스를 동시에 실행합니다.
- Rust 의 동시 실행:
  Rust 의 동시실행을 참고하세요.
- 상수:
  프로그램 실행 중에 변경되지 않는 값입니다.
- 제어 흐름:
  프로그램에서 개별 문 또는 명령이 실행되는 순서입니다.
- 비정상 종료:

프로그램의 예기치 않거나 처리되지 않은 오류 또는 종료입니다.
- enumeration:
  A data type that holds one of several named constants, possibly with an associated tuple or struct.
- 오류:
  예상 동작을 벗어나는 예기치 못한 상태나 결과입니다.
- 오류 처리:
  프로그램 실행 중에 발생하는 오류를 관리하고 이에 대응하는프로세스입니다.
- 연습:
  프로그래밍 기술을 연습하고 테스트하기 위한 과제 또는 문제입니다.
- 함수:
  특정 작업을 실행하는 재사용 가능한 코드 블록입니다.
- 가비지 컬렉터:
  더 이상 사용되지 않는 객체가 차지하는 메모리를 자동으로 해제하는메커니즘입니다.
- 제네릭:
  타입에 관한 자리표시자를 사용하여 코드를 작성할 수 있는 기능으로, 다양한 데이터 타입으로 코드를 재사용할 수 있습니다.
- 변경 불가능:
  생성 후에는 변경할 수 없습니다.
- 통합 테스트:
  시스템의 여러 부분 또는 구성요소 간의 상호작용을 확인하는 테스트타입입니다.
- 키워드:
  프로그래밍 언어에서 특정 의미를 가지며 식별자로 사용될 수 없는예약어입니다.
- 라이브러리:
  프로그램에서 사용할 수 있는 사전 컴파일된 루틴 또는 코드 모음입니다.
- 매크로:
  Rust 매크로는 이름의 `!`로 인식될 수 있습니다. 매크로는 일반함수가 충분하지 않을 때 사용됩니다. 일반적인 예로는 가변적인 인수 수를사용하는 `format!`이 있는데, 이는 Rust 함수에서 지원되지않습니다.
- `main` 함수:
  Rust 프로그램은 `main` 함수로 실행을 시작합니다.
- 일치:
  표현식 값에 대한 패턴 일치를 허용하는 Rust 의 제어 흐름 구성입니다.
- 메모리 누수:
  프로그램이 더 이상 필요하지 않은 메모리를 해제하지 못해 메모리사용량이 점차 늘어나는 상황입니다.
- 메서드:
  Rust 의 객체나 타입과 관련된 함수입니다.
- 모듈:
  Rust 에서 코드를 구성하기 위해 함수, 타입 또는 트레잇과 같은 정의가포함된 네임스페이스입니다.
- 이동:
  Rust 에서 한 변수에서 다른 변수로 값의 소유권을 이전하는 것입니다.
- mutable:
  선언된 후 변수를 수정할 수 있는 Rust 의 속성입니다.
- 소유권:
  값과 관련된 메모리를 관리하는 코드의 부분을 정의하는 Rust 의개념입니다.
- 패닉:
  Rust 에서 복구할 수 없는 오류 상태로, 프로그램이 종료됩니다.
- 매개변수:
  호출 시 함수나 메서드로 전달되는 값입니다.

- 패턴:
  Rust 의 표현식과 일치시킬 수 있는 값, 리터럴 또는 구조의 조합입니다.
- 페이로드:
  메시지, 이벤트 또는 데이터 구조에 의해 전달되는 데이터 또는정보입니다.
- 프로그램:
  컴퓨터가 특정 작업을 수행하거나 특정 문제를 해결하기 위해 실행할 수있는 일련의 명령입니다.
- 프로그래밍 언어: 컴퓨터에 명령을 전달하는 데 사용되는 공식시스템입니다 (예: Rust).
- 수신자:
  메서드가 호출되는 인스턴스를 나타내는 Rust 메서드의 첫 번째매개변수입니다.
- 참조 계산:
  객체에 대한 참조 수를 추적하고 개수가 0 에 도달하면 객체의 할당을해제하는 메모리 관리 기법입니다.
- return:
  함수에서 반환될 값을 나타내는 데 사용되는 Rust 의 키워드입니다.
- Rust:
  안전, 성능, 동시 실행에 중점을 둔 시스템 프로그래밍 언어입니다.
- Rust Fundamentals:
  Days 1 to 4 of this course.
- Android 의 Rust:
  Android 의 Rust 를참고하세요.
- Chromium 의 Rust:
  Chromium 의 Rust 를참고하세요.
- 안전:
  Rust 의 소유권 및 빌림 규칙을 준수하여 메모리 관련 오류를 방지하는코드를 나타냅니다.
- 범위:
  변수가 유효하여 사용할 수 있는 프로그램의 영역입니다.
- 표준 라이브러리:
  Rust 에서 필수 기능을 제공하는 모듈 모음입니다.
- static:
  Rust 에서 `'static` 전체 기간으로 정적 변수또는 항목을 정의하는 데 사용되는 키워드입니다.
- 문자열:
  텍스트 데이터를 저장하는 데이터 타입입니다. 자세한 내용은 `String` 및 `str` 을 참고하세요.
- 구조체:
  다양한 타입의 변수를 단일 이름으로 그룹화하는 Rust 의 복합 데이터타입입니다.
- test:
  다른 함수의 정확성을 테스트하는 함수가 포함된 Rust 모듈입니다.
- 스레드:
  프로그램의 별도 실행 시퀀스로, 동시 실행을 허용합니다.
- 스레드 안전:
  다중 스레드 환경에서 올바른 동작을 보장하는 프로그램의 속성입니다.
- 트레잇:
  알 수 없는 타입에 관해 정의된 메서드 모음으로, Rust 에서 다형성을달성하는 방법을 제공합니다.
- trait bound:
  An abstraction where you can require types to implement some traits of your interest.
- tuple:
  A composite data type that contains variables of different types. Tuple fields have no names, and are accessed by their ordinal numbers.
- 타입:
  Rust 에서 특정 종류의 값에 대해 어떤 작업을 실행할 수 있는지 지정하는분류입니다.
- 타입 추론:

변수나 표현식의 타입을 추론하는 Rust 컴파일러의 기능입니다.

- 정의되지 않은 동작:
  지정된 결과가 없는 Rust 의 작업 또는 조건으로, 종종 예측할 수 없는프로그램 동작을 초래합니다.
- union:
  한 번에 하나씩만 여러 타입의 값을 보유할 수 있는 데이터 타입입니다.
- 단위 테스트:
  Rust 에는 작은 단위 테스트와 대규모 통합 테스트를 실행할 수 있는 지원기능이 내장되어 있습니다. 단위테스트를 참고하세요.
- unit type:
  Type that holds no data, written as a tuple with no members.
- 안전하지 않음:
  _정의되지 않은 동작_을 트리거할 수 있는 Rust 의 하위 집합입니다. 안전하지않은 Rust 를 참고하세요.
- variable:
  A memory location storing data. Variables are valid in a *scope*.

제 69 장

러스트 참고자료

러스트 커뮤니티는 온라인에서 고품질의 무료 소스를 만들었습니다.

공식문서들

러스트 프로젝트에는 참조할 만한 자료가 많습니다. 일반적인 내용을 다루는몇가지 참고 문서들입니다:

- The Rust Programming Language: 러스트에 대한 무료 표준 서적입니다. 언어에 대한 자세한 설명과 사람들이빌드 할수 있는 몇가지 프로젝트를 포함합니다.
- Rust By Example: 여러 예제를 통해 러스트의 문법을 보여주며 때때로 코드를 확장하는약간의 연습문제들이 포함되어 있습니다.
- Rust Standard Library: 러스트표준 라이브러리 전체 문서입니다.
- The Rust Reference: 메모리 모델링과 러스트 문법을 설명하는 문서입니다.(아직 불완전하다함)

좀 더 전문적인 공식 가이드입니다:

- The Rustonomicon: 안전하지않은 러스트, FFI, raw 포인터 작업을 다룹니다.
- Asynchronous Programming in Rust: 러스트 북이 작성 된 이후 도입된 새로운 비동기프로그래밍 모델을 다룹니다.
- The Embedded Rust Book: 운영체제가 없는 임베디드 장치에서의 러스트 사용법을소개합니다.

비공식적 학습자료

러스트에 대한 기타 안내서와 튜토리얼의 일부입니다:

- Learn Rust the Dangerous Way: C 언어 프로그래머 관점에서 러스트를 다룹니다.
- Rust for Embedded C Programmers: 임베디드 C 개발자 (펌웨어 개발자) 를 위한 러스트가이드입니다.
- Rust for professionals: 다른 언어 (C/C++, Java, Python, Javascript) 와의병렬비교를 사용하여 러스트 문법을 다룹니다.
- Rust on Exercism: 러스트를배우는데 도움이 되는 100 개 이상의 연습문제
- Ferrous Teaching Material: 러스트 언어의 기본부터 고급을 전부 다루는 일련의작은 프레젠테이션, 웹 어셈블리, async/await 같은 부분도 함께 다룹니다.
- Beginner's Series to Rust, Take your first steps with Rust: 첫번째는 35 개의 시리즈 영상이며 두번째는러스트의 문법과 구조를 다루는 11 개의 모듈 세트입니다.

- Learn Rust With Entirely Too Many Linked Lists: 몇가지 유형의 리스트 자료구조를구현해 보면서 러스트의 메모리 관리 규칙들을 깊이있게 탐색합니다.

Little Book of Rust Books 에서 더많은 러스트 북을 확인해보세요.

제 70 장

도와주신분들

이 자료는 많은 훌륭한 러스트 문서들의 도움을 받아 작성되었습니다. 유용한자료의 전체 목록은 other resources 에서 살펴보시기 바랍니다.

Rust by Example

일부 예제와 연습문제는 Rust by Example 을참조하였습니다. 라이선스 조항을 포함하여 저장소의 `third_party/rust-by-example/` 폴더를 참조하시기 바랍니다.

Rust on Exercism

일부 연습문제는 Rust on Exercism 을 참조하였습니다. 라이선스 조항을 포함하여 저장소의 `third_party/rust-on-exercism/`폴더를 참조하시기 바랍니다.

CXX

4 일차 오후 강의 중 Interoperability with C++에서는 CXX 의 이미지를 사용하였습니다. 라이선스 조항을 포함하여 저장소의 `third_party/cxx/`폴더를참조하시기 바랍니다.