

HSPACE Rust 특강

Rust Basic #5 - 크레이트와 모듈

Chris Ohk

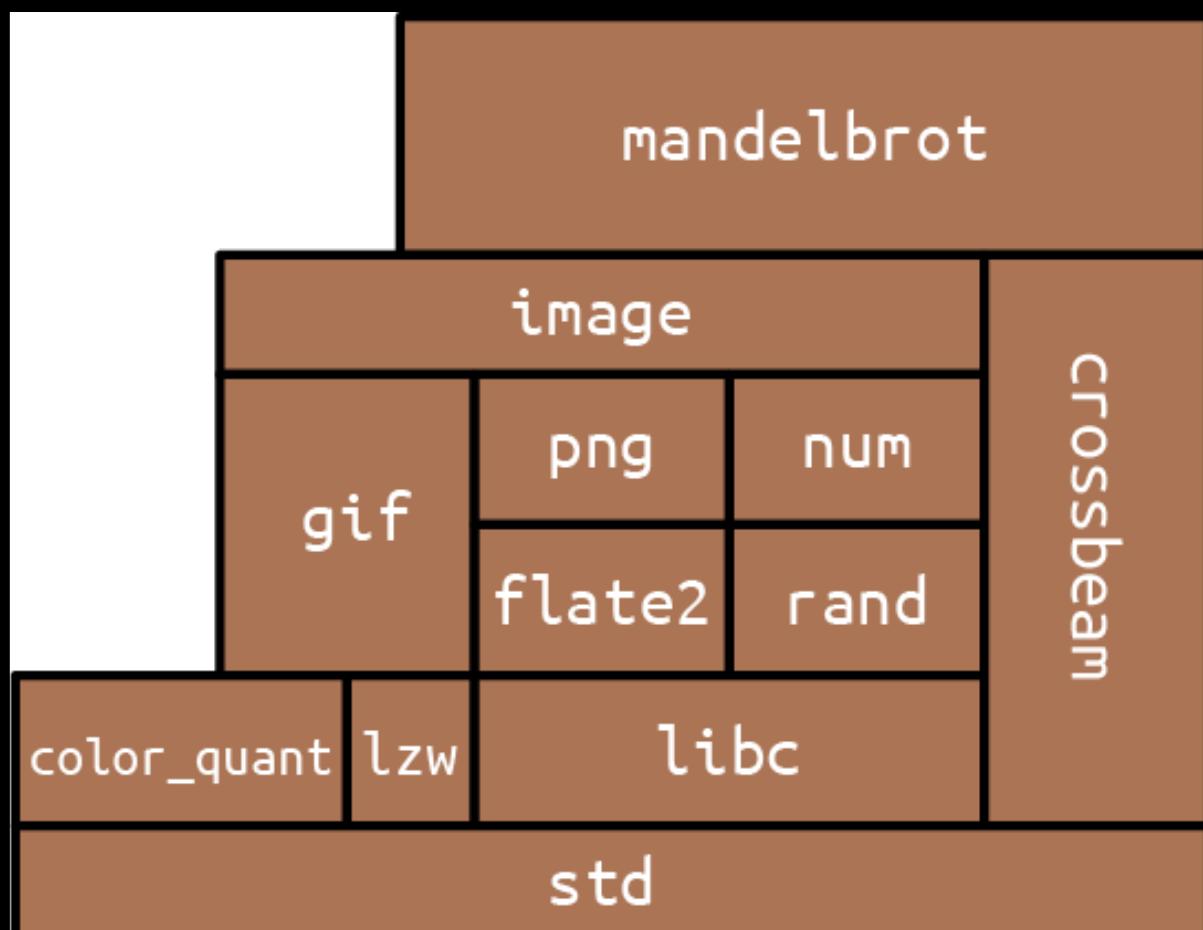
utilForever@gmail.com

크레이트와 모듈

HSPACE Rust 특강
Rust Basic #5 - 크레이트와 모듈

- 크레이트 (Crate)
- 모듈 (Module)
- 프로그램을 라이브러리로 바꾸기
- src/bin 디렉터리
- 어트리뷰트 (Attribute)
- 테스트와 문서화
- 의존성 지정하기
- crates.io에 크레이트 게시하기
- 워크스페이스 (Workspace)

- Rust 프로그램은 **크레이트(Crate)**로 구성된다.
 - 각 크레이트는 하나의 완전하고 응집력 있는 단위다.
 - 즉, 단일 라이브러리나 실행 파일의 모든 소스 코드를 비롯해 그와 연관된 테스트, 예제, 도구, 구성 등 여러 가지 것들이 독자적인 프로젝트를 이룬다는 뜻이다.



- Rust 프로그램은 **크레이트(Crate)**로 구성된다.
 - 프로그램을 만들 때 `main.rs`에 다른 크레이트에 있는 아이템을 사용하기 위한 여러 개의 `use` 선언문이 포함되어 있는 경우를 어렵지 않게 볼 수 있다.



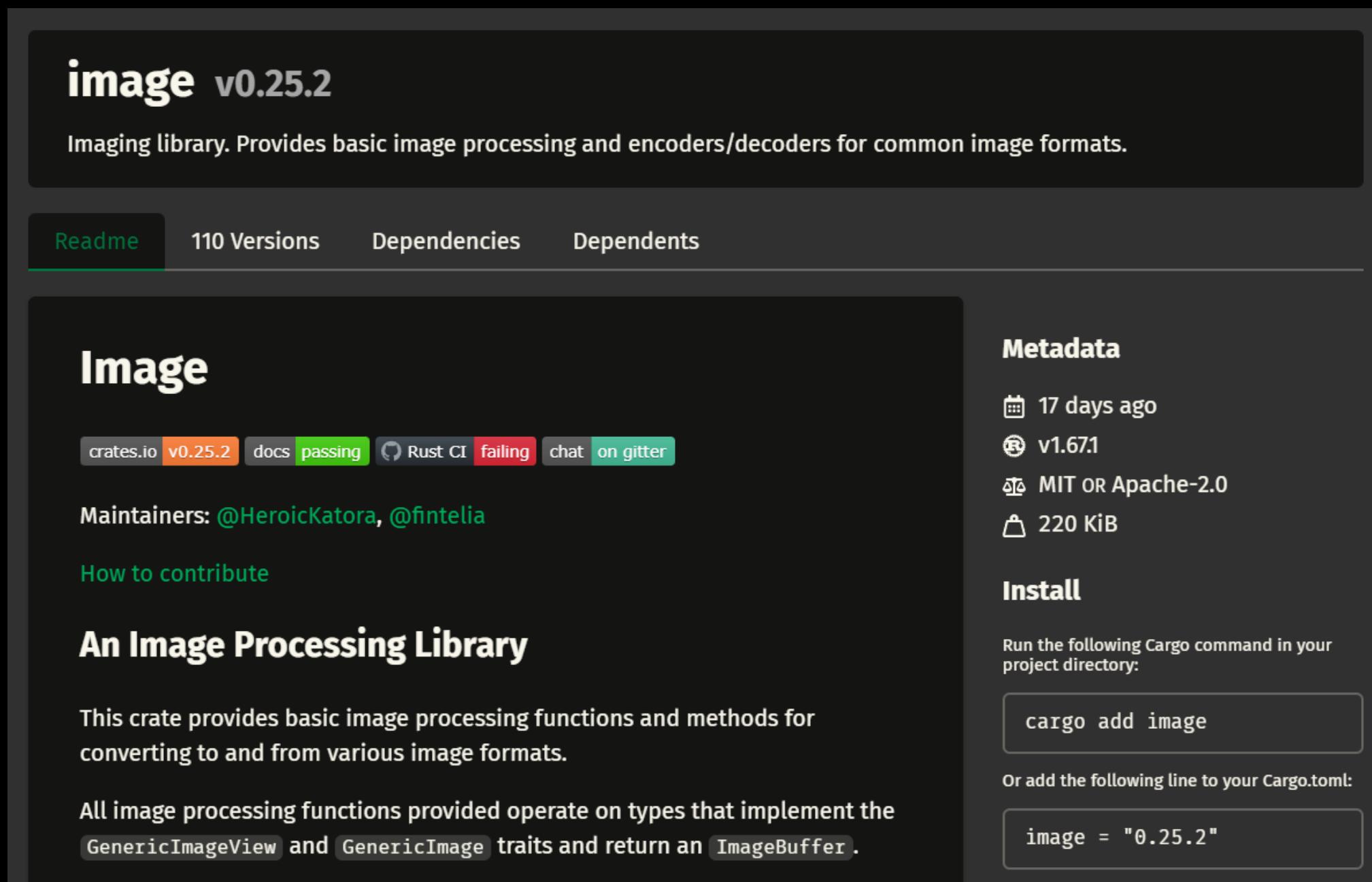
```
use num::Complex;
// ...
use image::ColorType;
use image::png::PNGEncoder;
```

- 이 크레이트들은 `Cargo.toml` 파일에 원하는 버전을 지정해 가져온다.



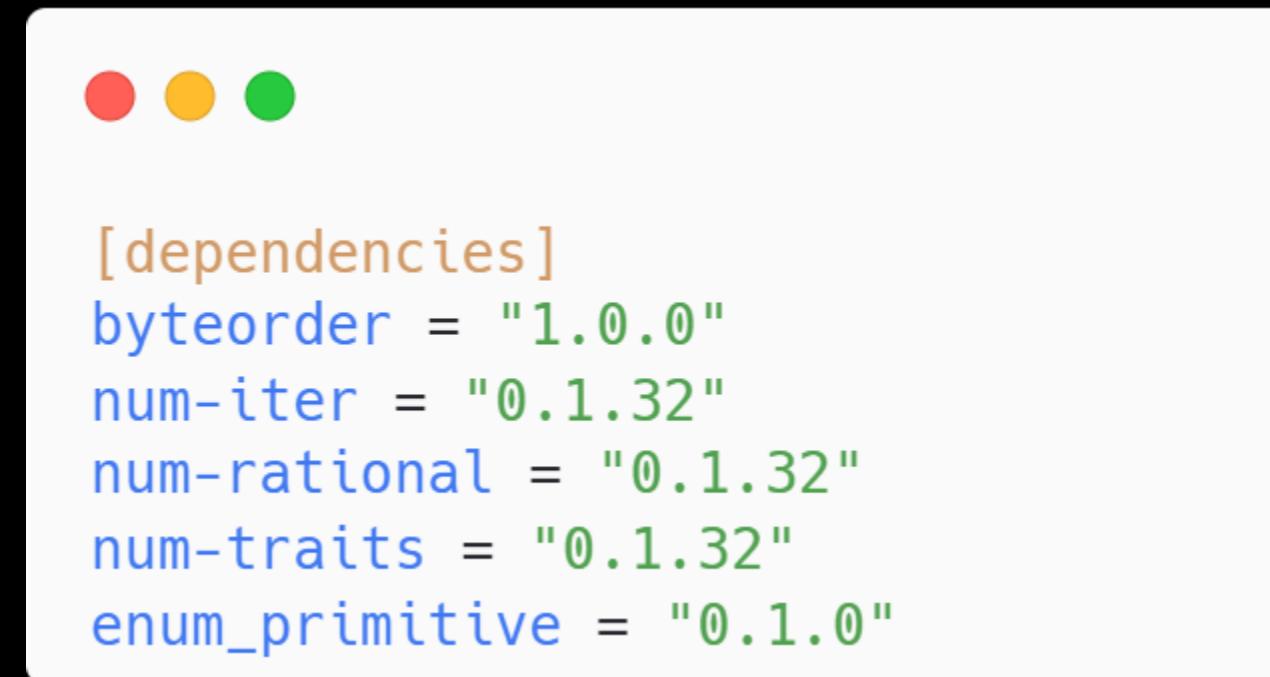
```
[dependencies]
num = "0.4"
image = "0.13"
crossbeam = "0.8"
```

- Rust 프로그램은 **크레이트(Crate)**로 구성된다.
 - 여기서 `dependencies`라는 키워드는 이 프로젝트가 사용하는 다른 크레이트들, 즉 의존하고 있는 코드를 뜻한다. 앞서 나온 크레이트들은 모두 Rust 커뮤니티의 오픈 소스 크레이트 사이트인 crates.io에서 찾은 것들이다.
 - crates.io의 각 크레이트 페이지는 해당 크레이트의 `README.md` 파일을 보여 주고, 문서와 소스에 대한 링크와 더불어 `image = "0.13"`과 같이 `Cargo.toml`에 복사해 넣을 수 있는 설정 코드를 제공한다.



- Rust 프로그램은 **크레이트(Crate)**로 구성된다.

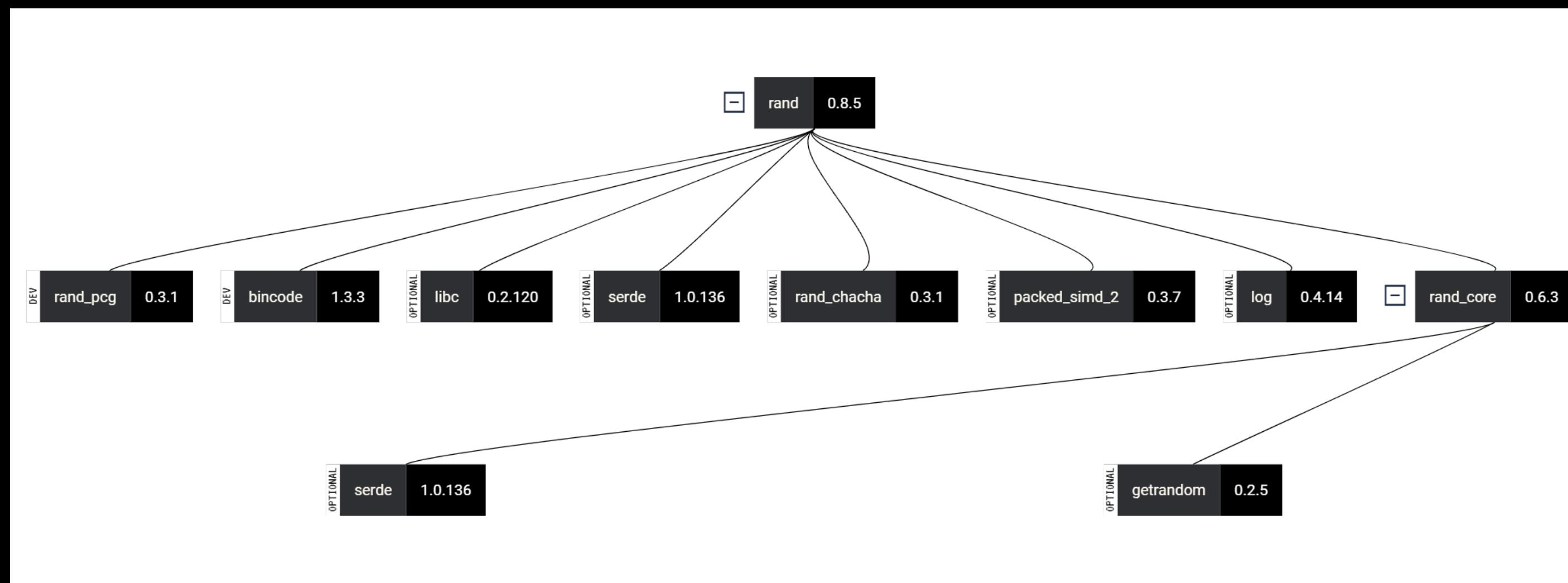
- Cargo로 빌드할 때 출력되는 내용을 살펴보면 이러한 정보가 어떻게 쓰이는지 알 수 있다.
- cargo build를 실행하면 Cargo는 먼저 지정된 버전의 크레이트 소스 코드를 crates.io에서 내려받는다. 그런 다음 해당 크레이트의 Cargo.toml 파일을 읽고 그가 가진 의존성을 내려받고, 이 과정을 재귀적으로 반복한다. 예를 들어, image 크레이트 버전의 0.13.0의 소스 코드에 있는 Cargo.toml 파일은 아래의 내용을 담고 있다.



```
[dependencies]
byteorder = "1.0.0"
num-iter = "0.1.32"
num-rational = "0.1.32"
num-traits = "0.1.32"
enum_primitive = "0.1.0"
```

- Cargo는 위 내용을 보고 image를 사용하기 위해서는 먼저 이 크레이트들을 가져와야 한다는 걸 알게 된다. Cargo가 소스 코드를 crates.io 말고 Git 저장소나 로컬 파일 시스템에서 가져오게 하는 방법도 있다.

- Rust 프로그램은 크레이트(Crate)로 구성된다.
 - 우리가 만드는 크레이트의 이름이 hspace라고 할 때, 이 크레이트들은 hspace가 image 크레이트를 통해 간접적으로 의존한다고 해서 hspace의 추이적인 의존성(Transitive Dependency)이라고 한다.
 - 이러한 의존성 관계들의 모음을 통틀어서 크레이트의 의존성 그래프(Dependency Graph)라고 하는데, Cargo는 이 의존성 그래프를 통해서 무슨 크레이트를 어떤 순서로 빌드해야 하는지 파악한다.
 - Cargo가 이를 알아서 처리해주기 때문에 프로그래머들의 시간과 노력이 크게 절약된다.



- Rust 프로그램은 **크레이트(Crate)**로 구성된다.

- 모든 크레이트의 소스 코드가 확보되고 나면 컴파일이 시작된다.

Cargo는 프로젝트의 의존성 그래프를 따라 각 크레이트에 대해 Rust 컴파일러인 `rustc`를 실행한다.

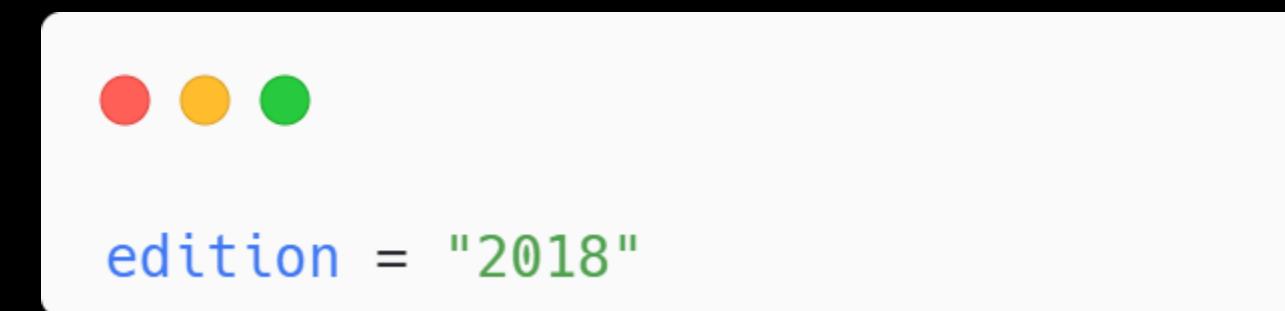
- 라이브러리를 컴파일할 때는 `--crate-type lib` 옵션이 쓰인다.

이렇게 하면 `rustc`가 `main()` 함수를 찾지 않고 `.rlib` 파일을 만들게 된다.

이 파일은 컴파일된 코드를 담고 있어서 나중에 바이너리와 다른 `.rlib` 파일을 만드는데 쓰일 수 있다.

- 프로그램을 컴파일할 때는 `--crate-type bin` 옵션이 쓰이며,
결과물로 대상 플랫폼의 바이너리 실행 파일이 만들어진다.

- 에디션
 - Rust는 호환성 보장이 매우 뛰어나다.
Rust 1.0으로 컴파일된 코드는 Rust 1.50은 물론, 그 이후 버전에서도 컴파일되어야 한다.
 - 그러나 가끔은 기존 코드가 더 이상 컴파일되지 않더라도 흥미로운 기능을 추가해 언어를 확장하고 싶을 때가 있다.
예를 들어, Rust는 오랜 논의 끝에 식별자 `async/await`를 키워드로 쓰는 비동기 프로그래밍을 위한 문법을 추가했다.
그러나 이로 인해 `async`나 `await`를 변수 이름으로 쓰는 기존 코드들이 모두 깨지고 말 것이다.
 - Rust는 기존 코드를 깨뜨리지 않고 진화하기 위해서 에디션(Edition)을 사용한다.
Rust 2015 에디션은 Rust 1.0과 호환된다. 2018 에디션은 `async`와 `await`를 키워드로 바꿨고
모듈 시스템을 간소화했으며, 2021 에디션은 배열의 사용성을 개선했고 널리 쓰이는 라이브러리 정의 몇 가지를
기본적으로 어디서나 쓸 수 있게 만들었다.
 - 각 크레이트는 `Cargo.toml` 파일 맨 위에 있는 `[package]` 부분에 자신이 사용하는 Rust의 에디션을 기재한다.



빌드 프로필

- Cargo.toml 파일에는 몇 가지 구성 설정을 둘 수 있는데,
이 설정값에 따라서 cargo가 생성하는 rustc 명령줄의 내용이 달라진다.

명령줄	사용되는 Cargo.toml 영역
cargo build	[profile.dev]
cargo build --release	[profile.release]
cargo test	[profile.test]

- 대부분의 경우는 기본 설정을 쓰면 되지만, 프로파일러를 쓰려고 하는 경우는 예외다.
 - 프로파일러는 프로그램에서 CPU 시간이 소비되는 곳을 측정하는 도구다.
 - 프로파일러에서 제대로 된 데이터를 얻으려면 최적화와 디버그 심볼이 모두 필요하다.이 둘을 모두 적용하려면 Cargo.toml에 다음 내용을 추가한다.



```
[profile.release]
debug = true # Enable debug symbols in release builds
```

- 빌드 프로필
 - Cargo.toml에서 변경할 수 있는 다른 여러 설정들에 대해서는 Cargo 문서(<https://doc.rust-lang.org/cargo/reference/manifest.html>)를 참고하자.

The Manifest Format

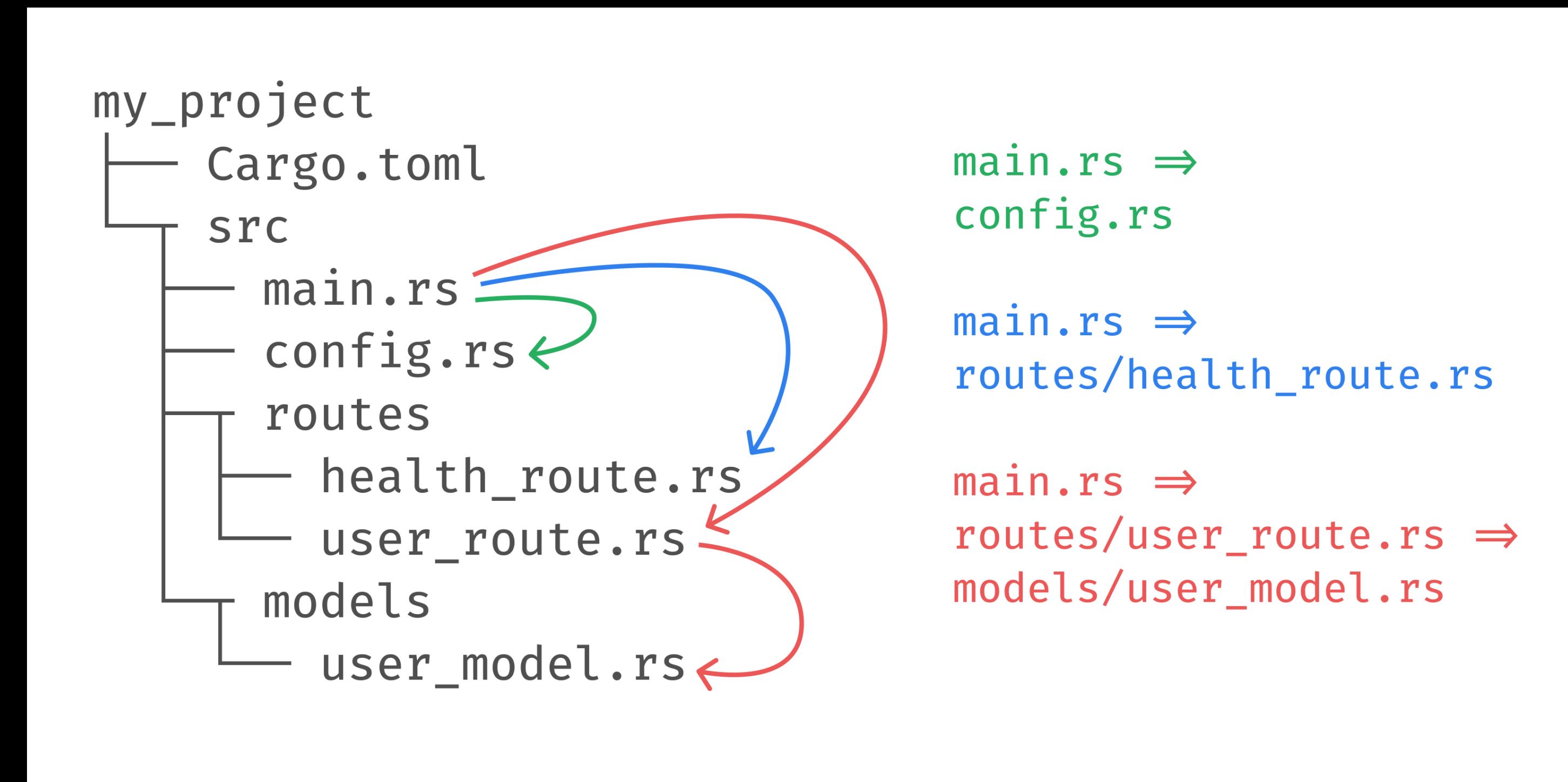
The `Cargo.toml` file for each package is called its *manifest*. It is written in the TOML format. It contains metadata that is needed to compile the package. Checkout the `cargo locate-project` section for more detail on how cargo finds the manifest file.

Every manifest file consists of the following sections:

- `cargo-features` — Unstable, nightly-only features.
- `[package]` — Defines a package.
 - `name` — The name of the package.
 - `version` — The version of the package.
 - `authors` — The authors of the package.
 - `edition` — The Rust edition.
 - `rust-version` — The minimal supported Rust version.
 - `description` — A description of the package.
 - `documentation` — URL of the package documentation.
 - `readme` — Path to the package's README file.
 - `homepage` — URL of the package homepage.
 - `repository` — URL of the package source repository.
 - `license` — The package license.
 - `license-file` — Path to the text of the license.
 - `keywords` — Keywords for the package.
 - `categories` — Categories of the package.

모듈

- 크레이트가 프로젝트 간의 코드 공유에 관한 것이라면
모듈(Module)은 프로젝트 내부의 코드 구성에 관한 것이다.
- Rust의 네임스페이스로 Rust 프로그램이나 라이브러리를 구성하는 함수, 타입, 상수 등을 담는 컨테이너 역할을 한다.



- 모듈은 다음과 같은 형태를 갖는다.

```
● ● ●

mod spores {
    use cells:::{Cell, Gene};

    /// A cell made by an adult fern. It disperses on the wind as part of the fern life cycle.
    /// A spore grows into a prothallus -- a whole separate organism, up to 5mm across
    /// -- which produces the zygote that grows into a new fern. (Plant sex is complicated.)
    pub struct Spore {
        // ...
    }

    /// Simulate the production of a spore by meiosis.
    pub fn produce_spore(factory: &mut Sporangium) -> Spore {
        // ...
    }

    /// Extract the genes in a particular spore.
    pub(crate) fn genes(spore: &Spore) -> Vec<Gene> {
        // ...
    }

    /// Mix genes to prepare for meiosis (part of interphase).
    fn recombine(parent: &mut Cell) {
        // ...
    }

    // ...
}
```

- 모듈은 아이템(Item)의 집합체다.
 - 아이템이란 이전 코드에 있는 Spore 구조체와 세 함수처럼 이름이 있는 기능을 말한다.
 - pub 키워드는 아이템을 모듈 바깥에서 접근할 수 있도록 공개한다.
 - 이전 코드를 보면 함수 하나가 pub(crate)로 표시되어 있는데, 이는 해당 함수를 이 크레이트 내부 어디서든 사용할 수 있게 만들되 외부 인터페이스의 일부로 노출하진 않겠다는 뜻이다.
따라서 해당 함수는 다른 크레이트에서 사용할 수 없고 이 크레이트의 문서에도 표시되지 않는다.
 - pub으로 표시되지 않은 것은 모두 비공개이며, 자신이 정의된 모듈이나 자식 모듈에서만 사용할 수 있다.

모듈

HSPACE Rust 특강
Rust Basic #5 - 크레이트와 모듈

- 중첩된 모듈

- 모듈은 중첩될 수 있는데, 다음처럼 한 모듈이 단순히 하위 모듈의 집합체로 쓰이는 경우를 심심찮게 볼 수 있다.



```
mod plant_structures {
    pub mod roots {
        // ...
    }

    pub mod stems {
        // ...
    }

    pub mod leaves {
        // ...
    }
}
```

- 중첩된 모듈

- 중첩된 모듈 안에 있는 아이템을 다른 크레이트에서 볼 수 있게 만들고 싶을 때는 아이템 자체와 바깥쪽 모듈들을 전부 공개로 표시해 둬야 한다. 그렇지 않으면 다음과 같은 경고를 보게 될 수도 있다.

```
warning: function is never used: `is_square`
--> src/crates_unused_items.rs:23:9
|
23 | /         pub fn is_square(root: &Root) -> bool {
24 | |             root.cross_section_shape().is_square()
25 | |         }
| |_____^
```

- 어쩌면 이 함수는 현재 시점에서 죽은 코드일 수 있다. 그러나 이 함수를 다른 크레이트에서 사용할 셈이었다면, 사실은 해당 함수가 다른 크레이트에서 볼 수 없는 상태에 있다는 걸 알려 주고 있는 것으로 봄다.
- 이럴 때는 바깥쪽 모듈들도 전부 pub로 되어 있는지 확인해봐야 한다.

- 중첩된 모듈

- 아이템을 부모 모듈에서만 볼 수 있게 만들고 싶을 때는 pub(super)를 지정하면 되고, 아이템을 원하는 부모 모듈과 그의 자식 모듈에서 볼 수 있게 만들고 싶을 때는 pub(in <경로>)를 지정하면 된다.
- 특히, 깊은 중첩된 모듈을 쓸 때 유용하다.

```
● ● ●

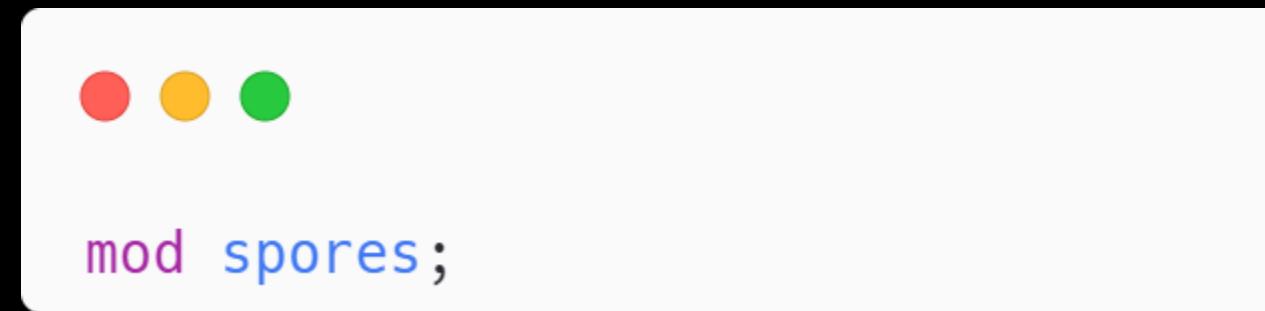
mod plant_structures {
    pub mod roots {
        pub mod products {
            pub(in crate::plant_structures::roots) struct Cytokinin {
                // ...
            }
        }

        use products::Cytokinin;      // OK: in `roots` module
    }

    use roots::products::Cytokinin; // Error: `Cytokinin` is private
}

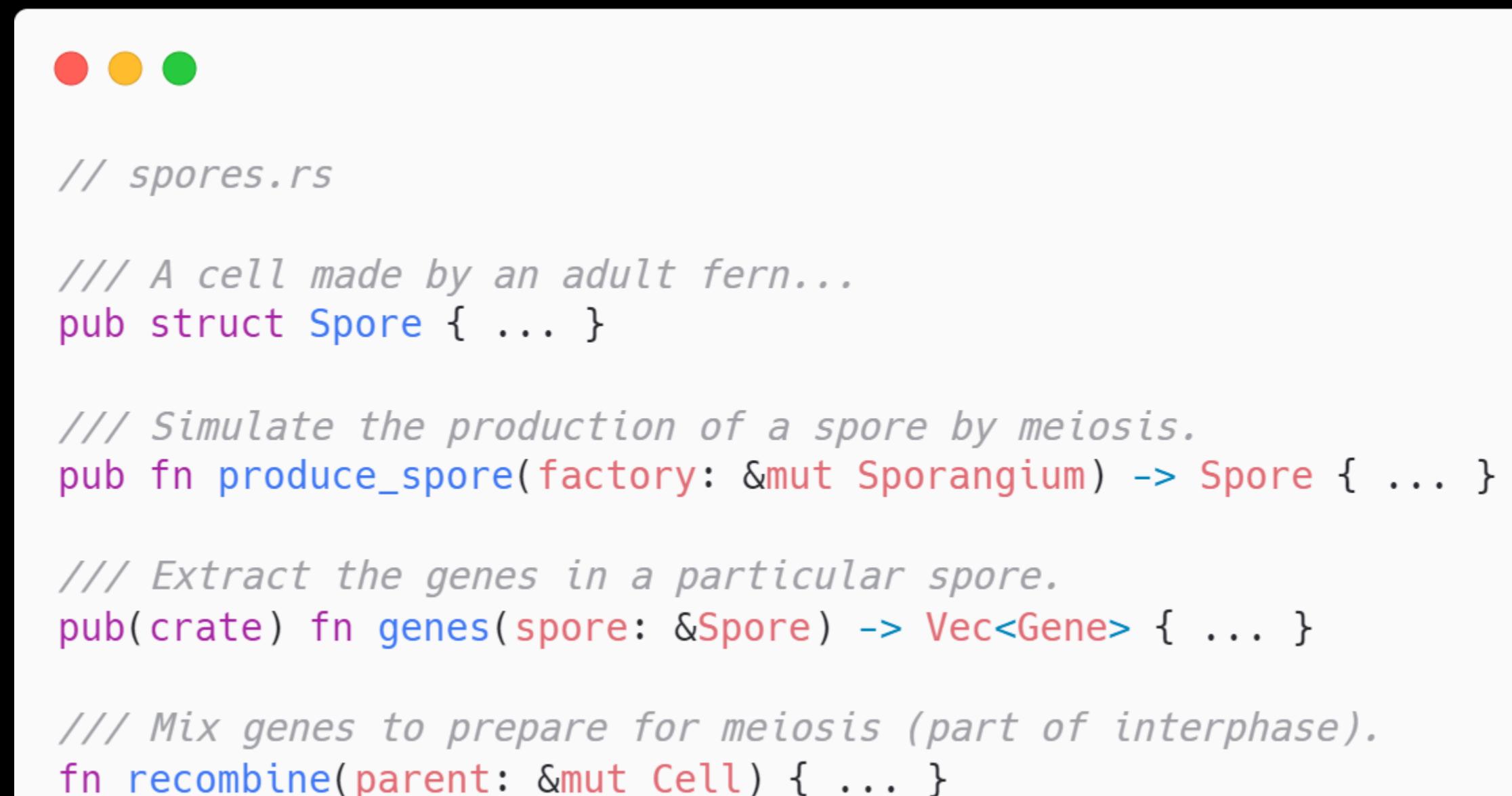
// Error: `Cytokinin` is private
use plant_structures::roots::products::Cytokinin;
```

- 분리된 파일에 있는 모듈
- 모듈은 다음처럼 작성할 수도 있다.



```
mod spores;
```

- 앞에서는 spores 모듈의 본문을 중괄호 안에 넣어 포함시켰다.
반면, 여기서는 Rust 컴파일러에게 spores 모듈이 spores.rs라고 하는 별도의 파일에 들어 있다고 알리고 있다.



```
// spores.rs

/// A cell made by an adult fern...
pub struct Spore { ... }

/// Simulate the production of a spore by meiosis.
pub fn produce_spore(factory: &mut Sporangium) -> Spore { ... }

/// Extract the genes in a particular spore.
pub(crate) fn genes(spore: &Spore) -> Vec<Gene> { ... }

/// Mix genes to prepare for meiosis (part of interphase).
fn recombine(parent: &mut Cell) { ... }
```

- **분리된 파일에 있는 모듈**

- 이전 슬라이드에서 `spores` 모듈과 더 이전 슬라이드에 있던 `spores` 모듈의 유일한 차이점은 코드의 위치다.
 - 공개인 것과 비공개인 것을 정하는 규칙은 두 경우 모두 동일하다.
 - Rust는 모듈이 분리된 파일에 있더라도 절대 따로 컴파일하지 않는다.
Rust 크레이트를 빌드하면 그때 여기에 속한 모든 모듈이 다시 컴파일된다.
- 모듈은 자기만의 디렉터리를 가질 수 있다.
 - Rust는 `mod spores;`를 만나면 `spores.rs`나 `spores/mod.rs`가 있는지를 확인한다. 둘 다 없거나 둘 다 있으면 오류다.
 - 이전 슬라이드에서는 `spores` 모듈이 하위 모듈을 갖지 않기 때문에 `spores.rs`를 사용했다.
그러나 더 이전 슬라이드에 있던 `plant_structures` 모듈을 생각해 보자.
이 모듈과 그 안에 있는 세 하위 모듈을 각각 분리된 파일로 나눈다면, 최종 프로젝트의 모습은 다음과 같다.

```
fern_sim/
└── Cargo.toml
└── src/
    ├── main.rs
    └── spores.rs
        └── plant_structures/
            ├── mod.rs
            ├── leaves.rs
            ├── roots.rs
            └── stems.rs
```

- 분리된 파일에 있는 모듈
 - plant_structures 모듈은 main.rs에 선언한다.

```
pub mod plant_structures;
```
 - 그러면 Rust는 다음과 같이 세 하위 모듈이 선언되어 있는 plant_structures/mod.rs를 로드하게 된다.

```
// in plant_structures/mod.rs
pub mod roots;
pub mod stems;
pub mod leaves;
```
 - 이 세 모듈의 내용은 각각 leaves.rs, roots.rs, stems.rs라는 이름을 가진 파일에 저장된다.
이 파일들은 mod.rs가 있는 plant_structures 디렉터리에 들어 있다.

- **분리된 파일에 있는 모듈**

- 같은 이름으로 된 파일과 디렉터리를 써서 모듈을 구성할 수도 있다. 예를 들어, `stems`가 `xylem`과 `phloem`이라는 모듈을 포함해야 한다면, `stems`는 `plant_structures/stems.rs`에 놓둔 채 `stems` 디렉터리를 추가할 수 있다.

```
fern_sim/
└── Cargo.toml
└── src/
    ├── main.rs
    └── spores.rs
        └── plant_structures/
            ├── mod.rs
            ├── leaves.rs
            ├── roots.rs
            └── stems/
                ├── phloem.rs
                └── xylem.rs
            └── stems.rs
```

- 그런 다음 새로운 두 하위 모듈을 `stems.rs`에 선언한다.



```
// in plant_structures/stems.rs
pub mod xylem;
pub mod phloem;
```

모듈

HSPACE Rust 특강
Rust Basic #5 - 크레이트와 모듈

- **분리된 파일에 있는 모듈**

- Rust의 모듈 시스템은 다음 세 가지 옵션을 통해서 프로젝트 구조를 여러분이 원하는 대로 꾸려갈 수 있게 도와준다.
- 자체 파일로 된 모듈
- 자체 디렉토리와 mod.rs로 된 모듈
- 자체 파일과 모듈을 포함하는 보조 디렉토리로 된 모듈

- 경로와 가져오기

- :: 연산자는 모듈의 기능에 접근하기 위한 용도로 쓰인다.

경로를 쓰면 프로젝트에 있는 코드 어디서든 표준 라이브러리의 기능을 참조할 수 있다.



```
if s1 > s2 {  
    std::mem::swap(&mut s1, &mut s2);  
}
```

- std는 표준 라이브러리 이름이다. 경로 std는 표준 라이브러리의 최상위 모듈을 가리킨다.
std::mem은 표준 라이브러리 안에 있는 하위 모듈이고 std::mem::swap은 그 모듈에 있는 공개 함수다.

- 경로와 가져오기

- 모든 코드를 이런 식으로 작성할 수도 있겠지만, 원이나 사전이 필요할 때마다 매번 `std::f64::consts::PI`나 `std::collections::HashMap::new`라고 적는 건 타이핑하기도 귀찮고 읽기도 어렵다.
이럴 땐 사용하려는 기능을 모듈 안으로 가져오면 편하다.

```
● ● ●

use std::mem;

if s1 > s2 {
    mem::swap(&mut s1, &mut s2);
}
```

- `use` 선언문은 바깥쪽 블록이나 모듈 전역에서 `std::mem`을 `mem`으로 쓸 수 있게 해준다. 즉, `mem`이란 이름이 `std::mem`의 지역 별칭이 된다.
- `use std::mem::swap;`이라고 써서 `mem` 모듈 대신 `swap` 함수 자체를 가져올 수도 있다. 그러나 위에서 한 것처럼 타입, 트레이트, (`std::mem` 같은) 모듈을 가져오고 나서 상대 경로로 안에 있는 함수와 상수를 비롯한 다른 멤버에 접근하는 것이 대개는 바람직한 스타일이다.

- 경로와 가져오기
 - 한 번에 여러 이름을 가져올 수도 있다.

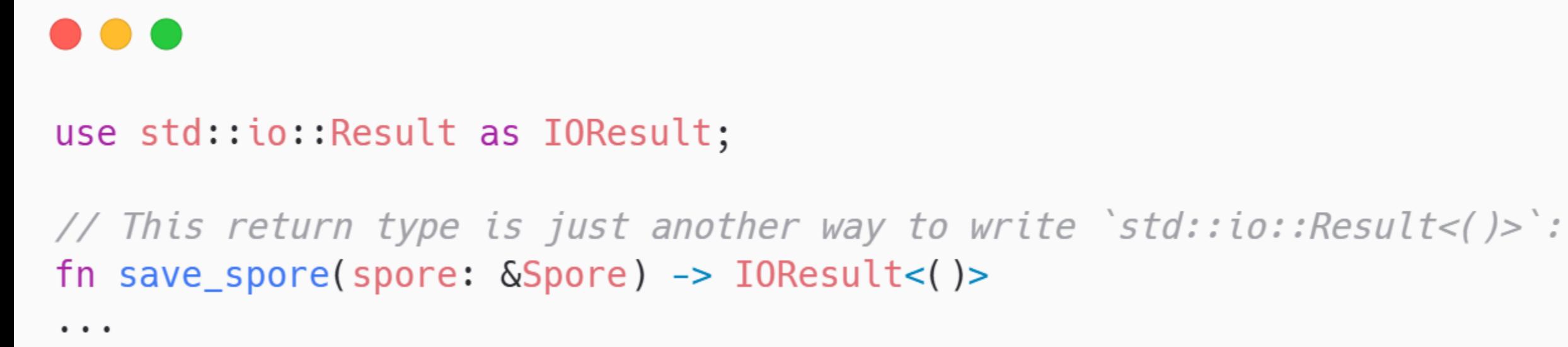
```
● ● ●  
  
use std::collections::{HashMap, HashSet};  
  
use std::fs::{self, File};  
  
use std::io::prelude::*;


```

- 앞의 코드는 모든 걸 일일이 적어 가져오는 코드의 축약 표기일 뿐이다.

```
● ● ●  
  
use std::collections::HashMap;  
use std::collections::HashSet;  
  
use std::fs;  
use std::fs::File;  
  
use std::io::prelude::Read;  
use std::io::prelude::Write;  
use std::io::prelude::BufRead;  
use std::io::prelude::Seek;
```

- 경로와 가져오기
 - as를 쓰면 아이템을 가져올 때 지역적으로 다른 이름을 줄 수 있다.



The screenshot shows a terminal window with three colored window control buttons (red, yellow, green) at the top. The terminal itself is black with white text. It contains the following Rust code:

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

// This return type is just another way to write `std::io::Result<()>`:
fn save_spore(spore: &Spore) -> IOResult<()>
...
```

- 경로와 가져오기

- 모듈은 부모 모듈이 가진 이름을 자동으로 상속하지 않는다.

예를 들어, `proteins/mod.rs`의 내용이 다음과 같다고 하자.



```
// proteins/mod.rs
pub enum AminoAcid { ... }
pub mod synthesis;
```

- 그렇다고 해서 `synthesis.rs`에 있는 코드가 `AminoAcid` 타입을 자동으로 볼 수 있게 되는 건 아니다.



```
// proteins/synthesis.rs
pub fn synthesize(seq: &[AminoAcid]) // Error: can't find type `AminoAcid`
...
```

모듈

HSPACE Rust 특강
Rust Basic #5 - 크레이트와 모듈

- 경로와 가져오기
 - 각 모듈은 백지상태에서 시작하기 때문에 반드시 자신이 사용하는 이름들을 가져와야 한다.



```
// proteins/synthesis.rs
use super::AminoAcid; // Explicitly import from parent

pub fn synthesize(seq: &[AminoAcid]) // OK
...
```

- 기본적으로 경로는 현재 모듈을 기준으로 한다.



```
// in proteins/mod.rs

// import from a submodule
use synthesis::synthesize;
```

- 경로와 가져오기
 - self는 현재 모듈의 동의어이기도 하기 때문에 다음처럼 쓸 수도 있다.



```
// in proteins/mod.rs

// import names from an enum,
// so we can write `Lys` for lysine, rather than `AminoAcid::Lys`
use self::AminoAcid::*;


```

- 아니면 간단하게 다음처럼 써도 된다.



```
// in proteins/mod.rs

use AminoAcid::*;


```

- 경로와 가져오기

- super와 crate 키워드는 경로에서 특별한 의미를 갖는다.
 - super : 부모 모듈
 - crate : 현재 모듈을 포함하고 있는 크레이트
- crate를 키워드를 쓰면 현재 모듈의 경로가 바뀌더라도 가져오기가 깨지지 않기 때문에 코드를 옮기기가 수월해진다. 예를 들어, synthesis.rs에서 crate를 쓰도록 고치면 다음과 같다.

```
// proteins/synthesis.rs
// Explicitly import relative to crate root
use crate::proteins::AminoAcid;

pub fn synthesize(seq: &[AminoAcid]) // OK
...
```

- 만약 하위 모듈에서 부모 모듈에 있는 비공개 아이템에 접근하고 싶다면, use super::*라고 쓰면 된다.

- 경로와 가져오기

- 사용 중인 크레이트와 같은 이름을 가진 모듈을 가지고 있을 때는 내용을 참조할 때 주의해야 한다.
예를 들어, Cargo.toml 파일에 image 크레이트를 의존성으로 등록해 둔 프로그램이 image라는 이름의 모듈을 가지고 있으면 image로 시작하는 경로들이 모호해진다.



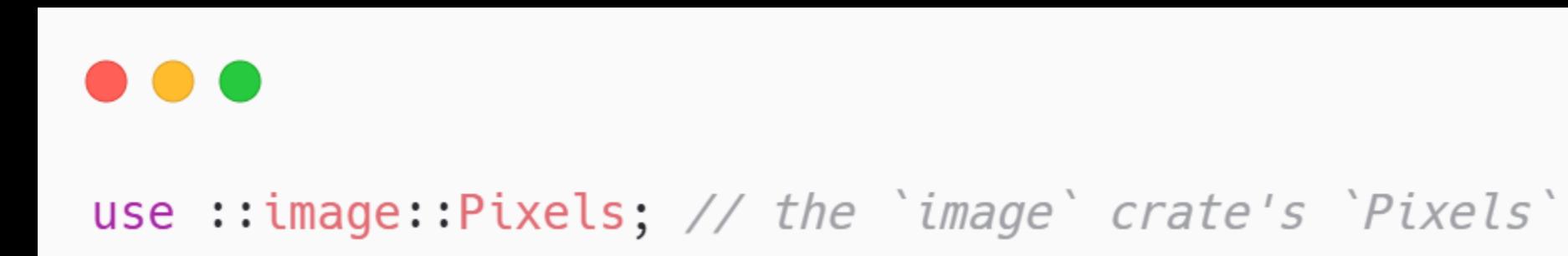
```
mod image {
    pub struct Sampler {
        ...
    }
}

// Error: Does this refer to our `image` module, or the `image` crate?
use image::Pixels;
```

- 위 코드에서 image 모듈이 Pixels 타입을 가지고 있지 않더라도 여전히 오류가 발생한다.
왜냐하면 나중에 그런 정의가 추가될 경우 프로그램에 있는 경로들의 지칭 대상이 바뀔 수 있어서 혼란을 불러올 여지가 있기 때문이다.

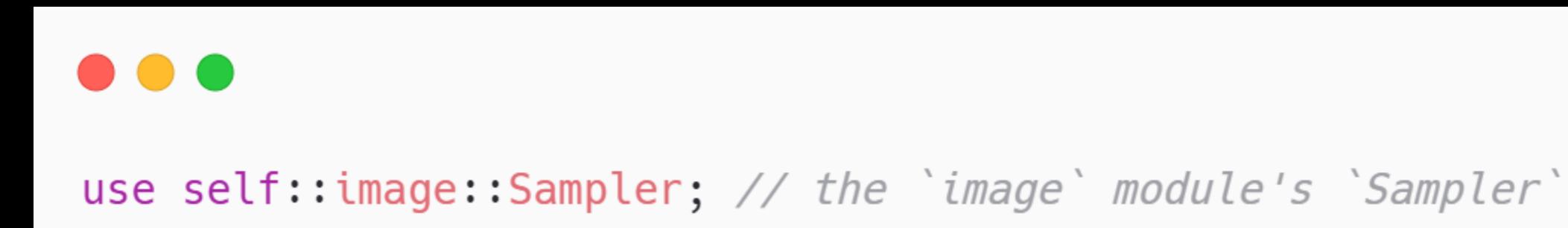
- 경로와 가져오기

- 이런 모호함을 해결하기 위해서 Rust는 **절대 경로(Absolute Path)**라고 하는 특별한 종류의 경로를 가지고 있다.
절대 경로는 ::로 시작하며, 항상 외부 크레이트를 참조한다.
- image 크레이트에 있는 Pixels 타입을 참조하려면 다음처럼 쓰면 된다.



```
use ::image::Pixels; // the `image` crate's `Pixels`
```

- 사용자 정의 모듈에 있는 Sampler 타입을 참조하려면 다음처럼 쓰면 된다.

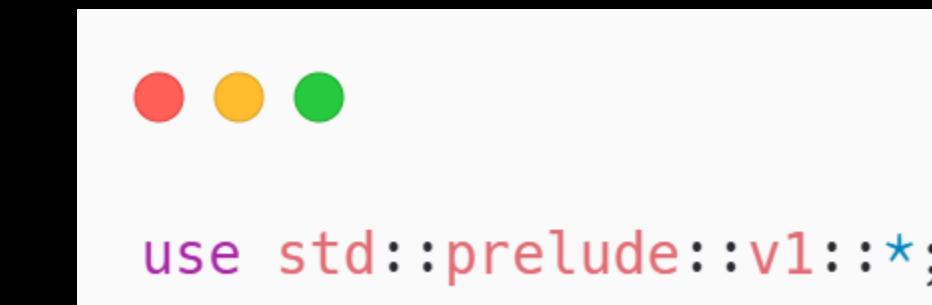


```
use self::image::Sampler; // the `image` module's `Sampler`
```

- 모듈이 파일과 같진 않지만, 모듈하고 Unix 파일 시스템의 파일과 디렉토리 사이에는 타고난 유사점이 있다.
ln 명령이 링크를 만들듯이, use 키워드는 별칭을 만든다. 경로는 파일 이름처럼 절대형과 상대형으로 쓸 수 있다.
self와 super는 특수 디렉토리인 .과 ..하고 비슷하다.

• 표준 프렐류드

- 앞에서 “각 모듈은 백지상태에서 시작한다”고 했지만, 완전한 백지는 아니다.
- 우선, 모든 프로젝트에는 표준 라이브러리 std가 자동으로 연결된다. 이 말은 `use std::whatever`라고 쓰거나 `std::mem::swap()`과 같은 식으로 코드 안에 이름을 인라인 처리하면 언제든 std 아이템을 참조할 수 있다는 뜻이다. 그뿐만 아니라 Vec과 Result처럼 자주 쓰이는 이름 몇 가지가 **표준 프렐류드(Standard Prelude)**에 들어 있어서 자동으로 포함된다.
- Rust는 루트 모듈을 포함한 모든 모듈의 맨 앞에 다음의 가져오기가 있는 것처럼 행동한다. 표준 프렐류드에는 자주 쓰이는 트레잇과 타입 수십 가지가 포함되어 있다.



- `use` 선언을 `pub`으로 만들기
 - `use` 선언은 별칭일 뿐이지만 공개될 수 있다.



```
// in plant_structures/mod.rs
...
pub use self::leaves::Leaf;
pub use self::roots::Root;
```

- 위 코드는 `Leaf`와 `Root`가 `plant_structures` 모듈의 공개 아이템이라는 뜻이다.
이들은 여전히 `plant_structures::leaves::Leaf`와 `plant_structures::roots::Root`의 단순한 별칭이다.
- 표준 프렐류드에는 이런 식으로 일련의 `pub` 가져오기들이 나열되어 있다.

- 구조체 필드를 pub으로 만들기
 - 모듈은 struct 키워드로 정의하는 사용자 정의 구조체 타입을 포함할 수 있다.



```
pub struct Fern {  
    pub roots: RootSet,  
    pub stems: StemSet  
}
```

- 구조체의 필드는 설령 비공개라 하더라도 그 구조체가 선언된 모듈과 그의 하위 모듈 전역에서 접근할 수 있다.
모듈 바깥에서는 공개 필드만 접근할 수 있다.

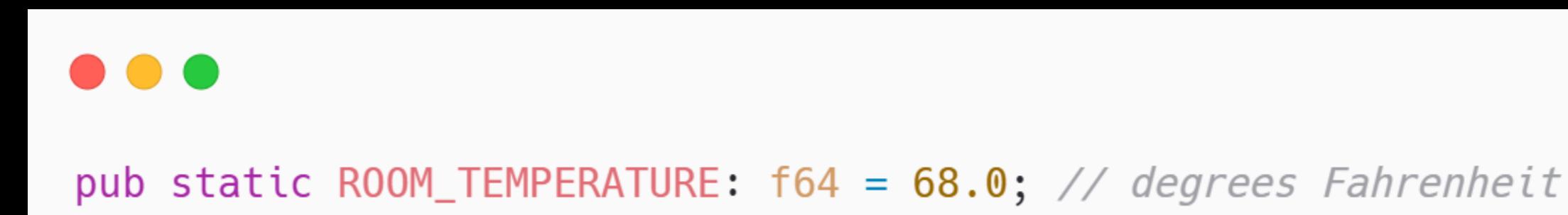
- **스태틱과 상수**

- 모듈은 함수, 타입, 중첩된 모듈 외에도 **상수(Constant)**과 **스태틱(Static)**을 정의할 수 있다.
- 상수는 `const` 키워드로 정의한다. 문법은 `let`과 비슷하지만 `pub`으로 표시할 수 있다는 점과 타입을 꼭 적어야 한다는 점이 다르다.
상수의 이름은 `UPPERCASE_NAMES`처럼 짓는 게 관례다.



```
pub const ROOM_TEMPERATURE: f64 = 20.0; // degrees Celsius
```

- 스탠틱 아이템은 `static` 키워드로 정의하며, 상수와 거의 비슷하다.



```
pub static ROOM_TEMPERATURE: f64 = 68.0; // degrees Fahrenheit
```

- **스태틱과 상수**

- 모듈은 함수, 타입, 중첩된 모듈 외에도 **상수(Constant)**과 **스태틱(Static)**을 정의할 수 있다.
 - 상수는 C++의 `#define`과 약간 비슷한데, 코드에서 상수가 쓰이는 모든 곳에 해당 값이 컴파일되어 들어간다.
 - 스탠틱은 프로그램이 시작되기 전에 만들어져서 종료될 때까지 지속되는 변수다.
 - 코드에 있는 매직 넘버와 문자열에 대해서는 상수를 쓰면 되고,
데이터 양이 많거나 상수값의 레퍼런스를 빌려와야 할 때는 스탠틱을 쓰면 된다.
- `mut` 상수라는 건 없다. 또한 Rust에는 `mut` 스탠틱에 대해서 배타적 접근 규칙을 시행할 방법이 마련되어 있지 않다.
따라서 이들은 태생적으로 스레드 세이프하지 않으며, 안전한 코드에서는 절대로 쓸 수 없다.



```
static mut PACKETS_SERVED: usize = 0;

println!("{} served", PACKETS_SERVED); // Error: use of mutable static
```

감사합니다.

utilForever@gmail.com

<https://github.com/utilForever>

Facebook, Twitter: @utilForever