Make Interpreter in Rust, Week 2

A Tour of Rust, Part 2

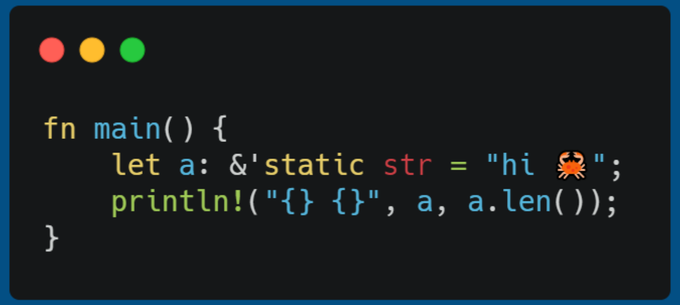
Chris Ohk
utilForever@gmail.com

목차

- Rust란?
- Part 1
  - 기초
  - 기초적인 흐름 제어
  - 기본 데이터 구조 자료형
  - Generic 자료형
  - 소유권과 데이터 대여
- Part 2
  - 텍스트
  - 객체 지향 프로그래밍
  - 스마트 포인터
  - 프로젝트 구성과 구조

문자열

- 문자열의 자료형은 &’static str이다.
  - &은 메모리 내의 장소를 참조하고 있다는 의미 (mut가 없으므로 값의 변경을 허용하지 않음)
  - ‘static은 문자열 데이터가 프로그램이 끝날 때까지 유효하다는 의미 (절대로 Drop되지 않음)
  - str은 언제나 유효한 UTF-8 바이트 열을 가리키고 있다는 의미



예외 처리 문자

- 어떤 문자들은 시각적으로 표현하기 어려우므로, 예외 처리 코드로 대체해서 쓴다.
  - ₩n : 줄바꿈
  - ₩r : 캐리지 리턴
  - ₩t : 탭
  - ₩₩ : 역슬래시
  - ₩0 : NULL
  - ₩' : 작은 따옴표

```
fn main() {
    let a: &'static str = "Ferris says:\t\"hello\"";
    println!("{}", a);
}
```

원시 문자열

- 원시 문자열은 r#"로 시작하고 "#으로 끝나며 문자열을 있는 그대로 쓰는 데 사용한다.

```
fn main() {
    let a: &'static str = r#"
        <div class="advice">
            Raw strings are useful for some situations.
        </div>
        "#;
    println!("{}", a);
}
```

파일에서 문자열 가져오기

- 매우 큰 텍스트가 필요하다면 include_str! 매크로를 이용해 로컬 파일에서 텍스트를 읽어오는 방식을 고려해보자.

```
let 00_html = include_str!("00_ko.html");
```

문자열 슬라이스

- 메모리 상의 바이트 열에 대한 참조이며, 언제나 유효한 UTF-8이어야 한다.
- &str에서 자주 사용하는 메소드는 다음과 같다.
  - len은 문자열의 바이트 길이를 가져온다. (글자수 아님)
  - starts_with와 ends_with는 기본적인 비교에 쓰인다.
  - is_empty는 길이가 0일 경우 true를 리턴한다.
  - find는 주어진 텍스트가 처음 등장하는 위치인 Option<usize> 값을 리턴한다.

```rust
fn main() {
    let a = "hi 🦀";
    println!("{}", a.len());
    let first_word = &a[0..2];
    let second_word = &a[3..7];
    // let half_crab = &a[3..5]; FAILS
    println!("{} {}", first_word, second_word);
}
```

문자

- Rust에서는 UTF-8 바이트 열을 char 타입의 벡터로 돌려주는 기능을 제공한다.
- char 하나는 4바이트다.

```rust
fn main() {
    let chars = "hi 🦀".chars().collect::<Vec<char>>();
    println!("{}", chars.len());
    println!("{}", chars[3] as u32);
}
```

스트링

- UTF-8 바이트 열을 힙 메모리에 소유하는 구조체
- 문자열과는 달리 변경하거나 기타 등등을 할 수 있다. (메모리가 힙에 있기 때문)
- 자주 사용하는 메소드는 다음과 같다.
  - push_str은 스트링의 맨 뒤에 UTF-8 바이트들을 더 붙일 때 사용한다.
  - replace는 UTF-8 바이트 열을 다른 것으로 교체할 때 사용한다.
  - to_lowercase와 to_uppercase는 대소문자를 비교할 때 사용한다.
  - trim은 공백을 제거할 때 사용한다.

```rust
fn main() {
    let mut helloworld = String::from("hello");
    helloworld.push_str(" world");
    helloworld = helloworld + "!";
    println!("{}", helloworld);
}
```

함수 매개변수로서의 텍스트

- 문자열과 스트링은 일반적으로 함수에 문자열 슬라이스 형태로 전달된다.
- 이 방법은 소유권을 넘길 필요가 없어 대부분의 경우에 유연하다.

```rust
fn say_it_loud(msg: &str) {
    println!("{}!!!", msg.to_string().to_uppercase());
}

fn main() {
    say_it_loud("hello");
    say_it_loud(&String::from("goodbye"));
}
```

스트링 만들기

- concat와 join은 스트링을 만드는 간단하지만 강력한 방법

```
fn main() {
    let helloworld = ["hello", " ", "world", "!"].concat();
    let abc = ["a", "b", "c"].join(",");
    println!("{}", helloworld);
    println!("{}", abc);
}
```

스트링 양식 만들기

- format! 매크로는 값이 어디에 어떻게 놓일지 매개변수화된 스트링을 정의해 생성한다.
- println!과 같이 매개변수화된 스트링을 사용한다.

```
fn main() {
    let a = 42;
    let f = format!("secret to life: {}", a);
    println!("{}", f);
}
```

스트링 변환

- to_string을 사용해 많은 데이터 타입을 스트링으로 변환할 수 있다.
- 제네릭 함수 parse로 스트링이나 문자열을 다른 데이터 타입을 갖는 값으로 변환할 수 있다. 이 함수는 실패할 수도 있기 때문에 Result를 리턴한다.

```rust
fn main() -> Result<(), std::num::ParseIntError> {
    let a = 42;
    let a_string = a.to_string();
    let b = a_string.parse::<i32>()?;
    println!("{} {}", a, b);
    Ok(())
}
```

OOP란 무엇인가?

- 객체 지향 프로그래밍은 다음과 같은 상징적 특징을 갖는 프로그래밍 언어를 뜻한다.
  - 캡슐화(Encapsulation) : 객체라 불리는 단일 타입의 개념적 단위에 데이터와 함수를 연결지음
  - 추상화(Abstraction) : 데이터와 함수를 숨겨 객체의 상세 구현 사항을 알기 어렵게 함
  - 다형성(Polymorphism) : 다른 기능적 관점에서 객체와 상호 작용하는 능력
  - 상속(Inheritance) : 다른 객체로부터 데이터와 동작을 상속받는 능력

Rust는 OOP가 아니다

- Rust에서는 어떠한 방법으로도 데이터와 동작의 상속이 불가능하다.
  - 구조체는 부모 구조체로부터 필드를 상속받을 수 없다.
  - 구조체는 부모 구조체로부터 함수를 상속받을 수 없다.

메소드 캡슐화하기

- Rust는 메소드가 연결된 구조체인 객체라는 개념을 지원한다.
- 모든 메소드의 첫번째 매개변수는 메소드 호출과 연관된 인스턴스에 대한 참조여야 한다.
  - `&self` : 인스턴스에 대한 변경 불가능한 참조
  - `&mut self` : 인스턴스에 대한 변경 가능한 참조
- 메소드는 `impl` 키워드를 쓰는 구현 블록 안에 정의한다.

```
struct SeaCreature {
    noise: String,
}

impl SeaCreature {
    fn get_sound(&self) -> &str {
        &self.noise
    }
}

fn main() {
    let creature = SeaCreature {
        noise: String::from("blub"),
    };
    println!("{}", creature.get_sound());
}
```

선택적 노출을 통한 추상화

- Rust는 객체의 내부 동작을 숨길 수 있다.
- 기본적으로, 필드와 메소드들은 그들이 속한 모듈에서만 접근 가능하다.
- pub 키워드는 구조체의 필드와 메소드를 모듈 밖으로 노출시킨다.

```rust
struct SeaCreature {
    pub name: String,
    noise: String,
}

impl SeaCreature {
    pub fn get_sound(&self) -> &str {
        &self.noise
    }
}

fn main() {
    let creature = SeaCreature {
        name: String::from("Ferris"),
        noise: String::from("blub"),
    };
    println!("{}", creature.get_sound());
}
```

다형성과 트레잇

- Rust는 트레잇으로 다형성을 지원한다.
  트레잇은 메소드의 집합을 구조체 데이터 타입에 연결할 수 있게 해준다.
- 먼저 트레잇 안에 메소드 원형을 정의한다. 구조체가 트레잇을 구현할 때,
  실제 데이터 타입이 무엇인지 알지 못하더라도 트레잇 데이터 타입을 통해
  간접적으로 구조체와 상호 작용할 수 있도록 협약을 맺게 된다.
- 구조체의 구현된 트레잇 메소드들은 구현 블록 안에 정의된다.

다형성과 트레잇


```rust
struct SeaCreature {
    pub name: String,
    noise: String,
}

impl SeaCreature {
    pub fn get_sound(&self) -> &str {
        &self.noise
    }
}

trait NoiseMaker {
    fn make_noise(&self);
}

impl NoiseMaker for SeaCreature {
    fn make_noise(&self) {
        println!("{}", &self.get_sound());
    }
}

fn main() {
    let creature = SeaCreature {
        name: String::from("Ferris"),
        noise: String::from("blub"),
    };
    creature.make_noise();
}
```

트레잇에 구현된 메소드

- 트레잇에 메소드를 구현해 넣을 수 있다.
- 함수가 구조체 내부의 필드에 직접 접근할 수는 없지만,
  트레잇 구현체들 사이에서 동작을 공유할 때 유용하게 쓰인다.

```
struct SeaCreature {
    pub name: String,
    noise: String,
}

impl SeaCreature {
    pub fn get_sound(&self) -> &str {
        &self.noise
    }
}

trait NoiseMaker {
    fn make_noise(&self);
    fn make_alot_of_noise(&self) {
        self.make_noise();
        self.make_noise();
        self.make_noise();
    }
}
```

```
impl NoiseMaker for SeaCreature {
    fn make_noise(&self) {
        println!("{}", &self.get_sound());
    }
}

fn main() {
    let creature = SeaCreature {
        name: String::from("Ferris"),
        noise: String::from("blub"),
    };
    creature.make_alot_of_noise();
}
```

트레잇 상속

- 트레잇은 다른 트레잇의 메소드들을 상속 받을 수 있다.

```rust
struct SeaCreature {
    pub name: String,
    noise: String,
}

impl SeaCreature {
    pub fn get_sound(&self) -> &str {
        &self.noise
    }
}

trait NoiseMaker {
    fn make_noise(&self);
}
```

```rust
trait LoudNoiseMaker: NoiseMaker {
    fn make_alot_of_noise(&self) {
        self.make_noise();
        self.make_noise();
        self.make_noise();
    }
}

impl NoiseMaker for SeaCreature {
    fn make_noise(&self) {
        println!("{}", &self.get_sound());
    }
}

impl LoudNoiseMaker for SeaCreature {}

fn main() {
    let creature = SeaCreature {
        name: String::from("Ferris"),
        noise: String::from("blub"),
    };
    creature.make_alot_of_noise();
}
```

동적 vs 정적 디스패치

- 메소드는 다음의 두 가지 방식으로 실행된다.
  - 정적 디스패치(Static Dispatch) : 인스턴스의 데이터 타입을 알고 있는 경우, 어떤 함수를 호출해야 하는지 정확히 알고 있다.
  - 동적 디스패치(Dynamic Dispatch) : 인스턴스의 데이터 타입을 모르는 경우, 올바른 함수를 호출할 방법을 찾아야 한다.
- 트레잇의 자료형인 `&dyn MyTrait`은 동적 디스패치를 통해 객체의 인스턴스들을 간접적으로 작동시킬 수 있게 해준다.
- Rust에서는 동적 디스패치를 사용할 경우 사람들이 알 수 있도록 트레잇 자료형 앞에 `dyn`을 붙일 것을 권고한다.

동적 vs 정적 디스패치


```rust
struct SeaCreature {
    pub name: String,
    noise: String,
}

impl SeaCreature {
    pub fn get_sound(&self) -> &str {
        &self.noise
    }
}

trait NoiseMaker {
    fn make_noise(&self);
}

impl NoiseMaker for SeaCreature {
    fn make_noise(&self) {
        println!("{}", &self.get_sound());
    }
}
```

```rust
fn static_make_noise(creature: &SeaCreature) {
    creature.make_noise();
}

fn dynamic_make_noise(noise_maker: &dyn NoiseMaker) {
    noise_maker.make_noise();
}

fn main() {
    let creature = SeaCreature {
        name: String::from("Ferris"),
        noise: String::from("blub"),
    };
    static_make_noise(&creature);
    dynamic_make_noise(&creature);
}
```

트레잇 객체

- 객체의 인스턴스를 `&dyn MyTrait` 데이터 타입을 가진 매개 변수로 넘길 때,
  이를 트레잇 객체라고 한다.
- 트레잇 객체는 인스턴스의 올바른 메소드를 간접적으로 호출할 수 있게 해주며,
  인스턴스에 대한 포인터와 인스턴스 메소드들에 대한 함수 포인터 목록을 갖는 구조체다.

크기를 알 수 없는 데이터 다루기

- 트레잇은 원본 구조체를 알기 어렵게 하느라 원래 크기 또한 알기 어렵다.
- Rust에서 크기를 알 수 없는 값이 구조체에 저장될 때는 두 가지 방법으로 처리된다.
  - generics : 매개 변수의 데이터 타입을 효과적으로 활용해 알려진 데이터 타입 및 크기의 구조체/함수를 생성한다.
  - indirection : 인스턴스를 힙에 올림으로써 실제 데이터 타입의 크기 걱정 없이 그 포인터만 저장할 수 있는 간접적인 방법을 제공한다.

제네릭 함수

- Rust의 제네릭은 트레잇과 함께 작동한다. 매개 변수 데이터 타입 T를 정의할 때 해당 인자가 어떤 트레잇을 구현해야 하는지 나열함으로써 인자에 어떤 데이터 타입을 쓸 수 있는지 제한할 수 있다.
- 제네릭을 이용하면 컴파일 시 데이터 타입과 크기를 알 수 있는 정적 데이터 타입의 함수가 만들어지며, 따라서 정적 디스패치와 함께 크기가 정해진 값으로 저장할 수 있게 된다.

제네릭 함수


```rust
struct SeaCreature {
    pub name: String,
    noise: String,
}

impl SeaCreature {
    pub fn get_sound(&self) -> &str {
        &self.noise
    }
}

trait NoiseMaker {
    fn make_noise(&self);
}
```

```rust
impl NoiseMaker for SeaCreature {
    fn make_noise(&self) {
        println!("{}", &self.get_sound());
    }
}

fn generic_make_noise<T>(creature: &T)
where T: NoiseMaker,
{
    creature.make_noise();
}

fn main() {
    let creature = SeaCreature {
        name: String::from("Ferris"),
        noise: String::from("blub"),
    };
    generic_make_noise(&creature);
}
```

제네릭 함수 줄여쓰기

- 트레잇으로 제한한 제네릭은 줄여쓸 수 있다.

```rust
fn generic_make_noise<T>(creature: &T)
where T: NoiseMaker,
{
    creature.make_noise();
}
```

```rust
fn generic_make_noise(creature: &impl NoiseMaker)
{
    creature.make_noise();
}
```

Box

- Box는 스택에 있는 데이터를 힙으로 옮길 수 있게 해주는 자료 구조다.
- 스마트 포인터로도 알려진 구조체이며 힙에 있는 데이터를 가리키는 포인터를 들고 있다.
- 크기가 알려져 있는 구조체이므로 (왜냐하면 그저 포인터만 들고 있으므로)
  필드의 크기를 알아야 하는 구조체에 뭔가의 참조를 저장할 때 종종 사용된다.

Box


```rust
struct SeaCreature {
    pub name: String,
    noise: String,
}

impl SeaCreature {
    pub fn get_sound(&self) -> &str {
        &self.noise
    }
}

trait NoiseMaker {
    fn make_noise(&self);
}

impl NoiseMaker for SeaCreature {
    fn make_noise(&self) {
        println!("{}", &self.get_sound());
    }
}

struct Ocean {
    animals: Vec<Box<dyn NoiseMaker>>,
}
```

```rust
fn main() {
    let ferris = SeaCreature {
        name: String::from("Ferris"),
        noise: String::from("blub"),
    };
    let sarah = SeaCreature {
        name: String::from("Sarah"),
        noise: String::from("swish"),
    };
    let ocean = Ocean {
        animals: vec![Box::new(ferris), Box::new(sarah)],
    };
    for a in ocean.animals.iter() {
        a.make_noise();
    }
}
```

참조 다시 보기

- 참조는 근본적으로 메모리 상의 어떤 바이트들의 시작 위치를 가리키는 숫자일 뿐이며, 유일한 용도는 특정 타입의 데이터가 어디에 존재하는지에 대한 개념을 나타내는 것이다.
- 일반 숫자와의 차이점은 Rust에서 참조가 가리키는 값보다 더 오래 살지 않도록 수명을 검증한다는 거다. (안그러면 그걸 사용했을 때 오류가 날 것이다!)

원시 포인터

- 참조는 더 원시적인 자료형인 원시 포인터로 변환될 수 있다.
- 원시 포인터는 숫자와 마찬가지로 거의 제한없이 여기저기 복사하고 이동할 수 있다.
- Rust는 원시 포인터가 가리키는 메모리 위치의 유효성을 보증하지 않는다.
- 원시 포인터에는 두 종류가 있다.
  - `*const T` : 데이터 타입 T의 데이터를 가리키는 절대 변경되지 않는 원시 포인터
  - `*mut T` : 데이터 타입 T의 데이터를 가리키는 변경될 수 있는 원시 포인터
- 원시 포인터는 숫자와 상호 변환이 가능하다. (예 : `usize`)
- 원시 포인터는 안전하지 않은 코드의 데이터에 접근할 수 있다.

```rust
fn main() {
    let a = 42;
    let memory_location = &a as *const i32 as usize;
    println!("Data is here {}", memory_location);
}
```

역참조

- 참조(예 : &i32)를 통해 참조되는 데이터를 접근/변경하는 과정을 역참조라고 한다.
- 참조로 데이터를 접근/변경하는 데에는 두 가지 방법이 있다.
  - 변수 할당 중에 참조되는 데이터에 접근
  - 참조되는 데이터의 필드나 메소드에 접근
- Rust에는 이를 가능케 하는 강력한 연산자가 있다.

* 연산자

- `*` 연산자는 참조를 역참조하는 명시적인 방법이다.

```rust
fn main() {
    let a: i32 = 42;
    let ref_ref_ref_a: &&&i32 = &&&a;
    let ref_a: &i32 = **ref_ref_ref_a;
    let b: i32 = *ref_a;
    println!("{}", b)
}
```

연산자

- `.` 연산자는 참조의 필드와 메소드에 접근하는 데에 쓰인다.
  (좀 더 미묘하게 동작하는데, 참조 열을 자동으로 역참조한다.)

```rust
struct Foo {
    value: i32
}

fn main() {
    let f = Foo { value: 42 };
    let ref_ref_ref_f = &&&f;
    println!("{}", ref_ref_ref_f.value);
}
```

```rust
struct Foo {
    value: i32
}

fn main() {
    let f = Foo { value: 42 };
    let ref_ref_ref_f = &&&f;
    println!("{}", (***ref_ref_ref_f).value);
}
```

스마트 포인터

- Rust에서는 `&` 연산자로 이미 존재하는 데이터의 참조를 생성하는 기능과 더불어, 스마트 포인터라 불리는 참조 같은 구조체를 생성하는 기능을 제공한다.
- 고수준에서 보자면 참조는 다른 데이터 타입에 대한 접근을 제공하는 데이터 타입이라고 볼 수 있다. 스마트 포인터가 일반적인 참조와 다른 점은, 프로그래머가 작성하는 내부 로직에 기반해 동작하는 거다.
- 일반적으로 스마트 포인터는 구조체가 `*`와 `.` 연산자로 역참조될 때 무슨 일이 발생할 지 지정하기 위해 `Deref`, `DerefMut`, 그리고 `Drop` 트레잇을 구현한다.

스마트 포인터


```rust
use std::ops::Deref;

struct TattleTell<T> {
    value: T,
}

impl<T> Deref for TattleTell<T> {
    type Target = T;

    fn deref(&self) -> &T {
        println!("{} was used!", std::any::type_name::<T>());
        &self.value
    }
}

fn main() {
    let foo = TattleTell {
        value: "secret message",
    };
    println!("{}", foo.len());
}
```

위험한 스마트 코드

- 스마트 포인터는 안전하지 않은 코드를 꽤 자주 쓰는 경향이 있다. 앞서 말했듯이, 스마트 포인터는 Rust에서 가장 저수준의 메모리를 다루기 위한 일반적인 도구다.
- 무엇이 안전하지 않은 코드일까? 안전하지 않은 코드는 Rust 컴파일러가 보증할 수 없는 몇 가지 기능이 있다는 예외사항을 제외하고는 일반적인 코드와 완전히 똑같이 동작한다.
- 안전하지 않은 코드의 주 기능은 원시 포인터를 역참조하는 것이다. 이는 원시 포인터를 메모리 상의 위치에 가져다 놓고 “데이터 구조가 여기있다!”고 선언한 뒤 사용할 수 있는 데이터 표현으로 변환하는 걸 의미한다. (즉, `*const u8`을 `u8`로)
- Rust에는 메모리에 쓰여지는 모든 바이트의 의미를 추적하는 방법이 없다. 원시 포인터로 쓰이는 임의의 숫자에 무엇이 존재하는지 보증할 수 없기 때문에, 역참조를 `unsafe { ... }` 블록 안에 넣는다.

위험한 스마트 코드


```rust
fn main() {
    let a: [u8; 4] = [86, 14, 73, 64];
    let pointer_a = &a as *const u8 as usize;
    println!("Data memory location: {}", pointer_a);

    let pointer_b = pointer_a as *const f32;
    let b = unsafe {
        *pointer_b
    };
    println!("I swear this is a pie! {}", b);
}
```

익숙한 친구들

- 이미 본 적 있는 `Vec<T>`나 `String` 같은 스마트 포인터를 생각해 보자.
  - `Vec<T>`는 바이트들의 메모리 영역을 소유하는 스마트 포인터다. Rust 컴파일러는 이 바이트들에 뭐가 존재하는지 모른다. 스마트 포인터는 관리하는 메모리 영역에서 내용물을 꺼내기 위해 자기가 뭘 의미하는지 해석하고, 데이터 구조가 그 바이트들 내 어디에서 시작하고 끝나는지 추적하며, 마지막으로 원시 포인터를 데이터 구조로, 또 쓰기 편한 깔끔한 인터페이스로 역참조한다. (예 : `my_vec[3]`)
  - `String`은 바이트들의 메모리 영역을 추적하며, 쓰여지는 내용물이 언제나 유효한 UTF-8이도록 프로그램적으로 제한하며, 그 메모리 영역을 `&str` 데이터 타입으로 역참조할 수 있도록 도와준다.
- 두 데이터 구조 모두 원시 포인터에 대한 안전하지 않은 역참조를 사용한다.

익숙한 친구들


```rust
use std::alloc::{alloc, Layout};
use std::ops::Deref;

struct Pie {
    secret_recipe: usize,
}

impl Pie {
    fn new() -> Self {
        let layout = Layout::from_size_align(4, 1).unwrap();

        unsafe {
            let ptr = alloc(layout) as *mut u8;

            ptr.write(86);
            ptr.add(1).write(14);
            ptr.add(2).write(73);
            ptr.add(3).write(64);

            Pie { secret_recipe: ptr as usize }
        }
    }
}
```

```rust
impl Deref for Pie {
    type Target = f32;
    fn deref(&self) -> &f32 {
        let pointer = self.secret_recipe as *const f32;
        unsafe { &*pointer }
    }
}

fn main() {
    let p = Pie::new();
    println!("{:?}", *p);
}\
```

힙에 할당된 메모리

- Box는 데이터를 스택에서 힙으로 옮길 수 있게 해주는 스마트 포인터다. 이를 역참조하면 마치 원래 데이터 타입이었던 것처럼 힙에 할당된 데이터를 편하게 쓸 수 있다.

```rust
struct Pie;

impl Pie {
    fn eat(&self) {
        println!("tastes better on the heap!")
    }
}

fn main() {
    let heap_pie = Box::new(Pie);
    heap_pie.eat();
}
```

참조 카운팅

- Rc는 스택에 있는 데이터를 힙으로 옮겨주는 스마트 포인터다. 이는 힙에 놓인 데이터를 변경 불가능하게 대여하는 기능을 갖는 다른 Rc 스마트 포인터를 복제할 수 있게 해준다.
- 마지막 스마트 포인터가 Drop될 때에만 힙에 있는 데이터가 할당 해제된다.

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

struct Pie;

impl Pie {
    fn eat(&self) {
        println!("tastes better on the heap!")
    }
}

fn main() {
    let heap_pie = Rc::new(Pie);
    let heap_pie2 = heap_pie.clone();
    let heap_pie3 = heap_pie2.clone();

    heap_pie3.eat();
    heap_pie2.eat();
    heap_pie.eat();
}
```

접근 공유하기

- RefCell은 보통 스마트 포인터가 보유하는 컨테이너 데이터 구조로서, 데이터를 가져오거나 안에 있는 것에 대한 변경 가능한 또는 불가능한 참조를 대여할 수 있게 해준다.
- 데이터를 대여할 때, Rust는 런타임에 메모리 안전 규칙을 적용해 남용을 방지한다.
  “단 하나의 변경 가능한 참조 또는 여러 개의 변경 불가능한 참조만 허용하며, 둘 다는 안됨!”
  이 규칙을 어기면 RefCell은 패닉을 일으킨다.

접근 공유하기


```rust
use std::cell::RefCell;

struct Pie {
    slices: u8
}

impl Pie {
    fn eat(&mut self) {
        println!("tastes better on the heap!");
        self.slices -= 1;
    }
}

fn main() {
    let pie_cell = RefCell::new(Pie{slices:8});

    {
        let mut mut_ref_pie = pie_cell.borrow_mut();
        mut_ref_pie.eat();
        mut_ref_pie.eat();
    }

     let ref_pie = pie_cell.borrow();
     println!("{} slices left",ref_pie.slices);
}
```

스레드 간에 공유하기

- Mutex는 보통 스마트 포인터가 보유하는 컨테이너 데이터 구조로서, 데이터를 가져오거나 안에 있는 것에 대한 변경 가능한 또는 불가능한 참조를 대여할 수 있게 해준다.
- 잠긴 대여를 통해 운영체제가 동시에 오직 하나의 CPU만 데이터에 접근 가능하도록 하고, 원래 스레드가 끝날 때까지 다른 스레드들을 막음으로써 대여가 남용을 방지한다.
- Mutex는 여러 개의 CPU 스레드가 같은 데이터에 접근하는 걸 조율하는 방법이다.
- 특별한 스마트 포인터인 Arc도 있는데, 스레드-안전성을 가진 참조 카운트 증가 방식을 사용한다는 걸 제외하고는 Rc와 동일하다.
  - 동일한 Mutex에 다수의 참조를 가질 때 종종 사용되곤 한다.

스레드 간에 공유하기


```
use std::sync::Mutex;

struct Pie;

impl Pie {
    fn eat(&self) {
        println!("only I eat the pie right now!");
    }
}

fn main() {
    let mutex_pie = Mutex::new(Pie);

    let ref_pie = mutex_pie.lock().unwrap();
    ref_pie.eat();
}
```

스마트 포인터 조합하기

- 스마트 포인터는 한계가 있는 것처럼 보이지만, 조합해서 사용하면 매우 강력해질 수 있다.
  - `Rc<Vec<Foo>>`
    힙에 있는 변경 불가능한 데이터 구조의 동일한 벡터를 대여할 수 있는 복수의 스마트 포인터를 복제할 수 있게 해준다.
  - `Rc<RefCell<Foo>>`
    복수의 스마트 포인터가 동일한 `Foo` 구조체를 변경 가능하게 또는 불가능하게 대여할 수 있게 해준다.
  - `Arc<Mutex<Foo>>`
    복수의 스마트 포인터가 임시의 변경 가능한 또는 불가능한 대여를 CPU 스레드 독점 방식으로 잠글 수 있게 해준다.

스마트 포인터 조합하기


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

struct Pie {
    slices: u8,
}

impl Pie {
    fn eat_slice(&mut self, name: &str) {
        println!("{} took a slice!", name);
        self.slices -= 1;
    }
}

struct SeaCreature {
    name: String,
    pie: Rc<RefCell<Pie>>,
}

impl SeaCreature {
    fn eat(&self) {
        let mut p = self.pie.borrow_mut();
        p.eat_slice(&self.name);
    }
}
```

```rust
fn main() {
    let pie = Rc::new(RefCell::new(Pie { slices: 8 }));
    let ferris = SeaCreature {
        name: String::from("ferris"),
        pie: pie.clone(),
    };
    let sarah = SeaCreature {
        name: String::from("sarah"),
        pie: pie.clone(),
    };
    ferris.eat();
    sarah.eat();

    let p = pie.borrow();
    println!("{} slices left", p.slices);
}
```

모듈

- 모든 Rust 프로그램이나 라이브러리는 크레이트(Crate)다.
- 모든 크레이트는 모듈의 계층 구조로 이뤄져 있다.
- 모든 크레이트에는 최상위 모듈이 있다.
- 모듈에는 전역 변수, 함수, 구조체, 트레잇, 또는 다른 모듈까지도 포함될 수 있다!
- Rust에서는 파일과 모듈 트리 계층 구조 간의 1:1 대응이 없다.
  모듈의 트리 구조는 코드로 직접 작성해야 한다.

프로그램/라이브러리 작성하기

- 프로그램은 main.rs라 불리는 파일에 최상위 모듈을 갖고 있다.
- 라이브러리는 lib.rs라 불리는 파일에 최상위 모듈을 갖고 있다.

다른 모듈과 크레이트 참조하기

- 모듈 내의 항목은 전체 모듈 경로인 std::f64::consts::PI를 이용해 참조할 수 있다.
- 더 간단한 방법은 use 키워드를 사용하는 거다. 이를 이용하면 모듈에서 쓰고자 하는 특정 항목을 전체 경로를 쓰지 않고도 코드 어디에서든 사용할 수 있다.
  - 예를 들어, use std::f64::consts::PI를 쓰면 main 함수에서 PI만으로 사용할 수 있다.
- std는 유용한 데이터 구조 및 OS와 상호 작용할 수 있는 함수로 가득한 표준 라이브러리(Standard Library)의 크레이트다.

여러 개의 항목을 참조하기

- 복수의 항목을 하나의 모듈 경로로 참조하고 싶다면 다음과 같이 사용하면 된다.

```
use std::f64::consts::{PI, TAU}
```

모듈 작성하기

- 코드를 생각할 때 보통은 디렉토리로 구성된 파일 구조를 떠올린다.
- Rust에서 모듈을 선언하는 두 가지 방법이 있다.
  예를 들어, foo 모듈은 다음과 같이 나타낼 수 있다.
  - foo.rs라는 이름의 파일
  - foo라는 이름의 디렉토리에 들어있는 파일 mod.rs

모듈 계층구조

- 한 모듈은 다른 모듈에 의존할 수 있다. 모듈과 하위 모듈 사이에 관계를 지어주려면,
  부모 모듈에 코드 `mod foo;`를 작성하면 된다.
  `foo.rs` 파일이나 `foo/mod.rs` 파일을 찾아 범위 내의 `foo` 모듈 안에 그 내용물을 삽입한다.

인라인 모듈

- 하위 모듈은 모듈의 코드 내에 직접 치환될 수 있다.
- 인라인 모듈의 가장 흔한 용도는 유닛 테스트를 만들 때다.
  Rust가 테스트에 쓰일 때에만 존재하는 인라인 모듈을 만들 수 있다.

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

    ... tests go here ...
}
```

내부 모듈 참조하기

- Rust에서는 use 경로에 사용할 수 있는 몇 가지 키워드를 통해 원하는 모듈을 빠르게 가져다 쓸 수 있다.
  - crate : 최상위 모듈
  - super : 현재 모듈의 부모 모듈
  - self : 현재 모듈

내보내기

- 기본적으로 모듈의 구성원들은 외부에서 접근이 불가능하다. (자식 모듈에게까지도!)
  pub 키워드를 사용하면 모듈의 구성원들을 접근 가능하게 할 수 있다.
- 기본적으로 크레이트의 구 성원들도 외부에서 접근이 불가능하다.
  크레이트의 최상위 모듈에 pub을 표시하면 구성원들을 접근 가능하게 할 수 있다.

구조체 가시성

- 구조체도 함수와 마찬가지로 pub를 사용해 모듈 외부로 무엇을 노출할 지 선언할 수 있다.

```rust
pub struct SeaCreature {
    pub animal_type: String,
    pub name: String,
    pub arms: i32,
    pub legs: i32,
    weapon: String,
}
```

Prelude

- use로 가져오지도 않았는데 어떻게 어디서나 Vec나 Box를 쓸 수 있을까?
  이는 표준 라이브러리의 prelude 모듈 덕분이다.
- Rust의 표준 라이브러리에서는 std::prelude::*로 내보내기된 모든 것들이 어디에서든 자동으로 사용 가능하다는 점을 알아두자. Vec와 Box가 바로 이런 경우이며, 다른 것들 (Option, Copy, 기타 등등)도 마찬가지다.

참고 자료

- https://www.rust-lang.org/
- https://doc.rust-lang.org/book/
- https://tourofrust.com/
- The Rust Programming Language (No Starch Press, 2019)

Thank you!