The Rust Guidebook 2026: 차세대 시스템 프로그래밍

> "안전하고, 병렬적이며, 실용적인 시스템 프로그래밍을 위한 완벽 가이드"

이 책은 Rust의 기초부터 고급 비동기 프로그래밍, 그리고 아키텍처 설계까지 포괄하는 실전 가이드입니다. 이론적인 "문법 설명"을 넘어, **왜 Rust 인가?** (Why Rust?)에 대한 깊은 통찰과 **실무 모범 사례** (Best Practices)를 담았습니다.

✍️ 저자/기여자

NeuralFoundry-Coder (AI Systems Architect)

- *Philosophy*: "비용 없는 추상화, 타협 없는 안전성."

📚 목차 (Table of Contents)

- Chapter 5: 열거형 (Enums) - 대수적 데이터 타입과 Null 안전성
- Chapter 6: 모듈 시스템 - 캡슐화와 아키텍처 경계

Part 3: 데이터와 로직 (Data & Logic)

- Chapter 7: 컬렉션 - Vec, String, HashMap의 내부 구조
- Chapter 8: 에러 처리 - Result와 ? 연산자
- Chapter 9: 제네릭 - 단형성을 통한 런타임 오버헤드 제거

Part 4: 추상화와 검증 (Abstraction)

- Chapter 10: 트레이트 - 동작의 정의와 고아 규칙
- Chapter 11: 라이프타임 - 참조 유효성 검증
- Chapter 12: [프로젝트] 파일 정리 CLI - 실전 도구 만들기

Part 5: 고급 기능 (Advanced)

- Chapter 13: 함수형 기능 - 이터레이터 최적화
- Chapter 14: 스마트 포인터 - Box, Rc, RefCell, Interior Mutability
- Chapter 15: 병행성 - Send/Sync, Mutex, Channel

Part 6: 마스터리 (Mastery)

- Chapter 16: OOP 특징
- Chapter 17: 패턴 매칭 심화
- Chapter 18: [프로젝트] 웹 서버 - 멀티스레드 아키텍처 구현

🚀 학습 트랙

- **7일 완성 코스**: 1~3장(기초) -> 6장(모듈) -> 12장(CLI) -> 18장(웹서버)
- **30일 마스터 코스**: 1장부터 18장까지 순서대로, 매일 예제를 직접 타이핑하며 학습

Chapter 1: Rust의 세계로 (Introduction to Rust)

> “Rust는 단순히 안전한 언어가 아닙니다. Rust는 여러분이 더 나은 프로그래머가 되도록 돕는 멘토입니다.”

1.1 왜 지금 Rust인가? (Why Rust Now?)

지난 수십 년간 시스템 프로그래밍은 C와 C++가 지배해왔습니다. 하드웨어를 직접 제어할 수 있는 강력함을 제공했지만, 그 대가는 혹독했습니다. 바로 **메모리 안전성(Memory Safety)** 문제였습니다. 버퍼 오버플로우, 댕글링 포인터(Dangling Pointer), 데이터 레이스(Data Race)... 수많은 보안 취약점이 여기서 발생했습니다.

Rust는 이 딜레마를 해결하기 위해 탄생했습니다.

Rust의 3가지 핵심 약속

1. **메모리 안전성 (Memory Safety without GC)**: 가비지 컬렉터(Garbage Collector) 없이도 컴파일 타임에 메모리 오류를 잡아냅니다. 이는 예측 가능한 성능을 보장하면서도 안전하다는 뜻입니다.
2. **비용 없는 추상화 (Zero-cost Abstractions)**: 고수준의 추상화(맵, 필터 등)를 사용해도 어셈블리어로 짰을 때와 거의 동일한 속도로 동작합니다.
3. **두려움 없는 병행성 (Fearless Concurrency)**: 데이터 레이스 상태를 컴파일러가 차단하므로, 멀티스레드 프로그래밍을 두려움 없이 할 수 있습니다.

1.2 개발 환경 구축 (Installation)

Rust를 시작하는 가장 좋은 방법은 `rustup`을 사용하는 것입니다. `rustup`은 Rust 컴파일러(`rustc`)와 패키지 매니저(`cargo`), 그리고 표준 라이브러리 문서를 관리해주는 도구입니다.

macOS / Linux 설치 (터미널)

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

설치 스크립트가 실행되면 `1`을 눌러 기본 설치(Default installation)를 진행합니다.

설치 확인

새로운 터미널 창을 열고 다음 명령어를 입력하여 설치가 잘 되었는지 확인해 봅시다.

```
rustc --version
cargo --version
```

버전 정보가 출력된다면 성공입니다!

IDE 추천

생산성을 위해 VS Code(Visual Studio Code)에 **rust-analyzer** 확장을 설치하는 것을 강력히 추천합니다. 코드 자동 완성, 타입 힌트, 실시간 에러 검사를 완벽하게 지원합니다.

1.3 Hello, World! 그 이면의 동작 원리

전통에 따라 `Hello, World!`를 출력해보겠습니다. 하지만 우리는 단순히 코드를 복사해서 붙여넣는 것에 그치지 않고, 프로젝트 구조부터 제대로 잡아보겠습니다.

Cargo: Rust의 든든한 지원군

Rust 생태계에서 `cargo` 없이 개발하는 것은 상상하기 어렵습니다. Cargo는 빌드 시스템이자 패키지 매니저입니다.

새로운 프로젝트를 생성해 봅시다.

```
cargo new hello_rust
cd hello_rust
```

이 명령어를 실행하면 다음과 같은 디렉토리 구조가 생성됩니다.

```
hello_rust/
├── Cargo.toml    # 프로젝트 메타데이터 및 의존성 설정 파일
└── src/
    └── main.rs   # 실제 소스 코드
```

코드 분석 (`src/main.rs`)

`src/main.rs` 파일을 열어보면 다음과 같은 코드가 있습니다.

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

이 짧은 코드에도 Rust의 중요한 철학들이 담겨 있습니다.

- `fn main()`: 프로그램의 진입점(Entry Point)입니다.
- `println!`: 함수가 아니라 **매크로(Macro)**입니다. Rust에서 `!`로 끝나는 것은 매크로를 의미합니다. 매크로는 컴파일 타임에 코드를 확장하여 런타임 오버헤드를 줄이거나 가변 인자를 처리하는 등의 강력한 기능을 제공합니다.
- 세미콜론(`;`): Rust는 표현식(Expression) 기반 언어이지만, 구문(Statement)의 끝에는 세미콜론을 붙입니다.

실행하기 (Build & Run)

프로젝트 루트에서 다음 명령어로 실행합니다.

```
cargo run
```

```
   Compiling hello_rust v0.1.0 (/path/to/hello_rust)
    Finished dev [unoptimized + debuginfo] target(s) in 0.45s
     Running `target/debug/hello_rust`
Hello, world!
```

`cargo run`은 컴파일(`cargo build`)과 실행을 한 번에 처리해줍니다. 결과물은 `target/debug/` 디렉토리에 생성됩니다.

📚 1.4 핵심 요약

- Rust는 안전성과 속도를 모두 잡은 시스템 프로그래밍 언어입니다.
- `rustup`으로 툴체인을 설치하고, **VS Code + rust-analyzer** 조합을 추천합니다.
- `cargo`는 프로젝트 생성, 빌드, 의존성 관리를 담당하는 필수 도구입니다.

- `main` 함수는 프로그램의 시작점이며, `println!`은 화면 출력을 담당하는 매크로입니다.

다음 챕터에서는 Rust의 변수와 데이터 타입 기본기를 다지며 본격적인 프로그래밍을 시작해보겠습니다.

Chapter 2: 일반적인 프로그래밍 개념 (Common Programming Concepts)

> "Rust는 **불변성(Immutability)을 기본값**으로 채택함으로써, 예측 가능하고 안전한 코드를 작성하도록 유도합니다."

이 장에서는 거의 모든 프로그래밍 언어에 존재하는 공통 개념들이 Rust에서 어떻게 표현되는지 깊이 있게 살펴봅니다. 변수, 데이터 타입, 함수, 그리고 제어 흐름—이 네 가지 기둥은 모든 Rust 프로그램의 토대입니다.

2.1 변수와 가변성 (Variables and Mutability)

2.1.1 변수 선언: `let` 키워드

Rust에서 변수를 선언하려면 `let` 키워드를 사용합니다. 이것은 다른 많은 언어와 유사하지만, Rust만의 특별한 점이 있습니다.

```
fn main() {
    // 기본적인 변수 선언
    let x = 5;
    let pi = 3.14159;
    let greeting = "Hello, Rust!";

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

핵심 개념: Rust 컴파일러는 **타입 추론(Type Inference)**을 수행합니다. 위 코드에서:

- `x`는 값 `5`를 보고 `i32` 타입으로 추론됩니다
- `pi`는 소수점이 있으므로 `f64` 타입으로 추론됩니다
- `greeting`은 문자열 리터럴이므로 `&str` 타입입니다

2.1.2 불변성 (Immutability): 왜 기본이 불변인가?

대부분의 프로그래밍 언어와 달리, Rust의 변수는 기본적으로 **불변(Immutable)**입니다. 이것은 의도적인 설계 결정입니다.

```
fn main() {
    let x = 5;
    x = 6;  // ❌ 컴파일 에러!
    //  ^^^  cannot assign twice to immutable variable
}
```

왜 불변성이 중요한가?

1. **예측 가능성**: 코드를 읽을 때 "이 변수는 절대 변하지 않는다"는 확신을 가질 수 있습니다
2. **동시성 안전성**: 불변 데이터는 여러 스레드에서 동시에 읽어도 안전합니다
3. **버그 예방**: 실수로 값을 변경하는 것을 방지합니다

```rust
// 실제 상황 예시: 구성 설정
fn main() {
    let max_connections = 100;  // 이 값은 프로그램 내내 변하지 않아야 함

    // ... 수백 줄의 코드 ...

    // 누군가 실수로 이 값을 바꾸려 해도 컴파일러가 막아줍니다!
    // max_connections = 200;  // ✗ 컴파일 에러

    println!("최대 연결 수: {}", max_connections);
}
```

2.1.3 가변 변수: `mut` 키워드

값을 변경해야 할 때는 명시적으로 `mut` 키워드를 붙입니다. 이것은 **"이 변수는 변경될 것입니다"**라는 개발자의 의도를 코드에 명확히 표현합니다.

```rust
fn main() {
    let mut counter = 0;  // mut로 가변성 선언

    println!("초기값: {}", counter);  // 0

    counter = counter + 1;
    println!("1 증가: {}", counter);  // 1

    counter += 1;  // 복합 대입 연산자 사용
    println!("1 더 증가: {}", counter);  // 2
}
```

실전 예제: 사용자 입력 처리

```rust
use std::io;

fn main() {
    let mut input = String::new();  // 사용자 입력을 저장할 가변 문자
열

    println!("이름을 입력하세요:");
    io::stdin()
        .read_line(&mut input)
        .expect("입력 읽기 실패");

    let trimmed = input.trim();  // 공백 제거
    println!("안녕하세요, {}님!", trimmed);
}
```

2.1.4 상수 (Constants)

상수는 `const` 키워드로 선언하며, 몇 가지 특별한 규칙이 있습니다:

특성	변수 (`let`)	상수 (`const`)
가변성	`mut` 가능	항상 불변
타입 명시	선택적	필수
값 결정 시점	런타임 가능	컴파일 타임만
스코프	블록 내	전역 가능
섀도잉	가능	불가능

```rust
// 상수 선언: 타입 명시 필수, SCREAMING_SNAKE_CASE 관례
const MAX_POINTS: u32 = 100_000;
const PI: f64 = 3.141592653589793;
const APP_NAME: &str = "Rust Learning App";

fn main() {
    println!("앱 이름: {}", APP_NAME);
    println!("최대 점수: {}", MAX_POINTS);
    println!("원주율: {}", PI);

    // 상수는 컴파일 타임에 평가되므로 복잡한 표현식 불가
    // const COMPUTED: u32 = some_function();  // ❌ 에러
}
```

언더스코어 숫자 구분자: `100_000`처럼 숫자 사이에 언더스코어를 넣어 가독성을 높일 수 있습니다. `100000` = `100_000` = `10_00_00` (같은 값)

2.1.5 섀도잉 (Shadowing)

같은 이름의 변수를 `let`으로 다시 선언하면 이전 변수를 **"가립니다(Shadow)"**. 이것은 변수 재할당과 다른 개념입니다.

```rust
fn main() {
    let x = 5;
    println!("처음 x: {}", x);  // 5

    let x = x + 1;  // 새로운 x 생성 (이전 x는 가려짐)
    println!("두 번째 x: {}", x);  // 6

    {
        let x = x * 2;  // 이 스코프 내에서만 유효한 새로운 x
        println!("내부 스코프 x: {}", x);  // 12
    }

    println!("외부 스코프 x: {}", x);  // 다시 6 (내부 x는 스코프 종료)
}
```

섀도잉 vs mut: 핵심 차이

특성	섀도잉 (`let x = ...`)	가변 (`mut`)
타입 변경	✅ 가능	❌ 불가능
새 변수 생성	✅ 예	❌ 아니오
불변성 유지	✅ 새 변수는 불변	❌ 변경 가능

```rust
fn main() {
    // 섀도잉: 타입 변경 가능!
    let spaces = "   ";           // &str 타입
    let spaces = spaces.len();    // usize 타입으로 변경!
    println!("공백 개수: {}", spaces);  // 3

    // mut로는 타입 변경 불가
    // let mut spaces2 = "   ";
    // spaces2 = spaces2.len();  // ❌ 타입 불일치 에러
}
```

2.2 데이터 타입 (Data Types)

Rust는 **정적 타입(Statically Typed)** 언어입니다. 컴파일 시점에 모든 변수의 타입이 결정되어야 합니다. 타입은 크게 **스칼라 타입**과 **복합 타입**으로 나뉩니다.

2.2.1 스칼라 타입 (Scalar Types)

스칼라 타입은 **단일 값**을 나타냅니다. Rust에는 네 가지 기본 스칼라 타입이 있습니다.

정수 타입 (Integer Types)

크기	부호 있음	부호 없음	범위
8비트	`i8`	`u8`	-128~127 / 0~255
16비트	`i16`	`u16`	-32,768~32,767 / 0~65,535
32비트	`i32`	`u32`	~±21억 / 0~43억

크기	부호 있음	부호 없음	범위
64비트	`i64`	`u64`	매우 큰 범위
128비트	`i128`	`u128`	극도로 큰 범위
아키텍처	`isize`	`usize`	플랫폼 의존적

```rust
fn main() {
    // 다양한 정수 타입
    let a: i8 = -128;        // 최솟값
    let b: u8 = 255;         // 최댓값
    let c: i32 = -2_147_483_648;
    let d: u64 = 18_446_744_073_709_551_615u64;

    // 다양한 리터럴 표기법
    let decimal = 98_222;         // 십진수
    let hex = 0xff;               // 16진수
    let octal = 0o77;             // 8진수
    let binary = 0b1111_0000;     // 2진수
    let byte = b'A';              // 바이트 (u8만)

    println!("10진수: {}", decimal);  // 98222
    println!("16진수: {}", hex);      // 255
    println!("8진수: {}", octal);     // 63
    println!("2진수: {}", binary);    // 240
    println!("바이트: {}", byte);     // 65 (A의 ASCII값)
}
```

정수 오버플로우 처리:

- **디버그 모드**: 오버플로우 시 패닉
- **릴리즈 모드**: 래핑 (최댓값 → 최솟값으로 순환)

```rust
fn main() {
    let max: u8 = 255;
    // let overflow = max + 1;  // 디버그: 패닉, 릴리즈: 0

    // 명시적 오버플로우 처리
    let wrapped = max.wrapping_add(1);      // 항상 래핑: 0
    let checked = max.checked_add(1);       // Option 반환: None
    let saturating = max.saturating_add(1); // 포화: 255 (최댓값 유지)

    println!("래핑: {}", wrapped);           // 0
    println!("체크: {:?}", checked);         // None
    println!("포화: {}", saturating);        // 255
}
```

부동소수점 타입 (Floating-Point Types)

타입	크기	정밀도
`f32`	32비트	단정밀도 (~7자리)
`f64`	64비트	배정밀도 (~15자리) 기본값

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

    // 산술 연산
    let sum = 5.0 + 10.0;
    let difference = 95.5 - 4.3;
    let product = 4.0 * 30.0;
    let quotient = 56.7 / 32.2;
    let remainder = 43.0 % 5.0;

    println!("합: {}", sum);
    println!("차: {}", difference);
    println!("곱: {}", product);
    println!("몫: {}", quotient);
    println!("나머지: {}", remainder);
}
```

불리언 타입 (Boolean Type)

```rust
fn main() {
    let t = true;
    let f: bool = false;

    // 논리 연산
    let and = t && f;    // false
    let or = t || f;     // true
    let not = !t;        // false

    println!("AND: {}, OR: {}, NOT: {}", and, or, not);
}
```

문자 타입 (Character Type)

Rust의 `char`는 **4바이트 유니코드 스칼라 값**입니다. ASCII만 지원하는 다른 언어와 다릅니다!

```
fn main() {
    let c = 'z';
    let z: char = 'ℤ';
    let heart_eyed_cat = '😻';
    let korean = '가';
    let japanese = '日';
    let emoji = '🦀';

    println!("문자들: {} {} {} {} {} {}", c, z, heart_eyed_cat,
korean, japanese, emoji);

    // 문자 크기 확인
    println!("char 크기: {} 바이트", std::mem::size_of::<char>
());  // 4
}
```

2.2.2 복합 타입 (Compound Types)

복합 타입은 여러 값을 하나로 묶습니다.

튜플 (Tuple)

고정 길이, **다양한 타입**을 담을 수 있는 복합 타입입니다.

```rust
fn main() {
    // 튜플 생성
    let tup: (i32, f64, u8) = (500, 6.4, 1);

    // 구조 분해 (Destructuring)
    let (x, y, z) = tup;
    println!("x: {}, y: {}, z: {}", x, y, z);

    // 인덱스 접근 (0부터 시작)
    let first = tup.0;
    let second = tup.1;
    let third = tup.2;
    println!("첫 번째: {}, 두 번째: {}, 세 번째: {}", first, second,
third);

    // 빈 튜플 = 유닛 타입 ()
    let unit: () = ();
    println!("유닛 타입: {:?}", unit);
}
```

실전 예제: 함수에서 여러 값 반환

```rust
fn calculate_stats(numbers: &[i32]) -> (i32, i32, f64) {
    let min = *numbers.iter().min().unwrap();
    let max = *numbers.iter().max().unwrap();
    let avg = numbers.iter().sum::<i32>() as f64 /
numbers.len() as f64;
    (min, max, avg)
}

fn main() {
    let data = [3, 1, 4, 1, 5, 9, 2, 6];
    let (min, max, avg) = calculate_stats(&data);
    println!("최솟값: {}, 최댓값: {}, 평균: {:.2}", min, max, avg);
}
```

배열 (Array)

고정 길이, 같은 타입의 요소들. 스택에 할당됩니다.

```rust
fn main() {
    // 배열 생성
    let a = [1, 2, 3, 4, 5];  // [i32; 5]

    // 타입 명시
    let b: [i32; 5] = [1, 2, 3, 4, 5];

    // 같은 값으로 초기화
    let c = [3; 5];  // [3, 3, 3, 3, 3]

    // 요소 접근
    let first = a[0];
    let second = a[1];
    println!("first: {}, second: {}", first, second);

    // 배열 순회
    for element in a {
        print!("{} ", element);
    }
    println!();
}
```

배열 vs Vec: 언제 무엇을 쓸까?

상황	배열 `[T; N]`	Vec `Vec<T>`
크기	컴파일 타임 고정	런타임에 가변
저장 위치	스택	힙
성능	약간 빠름	유연함
사용 예	월 이름, 고정 설정	사용자 입력, 동적 데이터

2.3 함수 (Functions)

함수는 Rust 프로그램의 핵심 구성 요소입니다. `fn` 키워드로 정의하며, **snake_case** 네이밍 규칙을 따릅니다.

2.3.1 기본 함수 정의

```rust
fn main() {
    println!("Hello from main!");

    another_function();
    greet("Rustacean");
}

fn another_function() {
    println!("Another function called!");
}

fn greet(name: &str) {
    println!("Hello, {}!", name);
}
```

2.3.2 매개변수 (Parameters)

모든 매개변수는 **타입 명시가 필수**입니다.

```rust
fn print_labeled_measurement(value: i32, unit_label: char) {
    println!("측정값: {}{}", value, unit_label);
}

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

2.3.3 구문(Statement) vs 표현식(Expression)

Rust는 **표현식 기반(Expression-based)** 언어입니다. 이 구분은 매우 중요합니다!

구분	설명	예시	값 반환
구문	동작을 수행	`let y = 6;`	❌
표현식	값으로 평가	`5 + 6`, `{ x + 1 }`	✅

```rust
fn main() {
    // 구문: 값을 반환하지 않음
    let x = 5;  // let 구문

    // ❌ 이것은 에러! (let은 값을 반환하지 않음)
    // let y = (let z = 6);

    // 표현식: 값을 반환함
    let y = {
        let x = 3;
        x + 1  // 세미콜론 없음 = 표현식 = 이 블록의 반환값
    };

    println!("y = {}", y);  // y = 4

    // 세미콜론을 붙이면 구문이 됨 (유닛 타입 반환)
    let z = {
        let x = 3;
        x + 1;  // 세미콜론 있음 = 구문
    };
    println!("z = {:?}", z);  // z = () (유닛 타입)
}
```

2.3.4 반환 값이 있는 함수

```rust
fn five() -> i32 {
    5  // 세미콜론 없음 = 이 값을 반환
}

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

fn main() {
    let x = five();
    println!("five() = {}", x);  // 5

    let y = plus_one(5);
    println!("plus_one(5) = {}", y);  // 6
}
```

`return` 키워드: 조기 반환 시 사용

```rust
fn absolute_value(x: i32) -> i32 {
    if x < 0 {
        return -x;  // 조기 반환
    }
    x  // 마지막 표현식 (암묵적 반환)
}

fn main() {
    println!("|5| = {}", absolute_value(5));    // 5
    println!("|-5| = {}", absolute_value(-5));  // 5
}
```

2.4 제어 흐름 (Control Flow)

2.4.1 조건문: `if` 표현식

Rust의 `if`는 표현식입니다. 따라서 값을 반환할 수 있습니다!

```
fn main() {
    let number = 7;

    // 기본 if-else
    if number < 5 {
        println!("조건이 참");
    } else {
        println!("조건이 거짓");
    }

    // else if 체인
    if number % 4 == 0 {
        println!("4로 나누어 떨어짐");
    } else if number % 3 == 0 {
        println!("3으로 나누어 떨어짐");
    } else if number % 2 == 0 {
        println!("2로 나누어 떨어짐");
    } else {
        println!("2, 3, 4로 나누어 떨어지지 않음");
    }
}
```

if를 표현식으로 사용

```
fn main() {
    let condition = true;
    let number = if condition { 5 } else { 6 };

    println!("number = {}", number);  // 5

    // 주의: 양쪽 타입이 같아야 함!
    // let error = if condition { 5 } else { "six" };  // ❌ 타입 불일치
}
```

2.4.2 반복문

Rust에는 세 가지 반복문이 있습니다: `loop`, `while`, `for`

`loop`: 무한 루프

```
fn main() {
    let mut counter = 0;

    // loop는 값을 반환할 수 있음!
    let result = loop {
        counter += 1;

        if counter == 10 {
            break counter * 2;  // 값을 반환하며 탈출
        }
    };

    println!("결과: {}", result);  // 20
}
```

레이블을 사용한 중첩 루프 탈출

```rust
fn main() {
    let mut count = 0;

    'counting_up: loop {  // 루프 레이블
        println!("count = {}", count);
        let mut remaining = 10;

        loop {
            println!("remaining = {}", remaining);
            if remaining == 9 {
                break;  // 내부 루프만 탈출
            }
            if count == 2 {
                break 'counting_up;  // 외부 루프까지 탈출
            }
            remaining -= 1;
        }

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

`while`: 조건 루프

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

    while number != 0 {
        println!("{}!", number);
        number -= 1;
    }

    println!("LIFTOFF!");
}
```

`for`: 컬렉션 순회 (가장 많이 사용!)

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

    // 배열 순회
    for element in a {
        println!("값: {}", element);
    }

    // 범위 순회
    for number in 1..4 {  // 1, 2, 3 (4 미포함)
        println!("{}", number);
    }

    // 범위 (끝 포함)
    for number in 1..=3 {  // 1, 2, 3
        println!("{}", number);
    }

    // 역순
    for number in (1..4).rev() {  // 3, 2, 1
        println!("{}!", number);
    }

    // 인덱스와 함께
    for (index, value) in a.iter().enumerate() {
        println!("a[{}] = {}", index, value);
    }
}
```

🚀 실습 과제

다음 함수들을 구현해보세요:

1. **피보나치 수열**: n번째 피보나치 수를 반환하는 함수
2. **화씨↔섭씨 변환기**: 온도를 변환하는 함수
3. **FizzBuzz**: 1부터 100까지 출력하되, 3의 배수는 “Fizz”, 5의 배수는 “Buzz”, 15의 배수는 “FizzBuzz”

```
// 힌트: 피보나치
fn fibonacci(n: u32) -> u32 {
    // 구현해보세요!
    todo!()
}

// 힌트: 온도 변환
fn celsius_to_fahrenheit(c: f64) -> f64 {
    // 공식: F = C × 9/5 + 32
    todo!()
}
```

Chapter 3: 소유권 (Understanding Ownership)

> "소유권은 Rust를 **특별**하게 만드는 핵심 개념입니다. 가비지 컬렉터 없이도 메모리 안전성을 보장하며, 프로그램 실행 속도에 전혀 영향을 주지 않습니다."

소유권을 이해하면 Rust의 나머지 기능들도 자연스럽게 이해됩니다. 이 장은 Rust 학습에서 **가장 중요한 장**입니다. 천천히, 그리고 반복해서 읽어주세요.

3.1 메모리 관리의 역사: 왜 소유권인가?

3.1.1 세 가지 메모리 관리 방식

프로그래밍 역사에서 메모리 관리는 항상 어려운 문제였습니다. 크게 세 가지 접근법이 있습니다:

방식	언어	장점	단점
수동 관리	C, C++	최고의 성능, 완전한 제어	버그 다발 (메모리 누수, 댕글링 포인터, 이중 해제)
가비지 컬렉션 (GC)	Java, Python, Go	편리함, 안전함	런타임 오버헤드, 예측 불가한 멈춤(Stop-the-world)
소유권 시스템	**Rust**	안전함 + 성능	학습 곡선

수동 관리의 위험성 (C 예시)

```c
// C 코드 - 이중 해제(Double Free) 버그
char* ptr = malloc(100);
free(ptr);
free(ptr);  // 💀 위험! 이중 해제 → 보안 취약점
```

```c
// C 코드 - 댕글링 포인터(Dangling Pointer)
char* get_string() {
    char buffer[100] = "Hello";
    return buffer;  // 💀 스택 메모리 반환 → 스코프 종료 후 무효!
}
```

Rust의 소유권 시스템은 이러한 버그들을 **컴파일 타임에 완전히 차단**합니다.

3.1.2 스택(Stack)과 힙(Heap)

소유권을 이해하려면 먼저 스택과 힙을 알아야 합니다.

스택 (Stack)

- **LIFO** (Last In, First Out) 구조
- 데이터 크기가 **컴파일 타임에 고정**되어야 함
- 매우 빠름 (포인터만 이동)
- 함수 호출 시 지역 변수 저장

```rust
fn main() {
    let x: i32 = 5;          // 스택에 4바이트 할당
    let y: f64 = 3.14;       // 스택에 8바이트 할당
    let arr: [i32; 3] = [1, 2, 3];  // 스택에 12바이트 할당
}  // 스코프 종료: 스택에서 자동 정리 (역순)
```

힙 (Heap)

- 런타임에 크기가 결정되는 데이터
- 메모리 할당자(Allocator)가 빈 공간을 찾아 할당
- 접근이 느림 (포인터 역참조 필요)
- `String`, `Vec`, `Box` 등이 힙 사용

```rust
fn main() {
    // String은 힙에 데이터를 저장
    let s = String::from("hello");

    // 스택: [ptr, len, capacity] (24바이트 정도)
    // 힙: 실제 문자열 데이터 "hello" (5바이트)

    println!("{}", s);
}
```

메모리 레이아웃 시각화:

```
스택                    힙
+---------------+       +---+---+---+---+---+
| ptr --------->|------>| h | e | l | l | o |
| len: 5        |       +---+---+---+---+---+
| capacity: 5   |
+---------------+
```

3.2 소유권 규칙 (The Ownership Rules)

> ⚠️ 이 세 가지 규칙을 반드시 암기하세요! Rust의 모든 것이 여기서 시작됩니다.

규칙 1: 각 값은 **소유자(Owner)**를 가진다

```rust
fn main() {
    let s = String::from("hello");  // s가 이 String의 소유자
}
```

규칙 2: 소유자는 단 하나뿐이다

```rust
fn main() {
    let s1 = String::from("hello");
    let s2 = s1;  // 소유권이 s1에서 s2로 이동!

    // println!("{}", s1);  // ❌ 에러! s1은 더 이상 유효하지 않음
    println!("{}", s2);     // ✅ OK
}
```

규칙 3: 소유자가 스코프를 벗어나면 값이 버려진다(drop)

```
fn main() {
    {
        let s = String::from("hello");
        println!("{}", s);
    }  // s가 스코프를 벗어남 → drop 호출 → 힙 메모리 해제

    // println!("{}", s);  // ✗ 에러! s는 존재하지 않음
}
```

3.3 이동(Move)과 복사(Copy)

3.3.1 이동 (Move) - 힙 데이터의 기본 동작

핵심 개념: Rust에서 대입은 기본적으로 **이동(Move)**입니다.

```
fn main() {
    let s1 = String::from("hello");
    let s2 = s1;  // s1의 스택 데이터가 s2로 복사되고, s1은 무효화됨

    // 왜 이렇게 설계했을까?
    // 만약 s1과 s2가 같은 힙 데이터를 가리킨다면...
    // 둘 다 스코프를 벗어날 때 같은 메모리를 두 번 해제하게 됨!
    // → 이중 해제(Double Free) 버그!

    // Rust는 이를 방지하기 위해 s1을 무효화한다.
}
```

이동 전후 메모리 다이어그램:

```
이동 전:
s1: [ptr|len|cap] ---> 힙: "hello"
s2: (존재하지 않음)

이동 후:
s1: (무효화됨)
s2: [ptr|len|cap] ---> 힙: "hello"
```

3.3.2 복사 (Copy) - 스택 데이터의 동작

정수, 부동소수점, 불리언 등 **스택에만 저장되는** 단순 타입은 `Copy` 트레이트를 구현합니다.

```rust
fn main() {
    let x = 5;
    let y = x;  // 복사! (x는 여전히 유효)

    println!("x = {}, y = {}", x, y);  // ✅ 둘 다 사용 가능

    // 이유: 정수는 스택에 저장되고, 복사 비용이 거의 없음
}
```

Copy 가능한 타입들:

- 모든 정수 타입 (`i32`, `u64`, 등)
- 부동소수점 (`f32`, `f64`)
- 불리언 (`bool`)
- 문자 (`char`)
- Copy 타입으로만 구성된 튜플 (`(i32, i32)` ✅, `(i32, String)` ❌)

3.3.3 Clone - 명시적 깊은 복사

힙 데이터를 복사하고 싶다면 `clone()`을 사용합니다.

```
fn main() {
    let s1 = String::from("hello");
    let s2 = s1.clone();  // 힙 데이터까지 복사!

    println!("s1 = {}, s2 = {}", s1, s2);  // ✅ 둘 다 유효

    // 메모리 상태:
    // s1 ---> 힙: "hello" (원본)
    // s2 ---> 힙: "hello" (복사본, 별도 메모리)
}
```

> ⚠️ `clone()`은 **비용이 있습니다**. 힙 데이터 전체를 복사하므로 필요할 때만 사용하세요.

3.4 소유권과 함수

3.4.1 함수에 값 전달하기

함수에 값을 전달하면 소유권도 함께 **이동**하거나 **복사**됩니다.

```rust
fn main() {
    let s = String::from("hello");
    takes_ownership(s);           // s의 소유권이 함수로 이동

    // println!("{}", s);         // ❌ 에러! s는 더 이상 유효하지 않음

    let x = 5;
    makes_copy(x);                // x는 Copy이므로 복사됨

    println!("x = {}", x);        // ✅ 여전히 사용 가능
}

fn takes_ownership(some_string: String) {
    println!("받은 문자열: {}", some_string);
}  // some_string이 스코프를 벗어나며 drop됨

fn makes_copy(some_integer: i32) {
    println!("받은 정수: {}", some_integer);
}
```

3.4.2 반환 값으로 소유권 돌려주기

```
fn main() {
    let s1 = gives_ownership();         // 반환값으로 소유권을 받음

    let s2 = String::from("hello");
    let s3 = takes_and_gives_back(s2);  // s2를 주고 s3로 받음

    // println!("{}", s2);               // ❌ s2는 이동됨
    println!("{}", s3);                  // ✅ s3는 유효
}

fn gives_ownership() -> String {
    let some_string = String::from("yours");
    some_string  // 소유권을 반환
}

fn takes_and_gives_back(a_string: String) -> String {
    a_string  // 받은 소유권을 그대로 반환
}
```

3.4.3 튜플로 여러 값 반환하기

소유권을 돌려받으면서 추가 정보도 반환하고 싶다면:

```rust
fn main() {
    let s1 = String::from("hello");
    let (s2, len) = calculate_length(s1);

    println!("'{}' 의 길이는 {}", s2, len);
}

fn calculate_length(s: String) -> (String, usize) {
    let length = s.len();
    (s, length)  // 소유권과 길이를 함께 반환
}
```

🤔 매번 이렇게 소유권을 주고받는 것은 번거롭습니다. 다음 절에서 **참조(Reference)**를 배워 이 문제를 해결합니다!

3.5 빌림(Borrowing)과 참조(References)

3.5.1 참조란?

참조(Reference)는 소유권을 넘기지 않고 값을 “빌려” 사용하는 방법입니다.

```rust
fn main() {
    let s1 = String::from("hello");
    let len = calculate_length(&s1);  // &s1: s1에 대한 참조

    println!("'{}' 의 길이는 {}", s1, len);  // ✅ s1은 여전히 유효!
}

fn calculate_length(s: &String) -> usize {  // &String: String
에 대한 참조
    s.len()
}  // s가 스코프를 벗어남. 하지만 참조이므로 drop되지 않음!
```

참조 메모리 다이어그램:

```
스택                          힙
s1: [ptr|len|cap] ----------> "hello"
        ↑
       &s1 (참조)
```

3.5.2 불변 참조 (`&T`)

기본적으로 참조는 **불변**입니다. 빌린 값을 수정할 수 없습니다.

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

    change(&s);  // 불변 참조 전달
}

fn change(some_string: &String) {
    // some_string.push_str(", world");  // ❌ 에러! 불변 참조로는 수정 불가
    println!("읽기만 가능: {}", some_string);
}
```

3.5.3 가변 참조 (`&mut T`)

값을 수정하려면 가변 참조가 필요합니다.

```rust
fn main() {
    let mut s = String::from("hello");  // 원본도 mut이어야 함!

    change(&mut s);

    println!("{}", s);  // "hello, world"
}

fn change(some_string: &mut String) {
    some_string.push_str(", world");  // ✅ 가변 참조로 수정 가능
}
```

3.5.4 참조의 규칙 (The Rules of References)

> 🏛️ 도서관 비유: 이 규칙들을 도서관에 비유하면 쉽게 이해할 수 있습니다.

규칙 1: 불변 참조는 여러 개 존재 가능

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

    let r1 = &s;  // ✅ OK
    let r2 = &s;  // ✅ OK
    let r3 = &s;  // ✅ OK - 여러 명이 동시에 책을 읽을 수 있음

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

도서관 비유: 열람실의 책은 수십 명이 동시에 읽을 수 있습니다. 하지만 누구도 책에 낙서하거나 찢을 수 없습니다.

규칙 2: 가변 참조는 단 하나만 존재 가능

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

    let r1 = &mut s;
    // let r2 = &mut s;  // ❌ 에러! 두 개의 가변 참조 불가

    println!("{}", r1);
}
```

도서관 비유: 편집자가 원고를 수정하러 가져갔다면, 다른 누구도 그 책을 읽거나 수정할 수 없습니다 (배타적 접근).

규칙 3: 불변 참조와 가변 참조는 동시에 존재 불가

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

    let r1 = &s;       // ✅ 불변 참조
    let r2 = &s;       // ✅ 불변 참조
    // let r3 = &mut s;  // ❌ 불변 참조가 있는 동안 가변 참조 불가!

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

이 규칙의 목적: 컴파일 타임에 **데이터 레이스(Data Race)**를 방지합니다.

데이터 레이스 발생 조건:

1. 둘 이상의 포인터가 동시에 같은 데이터에 접근
2. 그중 하나 이상이 쓰기(write) 작업
3. 동기화 메커니즘 없음

Rust는 **"쓰기 작업 시 배타적 접근만 허용"**하여 이를 차단합니다.

3.5.5 Non-Lexical Lifetimes (NLL)

Rust 2018 에디션부터, 참조의 유효 범위가 더 정밀해졌습니다.

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

    let r1 = &s;
    let r2 = &s;
    println!("{} and {}", r1, r2);
    // r1과 r2는 여기서 마지막으로 사용됨 (더 이상 필요 없음)

    let r3 = &mut s;  // ✅ OK! r1, r2가 더 이상 사용되지 않으므로
    println!("{}", r3);
}
```

3.5.6 댕글링 참조 방지

Rust는 **댕글링 참조(Dangling Reference)**를 컴파일 타임에 방지합니다.

```rust
fn main() {
    // let reference_to_nothing = dangle();  // ❌ 컴파일 에러
    let valid_string = no_dangle();
    println!("{}", valid_string);
}

// ❌ 이 코드는 컴파일되지 않습니다!
// fn dangle() -> &String {
//     let s = String::from("hello");
//     &s  // s가 함수 끝에서 drop됨 → 참조가 무효해짐!
// }

fn no_dangle() -> String {
    let s = String::from("hello");
    s  // 소유권을 이동시켜 반환 (댕글링 아님)
}
```

3.6 슬라이스 (Slices)

슬라이스는 컬렉션의 **연속된 일부분**을 참조합니다. 소유권을 갖지 않습니다.

3.6.1 문자열 슬라이스 (`&str`)

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

    // 슬라이스 문법: &변수[시작..끝]
    let hello = &s[0..5];   // "hello"
    let world = &s[6..11];  // "world"

    println!("{} {}", hello, world);

    // 문법 설탕 (Syntactic Sugar)
    let slice1 = &s[..5];   // 처음부터: &s[0..5]
    let slice2 = &s[6..];   // 끝까지: &s[6..len]
    let slice3 = &s[..];    // 전체: &s[0..len]
}
```

3.6.2 슬라이스와 안전성

슬라이스는 **원본 데이터의 무효화를 방지**합니다.

```rust
fn main() {
    let mut s = String::from("hello world");
    let word = first_word(&s);  // 불변 빌림

    // s.clear();  // ❌ 에러! 불변 빌림 중에 가변 접근 불가

    println!("첫 단어: {}", word);

    // word 사용이 끝난 후에야 수정 가능
    s.clear();  // ✅ OK
}

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

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

    &s[..]
}
```

3.6.3 배열 슬라이스

```
fn main() {
    let a = [1, 2, 3, 4, 5];
    let slice = &a[1..3];  // [2, 3]

    println!("{:?}", slice);

    assert_eq!(slice, &[2, 3]);
}
```

3.6.4 `&String` vs `&str` - 어느 것을 사용할까?

```
// &str을 받으면 더 유연함!
fn process(s: &str) {
    println!("{}", s);
}

fn main() {
    let owned = String::from("hello");
    let literal = "world";

    process(&owned);    // ✅ String에서 &str로 자동 변환
    process(literal);   // ✅ &str 직접 전달
    process(&owned[0..2]);  // ✅ 슬라이스도 가능
}
```

> 팁: 함수 매개변수로 `&String` 대신 `&str`을 사용하면 `String`과 `&str` 모두 받을 수 있어 더 유연합니다.

3.7 소유권 정리: 한눈에 보기

상황	동작	예시
스택 타입 대입	복사	`let y = x;` (x는 여전히 유효)
힙 타입 대입	이동	`let s2 = s1;` (s1 무효화)
함수에 전달	복사 또는 이동	타입에 따라 다름
참조 전달	빌림	`fn foo(s: &String)`
가변 참조 전달	가변 빌림	`fn foo(s: &mut String)`
`.clone()` 호출	깊은 복사	`let s2 = s1.clone();`

🚀 실습 과제

1. **문자열 역순**: 문자열을 받아 역순으로 반환하는 함수 작성
2. **단어 개수 세기**: 문자열에서 단어 개수를 반환하는 함수 작성
3. **로그 분석기**: 로그 메시지에서 에러만 필터링하는 함수 작성

```rust
// 힌트 1: 역순
fn reverse(s: &str) -> String {
    s.chars().rev().collect()
}

// 힌트 2: 단어 개수
fn word_count(s: &str) -> usize {
    s.split_whitespace().count()
}
```

Chapter 4: 구조체로 데이터 구조화하기 (Using Structs)

> "구조체는 관련된 데이터를 하나로 묶어 의미를 부여하는 사용자 정의 타입입니다. 도메인 모델링의 핵심 도구이며, 메서드를 통해 데이터와 동작을 함께 캡슐화합니다."

왜 구조체가 필요한가?

프로그래밍에서 관련 있는 데이터를 함께 다루는 것은 매우 중요합니다. 예를 들어, 사용자 정보를 표현한다고 생각해 봅시다:

```
// ❌ 나쁜 예: 관련 데이터가 분리됨
let username = String::from("alice");
let email = String::from("alice@example.com");
let active = true;
let sign_in_count = 1;

// 함수에 전달할 때 모든 변수를 개별적으로 전달해야 함
fn print_user(username: String, email: String, active: bool,
count: u64) {
    // ...
}
```

위 코드의 문제점:

- 변수들 사이의 **관계가 명시적이지 않음**
- 함수에 여러 인수를 전달해야 하여 **실수 가능성** 증가

- 새 필드 추가 시 **모든 관련 코드 수정** 필요

구조체를 사용하면 이 모든 문제가 해결됩니다:

```
// ✅ 좋은 예: 관련 데이터가 하나의 단위로 묶임
struct User {
    username: String,
    email: String,
    active: bool,
    sign_in_count: u64,
}

fn print_user(user: &User) {
    println!("사용자: {}", user.username);
}
```

4.1 구조체 정의와 인스턴스화

기본 구조체 정의

구조체는 `struct` 키워드로 정의합니다. 각 필드는 이름과 타입을 가집니다:

```rust
// 구조체 정의: 사용자 정보를 표현하는 타입
struct User {
    active: bool,         // 활성 상태
    username: String,     // 사용자명
    email: String,        // 이메일
    sign_in_count: u64,   // 로그인 횟수
}

fn main() {
    // 인스턴스 생성: 모든 필드를 초기화해야 함
    let user1 = User {
        active: true,
        username: String::from("someuser123"),
        email: String::from("someone@example.com"),
        sign_in_count: 1,
    };

    // 필드 접근: 점(.) 표기법 사용
    println!("사용자명: {}", user1.username);
    println!("이메일: {}", user1.email);
    println!("활성 상태: {}", user1.active);
    println!("로그인 횟수: {}", user1.sign_in_count);
}
```

📌 중요: 구조체 인스턴스를 생성할 때 모든 필드를 초기화해야 합니다. 하나라도 빠지면 컴파일 에러가 발생합니다. 이는 Rust의 "초기화되지 않은 메모리 접근 방지" 철학의 일부입니다.

필드 순서

필드 초기화 순서는 정의 순서와 달라도 됩니다:

```rust
let user2 = User {
    email: String::from("user2@example.com"),  // 순서 변경 가능
    active: false,
    sign_in_count: 0,
    username: String::from("user2"),
};
```

가변 구조체

필드를 수정하려면 전체 인스턴스가 **가변(mut)**이어야 합니다:

```rust
fn main() {
    // mut 키워드로 가변 인스턴스 선언
    let mut user = User {
        active: true,
        username: String::from("alice"),
        email: String::from("alice@example.com"),
        sign_in_count: 1,
    };

    // 필드 수정 가능
    user.email = String::from("newemail@example.com");
    user.sign_in_count += 1;

    println!("새 이메일: {}", user.email);
    println!("로그인 횟수: {}", user.sign_in_count);
}
```

⚠️ **주의**: Rust는 **특정 필드만 가변으로 만드는 것을 허용하지 않습니다**. 인스턴스 전체가 가변이거나 불변이어야 합니다. 이는 데이터 일관성을 보장하기 위한 설계입니다.

가변성과 불변성 비교

특성	불변 인스턴스	가변 인스턴스
선언	`let user = User {...}`	`let mut user = User {...}`
필드 읽기	✅ 가능	✅ 가능
필드 수정	❌ 불가능	✅ 가능
스레드 안전성	더 안전	주의 필요

필드 초기화 축약 문법 (Field Init Shorthand)

변수명과 필드명이 같으면 축약할 수 있습니다:

```rust
fn build_user(email: String, username: String) -> User {
    // 축약 전
    // User {
    //     email: email,
    //     username: username,
    //     active: true,
    //     sign_in_count: 1,
    // }

    // 축약 후: 변수명과 필드명이 같으면 생략 가능
    User {
        email,              // email: email 축약
        username,           // username: username 축약
        active: true,       // 이것은 축약 불가 (변수가 아니라 리터럴)
        sign_in_count: 1,
    }
}

fn main() {
    let user = build_user(
        String::from("alice@example.com"),
        String::from("alice"),
    );
    println!("생성된 사용자: {}", user.username);
}
```

이 축약 문법은 빌더 패턴이나 팩토리 함수에서 특히 유용합니다.

구조체 갱신 문법 (Struct Update Syntax)

기존 인스턴스를 기반으로 새 인스턴스를 만들 때 `..` 문법을 사용합니다:

```rust
fn main() {
    let user1 = User {
        active: true,
        username: String::from("user1"),
        email: String::from("user1@example.com"),
        sign_in_count: 1,
    };

    // user1을 기반으로 user2 생성
    // email만 다르고 나머지는 user1에서 가져옴
    let user2 = User {
        email: String::from("user2@example.com"),
        ..user1  // 나머지 필드는 user1에서 복사
    };

    println!("user2 이메일: {}", user2.email);
    println!("user2 활성 상태: {}", user2.active);
}
```

⚠️ 소유권 주의: `..user1` 구문은 소유권 이동을 발생시킬 수 있습니다!

```rust
fn main() {
    let user1 = User {
        active: true,
        username: String::from("user1"),
        email: String::from("user1@example.com"),
        sign_in_count: 1,
    };

    let user2 = User {
        email: String::from("user2@example.com"),
        ..user1
    };

    // ❌ 컴파일 에러! user1.username이 user2로 이동됨
    // println!("{}", user1.username);

    // ✅ Copy 타입 필드는 여전히 사용 가능
    println!("user1.active: {}", user1.active);      // bool은 Copy
    println!("user1.sign_in_count: {}", user1.sign_in_count);
// u64도 Copy
}
```

소유권 이동 규칙

타입	갱신 문법 사용 시 동작	원본 사용 가능?
`bool`, `u64` 등 (Copy)	복사	✅ 가능
`String`, `Vec<T>` 등	이동	❌ 불가능

4.2 튜플 구조체 (Tuple Structs)

필드에 이름을 붙이지 않고, 타입에만 이름을 부여하고 싶을 때 사용합니다.

```
// 튜플 구조체 정의
struct Color(u8, u8, u8);       // RGB 색상
struct Point(i32, i32, i32);    // 3D 좌표

fn main() {
    let red = Color(255, 0, 0);
    let origin = Point(0, 0, 0);

    // 인덱스로 접근 (0부터 시작)
    println!("Red: R={}, G={}, B={}", red.0, red.1, red.2);
    println!("Origin: x={}, y={}, z={}", origin.0, origin.1,
origin.2);
}
```

Newtype 패턴

튜플 구조체의 중요한 특성: 내부 구조가 같아도 다른 타입으로 취급됩니다.

```
struct Meters(f64);
struct Kilometers(f64);

fn main() {
    let distance_m = Meters(1000.0);
    let distance_km = Kilometers(1.0);

    // ❌ 컴파일 에러! 타입이 다름
    // let sum = distance_m.0 + distance_km.0;  // 이건 가능하지
만...
    // let wrong: Meters = distance_km;  // 이건 불가능!

    // ✅ 명시적 변환 필요
    let km_in_meters = Meters(distance_km.0 * 1000.0);
}
```

💡 **Newtype 패턴의 장점**:

- 단위 실수 방지 (미터를 킬로미터로 잘못 사용하는 것 방지)
- 의미론적 명확성 (코드의 의도 전달)
- 추가 검증 로직 캡슐화 가능

실제 활용 예시: 강타입 ID

```
// 데이터베이스 ID를 구분하는 Newtype
struct UserId(u64);
struct OrderId(u64);

fn get_user(id: UserId) -> String {
    format!("사용자 #{}", id.0)
}

fn main() {
    let user_id = UserId(123);
    let order_id = OrderId(456);

    println!("{}", get_user(user_id));

    // ❌ 컴파일 에러! OrderId는 UserId가 아님
    // println!("{}", get_user(order_id));
}
```

4.3 유닛 구조체 (Unit-like Structs)

필드가 없는 구조체입니다. 주로 트레이트 구현에 사용됩니다.

```
// 필드 없는 구조체
struct AlwaysEqual;

fn main() {
    let subject = AlwaysEqual;
    // 타입 자체가 의미를 가짐
}
```

실제 사용 예시: 마커 타입

```rust
// 상태를 나타내는 마커 타입
struct Pending;
struct Approved;
struct Rejected;

struct Document<State> {
    content: String,
    _state: State,  // 마커 (실제 데이터 없음)
}

impl Document<Pending> {
    fn approve(self) -> Document<Approved> {
        Document {
            content: self.content,
            _state: Approved,
        }
    }
}

impl Document<Approved> {
    fn publish(&self) {
        println!("문서 게시: {}", self.content);
    }
}

fn main() {
    let doc = Document {
        content: String::from("중요한 내용"),
        _state: Pending,
    };

    // 승인 후에만 게시 가능
    let approved_doc = doc.approve();
    approved_doc.publish();
```

```
    // ❌ Pending 상태에서는 publish 불가
    // doc.publish();  // 메서드가 없음!
}
```

4.4 메서드 정의: `impl` 블록

Rust의 구조체는 **데이터(struct)**와 **동작(impl)**이 분리되어 있습니다.

기본 메서드 정의

```
#[derive(Debug)]
struct Rectangle {
    width: u32,
    height: u32,
}

impl Rectangle {
    // 메서드: &self를 첫 번째 인자로 받음
    fn area(&self) -> u32 {
        self.width * self.height
    }

    // 다른 인스턴스와 비교
    fn can_hold(&self, other: &Rectangle) -> bool {
        self.width > other.width && self.height > other.height
    }

    // 가변 메서드: &mut self 사용
    fn double_size(&mut self) {
        self.width *= 2;
        self.height *= 2;
    }
}

fn main() {
    let rect1 = Rectangle { width: 30, height: 50 };
    let rect2 = Rectangle { width: 10, height: 40 };

    println!("사각형 면적: {}", rect1.area());
    println!("rect1이 rect2를 담을 수 있나? {}",
rect1.can_hold(&rect2));
}
```

self의 세 가지 형태

형태	의미	사용 사례
`&self`	불변 참조	읽기 전용 메서드
`&mut self`	가변 참조	인스턴스 수정
`self`	소유권 가져감	변환 메서드, 소비 패턴

```rust
impl Rectangle {
    // &self: 읽기만 함
    fn area(&self) -> u32 {
        self.width * self.height
    }

    // &mut self: 수정함
    fn scale(&mut self, factor: u32) {
        self.width *= factor;
        self.height *= factor;
    }

    // self: 소유권을 가져감 (원본 사용 불가)
    fn into_square(self) -> Rectangle {
        let side = std::cmp::max(self.width, self.height);
        Rectangle { width: side, height: side }
    }
}

fn main() {
    let mut rect = Rectangle { width: 30, height: 50 };

    println!("면적: {}", rect.area());          // &self
    rect.scale(2);                               // &mut self
    println!("확대 후 면적: {}", rect.area());

    let square = rect.into_square();             // self (rect 소비)
    // println!("{:?}", rect);  // ❌ 에러! rect는 이동됨
    println!("정사각형: {:?}", square);
}
```

연관 함수 (Associated Functions)

`self`를 받지 않는 함수입니다. 주로 생성자로 사용됩니다.

```
impl Rectangle {
    // 연관 함수: self 없음, :: 으로 호출
    fn new(width: u32, height: u32) -> Self {
        Self { width, height }
    }

    fn square(size: u32) -> Self {
        Self {
            width: size,
            height: size,
        }
    }

    // 기본값으로 생성
    fn default() -> Self {
        Self { width: 1, height: 1 }
    }
}

fn main() {
    // 연관 함수 호출: :: 사용
    let rect = Rectangle::new(30, 50);
    let square = Rectangle::square(25);
    let default = Rectangle::default();

    println!("사각형: {:?}", rect);
    println!("정사각형: {:?}", square);
    println!("기본값: {:?}", default);
}
```

메서드 vs 연관 함수

특성	메서드	연관 함수
첫 번째 인자	`self`, `&self`, `&mut self`	없음
호출 문법	`인스턴스.메서드()`	`타입::함수()`
주요 용도	인스턴스 작업	생성자, 유틸리티

자동 참조와 역참조 (Automatic Referencing)

C/C++과 달리 Rust에서는 `object->method()` 문법이 없습니다. 대신 `.` 연산자가 자동으로 `&`, `&mut`, `*`를 추가합니다.

```
impl Rectangle {
    fn area(&self) -> u32 {
        self.width * self.height
    }
}

fn main() {
    let rect = Rectangle { width: 30, height: 50 };

    // 아래 두 호출은 완전히 동일합니다
    println!("방법 1: {}", rect.area());
    println!("방법 2: {}", (&rect).area());

    // Rust가 자동으로 &rect를 추가함!
}
```

> 💡 **이 기능이 작동하는 이유**: Rust는 메서드 시그니처를 보고 인스턴스에 적절한 `&`, `&mut`, `*`를 자동으로 추가합니다. 이것이 가능한 이유는 메서드가 `self`의 타입을 명확히 선언하기 때문입니다.

여러 `impl` 블록

동일한 타입에 대해 여러 `impl` 블록을 정의할 수 있습니다:

```rust
#[derive(Debug)]
struct Rectangle {
    width: u32,
    height: u32,
}

// 기본 메서드
impl Rectangle {
    fn area(&self) -> u32 {
        self.width * self.height
    }

    fn perimeter(&self) -> u32 {
        2 * (self.width + self.height)
    }
}

// 생성자들
impl Rectangle {
    fn new(width: u32, height: u32) -> Self {
        Self { width, height }
    }

    fn square(size: u32) -> Self {
        Self { width: size, height: size }
    }
}

// 변환 메서드
impl Rectangle {
    fn rotate(self) -> Self {
        Self {
            width: self.height,
            height: self.width,
        }
```

```
    }
}
```

여러 `impl` 블록으로 분리하면 제네릭이나 트레이트 구현 시 유용합니다.

4.5 `#[derive]`로 유용한 트레이트 추가하기

`#[derive]` 속성을 사용하면 컴파일러가 자동으로 트레이트를 구현해 줍니다.

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

fn main() {
    let p1 = Point { x: 1, y: 2 };
    let p2 = p1.clone();  // Clone

    println!("{:?}", p1);           // Debug
    println!("{:#?}", p1);          // Debug (예쁘게 출력)
    println!("같음: {}", p1 == p2); // PartialEq

    // Hash가 있으면 HashMap의 키로 사용 가능
    use std::collections::HashMap;
    let mut scores: HashMap<Point, i32> = HashMap::new();
    scores.insert(p1, 100);
}
```

주요 Derive 트레이트

트레이트	기능	예시
`Debug`	`{:?}` 포맷 출력	`println!("{:?}", value)`
`Clone`	`.clone()` 메서드	`let copy = original.clone()`
`Copy`	암묵적 복사 (스택 타입)	대입 시 자동 복사
`PartialEq`	`==`, `!=` 비교	`a == b`
`Eq`	완전 동등성	`HashMap` 키 가능
`Hash`	해시 계산	`HashMap`, `HashSet` 키
`Default`	기본값 생성	`Type::default()`
`PartialOrd`	`<`, `>`, `<=`, `>=`	부분 순서
`Ord`	완전 순서	`BTreeMap` 키, 정렬

Copy와 Clone의 차이

```rust
// Copy: 스택에 저장되는 간단한 타입만 가능
#[derive(Copy, Clone)]
struct Point2D {
    x: i32,  // Copy 타입
    y: i32,  // Copy 타입
}

// ❌ String은 Copy가 아니므로 이 구조체도 Copy 불가
// #[derive(Copy, Clone)]
// struct User {
//     name: String,  // Copy가 아님!
// }

// ✅ Clone만 가능
#[derive(Clone)]
struct User {
    name: String,
}

fn main() {
    let p1 = Point2D { x: 1, y: 2 };
    let p2 = p1;  // Copy! p1 여전히 사용 가능
    println!("p1: ({}, {})", p1.x, p1.y);

    let u1 = User { name: String::from("Alice") };
    let u2 = u1.clone();  // 명시적 clone 필요
    // let u3 = u1;  // Move! u1 사용 불가
}
```

4.6 도메인 주도 설계와 구조체

DDD 관점에서의 구조체

구조체는 도메인 모델을 코드로 표현하는 핵심 도구입니다:

```rust
// 값 객체 (Value Object): 불변, 동등성 비교
#[derive(Debug, Clone, PartialEq, Eq, Hash)]
struct Email(String);

impl Email {
    fn new(value: &str) -> Result<Self, &'static str> {
        if value.contains('@') {
            Ok(Email(value.to_string()))
        } else {
            Err("유효하지 않은 이메일 형식")
        }
    }

    fn as_str(&self) -> &str {
        &self.0
    }
}

// 엔티티 (Entity): 고유 ID로 식별
#[derive(Debug)]
struct User {
    id: UserId,
    email: Email,
    name: String,
}

#[derive(Debug, Clone, Copy, PartialEq, Eq, Hash)]
struct UserId(u64);

fn main() {
    let email = Email::new("alice@example.com").unwrap();
    let user = User {
        id: UserId(1),
        email,
        name: String::from("Alice"),
    };
```

```
    println!("사용자: {:?}", user);
}
```

빌더 패턴

복잡한 구조체 생성을 단순화합니다:

```rust
#[derive(Debug)]
struct Server {
    host: String,
    port: u16,
    max_connections: u32,
    timeout_ms: u64,
}

#[derive(Default)]
struct ServerBuilder {
    host: String,
    port: u16,
    max_connections: u32,
    timeout_ms: u64,
}

impl ServerBuilder {
    fn new() -> Self {
        ServerBuilder {
            host: String::from("localhost"),
            port: 8080,
            max_connections: 100,
            timeout_ms: 30000,
        }
    }

    fn host(mut self, host: &str) -> Self {
        self.host = host.to_string();
        self
    }

    fn port(mut self, port: u16) -> Self {
        self.port = port;
        self
    }

    fn max_connections(mut self, max: u32) -> Self {
```

```
        self.max_connections = max;
        self
    }

    fn timeout(mut self, ms: u64) -> Self {
        self.timeout_ms = ms;
        self
    }

    fn build(self) -> Server {
        Server {
            host: self.host,
            port: self.port,
            max_connections: self.max_connections,
            timeout_ms: self.timeout_ms,
        }
    }
}

fn main() {
    // 빌더 패턴으로 서버 구성
    let server = ServerBuilder::new()
        .host("0.0.0.0")
        .port(3000)
        .max_connections(500)
        .build();

    println!("서버 설정: {:?}", server);
}
```

4.7 메모리 레이아웃

구조체의 메모리 배치를 이해하면 성능 최적화에 도움이 됩니다:

```rust
use std::mem::size_of;

struct Example1 {
    a: u8,   // 1 바이트
    b: u64,  // 8 바이트
    c: u16,  // 2 바이트
}

struct Example2 {
    b: u64,  // 8 바이트 (가장 큰 것 먼저)
    c: u16,  // 2 바이트
    a: u8,   // 1 바이트
}

fn main() {
    println!("Example1 크기: {} 바이트", size_of::<Example1>());
    println!("Example2 크기: {} 바이트", size_of::<Example2>());
    // 패딩으로 인해 Example1이 더 클 수 있음!
}
```

`#[repr]` 속성

메모리 레이아웃을 직접 제어할 수 있습니다:

```rust
// C 스타일 레이아웃 (FFI에 필요)
#[repr(C)]
struct CCompatible {
    x: i32,
    y: i32,
}

// 최소 크기로 패킹
#[repr(packed)]
struct Packed {
    a: u8,
    b: u64,
}
```

🚀 실습 과제

과제 1: 은행 계좌 시스템

```rust
// TODO: Account 구조체 정의
// - id: u64
// - owner: String
// - balance: f64

// TODO: 메서드 구현
// - new(id, owner) -> Account (초기 잔액 0)
// - deposit(&mut self, amount) -> Result<(), &str>
// - withdraw(&mut self, amount) -> Result<(), &str>
// - transfer(&mut self, to: &mut Account, amount) ->
Result<(), &str>
```

과제 2: 2D 벡터 연산

```
// TODO: Vector2D 구조체 정의 (Copy, Clone 가능하게)
// - x: f64
// - y: f64

// TODO: 메서드 구현
// - new(x, y) -> Vector2D
// - magnitude(&self) -> f64  (벡터 크기)
// - normalize(&self) -> Vector2D  (단위 벡터)
// - dot(&self, other: &Vector2D) -> f64  (내적)
// - add(&self, other: &Vector2D) -> Vector2D  (덧셈)
```

🚀 실습 가이드

`examples/` 폴더에서 확인:

1. `struct_basics.rs`: 기본 구조체와 갱신 문법
2. `rectangles.rs`: 메서드와 연관 함수
3. `builder_pattern.rs`: 빌더 패턴 구현

📚 핵심 정리

개념	설명
구조체	관련 데이터를 묶는 사용자 정의 타입

개념	설명
튜플 구조체	이름 없는 필드, Newtype 패턴에 활용
유닛 구조체	필드 없음, 마커 타입으로 사용
`impl` 블록	메서드와 연관 함수 정의
`&self`	불변 참조로 인스턴스 접근
`&mut self`	가변 참조로 인스턴스 수정
`self`	소유권 가져감, 변환 메서드
`#[derive]`	트레이트 자동 구현
빌더 패턴	복잡한 생성 로직 단순화

Chapter 5: 열거형과 패턴 매칭 (Enums and Pattern Matching)

> “Rust의 enum은 단순한 상수 집합이 아닙니다. 각 variant에 데이터를 담을 수 있는 **대수적 데이터 타입(Algebraic Data Type)**으로, 타입 안전한 상태 모델링의 핵심 도구입니다.”

왜 열거형이 필요한가?

프로그래밍에서 “여러 가능한 상태 중 하나“를 표현해야 할 때가 많습니다:

```
// ❌ 나쁜 예: 문자열로 상태 표현
fn process_payment(method: &str) {
    match method {
        "credit" => { /* ... */ }
        "debit" => { /* ... */ }
        "cash" => { /* ... */ }
        _ => { /* 어떤 값이 올지 모름! */ }
    }
}

// 문제점:
// - 오타 발생 가능 ("credt" 등)
// - 컴파일러가 유효성 검사 불가
// - 새 방법 추가 시 모든 match 수정 필요 (누락 위험)
```

열거형을 사용하면 **타입 시스템이 유효성을 보장**합니다:

```rust
// ✅ 좋은 예: 열거형으로 상태 표현
enum PaymentMethod {
    Credit,
    Debit,
    Cash,
}

fn process_payment(method: PaymentMethod) {
    match method {
        PaymentMethod::Credit => { /* ... */ }
        PaymentMethod::Debit => { /* ... */ }
        PaymentMethod::Cash => { /* ... */ }
        // 모든 경우 처리 강제! 새 variant 추가 시 컴파일 에러
    }
}
```

5.1 열거형 정의하기

기본 열거형

가장 간단한 형태의 열거형입니다:

```rust
#[derive(Debug)]
enum Direction {
    North,
    South,
    East,
    West,
}

fn main() {
    let heading = Direction::North;

    println!("현재 방향: {:?}", heading);

    match heading {
        Direction::North => println!("북쪽으로 이동"),
        Direction::South => println!("남쪽으로 이동"),
        Direction::East => println!("동쪽으로 이동"),
        Direction::West => println!("서쪽으로 이동"),
    }
}
```

열거형의 크기

열거형의 크기는 **가장 큰 variant를 담을 수 있는 크기** + **태그(discriminant)**입니다:

```rust
use std::mem::size_of;

enum Simple {
    A,
    B,
    C,
}

fn main() {
    // 태그만 필요 (variant 구분용)
    println!("Simple 크기: {} 바이트", size_of::<Simple>());  // 보통 1 바이트
}
```

데이터를 담는 열거형 (Rust의 강력함!)

다른 언어와 달리, Rust의 enum은 **각 variant마다 서로 다른 타입과 양의 데이터**를 가질 수 있습니다:

```rust
#[derive(Debug)]
enum Message {
    Quit,                       // 데이터 없음 (유닛 variant)
    Move { x: i32, y: i32 },    // 익명 구조체 형태
    Write(String),              // 튜플 형태 (String 하나)
    ChangeColor(u8, u8, u8),    // 튜플 형태 (RGB)
}

fn main() {
    let m1 = Message::Quit;
    let m2 = Message::Move { x: 10, y: 20 };
    let m3 = Message::Write(String::from("hello"));
    let m4 = Message::ChangeColor(255, 0, 0);

    println!("{:?}", m1);  // Quit
    println!("{:?}", m2);  // Move { x: 10, y: 20 }
    println!("{:?}", m3);  // Write("hello")
    println!("{:?}", m4);  // ChangeColor(255, 0, 0)
}
```

왜 이것이 강력한가?

이 기능 덕분에 Rust의 enum은 **대수적 데이터 타입(ADT)**의 **합 타입(Sum Type)**을 표현합니다:

```rust
// 이 enum은 아래 struct들과 동등한 표현력을 가짐
struct QuitMessage;  // 유닛 구조체
struct MoveMessage { x: i32, y: i32 }
struct WriteMessage(String);  // 튜플 구조체
struct ChangeColorMessage(u8, u8, u8);

// 하지만 enum은 이들을 하나의 타입으로 통합!
// Vec<Message>처럼 다양한 종류의 메시지를 하나의 컬렉션에 담을 수 있음
```

메모리 레이아웃 비교

```
use std::mem::size_of;

enum Message {
    Quit,
    Move { x: i32, y: i32 },
    Write(String),
    ChangeColor(u8, u8, u8),
}

fn main() {
    // Message 크기 = max(variant 크기) + 태그
    println!("Message 크기: {} 바이트", size_of::<Message>());

    // 비교
    println!("String 크기: {} 바이트", size_of::<String>());  // 가장 큰 variant
    println!("(i32, i32) 크기: {} 바이트", size_of::<(i32, i32)>());
}
```

---

enum에 메서드 정의하기

구조체처럼 enum에도 `impl` 블록으로 메서드를 정의할 수 있습니다:

```rust
#[derive(Debug)]
enum Message {
    Quit,
    Move { x: i32, y: i32 },
    Write(String),
    ChangeColor(u8, u8, u8),
}

impl Message {
    fn call(&self) {
        match self {
            Message::Quit => {
                println!("프로그램 종료 요청");
            }
            Message::Move { x, y } => {
                println!("({}, {})로 이동", x, y);
            }
            Message::Write(text) => {
                println!("메시지 출력: {}", text);
            }
            Message::ChangeColor(r, g, b) => {
                println!("색상 변경: RGB({}, {}, {})", r, g, b);
            }
        }
    }

    // variant 판별 메서드
    fn is_quit(&self) -> bool {
        matches!(self, Message::Quit)
    }

    // 변환 메서드
    fn into_write_text(self) -> Option<String> {
        match self {
            Message::Write(text) => Some(text),
            _ => None,
        }
```

```
    }
}

fn main() {
    let messages = vec![
        Message::Move { x: 10, y: 20 },
        Message::Write(String::from("Hello")),
        Message::Quit,
    ];

    for msg in &messages {
        msg.call();
    }
}
```

5.2 Option 열거형: Null의 대안

10억 달러의 실수

> “null 참조를 발명한 것은 10억 달러의 실수였다.” - Tony Hoare (null의 발명자)

많은 언어에서 `null`은 “값이 없음“을 표현하지만, 심각한 문제가 있습니다:

```
// C/C++의 위험한 코드
char* name = get_user_name();  // null일 수 있음
printf("Hello, %s\n", name);    // 💥 null이면 Segmentation Fault!
```

```
// Java의 NullPointerException
String name = getUserName();  // null일 수 있음
System.out.println(name.length());  // 💥 NPE!
```

Rust의 해결책: Option<T>

Rust에는 **null이 없습니다**. 대신 `Option<T>`을 사용합니다:

```
// 표준 라이브러리에 정의됨
enum Option<T> {
    None,     // 값 없음
    Some(T),  // 값 있음
}
```

Option의 핵심 아이디어

```rust
fn main() {
    let some_number: Option<i32> = Some(5);
    let no_number: Option<i32> = None;

    // ❌ 컴파일 에러! Option<i32>와 i32는 다른 타입
    // let sum = some_number + 5;

    // ✅ 반드시 '없는 경우'를 처리해야 함
    let sum = match some_number {
        Some(n) => n + 5,
        None => 0,  // None 처리 강제!
    };

    println!("합: {}", sum);
}
```

> 💡 **핵심**: `Option<T>`와 `T`는 **다른 타입**입니다. 컴파일러가 None 처리를 강제하므로 null 관련 버그가 원천 차단됩니다.

Prelude에 포함

`Option`, `Some`, `None`은 prelude에 포함되어 있어 `Option::`을 생략할 수 있습니다:

```rust
fn main() {
    // 둘 다 동일
    let x: Option<i32> = Option::Some(5);
    let y: Option<i32> = Some(5);  // 축약

    let z: Option<i32> = Option::None;
    let w: Option<i32> = None;     // 축약
}
```

안전한 값 추출

Option에서 값을 추출하는 여러 방법이 있습니다:

```rust
fn main() {
    let maybe_value: Option<i32> = Some(42);

    // 방법 1: match (가장 안전하고 명시적)
    let result1 = match maybe_value {
        Some(value) => value * 2,
        None => 0,
    };
    println!("match 결과: {}", result1);

    // 방법 2: if let (하나의 경우만 처리)
    if let Some(v) = maybe_value {
        println!("if let 값: {}", v);
    }

    // 방법 3: unwrap_or (기본값 제공)
    let result3 = maybe_value.unwrap_or(0);
    println!("unwrap_or 결과: {}", result3);

    // 방법 4: unwrap_or_else (지연 평가)
    let result4 = maybe_value.unwrap_or_else(|| {
        println!("기본값 계산 중...");
        expensive_computation()
    });

    // 방법 5: map (값이 있으면 변환)
    let doubled = maybe_value.map(|v| v * 2);  // Some(84)

    // 방법 6: and_then (체이닝)
    let chained = maybe_value
        .map(|v| v + 10)
        .and_then(|v| if v > 50 { Some(v) } else { None });
}

fn expensive_computation() -> i32 {
```

```
    42
}
```

Option 메서드 정리

메서드	설명	None일 때
`unwrap()`	값 추출	패닉!
`expect(msg)`	값 추출 (커스텀 메시지)	패닉!
`unwrap_or(default)`	값 또는 기본값	기본값 반환
`unwrap_or_else(f)`	값 또는 함수 결과	함수 호출
`unwrap_or_default()`	값 또는 Default 트레이트	T::default()
`map(f)`	값 변환	None
`and_then(f)`	체이닝 (flatMap)	None
`is_some()`	Some인지 확인	false
`is_none()`	None인지 확인	true

unwrap vs expect

```rust
fn main() {
    let x: Option<i32> = None;

    // ❌ 위험! 메시지가 불명확
    // x.unwrap();  // thread 'main' panicked at 'called `Option::unwrap()` on a `None` value'

    // ⚠️ 조금 낫지만 여전히 위험
    // x.expect("값이 있어야 함");  // thread 'main' panicked at '값이 있어야 함'

    // ✅ 항상 None 처리
    match x {
        Some(v) => println!("값: {}", v),
        None => println!("값 없음"),
    }
}
```

5.3 match: 강력한 제어 흐름

`match`는 Rust의 가장 강력한 제어 흐름 구조입니다. 패턴 매칭과 **완전성 검사(Exhaustiveness)**를 제공합니다.

기본 match 문법

```rust
enum Coin {
    Penny,
    Nickel,
    Dime,
    Quarter,
}

fn value_in_cents(coin: Coin) -> u32 {
    match coin {
        Coin::Penny => {
            println!("행운의 페니!");
            1  // 블록 내 마지막 표현식이 반환값
        }
        Coin::Nickel => 5,
        Coin::Dime => 10,
        Coin::Quarter => 25,
    }
}

fn main() {
    let coin = Coin::Penny;
    println!("가치: {} 센트", value_in_cents(coin));
}
```

완전성 검사 (Exhaustiveness)

모든 가능한 경우를 처리하지 않으면 컴파일 에러가 발생합니다:

```rust
enum Color {
    Red,
    Green,
    Blue,
}

fn describe(color: Color) -> &'static str {
    match color {
        Color::Red => "빨강",
        Color::Green => "초록",
        // ❌ 컴파일 에러! Blue를 처리하지 않음
        // error[E0004]: non-exhaustive patterns: `Blue` not
covered
    }
}

// ✅ 새 variant 추가 시 컴파일러가 미처리 부분 알려줌
```

💡 **이것이 중요한 이유**: 새 enum variant를 추가하면, 관련된 모든 match 표현식에서 컴파일 에러가 발생합니다. 실수로 처리를 빼먹는 일이 없습니다!

패턴에서 값 추출하기

enum variant에 담긴 데이터를 추출할 수 있습니다:

```rust
#[derive(Debug)]
enum UsState {
    Alabama,
    Alaska,
    Arizona,
    California,
    // ...
}

enum Coin {
    Penny,
    Nickel,
    Dime,
    Quarter(UsState),  // 주(State) 정보 포함
}

fn value_in_cents(coin: Coin) -> u32 {
    match coin {
        Coin::Penny => 1,
        Coin::Nickel => 5,
        Coin::Dime => 10,
        Coin::Quarter(state) => {
            // state 변수에 UsState가 바인딩됨
            println!("{:?} 주의 쿼터!", state);
            25
        }
    }
}

fn main() {
    let coin = Coin::Quarter(UsState::Alaska);
    let value = value_in_cents(coin);
    println!("가치: {} 센트", value);
}
```

복잡한 패턴 매칭

```
enum Message {
    Quit,
    Move { x: i32, y: i32 },
    Write(String),
    ChangeColor(u8, u8, u8),
}

fn process_message(msg: Message) {
    match msg {
        Message::Quit => {
            println!("종료");
        }
        Message::Move { x, y } => {
            // 구조체 패턴 매칭
            println!("({}, {})로 이동", x, y);
        }
        Message::Write(text) => {
            // 튜플 variant 매칭
            println!("메시지: {}", text);
        }
        Message::ChangeColor(r, g, b) => {
            // 여러 값 매칭
            println!("색상: RGB({}, {}, {})", r, g, b);
        }
    }
}
```

_ 플레이스홀더

나머지 모든 경우를 처리합니다:

```rust
fn main() {
    let dice_roll = 9;

    match dice_roll {
        3 => add_fancy_hat(),
        7 => remove_fancy_hat(),
        _ => reroll(),  // 3, 7 외 모든 경우
    }
}

fn add_fancy_hat() {}
fn remove_fancy_hat() {}
fn reroll() {}
```

값을 사용하지 않는 _

```rust
fn main() {
    let dice_roll = 9;

    match dice_roll {
        3 => println!("모자 획득!"),
        7 => println!("모자 제거!"),
        other => println!("{}만큼 이동", other),  // 값 사용
        // or
        // _ => println!("아무 일도 안 함"),      // 값 무시
        // or
        // _ => (),                               // 아무것도 안 함
    }
}
```

catch-all 주의점

`_` 또는 바인딩은 **항상 마지막**에 와야 합니다:

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

    // ❌ 경고! 이후 패턴에 도달할 수 없음
    // match x {
    //     _ => println!("기본"),
    //     1 => println!("하나"),  // 절대 실행 안 됨!
    // }

    // ✅ 올바른 순서
    match x {
        1 => println!("하나"),
        2 => println!("둘"),
        _ => println!("기타"),
    }
}
```

패턴 가드 (Match Guards)

`if` 조건을 추가하여 더 세밀한 매칭이 가능합니다:

```rust
fn main() {
    let num = Some(4);

    match num {
        Some(x) if x < 5 => println!("5 미만: {}", x),
        Some(x) if x >= 5 => println!("5 이상: {}", x),
        None => println!("값 없음"),
        _ => unreachable!(),
    }

    // 복잡한 조건
    let (x, y) = (5, 10);
    match (x, y) {
        (a, b) if a == b => println!("같음"),
        (a, b) if a > b => println!("a가 큼"),
        (a, b) => println!("b가 큼: {} < {}", a, b),
    }
}
```

OR 패턴

`|`를 사용하여 여러 패턴을 결합합니다:

```rust
fn main() {
    let x = 1;

    match x {
        1 | 2 => println!("하나 또는 둘"),
        3..=5 => println!("3에서 5 사이"),  // 범위 패턴
        _ => println!("기타"),
    }

    // 문자 범위도 가능
    let c = 'a';
    match c {
        'a'..='z' => println!("소문자"),
        'A'..='Z' => println!("대문자"),
        _ => println!("기타"),
    }
}
```

@ 바인딩

패턴 매칭과 동시에 값을 바인딩합니다:

```rust
enum Message {
    Hello { id: i32 },
}

fn main() {
    let msg = Message::Hello { id: 5 };

    match msg {
        Message::Hello {
            id: id_variable @ 3..=7,  // 범위 체크하면서 바인딩
        } => println!("범위 내 id: {}", id_variable),

        Message::Hello { id: 10..=12 } => {
            println!("범위 10-12");
            // 여기서는 id 사용 불가!
        }

        Message::Hello { id } => println!("기타 id: {}", id),
    }
}
```

5.4 if let: 간결한 매칭

하나의 패턴만 처리하고 나머지는 무시할 때 사용합니다:

기본 사용법

```
fn main() {
    let config_max = Some(3u8);

    // match 버전: 장황함
    match config_max {
        Some(max) => println!("최댓값: {}", max),
        _ => (),  // 아무것도 안 함
    }

    // if let 버전: 간결함
    if let Some(max) = config_max {
        println!("최댓값: {}", max);
    }
}
```

else 블록

```
fn main() {
    let coin = Coin::Quarter(UsState::Alaska);

    // if let + else
    if let Coin::Quarter(state) = coin {
        println!("{:?} 주의 쿼터!", state);
    } else {
        println!("쿼터가 아님");
    }
}
```

let else (Rust 1.65+)

"값이 없으면 early return" 패턴에 유용합니다:

```rust
fn process_config(input: Option<String>) -> Result<(), &'static
str> {
    // let else: Some이 아니면 early return
    let Some(config) = input else {
        return Err("설정 필요");
    };

    // 여기서 config는 String (Option이 아님!)
    println!("설정: {}", config);
    Ok(())
}

fn main() {
    let _ = process_config(Some(String::from("debug=true")));
    let _ = process_config(None);
}
```

5.5 while let: 반복적 패턴 매칭

조건이 패턴에 맞는 동안 반복합니다:

```rust
fn main() {
    let mut stack = vec![1, 2, 3];

    // pop()은 Option<T>를 반환
    // Some(value)인 동안 반복
    while let Some(top) = stack.pop() {
        println!("{}", top);
    }
    // 출력: 3, 2, 1

    println!("스택이 비었습니다");
}
```

이터레이터와 함께

```rust
fn main() {
    let mut iter = (0..5).into_iter();

    while let Some(n) = iter.next() {
        println!("{}", n);
    }
    // 출력: 0, 1, 2, 3, 4
}
```

5.6 matches! 매크로

간단한 패턴 비교에 유용합니다:

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

    // matches! 매크로: 패턴 매치 결과를 bool로 반환
    assert!(matches!(x, Some(_)));
    assert!(!matches!(x, None));

    // 복잡한 조건도 가능
    let foo = 'f';
    assert!(matches!(foo, 'A'..='Z' | 'a'..='z'));

    // enum 확인
    enum Color { Red, Green, Blue }
    let color = Color::Red;
    assert!(matches!(color, Color::Red | Color::Green));
}
```

5.7 Result 열거형 미리보기

`Option`처럼 `Result`도 중요한 표준 라이브러리 열거형입니다:

```rust
enum Result<T, E> {
    Ok(T),   // 성공, 값 포함
    Err(E),  // 실패, 에러 포함
}

fn divide(a: f64, b: f64) -> Result<f64, String> {
    if b == 0.0 {
        Err(String::from("0으로 나눌 수 없음"))
    } else {
        Ok(a / b)
    }
}

fn main() {
    match divide(10.0, 2.0) {
        Ok(result) => println!("결과: {}", result),
        Err(e) => println!("에러: {}", e),
    }
}
```

> `Result`는 에러 처리 챕터에서 자세히 다룹니다.

5.8 상태 기계 패턴

enum은 **상태 기계(State Machine)** 구현에 이상적입니다:

```rust
#[derive(Debug)]
enum OrderState {
    Pending,
    Confirmed { confirmed_at: String },
    Shipped { tracking_number: String },
    Delivered { delivered_at: String },
    Cancelled { reason: String },
}

struct Order {
    id: u64,
    state: OrderState,
}

impl Order {
    fn new(id: u64) -> Self {
        Order {
            id,
            state: OrderState::Pending,
        }
    }

    fn confirm(&mut self) {
        match &self.state {
            OrderState::Pending => {
                self.state = OrderState::Confirmed {
                    confirmed_at: String::from("2024-01-15"),
                };
                println!("주문 #{} 확정됨", self.id);
            }
            _ => println!("이 상태에서는 확정할 수 없음"),
        }
    }

    fn ship(&mut self, tracking: String) {
        if let OrderState::Confirmed { .. } = &self.state {
            self.state = OrderState::Shipped {
```

```
                    tracking_number: tracking,
                };
                println!("주문 #{} 배송 시작", self.id);
            } else {
                println!("확정된 주문만 배송 가능");
            }
        }

        fn can_cancel(&self) -> bool {
            matches!(self.state, OrderState::Pending |
OrderState::Confirmed { .. })
        }
}

fn main() {
        let mut order = Order::new(1);
        println!("초기 상태: {:?}", order.state);

        order.confirm();
        println!("확정 후: {:?}", order.state);

        order.ship(String::from("TRACK123"));
        println!("배송 후: {:?}", order.state);
}
```

🚀 실습 과제

과제 1: 계산기 구현

```
// TODO: Operation enum 정의
// - Add(f64, f64)
// - Subtract(f64, f64)
// - Multiply(f64, f64)
// - Divide(f64, f64)

// TODO: calculate 함수 구현
// fn calculate(op: Operation) -> Option<f64>
// 0으로 나누기는 None 반환
```

과제 2: JSON 값 타입

```
// TODO: JsonValue enum 정의
// - Null
// - Bool(bool)
// - Number(f64)
// - String(String)
// - Array(Vec<JsonValue>)
// - Object(HashMap<String, JsonValue>)

// TODO: 타입 확인 메서드 구현
// fn is_null(&self) -> bool
// fn is_string(&self) -> bool
// fn as_string(&self) -> Option<&str>
```

🚀 실습 가이드

`examples/` 폴더에서 확인:

1. `enums.rs`: 기본 열거형과 데이터를 담는 열거형
2. `match_control.rs`: match와 if let 활용
3. `state_machine.rs`: 상태 기계 패턴

📚 핵심 정리

개념	설명
enum	가능한 값의 집합을 정의하는 타입
variant	enum의 각 가능한 값
데이터 variant	각 variant가 다른 타입의 데이터 보유 가능
`Option<T>`	null 대신 사용, `Some(T)` 또는 `None`
`match`	패턴 매칭, 완전성 검사
`if let`	단일 패턴 간결 매칭
`while let`	패턴 기반 반복
`matches!`	패턴 매칭 bool 결과
패턴 가드	`if` 조건 추가
`@` 바인딩	패턴 매칭하면서 값 바인딩

Chapter 6: 패키지, 크레이트, 모듈 (The Module System)

> “Rust의 모듈 시스템은 코드를 구조화하고, 캡슐화하며, 재사용 가능하게 만드는 도구입니다. 대규모 프로젝트에서 아키텍처의 근간을 형성합니다.”

왜 모듈 시스템이 필요한가?

소프트웨어가 커지면 여러 문제가 발생합니다:

1. **이름 충돌**: 같은 이름의 함수가 여러 곳에 있을 수 있음
2. **복잡성 관리**: 수천 줄의 코드를 하나의 파일에서 관리하기 어려움
3. **재사용성**: 코드를 다른 프로젝트에서 쉽게 재사용하고 싶음
4. **캡슐화**: 내부 구현을 숨기고 공개 인터페이스만 노출하고 싶음

Rust의 모듈 시스템은 이 모든 문제를 해결합니다:

```rust
// 모듈 시스템 없이: 모든 것이 전역
// fn connect() { ... }  // 데이터베이스 연결
// fn connect() { ... }  // 네트워크 연결 ❌ 이름 충돌!

// 모듈 시스템으로: 명확한 네임스페이스
mod database {
    pub fn connect() { /* ... */ }
}

mod network {
    pub fn connect() { /* ... */ }
}

fn main() {
    database::connect();  // ✅ 명확!
    network::connect();
}
```

6.1 핵심 개념 정리

Rust의 모듈 시스템은 네 가지 핵심 개념으로 구성됩니다:

용어	설명	비유
패키지 (Package)	`Cargo.toml`을 포함하는 프로젝트	책 전체
크레이트 (Crate)	컴파일 단위. 라이브러리 또는 바이너리	책의 한 권

용어	설명	비유
모듈 (Module)	코드 조직화 단위. `mod` 키워드로 정의	책의 장 (Chapter)
경로 (Path)	아이템(함수, 구조체 등)의 위치 지정	목차

계층 구조

```
📦 패키지 (my_project/)
├── Cargo.toml           # 패키지 메타데이터
├── src/
│   ├── main.rs          # 바이너리 크레이트 루트
│   ├── lib.rs           # 라이브러리 크레이트 루트 (선택)
│   └── bin/             # 추가 바이너리들
│       └── other.rs
└── tests/               # 통합 테스트
```

6.2 크레이트 (Crate)

크레이트는 Rust 컴파일러가 한 번에 고려하는 **최소 코드 단위**입니다.

바이너리 크레이트 (Binary Crate)

실행 파일로 컴파일됩니다:

```rust
// src/main.rs (바이너리 크레이트 루트)
fn main() {
    println!("Hello, world!");
}
```

- ✅ `main()` 함수 필수
- ✅ 실행 파일 생성
- 📍 기본 위치: `src/main.rs`

라이브러리 크레이트 (Library Crate)

다른 프로젝트에서 사용하는 기능을 제공합니다:

```rust
// src/lib.rs (라이브러리 크레이트 루트)
pub fn add(a: i32, b: i32) -> i32 {
    a + b
}
```

- ❌ `main()` 없음
- ✅ `.rlib` 파일 생성 (다른 크레이트에서 import)
- 📍 기본 위치: `src/lib.rs`

프로젝트 생성

```
# 바이너리 프로젝트 (기본)
cargo new my_app
# 생성: src/main.rs

# 라이브러리 프로젝트
cargo new my_library --lib
# 생성: src/lib.rs

# 하이브리드 (둘 다)
# src/main.rs와 src/lib.rs 모두 생성하면 됨
```

하이브리드 구조

많은 프로젝트가 라이브러리와 바이너리를 모두 갖습니다:

```
// src/lib.rs - 핵심 로직
pub mod calculator {
    pub fn add(a: i32, b: i32) -> i32 { a + b }
    pub fn multiply(a: i32, b: i32) -> i32 { a * b }
}

// src/main.rs - CLI 인터페이스
use my_project::calculator;

fn main() {
    println!("2 + 3 = {}", calculator::add(2, 3));
}
```

> 💡 **팁**: 로직은 라이브러리에, 인터페이스는 바이너리에 구현하면 테스트와 재사용이 쉬워집니다.

6.3 모듈 정의하기

인라인 모듈

파일 내에 직접 모듈을 정의합니다:

```rust
// src/lib.rs
mod front_of_house {
    pub mod hosting {
        pub fn add_to_waitlist() {
            println!("대기 명단에 추가");
        }

        pub fn seat_at_table() {
            println!("테이블 안내");
        }
    }

    mod serving {  // 비공개 모듈
        fn take_order() {}
        fn serve_order() {}
        fn take_payment() {}
    }
}

pub fn eat_at_restaurant() {
    // 절대 경로
    crate::front_of_house::hosting::add_to_waitlist();

    // 상대 경로
    front_of_house::hosting::seat_at_table();
}
```

모듈 트리

위 코드의 모듈 구조:

```
crate (lib.rs)
├── front_of_house
│   ├── hosting
│   │   ├── add_to_waitlist
│   │   └── seat_at_table
│   └── serving
│       ├── take_order
│       ├── serve_order
│       └── take_payment
└── eat_at_restaurant
```

파일 기반 모듈

모듈을 별도 파일로 분리할 수 있습니다. 두 가지 스타일이 있습니다:

스타일 1: 파일로 분리 (권장)

```
src/
├── lib.rs
├── front_of_house.rs          # mod front_of_house
└── front_of_house/
    └── hosting.rs             # mod hosting
```

```rust
// src/lib.rs
mod front_of_house;  // front_of_house.rs 또는
front_of_house/mod.rs 로드

pub use crate::front_of_house::hosting;

// src/front_of_house.rs
pub mod hosting;  // front_of_house/hosting.rs 로드

// src/front_of_house/hosting.rs
pub fn add_to_waitlist() {
    println!("대기 명단에 추가됨!");
}
```

스타일 2: mod.rs 사용 (구 스타일)

```
src/
├── lib.rs
└── front_of_house/
    ├── mod.rs                # mod front_of_house
    └── hosting.rs            # mod hosting
```

```rust
// src/lib.rs
mod front_of_house;

// src/front_of_house/mod.rs
pub mod hosting;

// src/front_of_house/hosting.rs
pub fn add_to_waitlist() {}
```

💡 추천: 스타일 1을 권장합니다. `mod.rs` 파일이 많아지면 혼동되기 쉽습니다.

실제 프로젝트 구조 예시

```
src/
├── lib.rs              # 크레이트 루트
├── config.rs           # mod config
├── database.rs         # mod database
├── database/
│   ├── connection.rs   # mod connection
│   └── query.rs        # mod query
├── handlers.rs         # mod handlers
└── handlers/
    ├── auth.rs         # mod auth
    └── user.rs         # mod user
```

```rust
// src/lib.rs
pub mod config;
pub mod database;
pub mod handlers;

// src/database.rs
pub mod connection;
pub mod query;

// src/handlers.rs
pub mod auth;
pub mod user;
```

6.4 경로 (Paths)

절대 경로와 상대 경로

아이템에 접근하는 두 가지 방법:

```
mod front_of_house {
    pub mod hosting {
        pub fn add_to_waitlist() {}
    }
}

pub fn eat_at_restaurant() {
    // 방법 1: 절대 경로 (크레이트 루트부터)
    crate::front_of_house::hosting::add_to_waitlist();

    // 방법 2: 상대 경로 (현재 모듈부터)
    front_of_house::hosting::add_to_waitlist();
}
```

언제 어떤 경로를 사용할까?

상황	권장 경로
코드 이동 시 안정성	절대 경로
간결함	상대 경로
다른 크레이트에서 사용	절대 경로 or `use`

```rust
// 절대 경로: 모듈 구조 변경에 영향 적음
crate::utils::helper::format_name();

// 상대 경로: 같은 모듈 내에서 호출 시 간결
helper::format_name();
```

---

super로 부모 참조

`super`는 상위 모듈을 참조합니다 (파일 시스템의 `..`와 유사):

```rust
fn deliver_order() {
    println!("주문 배달!");
}

mod back_of_house {
    fn fix_incorrect_order() {
        cook_order();
        super::deliver_order();  // 부모 모듈의 함수 호출
    }

    fn cook_order() {
        println!("요리 중...");
    }

    pub mod kitchen {
        pub fn prepare_meal() {
            super::cook_order();  // back_of_house::cook_order()
            super::super::deliver_order();  // 크레이트 루트의 deliver_order()
        }
    }
}
```

self 키워드

현재 모듈을 명시적으로 참조합니다:

```rust
mod foo {
    pub fn bar() {}
}

use self::foo::bar;  // 현재 모듈의 foo::bar

fn main() {
    bar();
}
```

6.5 가시성 (Visibility): `pub` 키워드

기본 규칙

> 핵심 원칙: Rust의 모든 아이템은 **기본적으로 비공개(Private)**입니다.

```rust
mod outer {
    fn private_fn() {  // 비공개
        println!("비공개");
    }

    pub fn public_fn() {  // 공개
        println!("공개");
        private_fn();  // ✅ 같은 모듈 내에서는 접근 가능
    }

    pub mod inner {
        pub fn inner_public() {}
        fn inner_private() {}  // outer 외부에서 접근 불가
    }
}

fn main() {
    outer::public_fn();        // ✅ OK
    outer::inner::inner_public();  // ✅ OK

    // outer::private_fn();     // ❌ 비공개
    // outer::inner::inner_private();  // ❌ 비공개
}
```

가시성 규칙 정리

규칙	설명
부모 → 자식	부모 모듈은 자식의 공개 아이템만 접근 가능
자식 → 부모	자식 모듈은 부모의 모든 아이템에 접근 가능
형제	같은 부모를 가진 모듈은 서로의 공개 아이템만 접근 가능

```rust
mod parent {
    fn parent_private() {}
    pub fn parent_public() {}

    mod child {
        fn access_parent() {
            super::parent_private();  // ✅ 자식은 부모의 비공개에 접근 가능
            super::parent_public();   // ✅
        }
    }

    fn access_child() {
        // child::???  // ❌ 부모는 자식의 비공개에 접근 불가
    }
}
```

구조체 필드 가시성

구조체 자체가 `pub`이어도 필드는 기본적으로 비공개입니다:

```rust
mod back_of_house {
    pub struct Breakfast {
        pub toast: String,          // 공개: 손님이 선택
        seasonal_fruit: String,     // 비공개: 주방장이 결정
    }

    impl Breakfast {
        // 연관 함수로 생성자 제공 (비공개 필드 초기화)
        pub fn summer(toast: &str) -> Breakfast {
            Breakfast {
                toast: String::from(toast),
                seasonal_fruit: String::from("복숭아"),
            }
        }

        // 비공개 필드 읽기용 메서드
        pub fn fruit(&self) -> &str {
            &self.seasonal_fruit
        }
    }
}

pub fn eat_at_restaurant() {
    // 생성자 사용 (비공개 필드 때문에 직접 생성 불가)
    let mut meal = back_of_house::Breakfast::summer("호밀");

    // 공개 필드: 읽기/쓰기 가능
    meal.toast = String::from("밀");
    println!("토스트: {}", meal.toast);  // ✅

    // 비공개 필드: 직접 접근 불가
    // meal.seasonal_fruit = String::from("블루베리");  // ❌

    // 메서드를 통한 간접 접근
    println!("과일: {}", meal.fruit());  // ✅
}
```

불변식 보호

비공개 필드는 **불변식(invariant)**을 보호하는 데 중요합니다:

```
pub struct PositiveNumber {
    value: i32,  // 비공개: 항상 양수여야 함
}

impl PositiveNumber {
    pub fn new(n: i32) -> Option<Self> {
        if n > 0 {
            Some(PositiveNumber { value: n })
        } else {
            None
        }
    }

    pub fn get(&self) -> i32 {
        self.value  // 항상 양수임이 보장됨!
    }
}
```

---

enum 가시성

enum은 구조체와 다르게 동작합니다:

> 규칙: enum이 `pub`이면 **모든 variant도 자동으로 `pub`**입니다.

```rust
mod menu {
    pub enum Appetizer {
        Soup,    // 자동으로 pub
        Salad,   // 자동으로 pub
        Bread,   // 자동으로 pub
    }

    // 비교: 구조체의 모든 필드를 pub으로 만들려면 각각 지정해야 함
    pub struct Meal {
        pub main: String,
        pub side: String,
        pub drink: String,
    }
}

fn main() {
    let soup = menu::Appetizer::Soup;  // ✅ OK
}
```

pub의 다양한 변형

Rust 2018부터 더 세밀한 가시성 제어가 가능합니다:

```rust
mod outer {
    pub(crate) fn crate_visible() {}      // 같은 크레이트 내에서만
    pub(super) fn parent_visible() {}     // 부모 모듈에서만
    pub(in crate::outer) fn specific() {} // 특정 경로에서만

    mod inner {
        pub(super) fn for_outer() {}      // outer에서만 접근 가능
    }
}
```

문법	가시성 범위
`pub`	완전 공개
`pub(crate)`	같은 크레이트 내
`pub(super)`	부모 모듈
`pub(in path)`	지정된 경로
(없음)	비공개

6.6 use로 경로 가져오기

기본 사용법

`use`로 경로를 현재 스코프로 가져옵니다:

```rust
mod front_of_house {
    pub mod hosting {
        pub fn add_to_waitlist() {}
    }
}

// use로 경로 단축
use crate::front_of_house::hosting;

pub fn eat_at_restaurant() {
    hosting::add_to_waitlist();  // 짧아짐!
    hosting::add_to_waitlist();
    hosting::add_to_waitlist();
}
```

use의 스코프

`use`는 선언된 스코프 내에서만 유효합니다:

```rust
mod customer {
    use crate::front_of_house::hosting;

    pub fn order() {
        hosting::add_to_waitlist();  // ✅ OK
    }
}

// hosting::add_to_waitlist();  // ❌ 스코프 밖!
```

관용적 경로 사용

Rust 커뮤니티의 관례:

```rust
// 함수: 부모 모듈까지만 가져옴 (출처 명확)
use std::io;

fn main() {
    io::stdin().read_line(&mut buffer).unwrap();
    // stdin()이 io 모듈에서 온 것이 명확함
}

// 구조체, enum, 기타: 전체 경로 가져옴
use std::collections::HashMap;
use std::io::Result;

fn main() {
    let map = HashMap::new();
    let result: Result<()> = Ok(());
}
```

예외: 이름 충돌 시

```rust
// 두 Result가 충돌
use std::fmt::Result;
use std::io::Result;  // ❌ 에러!

// 해결 1: 부모만 가져오기
use std::fmt;
use std::io;

fn function1() -> fmt::Result { Ok(()) }
fn function2() -> io::Result<()> { Ok(()) }

// 해결 2: as로 별칭
use std::fmt::Result;
use std::io::Result as IoResult;

fn function1() -> Result { Ok(()) }
fn function2() -> IoResult<()> { Ok(()) }
```

---

as로 별칭 지정

```rust
use std::fmt::Result;
use std::io::Result as IoResult;

fn format_output() -> Result {
    Ok(())
}

fn read_file() -> IoResult<String> {
    Ok(String::new())
}

// 긴 타입 축약에도 유용
use std::collections::HashMap as Map;
let m: Map<String, i32> = Map::new();
```

---

pub use로 재내보내기 (Re-export)

내부 구조를 숨기고 깔끔한 공개 API를 제공합니다:

```rust
// 내부 구조
mod front_of_house {
    pub mod hosting {
        pub fn add_to_waitlist() {}
    }
}

// 재내보내기: 외부에서 더 쉽게 접근
pub use crate::front_of_house::hosting;

// 외부에서:
// use my_crate::front_of_house::hosting::add_to_waitlist();
// 복잡
// use my_crate::hosting::add_to_waitlist();  // ✅ 재내보내기 덕분에 간단!
```

실제 활용: 깔끔한 API 설계

```rust
// src/lib.rs
mod internal {
    pub mod user {
        pub struct User { pub name: String }
    }
    pub mod database {
        pub fn connect() {}
    }
}

// 공개 API (내부 구조 숨김)
pub use internal::user::User;
pub use internal::database::connect;

// 외부에서:
// use my_lib::User;  // 깔끔!
// use my_lib::connect;
```

중첩 경로

여러 `use`를 하나로 결합합니다:

```rust
// 긴 버전
use std::cmp::Ordering;
use std::io;

// 축약 버전
use std::{cmp::Ordering, io};

// self 사용: 모듈과 그 안의 아이템
use std::io;
use std::io::Write;
// ↓ 축약
use std::io::{self, Write};
```

복잡한 중첩

```rust
// 여러 레벨 중첩
use std::collections::{HashMap, HashSet};
use std::io::{self, Read, Write, BufReader};

// 다른 모듈도 함께
use std::{
    collections::{HashMap, HashSet},
    io::{self, Read, Write},
    fs::File,
};
```

글로브 연산자 (*)

모듈의 모든 공개 아이템을 가져옵니다:

```rust
use std::collections::*;

fn main() {
    let _: HashMap<i32, i32> = HashMap::new();
    let _: HashSet<i32> = HashSet::new();
    let _: BTreeMap<i32, i32> = BTreeMap::new();
}
```

⚠️ 주의: 글로브는 어떤 이름이 스코프에 있는지 불명확하게 만듭니다. 테스트나 prelude 모듈에서만 사용하세요.

```rust
// ✅ 좋은 사용: 테스트
#[cfg(test)]
mod tests {
    use super::*;  // 테스트 대상 모듈 전체 가져오기

    #[test]
    fn test_something() {}
}

// ✅ 좋은 사용: prelude 패턴
pub mod prelude {
    pub use crate::User;
    pub use crate::Database;
    pub use crate::Error;
}

// 외부에서
use my_crate::prelude::*;
```

6.7 외부 크레이트 사용하기

Cargo.toml에 의존성 추가

```
[dependencies]
rand = "0.8"
serde = { version = "1.0", features = ["derive"] }
tokio = { version = "1", features = ["full"] }
```

코드에서 사용

```
// 외부 크레이트는 이름으로 직접 참조
use rand::Rng;
use serde::{Serialize, Deserialize};

fn main() {
    let mut rng = rand::thread_rng();
    let n: u32 = rng.gen_range(1..=100);
    println!("랜덤 숫자: {}", n);
}
```

표준 라이브러리

`std`는 자동으로 포함되지만, 아이템은 `use`해야 합니다:

```
// std는 자동 포함 (Cargo.toml에 추가 불필요)
use std::collections::HashMap;
use std::io::{self, Read};
use std::fs::File;
```

6.8 모듈 설계 패턴

Prelude 패턴

자주 사용되는 아이템을 한 곳에 모읍니다:

```
// src/lib.rs
pub mod user;
pub mod database;
pub mod error;

// src/prelude.rs
pub use crate::user::{User, UserRole};
pub use crate::database::Database;
pub use crate::error::{Error, Result};

// 사용자 코드
use my_crate::prelude::*;
```

Facade 패턴

복잡한 하위 시스템을 단순한 인터페이스로 노출:

```rust
// 복잡한 내부 구조
mod database {
    pub mod connection { /* ... */ }
    pub mod query { /* ... */ }
    pub mod transaction { /* ... */ }
}

// Facade: 단순한 인터페이스
pub mod db {
    use super::database;

    pub fn query(sql: &str) -> Vec<Row> {
        let conn = database::connection::pool().get();
        let tx = database::transaction::begin(&conn);
        let result = database::query::execute(&tx, sql);
        tx.commit();
        result
    }
}

// 사용자는 db::query()만 알면 됨
```

Feature 기반 모듈화

```
// src/lib.rs
#[cfg(feature = "async")]
pub mod async_support;

#[cfg(feature = "serde")]
pub mod serialization;

// Cargo.toml
// [features]
// async = ["tokio"]
// serde = ["serde"]
```

6.9 아키텍처 가이드라인

레이어드 아키텍처

```
src/
├── lib.rs
├── domain/           # 핵심 비즈니스 로직 (의존성 없음)
│   ├── mod.rs
│   ├── user.rs
│   └── order.rs
├── application/      # 유스케이스 (domain 의존)
│   ├── mod.rs
│   └── services.rs
├── infrastructure/   # 외부 시스템 (domain, application 의존)
│   ├── mod.rs
│   ├── database.rs
│   └── http.rs
└── presentation/     # UI/API (모든 레이어 의존)
    ├── mod.rs
    └── handlers.rs
```

의존성 방향

```
presentation → application → domain
      ↓
infrastructure
```

모듈 스코프 권장사항

권장	비권장
명확한 공개 API	내부 구현 노출
작은 모듈, 단일 책임	거대한 모듈
레이어 간 명확한 경계	순환 의존성
`pub use`로 API 정리	깊은 경로 노출

🚀 실습 과제

과제 1: 라이브러리 구조화

```
// TODO: 다음 구조로 계산기 라이브러리 만들기
// src/lib.rs
// src/operations/mod.rs
// src/operations/basic.rs  (add, subtract)
// src/operations/advanced.rs  (power, sqrt)

// prelude 패턴 적용
// pub use로 재내보내기
```

과제 2: 가시성 연습

```
// TODO: BankAccount 구조체 구현
// - balance: 비공개 (음수 불가능)
// - 생성자: new(initial_balance)
// - deposit, withdraw: 공개 메서드
// - internal_audit: 비공개 메서드
```

🚀 실습 가이드

`examples/` 폴더에서 확인:

1. `modules_lib.rs`: 모듈 정의와 가시성
2. `use_keyword.rs`: use를 활용한 경로 관리
3. `reexport.rs`: pub use 재내보내기 패턴

📚 핵심 정리

개념	설명
패키지	`Cargo.toml` 포함 프로젝트
크레이트	컴파일 단위 (바이너리/라이브러리)
모듈	`mod` 키워드로 정의, 코드 조직화
`pub`	공개 가시성 (기본은 비공개)

개념	설명
`use`	경로를 현재 스코프로 가져오기
`pub use`	재내보내기 (API 단순화)
`super`	부모 모듈 참조
`crate`	크레이트 루트 참조
글로브 `*`	모든 공개 아이템 가져오기 (주의)
Prelude 패턴	자주 쓰는 아이템 모아두기

Chapter 7: 일반적인 컬렉션 (Common Collections)

> "표준 라이브러리 컬렉션은 힙에 저장되어 런타임에 크기가 변할 수 있는 데이터 구조입니다. 배열과 튜플과 달리, 컴파일 타임에 크기를 알 필요가 없습니다."

왜 컬렉션이 필요한가?

프로그래밍에서 자주 마주치는 상황:

- **가변 개수의 요소**: 사용자가 몇 개의 항목을 입력할지 모름
- **동적 데이터**: 프로그램 실행 중 데이터가 추가/삭제됨
- **텍스트 처리**: 문자열을 조작하고 합치는 작업
- **키-값 저장**: 이름으로 데이터를 빠르게 찾기

Rust의 표준 라이브러리는 이 모든 상황을 위한 컬렉션을 제공합니다:

컬렉션	설명	사용 사례
`Vec<T>`	가변 크기 배열	리스트, 스택
`String`	가변 크기 UTF-8 문자열	텍스트 처리
`HashMap<K, V>`	키-값 저장소	캐시, 설정

7.1 벡터 (`Vec<T>`)

벡터는 Rust에서 가장 많이 사용되는 컬렉션입니다. **동일한 타입**의 값을 **가변 개수**로 저장합니다.

생성 방법

```
fn main() {
    // 방법 1: 빈 벡터 생성 (타입 명시 필요)
    let mut v1: Vec<i32> = Vec::new();

    // 방법 2: vec! 매크로 (타입 추론)
    let v2 = vec![1, 2, 3];

    // 방법 3: 용량 미리 할당 (성능 최적화)
    let mut v3: Vec<i32> = Vec::with_capacity(100);

    // 방법 4: 반복으로 생성
    let v4: Vec<i32> = (1..=5).collect();  // [1, 2, 3, 4, 5]

    // 방법 5: 같은 값으로 채우기
    let v5 = vec![0; 10];  // [0, 0, 0, 0, 0, 0, 0, 0, 0, 0]

    println!("v2: {:?}", v2);
    println!("v4: {:?}", v4);
    println!("v5 길이: {}", v5.len());
}
```

요소 추가와 제거

```
fn main() {
    let mut v = vec![1, 2, 3];

    // 끝에 추가
    v.push(4);
    v.push(5);
    println!("push 후: {:?}", v);  // [1, 2, 3, 4, 5]

    // 끝에서 제거 (Option 반환)
    let last = v.pop();
    println!("pop: {:?}, 남은 벡터: {:?}", last, v);  // Some(5),
[1, 2, 3, 4]

    // 특정 위치에 삽입
    v.insert(1, 10);  // 인덱스 1에 10 삽입
    println!("insert 후: {:?}", v);  // [1, 10, 2, 3, 4]

    // 특정 위치에서 제거
    let removed = v.remove(1);  // 인덱스 1 제거
    println!("remove: {}, 남은 벡터: {:?}", removed, v);  // 10,
[1, 2, 3, 4]

    // 모든 요소 제거
    v.clear();
    println!("clear 후: {:?}, 비었나?: {}", v, v.is_empty());
}
```

요소 읽기: 인덱싱 vs get

두 가지 방법이 있으며, 중요한 차이가 있습니다:

```rust
fn main() {
    let v = vec![1, 2, 3, 4, 5];

    // 방법 1: 인덱싱 ([] 연산자)
    // ⚠️ 범위 초과 시 패닉!
    let third: &i32 = &v[2];
    println!("인덱싱으로 세 번째: {}", third);

    // 방법 2: get 메서드 (Option 반환)
    // ✅ 범위 초과 시 None 반환 (안전)
    match v.get(2) {
        Some(value) => println!("get으로 세 번째: {}", value),
        None => println!("요소 없음"),
    }

    // 범위 초과 비교
    // let does_not_exist = &v[100];  // 💥 패닉!
    let does_not_exist = v.get(100);  // ✅ None
    println!("인덱스 100: {:?}", does_not_exist);
}
```

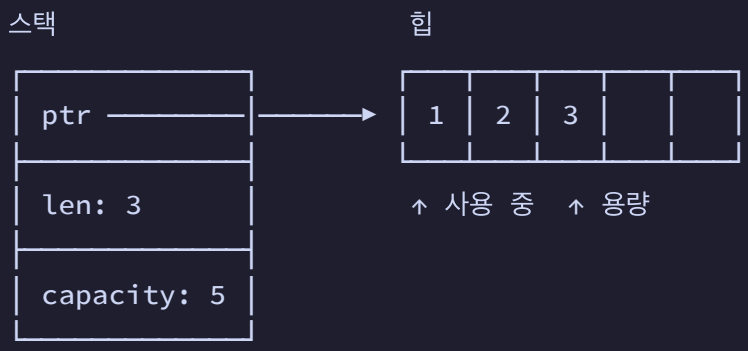
언제 어떤 방법을 사용할까?

방법	사용 상황
`v[i]`	인덱스가 항상 유효하다고 확신할 때 (버그 시 빠른 발견)
`v.get(i)`	인덱스가 유효하지 않을 수 있을 때 (사용자 입력 등)

```rust
fn main() {
    let v = vec![10, 20, 30];

    // 인덱싱: 확실한 경우
    for i in 0..v.len() {
        println!("{}", v[i]);  // 항상 유효
    }

    // get: 불확실한 경우
    fn get_element(v: &Vec<i32>, index: usize) -> Option<&i32>
{
        v.get(index)  // 안전하게 처리
    }
}
```

메모리 레이아웃과 재할당

벡터는 **연속된 메모리**에 요소를 저장합니다:



재할당과 소유권 규칙

중요한 소유권 규칙:

```rust
fn main() {
    let mut v = vec![1, 2, 3, 4, 5];

    let first = &v[0];  // 불변 빌림

    // ❌ 컴파일 에러! 불변 빌림 중 가변 변경 불가
    // v.push(6);

    println!("첫 요소: {}", first);

    // ✅ first 사용이 끝났으므로 이제 가능
    v.push(6);
}
```

왜 이런 제한이 있을까?

`push`가 재할당을 유발할 수 있기 때문입니다. 용량이 부족하면 벡터는:

1. 더 큰 메모리 블록 할당
2. 기존 데이터 복사
3. 기존 메모리 해제

이 과정에서 `first`가 가리키던 메모리가 해제될 수 있습니다!

용량 관리

```
fn main() {
    let mut v: Vec<i32> = Vec::new();

    println!("초기 - len: {}, capacity: {}", v.len(),
v.capacity());

    for i in 0..10 {
        v.push(i);
        println!("push({}) 후 - len: {}, capacity: {}", i,
v.len(), v.capacity());
    }

    // 용량 미리 할당으로 재할당 방지
    let mut efficient: Vec<i32> = Vec::with_capacity(10);
    for i in 0..10 {
        efficient.push(i);  // 재할당 없음!
    }

    // 남는 용량 정리
    efficient.shrink_to_fit();
}
```

순회

```rust
fn main() {
    let v = vec![100, 32, 57];

    // 불변 순회 (값 읽기만)
    println!("불변 순회:");
    for i in &v {
        println!("{}", i);
    }

    // 가변 순회 (값 수정)
    let mut v2 = vec![100, 32, 57];
    println!("\n가변 순회:");
    for i in &mut v2 {
        *i += 50;  // 역참조하여 수정
    }
    println!("{:?}", v2);  // [150, 82, 107]

    // 인덱스와 함께 순회
    println!("\n인덱스 순회:");
    for (index, value) in v.iter().enumerate() {
        println!("v[{}] = {}", index, value);
    }
}
```

이터레이터 어댑터

```
fn main() {
    let v = vec![1, 2, 3, 4, 5];

    // map: 변환
    let doubled: Vec<i32> = v.iter().map(|x| x * 2).collect();
    println!("doubled: {:?}", doubled);  // [2, 4, 6, 8, 10]

    // filter: 필터링
    let evens: Vec<&i32> = v.iter().filter(|x| *x % 2 ==
0).collect();
    println!("evens: {:?}", evens);  // [2, 4]

    // fold: 집계
    let sum: i32 = v.iter().fold(0, |acc, x| acc + x);
    println!("sum: {}", sum);  // 15

    // 체이닝
    let result: i32 = v.iter()
        .filter(|x| *x % 2 == 1)  // 홀수만
        .map(|x| x * 10)          // 10배
        .sum();                    // 합계
    println!("홀수 10배 합: {}", result);  // 90
}
```

---

enum으로 여러 타입 저장

벡터는 단일 타입만 저장하지만, enum을 사용하면 우회할 수 있습니다:

```rust
#[derive(Debug)]
enum SpreadsheetCell {
    Int(i32),
    Float(f64),
    Text(String),
}

fn main() {
    let row = vec![
        SpreadsheetCell::Int(3),
        SpreadsheetCell::Text(String::from("blue")),
        SpreadsheetCell::Float(10.12),
    ];

    for cell in &row {
        match cell {
            SpreadsheetCell::Int(i) => println!("정수: {}", i),
            SpreadsheetCell::Float(f) => println!("실수: {}",
f),
            SpreadsheetCell::Text(s) => println!("텍스트: {}",
s),
        }
    }

    println!("{:?}", row);
}
```

유용한 Vec 메서드

```rust
fn main() {
    let mut v = vec![3, 1, 4, 1, 5, 9, 2, 6];

    // 정렬
    v.sort();
    println!("정렬: {:?}", v);  // [1, 1, 2, 3, 4, 5, 6, 9]

    // 역순
    v.reverse();
    println!("역순: {:?}", v);

    // 중복 제거 (정렬 필요)
    v.sort();
    v.dedup();
    println!("중복 제거: {:?}", v);  // [1, 2, 3, 4, 5, 6, 9]

    // 검색
    let pos = v.iter().position(|&x| x == 4);
    println!("4의 위치: {:?}", pos);  // Some(3)

    // 포함 여부
    println!("5 포함?: {}", v.contains(&5));  // true

    // 슬라이스 (뷰)
    let slice = &v[1..4];
    println!("슬라이스: {:?}", slice);  // [2, 3, 4]
}
```

7.2 문자열 (`String`)

> ⚠️ Rust의 문자열은 "왜 이렇게 복잡한가요?"라는 질문을 자주 받습니다. UTF-8을 제대로 지원하기 때문입니다.

두 가지 문자열 타입

타입	설명	저장 위치	가변성
`&str`	문자열 슬라이스	어디든 (보통 정적)	불변
`String`	힙 할당 문자열	힙	가변

```
fn main() {
    // &str: 문자열 리터럴 (바이너리에 포함)
    let s1: &str = "Hello";

    // String: 힙에 할당
    let s2: String = String::from("Hello");

    // 변환
    let s3: String = s1.to_string();  // &str → String
    let s4: &str = &s2;               // String → &str

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

생성 방법

```rust
fn main() {
    // 다양한 생성 방법
    let s1 = String::new();                    // 빈 문자열
    let s2 = "hello".to_string();              // &str → String
    let s3 = String::from("hello");            // 명시적 생성
    let s4 = format!("hello, {}", "world");    // 포맷팅

    // UTF-8 문자열 (다국어 지원)
    let korean = String::from("안녕하세요");
    let emoji = String::from("Hello 👋 World 🌍");

    println!("{}", korean);
    println!("{}", emoji);
}
```

업데이트

```rust
fn main() {
    let mut s = String::from("foo");

    // push_str: 문자열 슬라이스 추가
    s.push_str("bar");
    println!("push_str: {}", s);  // foobar

    // push: 단일 문자 추가
    s.push('!');
    println!("push: {}", s);  // foobar!
}
```

문자열 연결

```
fn main() {
    // 방법 1: + 연산자
    // ⚠️ 왼쪽 String의 소유권이 이동됨!
    let s1 = String::from("Hello, ");
    let s2 = String::from("world!");
    let s3 = s1 + &s2;  // s1은 이동됨, s2는 빌림
    // println!("{}", s1);  // ❌ 에러!
    println!("{}", s3);     // Hello, world!

    // 방법 2: format! 매크로 (권장)
    // ✅ 소유권 이동 없음!
    let s4 = String::from("tic");
    let s5 = String::from("tac");
    let s6 = String::from("toe");
    let s7 = format!("{}-{}-{}", s4, s5, s6);
    println!("{}", s7);    // tic-tac-toe
    println!("{}", s4);    // ✅ 여전히 사용 가능
}
```

> **+ 연산자의 비밀**: `+`는 사실 `fn add(self, s: &str) -> String` 메서드입니다. 그래서 왼쪽은 `String`, 오른쪽은 `&str`이어야 합니다.

UTF-8과 인덱싱

Rust는 문자열 인덱싱을 허용하지 않습니다:

```rust
fn main() {
    let s = String::from("hello");
    // let h = s[0];  // ❌ 컴파일 에러!
}
```

이유: UTF-8은 가변 길이 인코딩입니다:

```rust
fn main() {
    // ASCII: 각 문자 1바이트
    let hello = String::from("hello");
    println!("hello: {} 바이트", hello.len());  // 5

    // 한글: 각 문자 3바이트
    let korean = String::from("안녕");
    println!("안녕: {} 바이트", korean.len());  // 6

    // 이모지: 가변 (보통 4바이트)
    let emoji = String::from("👋");
    println!("👋: {} 바이트", emoji.len());  // 4

    // s[0]이 1바이트면, "안녕"에서 s[0]은 '안'이 아니라 깨진 바이트!
}
```

문자열의 세 가지 관점

```
fn main() {
    let hindi = "नमस्ते";  // 힌디어 "나마스테"

    // 1. 바이트 (bytes)
    print!("바이트: ");
    for b in hindi.bytes() {
        print!("{} ", b);
    }
    println!();  // 224 164 168 ...

    // 2. 유니코드 스칼라 값 (chars)
    print!("chars: ");
    for c in hindi.chars() {
        print!("{} ", c);
    }
    println!();  // न म स ◌् त े

    // 3. 그래핌 클러스터 (외부 크레이트 필요)
    // unicode-segmentation 크레이트 사용
    // "나", "마", "스", "테" 로 분리됨
}
```

문자열 슬라이싱

바이트 인덱스로 슬라이싱은 가능하지만 위험합니다:

```rust
fn main() {
    let hello = "안녕하세요";

    // ✅ 유효한 UTF-8 경계
    let s = &hello[0..6];  // "안녕" (각 3바이트)
    println!("{}", s);

    // ❌ 패닉! 유효하지 않은 UTF-8 경계
    // let s2 = &hello[0..4];  // 중간에서 잘림!
}
```

안전한 문자열 조작

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

    // 부분 문자열 (안전)
    if let Some(comma_pos) = s.find(',') {
        let before = &s[..comma_pos];
        let after = &s[comma_pos + 2..];  // ", " 건너뛰기
        println!("콤마 앞: {}", before);
        println!("콤마 뒤: {}", after);
    }

    // 문자 반복
    let first_char = s.chars().next();
    println!("첫 문자: {:?}", first_char);  // Some('h')

    // n번째 문자
    let third = s.chars().nth(2);
    println!("세 번째 문자: {:?}", third);  // Some('l')

    // 문자열 분할
    for word in s.split_whitespace() {
        println!("단어: {}", word);
    }
}
```

유용한 String 메서드

```rust
fn main() {
    let s = String::from("  Hello, World!  ");

    // 공백 제거
    println!("trim: '{}'", s.trim());

    // 대소문자 변환
    println!("lowercase: {}", s.to_lowercase());
    println!("uppercase: {}", s.to_uppercase());

    // 포함 여부
    println!("contains 'World': {}", s.contains("World"));
    println!("starts_with '  H': {}", s.starts_with("  H"));

    // 치환
    println!("replace: {}", s.replace("World", "Rust"));

    // 반복
    println!("repeat: {}", "ha".repeat(3));  // hahaha
}
```

7.3 해시맵 (`HashMap<K, V>`)

키-값 쌍을 저장하는 컬렉션입니다. 키로 값을 O(1) 시간에 검색할 수 있습니다.

생성과 삽입

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

fn main() {
    // 빈 해시맵 생성
    let mut scores = HashMap::new();

    // 삽입
    scores.insert(String::from("Blue"), 10);
    scores.insert(String::from("Yellow"), 50);

    println!("{:?}", scores);

    // 초기값으로 생성
    let teams = HashMap::from([
        (String::from("Red"), 100),
        (String::from("Green"), 200),
    ]);

    println!("{:?}", teams);
}
```

소유권

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

fn main() {
    let field_name = String::from("color");
    let field_value = String::from("blue");

    let mut map = HashMap::new();

    // String은 이동됨!
    map.insert(field_name, field_value);

    // ❌ 더 이상 사용 불가
    // println!("{}", field_name);

    // 참조를 저장하려면 수명 필요
    // HashMap<&str, &str>
}
```

값 읽기

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

fn main() {
    let mut scores = HashMap::new();
    scores.insert(String::from("Blue"), 10);
    scores.insert(String::from("Yellow"), 50);

    // get: Option<&V> 반환
    let team_name = String::from("Blue");
    let score = scores.get(&team_name);

    match score {
        Some(s) => println!("{}팀 점수: {}", team_name, s),
        None => println!("팀을 찾을 수 없음"),
    }

    // copied()로 값 복사
    let score =
scores.get(&String::from("Blue")).copied().unwrap_or(0);
    println!("점수: {}", score);

    // 없는 키
    let unknown = scores.get(&String::from("Red"));
    println!("Red팀: {:?}", unknown);  // None
}
```

순회

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

fn main() {
    let mut scores = HashMap::new();
    scores.insert(String::from("Blue"), 10);
    scores.insert(String::from("Yellow"), 50);

    // 키-값 순회
    for (key, value) in &scores {
        println!("{}: {}", key, value);
    }

    // 키만 순회
    for key in scores.keys() {
        println!("키: {}", key);
    }

    // 값만 순회
    for value in scores.values() {
        println!("값: {}", value);
    }
}
```

업데이트 전략

1. 덮어쓰기

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

fn main() {
    let mut scores = HashMap::new();
    scores.insert(String::from("Blue"), 10);
    scores.insert(String::from("Blue"), 25);  // 덮어씀

    println!("{:?}", scores);  // {"Blue": 25}
}
```

2. 없을 때만 삽입 (`entry` + `or_insert`)

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

fn main() {
    let mut scores = HashMap::new();
    scores.insert(String::from("Blue"), 10);

    // Yellow: 없으므로 삽입
    scores.entry(String::from("Yellow")).or_insert(50);

    // Blue: 있으므로 무시
    scores.entry(String::from("Blue")).or_insert(50);

    println!("{:?}", scores);  // {"Blue": 10, "Yellow": 50}
}
```

3. 기존 값 기반 업데이트

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

fn main() {
    let text = "hello world wonderful world";
    let mut word_count = HashMap::new();

    for word in text.split_whitespace() {
        // or_insert는 &mut V를 반환
        let count = word_count.entry(word).or_insert(0);
        *count += 1;
    }

    println!("{:?}", word_count);
    // {"hello": 1, "world": 2, "wonderful": 1}
}
```

4. or_insert_with (지연 계산)

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

fn main() {
    let mut cache: HashMap<String, Vec<u8>> = HashMap::new();

    let key = String::from("large_data");

    // 값이 없을 때만 비용이 큰 연산 수행
    let data = cache.entry(key.clone()).or_insert_with(|| {
        println!("비용이 큰 연산 수행 중...");
        vec![0; 1000]  // 없을 때만 실행
    });

    println!("데이터 길이: {}", data.len());
}
```

해시 함수

기본적으로 HashMap은 **SipHash**를 사용합니다:

- ✅ HashDoS 공격에 안전
- ⚠️ 가장 빠른 해시 알고리즘은 아님

성능이 중요하다면 다른 해시 함수를 사용할 수 있습니다:

```rust
use std::collections::HashMap;
use std::hash::BuildHasherDefault;
// use ahash::AHasher;  // ahash 크레이트 필요

// 커스텀 해시 함수 사용 (예시)
// type FastHashMap<K, V> = HashMap<K, V,
BuildHasherDefault<AHasher>>;
```

7.4 기타 유용한 컬렉션

VecDeque (양방향 큐)

```rust
use std::collections::VecDeque;

fn main() {
    let mut deque = VecDeque::new();

    // 앞/뒤로 추가
    deque.push_back(1);
    deque.push_back(2);
    deque.push_front(0);

    println!("{:?}", deque);  // [0, 1, 2]

    // 앞/뒤에서 제거
    println!("앞: {:?}", deque.pop_front());  // Some(0)
    println!("뒤: {:?}", deque.pop_back());   // Some(2)
}
```

HashSet (중복 없는 집합)

```
use std::collections::HashSet;

fn main() {
    let mut set = HashSet::new();

    set.insert("apple");
    set.insert("banana");
    set.insert("apple");  // 중복, 무시됨

    println!("{:?}", set);  // {"apple", "banana"}
    println!("apple 포함?: {}", set.contains("apple"));

    // 집합 연산
    let set2: HashSet<_> = ["banana", "cherry"].into();

    println!("교집합: {:?}", set.intersection(&set2).collect::
<Vec<_>>());
    println!("합집합: {:?}", set.union(&set2).collect::<Vec<_>>
());
}
```

BTreeMap (정렬된 맵)

```rust
use std::collections::BTreeMap;

fn main() {
    let mut map = BTreeMap::new();

    map.insert(3, "three");
    map.insert(1, "one");
    map.insert(2, "two");

    // 키 순서대로 순회 (정렬됨)
    for (k, v) in &map {
        println!("{}: {}", k, v);
    }
    // 1: one, 2: two, 3: three
}
```

🚀 실습 과제

과제 1: 학생 성적 관리

```rust
// TODO: HashMap 사용
// - 학생 이름 → 점수 목록 저장
// - 평균 점수 계산 함수
// - 최고 점수 학생 찾기
```

과제 2: 단어 빈도수

```
// TODO: 주어진 텍스트에서
// - 각 단어의 빈도수 계산
// - 가장 많이 나온 단어 3개 출력
// - 알파벳 순 정렬 출력
```

🚀 실습 가이드

`examples/` 폴더에서 확인:

1. `vectors.rs`: 벡터 생성, 읽기, 수정
2. `strings.rs`: 문자열 조작과 UTF-8
3. `hashmaps.rs`: 해시맵 활용

📚 핵심 정리

컬렉션	특징	시간 복잡도
`Vec<T>`	순서 있음, 인덱스 접근	접근 O(1), 삽입 O(n)
`String`	UTF-8, 인덱싱 불가	-
`HashMap<K,V>`	키로 접근, 순서 없음	접근/삽입 O(1)
`VecDeque<T>`	양방향 추가/제거	앞/뒤 O(1)

컬렉션	특징	시간 복잡도
`HashSet<T>`	중복 없음	접근/삽입 O(1)
`BTreeMap<K,V>`	키 정렬됨	접근/삽입 O(log n)

Chapter 8: 에러 처리 (Error Handling)

> "Rust는 에러를 **복구 가능한 에러**와 **복구 불가능한 에러**로 명확히 구분합니다. 이 구분은 프로그램의 안정성과 사용자 경험에 직접적인 영향을 미칩니다."

에러 처리의 철학

많은 언어가 에러 처리에 예외(Exception)를 사용합니다. 하지만 예외에는 문제가 있습니다:

```
// Java: 예외를 놓치기 쉬움
void riskyOperation() {
    doSomething();  // 예외를 던질 수 있지만 호출자가 모를 수 있음
}
```

Rust는 다른 접근법을 취합니다:

- **복구 불가능한 에러**: `panic!` - 프로그램 버그, 즉시 종료
- **복구 가능한 에러**: `Result<T, E>` - 예상된 실패, 호출자가 처리

```
// Rust: 에러 가능성이 타입에 명시됨
fn risky_operation() -> Result<Data, Error> {
    // 호출자가 반드시 에러를 처리해야 함
}
```

언어	에러 처리 방식	호출자 강제 처리
C	반환값 규약	❌
Java/Python	예외	❌ (unchecked)
Go	`(value, error)` 튜플	△ (무시 가능)
Rust	`Result<T, E>`	✅ (컴파일러 경고)

8.1 복구 불가능한 에러: `panic!`

`panic!`은 프로그램이 더 이상 진행할 수 없는 상황에서 사용합니다.

기본 사용법

```
fn main() {
    panic!("치명적인 오류 발생!");
}
```

실행 결과:

```
thread 'main' panicked at '치명적인 오류 발생!', src/main.rs:2:5
```

패닉이 발생하면?

1. 에러 메시지 출력

2. 스택을 되감기(unwind) 또는 즉시 중단(abort)

3. 프로그램 종료

```
# Cargo.toml - 릴리스에서 바이너리 크기 줄이기
[profile.release]
panic = 'abort'   # 되감기 대신 즉시 중단
```

백트레이스 (Backtrace)

어디서 패닉이 발생했는지 추적합니다:

```
# 백트레이스 활성화
RUST_BACKTRACE=1 cargo run

# 전체 백트레이스
RUST_BACKTRACE=full cargo run
```

```
fn main() {
    let v = vec![1, 2, 3];
    v[99];  // 패닉! 범위 초과
}
```

백트레이스 출력:

```
thread 'main' panicked at 'index out of bounds: the len is 3
but the index is 99'
stack backtrace:
   0: rust_begin_unwind
   1: core::panicking::panic_fmt
   ...
  10: playground::main
             at src/main.rs:3:5
```

버퍼 오버플로우 방지

C/C++에서는 배열 범위 초과가 **정의되지 않은 동작**입니다:

```c
// C: 위험! 메모리 손상 가능
int arr[3] = {1, 2, 3};
printf("%d\n", arr[99]);  // 쓰레기 값 또는 크래시
```

Rust는 **패닉으로 안전하게 종료**합니다:

```rust
fn main() {
    let v = vec![1, 2, 3];
    let value = v[99];  // 패닉! 버퍼 오버플로우 방지
}
```

8.2 복구 가능한 에러: `Result<T, E>`

대부분의 에러는 프로그램 전체를 종료할 정도로 심각하지 않습니다.

Result 타입 정의

```
enum Result<T, E> {
    Ok(T),   // 성공: 값 T 포함
    Err(E),  // 실패: 에러 E 포함
}
```

파일 열기 예제

```
use std::fs::File;

fn main() {
    let f = File::open("hello.txt");

    let file = match f {
        Ok(file) => {
            println!("파일 열기 성공!");
            file
        }
        Err(error) => {
            panic!("파일 열기 실패: {:?}", error);
        }
    };
}
```

에러 종류에 따른 처리

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

fn main() {
    let f = File::open("hello.txt");

    let file = match f {
        Ok(file) => file,
        Err(error) => match error.kind() {
            // 파일이 없으면 생성
            ErrorKind::NotFound => match
File::create("hello.txt") {
                Ok(fc) => {
                    println!("파일 생성 성공");
                    fc
                }
                Err(e) => panic!("파일 생성 실패: {:?}", e),
            },
            // 권한 없음
            ErrorKind::PermissionDenied => {
                panic!("파일 접근 권한이 없습니다");
            }
            // 기타 에러
            other_error => {
                panic!("파일 열기 실패: {:?}", other_error);
            }
        },
    };
}
```

unwrap_or_else로 간결하게

중첩된 match는 읽기 어렵습니다. 클로저를 사용하면 간결해집니다:

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

fn main() {
    let file = File::open("hello.txt").unwrap_or_else(|error| {
        if error.kind() == ErrorKind::NotFound {
            File::create("hello.txt").unwrap_or_else(|error| {
                panic!("파일 생성 실패: {:?}", error);
            })
        } else {
            panic!("파일 열기 실패: {:?}", error);
        }
    });
}
```

map과 and_then으로 체이닝

```
use std::fs::File;
use std::io::{self, Read};

fn read_username_from_file() -> Result<String, io::Error> {
    File::open("hello.txt")
        .and_then(|mut file| {
            let mut s = String::new();
            file.read_to_string(&mut s).map(|_| s)
        })
}
```

8.3 `unwrap`과 `expect`

match를 매번 작성하는 것은 번거롭습니다. 단축 메서드가 있습니다.

unwrap

성공하면 값을 반환, 실패하면 패닉:

```
use std::fs::File;

fn main() {
    // ⚠️ 파일이 없으면 패닉!
    let file = File::open("hello.txt").unwrap();
}
```

패닉 메시지:

```
thread 'main' panicked at 'called `Result::unwrap()` on an
`Err` value: Os { ... }'
```

expect (권장)

커스텀 에러 메시지 제공:

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

fn main() {
    // ⚠️ 패닉하지만 메시지가 더 유용함
    let file = File::open("hello.txt")
        .expect("hello.txt 파일을 열 수 없습니다");
}
```

패닉 메시지:

```
thread 'main' panicked at 'hello.txt 파일을 열 수 없습니다: Os { ...
}'
```

언제 unwrap/expect를 사용할까?

상황	권장
프로토타입 작성	✅ `unwrap()` OK
예제 코드	✅ `unwrap()` OK
테스트 코드	✅ `unwrap()` OK (실패 = 테스트 실패)
절대 실패하지 않는 로직	△ `expect("이유")`
프로덕션 코드	❌ 적절한 에러 처리 필요

```rust
// 절대 실패하지 않음을 문서화
let home: IpAddr = "127.0.0.1"
    .parse()
    .expect("하드코딩된 IP 주소가 유효하지 않음");
```

8.4 에러 전파하기 (Propagating Errors)

함수가 에러를 직접 처리하지 않고 호출자에게 전달하는 패턴입니다.

긴 버전: match 사용

```rust
use std::fs::File;
use std::io::{self, Read};

fn read_username_from_file() -> Result<String, io::Error> {
    let f = File::open("hello.txt");

    let mut f = match f {
        Ok(file) => file,
        Err(e) => return Err(e),  // 에러 즉시 반환
    };

    let mut s = String::new();

    match f.read_to_string(&mut s) {
        Ok(_) => Ok(s),
        Err(e) => Err(e),
    }
}
```

? 연산자 (권장!)

?는 위 패턴의 **축약 문법**입니다:

```rust
use std::fs::File;
use std::io::{self, Read};

fn read_username_from_file() -> Result<String, io::Error> {
    let mut f = File::open("hello.txt")?;  // 에러면 즉시 반환
    let mut s = String::new();
    f.read_to_string(&mut s)?;             // 에러면 즉시 반환
    Ok(s)
}
```

?의 동작

```rust
// ?가 하는 일 (대략적으로)
let mut f = match File::open("hello.txt") {
    Ok(file) => file,
    Err(e) => return Err(From::from(e)),  // 에러 타입 변환 포함!
};
```

> 💡 **포인트**: ?는 `From` 트레이트를 사용하여 에러 타입을 자동 변환합니다.

체이닝

?는 체이닝이 가능합니다:

```rust
use std::fs::File;
use std::io::{self, Read};

fn read_username_from_file() -> Result<String, io::Error> {
    let mut s = String::new();
    File::open("hello.txt")?.read_to_string(&mut s)?;
    Ok(s)
}
```

가장 짧은 버전

표준 라이브러리에 이미 있습니다:

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

fn read_username_from_file() -> Result<String, io::Error> {
    fs::read_to_string("hello.txt")
}
```

8.5 `?` 연산자와 `Option`

`?`는 `Option<T>`에도 사용할 수 있습니다:

```rust
fn last_char_of_first_line(text: &str) -> Option<char> {
    text.lines()       // 줄 이터레이터
        .next()?       // 첫 줄 (없으면 None 반환)
        .chars()       // 문자 이터레이터
        .last()        // 마지막 문자
}

fn main() {
    assert_eq!(last_char_of_first_line("Hello\nWorld"),
Some('o'));
    assert_eq!(last_char_of_first_line(""), None);
    assert_eq!(last_char_of_first_line("\nhi"), None);
}
```

Option과 Result 혼합

`?`는 같은 타입끼리만 사용할 수 있습니다:

```rust
fn mix_example() -> Option<i32> {
    let value: Result<i32, _> = "42".parse();
    // let n = value?;  // ❌ 에러! Option 함수에서 Result에 ? 사용 불가

    // ✅ ok()로 변환
    let n = value.ok()?;
    Some(n * 2)
}

fn mix_example2() -> Result<i32, String> {
    let values = vec![1, 2, 3];
    // let first = values.first()?;  // ❌ 에러! Option에 ?

    // ✅ ok_or로 변환
    let first = values.first().ok_or("빈 벡터")?;
    Ok(*first * 2)
}
```

8.6 main에서 Result 반환

`main` 함수도 `Result`를 반환할 수 있습니다:

```rust
use std::error::Error;
use std::fs::File;

fn main() -> Result<(), Box<dyn Error>> {
    let f = File::open("hello.txt")?;
    // ?를 main에서 직접 사용 가능!

    Ok(())
}
```

종료 코드

```rust
use std::process::ExitCode;

fn main() -> ExitCode {
    if some_condition {
        ExitCode::SUCCESS  // 0
    } else {
        ExitCode::FAILURE  // 1
    }
}
```

8.7 언제 `panic!`하고 언제 `Result`를 쓸까?

panic! 사용 가이드라인

상황	권장	이유
예제/프로토타입	✅ `unwrap`	빠른 작성
테스트	✅ `unwrap`	실패 시 테스트 실패
절대 실패 안 하는 로직	△ `expect`	의도 문서화
계약 위반	✅ `panic!`	버그 표시
복구 불가능한 상태	✅ `panic!`	안전한 종료

Result 사용 가이드라인

상황	권장	이유
파일 I/O	✅ `Result`	파일이 없을 수 있음
네트워크	✅ `Result`	연결 실패 가능
사용자 입력	✅ `Result`	잘못된 입력 가능
파싱	✅ `Result`	형식 오류 가능
라이브러리	✅ `Result`	호출자가 결정

계약에 의한 설계 (Design by Contract)

```
/// 1에서 100 사이의 추측 값을 표현합니다.
pub struct Guess {
    value: i32,
}

impl Guess {
    /// 새 Guess를 생성합니다.
    ///
    /// # Panics
    ///
    /// value가 1~100 범위를 벗어나면 패닉합니다.
    pub fn new(value: i32) -> Guess {
        if value < 1 || value > 100 {
            panic!("Guess 값은 1~100 사이여야 합니다. 입력: {}",
value);
        }

        Guess { value }
    }

    pub fn value(&self) -> i32 {
        self.value
    }
}

fn main() {
    let guess = Guess::new(50);  // OK
    println!("추측: {}", guess.value());

    // let invalid = Guess::new(200);  // 패닉!
}
```

Result를 사용한 버전

```
#[derive(Debug)]
pub enum GuessError {
    TooLow(i32),
    TooHigh(i32),
}

pub struct Guess {
    value: i32,
}

impl Guess {
    pub fn new(value: i32) -> Result<Guess, GuessError> {
        if value < 1 {
            return Err(GuessError::TooLow(value));
        }
        if value > 100 {
            return Err(GuessError::TooHigh(value));
        }
        Ok(Guess { value })
    }
}

fn main() {
    match Guess::new(200) {
        Ok(g) => println!("유효한 추측: {}", g.value),
        Err(GuessError::TooHigh(v)) => println!("{} 은(는) 너무
높습니다", v),
        Err(GuessError::TooLow(v)) => println!("{} 은(는) 너무 낮
습니다", v),
    }
}
```

8.8 커스텀 에러 타입

간단한 커스텀 에러

```
use std::fmt;

#[derive(Debug)]
pub enum DataError {
    NotFound,
    InvalidFormat(String),
    IoError(std::io::Error),
}

impl fmt::Display for DataError {
    fn fmt(&self, f: &mut fmt::Formatter<'_>) -> fmt::Result {
        match self {
            DataError::NotFound => write!(f, "데이터를 찾을 수 없습
니다"),
            DataError::InvalidFormat(msg) => write!(f, "형식 오
류: {}", msg),
            DataError::IoError(e) => write!(f, "I/O 오류: {}",
e),
        }
    }
}

impl std::error::Error for DataError {}

// io::Error에서 자동 변환
impl From<std::io::Error> for DataError {
    fn from(error: std::io::Error) -> Self {
        DataError::IoError(error)
    }
}
```

사용 예시

```
use std::fs;

fn read_config(path: &str) -> Result<String, DataError> {
    let content = fs::read_to_string(path)?;  // IoError 자동 변
환

    if content.is_empty() {
        return Err(DataError::NotFound);
    }

    if !content.starts_with("[config]") {
        return Err(DataError::InvalidFormat(
            "설정 헤더가 없습니다".to_string()
        ));
    }

    Ok(content)
}
```

---

thiserror 크레이트 (권장)

`thiserror`를 사용하면 보일러플레이트를 줄일 수 있습니다:

```rust
use thiserror::Error;

#[derive(Error, Debug)]
pub enum DataError {
    #[error("데이터를 찾을 수 없습니다")]
    NotFound,

    #[error("형식 오류: {0}")]
    InvalidFormat(String),

    #[error("I/O 오류")]
    IoError(#[from] std::io::Error),

    #[error("{kind} 오류: {message}")]
    Custom { kind: String, message: String },
}
```

anyhow 크레이트 (빠른 개발)

애플리케이션에서 에러 타입을 상세히 정의하지 않아도 될 때:

```rust
use anyhow::{Context, Result};

fn read_config() -> Result<String> {
    let content = std::fs::read_to_string("config.toml")
        .context("설정 파일을 읽을 수 없습니다")?;

    Ok(content)
}

fn main() -> Result<()> {
    let config = read_config()?;
    println!("{}", config);
    Ok(())
}
```

8.9 에러 처리 패턴

조기 반환 (Early Return)

```
fn process_data(input: &str) -> Result<i32, String> {
    if input.is_empty() {
        return Err("입력이 비어있습니다".to_string());
    }

    let number: i32 = input.parse().map_err(|_| "숫자가 아닙니
다")?;

    if number < 0 {
        return Err("음수는 허용되지 않습니다".to_string());
    }

    Ok(number * 2)
}
```

에러 컨텍스트 추가

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

fn read_user_config(user: &str) -> Result<String, String> {
    let path = format!("/home/{}/config.toml", user);

    fs::read_to_string(&path)
        .map_err(|e| format!("{} 사용자의 설정 읽기 실패: {}", user,
e))
}
```

여러 에러 타입 결합

```
use std::num::ParseIntError;
use std::io;

#[derive(Debug)]
enum AppError {
    Io(io::Error),
    Parse(ParseIntError),
}

impl From<io::Error> for AppError {
    fn from(e: io::Error) -> Self {
        AppError::Io(e)
    }
}

impl From<ParseIntError> for AppError {
    fn from(e: ParseIntError) -> Self {
        AppError::Parse(e)
    }
}

fn run() -> Result<i32, AppError> {
    let content = std::fs::read_to_string("number.txt")?;
    let number: i32 = content.trim().parse()?;
    Ok(number)
}
```

🚀 실습 과제

과제 1: 설정 파일 파서

```
// TODO: Config 구조체 정의
// - load("config.toml") -> Result<Config, ConfigError>
// - 파일 없음, 형식 오류, 필수 키 누락 처리
```

과제 2: 재시도 로직

```
// TODO: retry 함수 구현
// fn retry<T, E, F>(f: F, max_attempts: u32) -> Result<T, E>
// - 최대 max_attempts까지 재시도
// - 모두 실패하면 마지막 에러 반환
```

🚀 실습 가이드

`examples/` 폴더에서 확인:

1. `panic.rs`: 패닉과 백트레이스
2. `result_match.rs`: Result와 match 사용
3. `propagation.rs`: ? 연산자로 에러 전파

📚 핵심 정리

개념	설명
`panic!`	복구 불가능한 에러, 프로그램 종료
`Result<T, E>`	복구 가능한 에러, `Ok(T)` 또는 `Err(E)`
`unwrap()`	`Ok` → 값, `Err` → 패닉
`expect(msg)`	`unwrap` + 커스텀 메시지
`?` 연산자	에러 전파 축약 문법
`From` 트레이트	에러 타입 자동 변환
`thiserror`	커스텀 에러 보일러플레이트 제거
`anyhow`	빠른 개발용 에러 처리

Chapter 9: 제네릭 타입 (Generic Types)

> "제네릭은 코드 중복을 제거하면서 타입 안전성을 유지하는 강력한 도구입니다. Rust의 제네릭은 **런타임 비용이 없습니다** - 이것이 Zero-cost Abstraction입니다."

왜 제네릭이 필요한가?

비슷한 로직을 여러 타입에 대해 반복 작성하는 것은 유지보수의 악몽입니다:

```rust
// ❌ 중복 코드
fn largest_i32(list: &[i32]) -> &i32 {
    let mut largest = &list[0];
    for item in list {
        if item > largest {
            largest = item;
        }
    }
    largest
}

fn largest_char(list: &[char]) -> &char {
    let mut largest = &list[0];
    for item in list {  // 완전히 같은 로직!
        if item > largest {
            largest = item;
        }
    }
    largest
}

fn largest_f64(list: &[f64]) -> &f64 {
    // 또 같은 코드...
}
```

제네릭으로 **한 번만 작성**하면 됩니다:

```rust
// ✅ 제네릭으로 통합
fn largest<T: PartialOrd>(list: &[T]) -> &T {
    let mut largest = &list[0];
    for item in list {
        if item > largest {
            largest = item;
        }
    }
    largest
}

fn main() {
    let numbers = vec![34, 50, 25, 100, 65];
    println!("가장 큰 수: {}", largest(&numbers));

    let chars = vec!['y', 'm', 'a', 'q'];
    println!("가장 큰 문자: {}", largest(&chars));

    let floats = vec![1.1, 3.3, 2.2];
    println!("가장 큰 실수: {}", largest(&floats));
}
```

9.1 함수에서의 제네릭

기본 문법

```rust
// T는 타입 매개변수 (관례적으로 대문자 한 글자)
fn identity<T>(value: T) -> T {
    value
}

fn main() {
    let int_val = identity(5);        // T = i32
    let str_val = identity("hello"); // T = &str

    // 명시적 타입 지정 (터보피시 문법)
    let float_val = identity::<f64>(3.14);

    println!("{}, {}, {}", int_val, str_val, float_val);
}
```

여러 타입 매개변수

```rust
fn swap<T, U>(a: T, b: U) -> (U, T) {
    (b, a)
}

fn main() {
    let result = swap(1, "hello");
    println!("{:?}", result);  // ("hello", 1)
}
```

타입 매개변수 네이밍 관례

매개변수	일반적 의미
`T`	Type (일반적인 타입)
`E`	Error (에러 타입)
`K`	Key (키 타입)
`V`	Value (값 타입)
`S`	State (상태)

```
// HashMap<K, V>: K=Key, V=Value
// Result<T, E>: T=성공 타입, E=에러 타입
```

9.2 구조체에서의 제네릭

단일 타입 매개변수

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

fn main() {
    let integer_point = Point { x: 5, y: 10 };
    let float_point = Point { x: 1.0, y: 4.0 };

    println!("{:?}", integer_point);  // Point { x: 5, y: 10 }
    println!("{:?}", float_point);    // Point { x: 1.0, y: 4.0
}

    // ❌ x와 y는 같은 타입이어야 함!
    // let mixed = Point { x: 5, y: 4.0 };  // 에러!
}
```

다중 타입 매개변수

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

fn main() {
    let integer_point = Point { x: 5, y: 10 };        // T=i32,
U=i32
    let float_point = Point { x: 1.0, y: 4.0 };       // T=f64,
U=f64
    let mixed_point = Point { x: 5, y: 4.0 };         // T=i32,
U=f64

    println!("{:?}", mixed_point);  // Point { x: 5, y: 4.0 }
}
```

실제 사용 예시: 결과 래퍼

```
#[derive(Debug)]
struct ApiResponse<T> {
    data: T,
    status_code: u16,
    message: String,
}

struct User {
    name: String,
}

struct Product {
    title: String,
    price: f64,
}

fn main() {
    let user_response = ApiResponse {
        data: User { name: String::from("Alice") },
        status_code: 200,
        message: String::from("Success"),
    };

    let product_response = ApiResponse {
        data: Product {
            title: String::from("Rust Book"),
            price: 29.99,
        },
        status_code: 200,
        message: String::from("Success"),
    };

    println!("User: {}", user_response.data.name);
```

```
    println!("Product: ${}", product_response.data.price);
}
```

9.3 열거형에서의 제네릭

이미 사용해본 `Option<T>`와 `Result<T, E>`가 제네릭 열거형입니다!

`Option<T>` 구조

```
// 표준 라이브러리 정의
enum Option<T> {
    Some(T),
    None,
}

fn main() {
    let some_number: Option<i32> = Some(5);
    let some_string: Option<String> =
Some(String::from("hello"));
    let no_value: Option<i32> = None;
}
```

`Result<T, E>` 구조

```
// 표준 라이브러리 정의
enum Result<T, E> {
    Ok(T),   // 성공: T 타입 값
    Err(E),  // 실패: E 타입 에러
}

use std::fs::File;
use std::io::Error;

fn main() {
    let file_result: Result<File, Error> =
File::open("hello.txt");

    match file_result {
        Ok(file) => println!("파일 열기 성공"),
        Err(e) => println!("에러: {}", e),
    }
}
```

커스텀 제네릭 열거형

```
#[derive(Debug)]
enum Tree<T> {
    Leaf(T),
    Node {
        value: T,
        left: Box<Tree<T>>,
        right: Box<Tree<T>>,
    },
    Empty,
}

fn main() {
    let tree = Tree::Node {
        value: 5,
        left: Box::new(Tree::Leaf(3)),
        right: Box::new(Tree::Node {
            value: 7,
            left: Box::new(Tree::Empty),
            right: Box::new(Tree::Leaf(9)),
        }),
    };

    println!("{:?}", tree);
}
```

9.4 메서드에서의 제네릭

기본 메서드 정의

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

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

    fn y(&self) -> &T {
        &self.y
    }

    fn new(x: T, y: T) -> Self {
        Self { x, y }
    }
}

fn main() {
    let p = Point::new(5, 10);
    println!("x = {}", p.x());
}
```

`impl<T>` 문법: `impl` 뒤의 `<T>`는 “T가 제네릭임을 선언“하고, `Point<T>` 뒤의 `<T>`는 “이 타입 매개변수를 사용“한다는 의미입니다.

특정 타입에만 메서드 구현

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

// 모든 Point<T>에 대해
impl<T> Point<T> {
    fn new(x: T, y: T) -> Self {
        Self { x, y }
    }
}

// Point<f32>에만
impl Point<f32> {
    fn distance_from_origin(&self) -> f32 {
        (self.x.powi(2) + self.y.powi(2)).sqrt()
    }
}

// Point<f64>에만
impl Point<f64> {
    fn distance_from_origin(&self) -> f64 {
        (self.x.powi(2) + self.y.powi(2)).sqrt()
    }
}

fn main() {
    let p_f32 = Point::<f32>::new(3.0, 4.0);
    println!("f32 거리: {}", p_f32.distance_from_origin());  //
5.0

    let p_i32 = Point::new(3, 4);
    // p_i32.distance_from_origin();  // ❌ Point<i32>에는 없음!
}
```

다른 타입 매개변수 사용

메서드가 구조체와 다른 타입 매개변수를 가질 수 있습니다:

```
#[derive(Debug)]
struct Point<X1, Y1> {
    x: X1,
    y: Y1,
}

impl<X1, Y1> Point<X1, Y1> {
    // 메서드만의 타입 매개변수 X2, Y2
    fn mixup<X2, Y2>(self, other: Point<X2, Y2>) -> Point<X1,
Y2> {
        Point {
            x: self.x,   // 원본의 x
            y: other.y,  // other의 y
        }
    }
}

fn main() {
    let p1 = Point { x: 5, y: 10.4 };
    let p2 = Point { x: "Hello", y: 'c' };

    let p3 = p1.mixup(p2);

    println!("p3 = {:?}", p3);  // Point { x: 5, y: 'c' }
    // p3의 타입: Point<i32, char>
}
```

9.5 단형성화 (Monomorphization)

> “제네릭은 **런타임 비용이 없습니다**.” (Zero-cost Abstraction)

컴파일 타임 최적화

Rust 컴파일러는 제네릭 코드를 **구체적인 타입으로 확장**합니다:

```
// 작성한 코드
let integer = Some(5);
let float = Some(5.0);

// 컴파일러가 생성하는 코드 (개념적으로)
enum Option_i32 {
    Some(i32),
    None,
}

enum Option_f64 {
    Some(f64),
    None,
}

let integer = Option_i32::Some(5);
let float = Option_f64::Some(5.0);
```

성능 비교

```rust
use std::time::Instant;

// 제네릭 버전
fn generic_sum<T: std::ops::Add<Output = T> + Copy>(slice: &
[T]) -> T
where
    T: Default,
{
    let mut sum = T::default();
    for &item in slice {
        sum = sum + item;
    }
    sum
}

// 구체적 버전
fn concrete_sum(slice: &[i32]) -> i32 {
    let mut sum = 0;
    for &item in slice {
        sum = sum + item;
    }
    sum
}

fn main() {
    let numbers: Vec<i32> = (0..1_000_000).collect();

    // 동일한 성능! (단형성화 덕분)
    let start = Instant::now();
    let _sum1 = generic_sum(&numbers);
    println!("제네릭: {:?}", start.elapsed());

    let start = Instant::now();
    let _sum2 = concrete_sum(&numbers);
```

```
    println!("구체적: {:?}", start.elapsed());
}
```

9.6 트레이트 바운드 (Trait Bounds)

제네릭 타입에 **제약**을 걸어 특정 기능을 보장합니다.

기본 문법

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

// T는 Display를 구현해야 함
fn print_value<T: Display>(value: T) {
    println!("값: {}", value);
}

fn main() {
    print_value(42);
    print_value("hello");
    // print_value(vec![1, 2, 3]);  // ❌ Vec<i32>는 Display 미구현
}
```

여러 트레이트 요구

```rust
use std::fmt::{Debug, Display};

// 방법 1: + 문법
fn compare_and_display<T: PartialOrd + Display>(a: T, b: T) {
    if a > b {
        println!("{} > {}", a, b);
    } else {
        println!("{} <= {}", a, b);
    }
}

// 방법 2: where 절 (가독성 향상)
fn complex_function<T, U>(t: &T, u: &U) -> i32
where
    T: Display + Clone,
    U: Clone + Debug,
{
    // ...
    0
}
```

where 절을 사용하는 이유

```
// ❌ 읽기 어려움
fn some_function<T: Display + Clone + PartialEq, U: Clone +
Debug + Default>(t: &T, u: &U) {
    // ...
}

// ✅ 훨씬 명확함
fn some_function<T, U>(t: &T, u: &U)
where
    T: Display + Clone + PartialEq,
    U: Clone + Debug + Default,
{
    // ...
}
```

조건부 메서드 구현

특정 트레이트를 구현한 타입에만 메서드 추가:

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

struct Pair<T> {
    x: T,
    y: T,
}

impl<T> Pair<T> {
    fn new(x: T, y: T) -> Self {
        Self { x, y }
    }
}

// Display + PartialOrd를 구현한 T에만
impl<T: Display + PartialOrd> Pair<T> {
    fn cmp_display(&self) {
        if self.x >= self.y {
            println!("가장 큰 값: x = {}", self.x);
        } else {
            println!("가장 큰 값: y = {}", self.y);
        }
    }
}

fn main() {
    let pair = Pair::new(5, 10);
    pair.cmp_display();  // ✅ i32는 Display + PartialOrd 구현

    // 커스텀 타입
    struct NoDisplay(i32);
    let pair2 = Pair::new(NoDisplay(1), NoDisplay(2));
    // pair2.cmp_display();  // ❌ NoDisplay는 Display 미구현
}
```

9.7 impl Trait (간결한 문법)

매개변수에서 impl Trait

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

// impl Trait 문법 (간결)
pub fn notify(item: &impl Summary) {
    println!("Breaking news! {}", item.summarize());
}

// 동일: 트레이트 바운드 문법 (명시적)
pub fn notify_verbose<T: Summary>(item: &T) {
    println!("Breaking news! {}", item.summarize());
}
```

언제 어떤 문법을 사용할까?

```
// impl Trait: 간단한 경우
fn process(item: &impl Display) { ... }

// 트레이트 바운드: 같은 타입 강제 시
fn compare<T: Display>(a: &T, b: &T) { ... }

// impl Trait로는 불가능:
// fn compare_wrong(a: &impl Display, b: &impl Display)
// ↑ a와 b가 다른 타입일 수 있음!
```

반환 타입에서 impl Trait

```rust
trait Summary {
    fn summarize(&self) -> String;
}

struct Article {
    title: String,
    content: String,
}

impl Summary for Article {
    fn summarize(&self) -> String {
        format!("{}: {}...", self.title,
&self.content[..50.min(self.content.len())])
    }
}

struct Tweet {
    username: String,
    text: String,
}

impl Summary for Tweet {
    fn summarize(&self) -> String {
        format!("@{}: {}", self.username, self.text)
    }
}

// 구체적 타입을 숨기고 트레이트만 노출
fn create_article() -> impl Summary {
    Article {
        title: String::from("Rust 2024"),
        content: String::from("Rust continues to grow..."),
    }
}
```

```rust
fn main() {
    let item = create_article();
    println!("{}", item.summarize());
}
```

⚠️ 제한사항

`impl Trait` 반환은 **단일 타입만** 반환할 수 있습니다:

```rust
// ❌ 컴파일 에러!
fn random_summary(switch: bool) -> impl Summary {
    if switch {
        Article { ... }  // Article 타입
    } else {
        Tweet { ... }    // Tweet 타입 - 다른 타입!
    }
}
```

조건에 따라 다른 타입을 반환하려면 **트레이트 객체**를 사용해야 합니다:

```rust
// ✅ 트레이트 객체 사용
fn random_summary(switch: bool) -> Box<dyn Summary> {
    if switch {
        Box::new(Article { ... })
    } else {
        Box::new(Tweet { ... })
    }
}
```

9.8 블랭킷 구현 (Blanket Implementation)

특정 트레이트를 구현한 모든 타입에 다른 트레이트를 자동 구현:

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

// 표준 라이브러리의 실제 코드 (개념적)
impl<T: Display> ToString for T {
    fn to_string(&self) -> String {
        format!("{}", self)
    }
}

fn main() {
    // Display를 구현한 모든 타입에서 to_string() 사용 가능!
    let s = 3.to_string();
    let s2 = "hello".to_string();
    let s3 = 3.14.to_string();
}
```

커스텀 블랭킷 구현

```rust
trait Describable {
    fn describe(&self) -> String;
}

// Debug를 구현한 모든 타입에 Describable 자동 구현
impl<T: std::fmt::Debug> Describable for T {
    fn describe(&self) -> String {
        format!("Debug 표현: {:?}", self)
    }
}

fn main() {
    let vec = vec![1, 2, 3];
    println!("{}", vec.describe());  // "Debug 표현: [1, 2, 3]"

    let option = Some(42);
    println!("{}", option.describe());  // "Debug 표현:
Some(42)"
}
```

9.9 Const 제네릭

배열 크기 등 **컴파일 타임 상수**를 타입 매개변수로 사용:

```rust
// N은 const 제네릭 (컴파일 타임 상수)
fn print_array<T: std::fmt::Debug, const N: usize>(arr: [T; N])
{
    println!("{:?}", arr);
}

fn main() {
    print_array([1, 2, 3]);      // N = 3
    print_array([1, 2, 3, 4, 5]); // N = 5
}
```

실용 예시: 고정 크기 버퍼

```
struct Buffer<const SIZE: usize> {
    data: [u8; SIZE],
    len: usize,
}

impl<const SIZE: usize> Buffer<SIZE> {
    fn new() -> Self {
        Buffer {
            data: [0; SIZE],
            len: 0,
        }
    }

    fn capacity(&self) -> usize {
        SIZE
    }

    fn push(&mut self, byte: u8) -> bool {
        if self.len < SIZE {
            self.data[self.len] = byte;
            self.len += 1;
            true
        } else {
            false
        }
    }
}

fn main() {
    let mut small: Buffer<16> = Buffer::new();
    let mut large: Buffer<1024> = Buffer::new();

    println!("Small 용량: {}", small.capacity());  // 16
```

```
    println!("Large 용량: {}", large.capacity());  // 1024
}
```

🚀 실습 과제

과제 1: 제네릭 스택

```
// TODO: Stack<T> 구현
// - push(item: T)
// - pop() -> Option<T>
// - peek() -> Option<&T>
// - is_empty() -> bool
// - len() -> usize
```

과제 2: 제네릭 캐시

```
// TODO: Cache<K, V> 구현
// - get(key: &K) -> Option<&V>
// - insert(key: K, value: V)
// - remove(key: &K) -> Option<V>
// - contains(key: &K) -> bool
```

🚀 실습 가이드

`examples/` 폴더에서 확인:

1. `generic_functions.rs`: 제네릭 함수
2. `generic_structs.rs`: 제네릭 구조체와 메서드
3. `trait_bounds.rs`: 트레이트 바운드 활용

📚 핵심 정리

개념	설명
제네릭	타입을 매개변수화 `<T>`
단형성화	컴파일 타임에 구체 타입으로 확장
Zero-cost	런타임 오버헤드 없음
트레이트 바운드	제네릭 타입의 기능 제약
where 절	복잡한 바운드 가독성 향상
`impl Trait`	간결한 트레이트 바운드/반환
블랭킷 구현	조건부 자동 트레이트 구현
Const 제네릭	컴파일 타임 상수 매개변수

Chapter 10: 트레이트 (Traits)

> “트레이트는 타입이 구현해야 하는 **공유 동작(Shared Behavior)**을 정의합니다. Java의 Interface, Haskell의 Typeclass와 유사하지만, Rust만의 강력한 기능을 제공합니다.”

왜 트레이트가 필요한가?

서로 다른 타입이 **같은 방식으로 동작**해야 할 때가 있습니다:

```
// ❌ 타입마다 별도 함수 필요
fn print_article_summary(article: &NewsArticle) { ... }
fn print_tweet_summary(tweet: &Tweet) { ... }
fn print_blog_summary(blog: &BlogPost) { ... }
// 새 타입 추가할 때마다 함수 추가...
```

트레이트를 사용하면:

```
// ✅ 하나의 함수로 모든 Summary 구현체 처리
fn print_summary(item: &impl Summary) {
    println!("{}", item.summarize());
}
```

트레이트의 핵심 역할

역할	설명
추상화	구현 세부사항을 숨기고 인터페이스만 노출
다형성	다양한 타입을 동일하게 처리
코드 재사용	공통 동작을 한 번만 정의
제네릭 제약	타입 매개변수의 기능 보장

10.1 트레이트 정의하기

트레이트는 `trait` 키워드로 정의합니다:

```
pub trait Summary {
    // 메서드 시그니처만 정의 (구현은 타입에 맡김)
    fn summarize(&self) -> String;
}
```

여러 메서드와 연관 타입

```
pub trait Container {
    // 연관 타입 (Associated Type)
    type Item;

    // 여러 메서드
    fn len(&self) -> usize;
    fn is_empty(&self) -> bool;
    fn get(&self, index: usize) -> Option<&Self::Item>;
    fn push(&mut self, item: Self::Item);
}
```

연관 상수

```
trait Bounded {
    const MIN: Self;
    const MAX: Self;
}

impl Bounded for i32 {
    const MIN: i32 = i32::MIN;
    const MAX: i32 = i32::MAX;
}
```

10.2 타입에 트레이트 구현하기

`impl TraitName for TypeName` 문법을 사용합니다:

```rust
pub struct NewsArticle {
    pub headline: String,
    pub location: String,
    pub author: String,
    pub content: String,
}

impl Summary for NewsArticle {
    fn summarize(&self) -> String {
        format!("{}, by {} ({})", self.headline, self.author,
self.location)
    }
}

pub struct Tweet {
    pub username: String,
    pub content: String,
    pub reply: bool,
    pub retweet: bool,
}

impl Summary for Tweet {
    fn summarize(&self) -> String {
        format!("@{}: {}", self.username, self.content)
    }
}

fn main() {
    let article = NewsArticle {
        headline: String::from("Rust 2.0 Released!"),
        location: String::from("서울"),
        author: String::from("김개발"),
        content: String::from("..."),
    };

    let tweet = Tweet {
        username: String::from("rustacean"),
```

```
        content: String::from("Rust is amazing!"),
        reply: false,
        retweet: false,
    };

    // 두 타입 모두 summarize() 호출 가능
    println!("기사: {}", article.summarize());
    println!("트윗: {}", tweet.summarize());
}
```

10.3 기본 구현 (Default Implementation)

트레이트 메서드에 기본 구현을 제공할 수 있습니다:

```rust
pub trait Summary {
    // 기본 구현 없음 (구현 필수)
    fn summarize_author(&self) -> String;

    // 기본 구현 제공 (오버라이드 가능)
    fn summarize(&self) -> String {
        format!("(Read more from {}...)",
self.summarize_author())
    }
}

// Tweet: summarize_author만 구현, summarize는 기본 구현 사용
impl Summary for Tweet {
    fn summarize_author(&self) -> String {
        format!("@{}", self.username)
    }
    // summarize()는 기본 구현 사용: "(Read more from
@username...)"
}

// NewsArticle: 둘 다 직접 구현
impl Summary for NewsArticle {
    fn summarize_author(&self) -> String {
        self.author.clone()
    }

    // 기본 구현을 오버라이드
    fn summarize(&self) -> String {
        format!("{} - {}", self.headline, self.author)
    }
}
```

기본 구현에서 다른 메서드 호출

```rust
pub trait Greet {
    fn name(&self) -> &str;

    fn greeting(&self) -> String {
        String::from("Hello")
    }

    // 다른 메서드 호출
    fn greet(&self) -> String {
        format!("{}, {}!", self.greeting(), self.name())
    }
}

struct Person {
    name: String,
}

impl Greet for Person {
    fn name(&self) -> &str {
        &self.name
    }
    // greeting()과 greet()은 기본 구현 사용
}

struct FormalPerson {
    name: String,
}

impl Greet for FormalPerson {
    fn name(&self) -> &str {
        &self.name
    }

    fn greeting(&self) -> String {
        String::from("Good evening")
```

```
    }
    // greet()은 기본 구현 사용하지만 greeting()이 오버라이드되어
    // "Good evening, 이름!" 출력
}
```

10.4 트레이트를 매개변수로 사용

impl Trait 문법 (간결)

```
pub fn notify(item: &impl Summary) {
    println!("Breaking news! {}", item.summarize());
}

// Summary를 구현한 어떤 타입이든 받을 수 있음
fn main() {
    notify(&article);  // NewsArticle
    notify(&tweet);    // Tweet
}
```

트레이트 바운드 문법 (명시적)

```
pub fn notify<T: Summary>(item: &T) {
    println!("Breaking news! {}", item.summarize());
}
```

두 매개변수가 같은 타입이어야 할 때

```rust
// impl Trait: 다른 타입 가능
pub fn notify_different(item1: &impl Summary, item2: &impl
Summary) {
    // item1과 item2가 다른 타입일 수 있음
}

// 트레이트 바운드: 같은 타입 강제
pub fn notify_same<T: Summary>(item1: &T, item2: &T) {
    // item1과 item2는 반드시 같은 타입
}
```

여러 트레이트 바운드

```rust
use std::fmt::{Display, Debug};

// + 문법으로 여러 트레이트 요구
pub fn notify(item: &(impl Summary + Display)) {
    println!("{}", item);  // Display
    println!("{}", item.summarize());  // Summary
}

// 트레이트 바운드 버전
pub fn notify<T: Summary + Display>(item: &T) {
    println!("{}", item);
    println!("{}", item.summarize());
}

// 여러 타입 매개변수와 여러 바운드
pub fn complex_function<T: Display + Clone, U: Clone + Debug>
(t: &T, u: &U) {
    // ...
}
```

where 절 (가독성 향상)

바운드가 복잡해지면 `where` 절을 사용합니다:

```rust
// ❌ 읽기 어려움
fn some_function<T: Display + Clone + PartialEq, U: Clone +
Debug + Default>(t: &T, u: &U) -> i32 {
    // ...
}

// ✅ where 절로 가독성 향상
fn some_function<T, U>(t: &T, u: &U) -> i32
where
    T: Display + Clone + PartialEq,
    U: Clone + Debug + Default,
{
    // ...
}
```

10.5 트레이트를 반환 타입으로

impl Trait 반환

구체적 타입을 숨기고 트레이트만 노출합니다:

```rust
fn returns_summarizable() -> impl Summary {
    Tweet {
        username: String::from("horse_ebooks"),
        content: String::from("of course, as you probably
already know, people"),
        reply: false,
        retweet: false,
    }
}

fn main() {
    let item = returns_summarizable();
    println!("{}", item.summarize());
    // item의 구체적 타입(Tweet)은 알 수 없음
}
```

⚠️ 제한: 단일 타입만 반환 가능

```rust
// ❌ 컴파일 에러! 다른 타입 반환
fn returns_summarizable(switch: bool) -> impl Summary {
    if switch {
        NewsArticle { ... }  // NewsArticle 타입
    } else {
        Tweet { ... }        // Tweet 타입 - 다름!
    }
}
```

조건에 따라 다른 타입을 반환하려면 트레이트 객체를 사용합니다:

```rust
// ✅ 트레이트 객체 사용
fn returns_summarizable(switch: bool) -> Box<dyn Summary> {
    if switch {
        Box::new(NewsArticle { ... })
    } else {
        Box::new(Tweet { ... })
    }
}
```

10.6 트레이트 바운드로 조건부 구현

특정 트레이트를 구현한 타입에만 메서드를 추가할 수 있습니다:

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

struct Pair<T> {
    x: T,
    y: T,
}

// 모든 Pair<T>에 대해
impl<T> Pair<T> {
    fn new(x: T, y: T) -> Self {
        Self { x, y }
    }
}

// T가 Display + PartialOrd를 구현할 때만
impl<T: Display + PartialOrd> Pair<T> {
    fn cmp_display(&self) {
        if self.x >= self.y {
            println!("가장 큰 값: x = {}", self.x);
        } else {
            println!("가장 큰 값: y = {}", self.y);
        }
    }
}

fn main() {
    let pair_i32 = Pair::new(5, 10);
    pair_i32.cmp_display();  // ✅ i32는 Display + PartialOrd

    struct NoDisplay(i32);
    let pair_no = Pair::new(NoDisplay(1), NoDisplay(2));
    // pair_no.cmp_display();  // ❌ NoDisplay는 Display 미구현
}
```

10.7 블랭킷 구현 (Blanket Implementations)

특정 트레이트를 구현한 모든 타입에 다른 트레이트를 자동 구현합니다:

```
// 표준 라이브러리의 실제 코드 (개념적)
impl<T: Display> ToString for T {
    fn to_string(&self) -> String {
        format!("{}", self)
    }
}
```

이 덕분에:

```
fn main() {
    // Display를 구현한 모든 타입에서 to_string() 사용 가능!
    let s = 3.to_string();           // i32
    let s2 = 3.14.to_string();       // f64
    let s3 = true.to_string();       // bool
    let s4 = "hello".to_string();    // &str
}
```

커스텀 블랭킷 구현

```
trait Printable {
    fn print(&self);
}

// Debug를 구현한 모든 타입에 Printable 자동 구현
impl<T: std::fmt::Debug> Printable for T {
    fn print(&self) {
        println!("{:?}", self);
    }
}

fn main() {
    vec![1, 2, 3].print();      // [1, 2, 3]
    Some(42).print();           // Some(42)
    "hello".print();             // "hello"
}
```

10.8 고아 규칙 (Orphan Rule)

트레이트 구현의 일관성을 보장하기 위한 규칙입니다:

> 규칙: 트레이트나 타입 중 하나는 반드시 우리 크레이트에 정의되어야 합니다.

상황	허용 여부
내 타입 + 외부 트레이트	✅

상황	허용 여부
외부 타입 + 내 트레이트	✅
외부 타입 + 외부 트레이트	❌

```rust
// ✅ OK: 우리 타입(Tweet)에 표준 트레이트(Display) 구현
impl std::fmt::Display for Tweet {
    fn fmt(&self, f: &mut std::fmt::Formatter) ->
std::fmt::Result {
        write!(f, "@{}: {}", self.username, self.content)
    }
}

// ✅ OK: 표준 타입(Vec<i32>)에 우리 트레이트(Summary) 구현
impl Summary for Vec<i32> {
    fn summarize(&self) -> String {
        format!("벡터: {} 요소", self.len())
    }
}

// ❌ 금지: 표준 타입(Vec<i32>)에 표준 트레이트(Display) 구현
// impl std::fmt::Display for Vec<i32> { ... }
// 에러: orphan rule 위반!
```

Newtype 패턴으로 우회

```
// Vec<String>에 Display를 구현하고 싶다면?
struct Wrapper(Vec<String>);

impl std::fmt::Display for Wrapper {
    fn fmt(&self, f: &mut std::fmt::Formatter) ->
std::fmt::Result {
        write!(f, "[{}]", self.0.join(", "))
    }
}

fn main() {
    let w = Wrapper(vec!["hello".to_string(),
"world".to_string()]);
    println!("{}", w);  // [hello, world]
}
```

10.9 트레이트 객체 (Trait Objects)

정적 디스패치 vs 동적 디스패치

```
// 정적 디스패치: 컴파일 타임에 단형성화
fn static_dispatch(item: &impl Summary) {
    println!("{}", item.summarize());
}

// 동적 디스패치: 런타임에 메서드 결정
fn dynamic_dispatch(item: &dyn Summary) {
    println!("{}", item.summarize());
}
```

특성	정적 디스패치	동적 디스패치
키워드	`impl Trait`	`dyn Trait`
타입 결정	컴파일 타임	런타임
성능	인라인 가능, 더 빠름	vtable 조회 오버헤드
바이너리 크기	더 큼 (단형성화)	더 작음
유연성	단일 타입	다양한 타입 혼합

트레이트 객체 사용 예시

```rust
// 다양한 타입을 하나의 벡터에 저장
fn main() {
    let article = NewsArticle { ... };
    let tweet = Tweet { ... };

    // Box<dyn Trait>로 런타임 다형성
    let items: Vec<Box<dyn Summary>> = vec![
        Box::new(article),
        Box::new(tweet),
    ];

    for item in items {
        println!("{}", item.summarize());
    }
}
```

객체 안전성 (Object Safety)

트레이트 객체로 사용하려면 트레이트가 객체 안전해야 합니다:

```rust
// ✅ 객체 안전
trait Draw {
    fn draw(&self);
}

// ❌ 객체 불안전: Self를 반환
trait Clone {
    fn clone(&self) -> Self;
}

// ❌ 객체 불안전: 제네릭 메서드
trait NotObjectSafe {
    fn foo<T>(&self, item: T);
}
```

객체 안전 규칙:

1. 반환 타입이 `Self`가 아니어야 함
2. 제네릭 타입 매개변수가 없어야 함

10.10 수퍼트레이트 (Supertraits)

트레이트가 다른 트레이트를 요구할 수 있습니다:

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

// OutlinePrint는 Display를 요구
trait OutlinePrint: Display {
    fn outline_print(&self) {
        let output = self.to_string();  // Display 메서드 사용 가능
        let len = output.len();

        println!("{}", "*".repeat(len + 4));
        println!("* {} *", output);
        println!("{}", "*".repeat(len + 4));
    }
}

struct Point { x: i32, y: i32 }

// Display를 먼저 구현
impl Display for Point {
    fn fmt(&self, f: &mut std::fmt::Formatter) ->
std::fmt::Result {
        write!(f, "({}, {})", self.x, self.y)
    }
}

// 그 후 OutlinePrint 구현 가능
impl OutlinePrint for Point {}

fn main() {
    let p = Point { x: 1, y: 3 };
    p.outline_print();
    // **********
    // * (1, 3) *
    // **********
}
```

10.11 표준 라이브러리의 중요 트레이트

자주 사용되는 트레이트

트레이트	용도	메서드
`Debug`	디버그 출력	`{:?}` 포맷
`Display`	사용자 출력	`{}` 포맷
`Clone`	명시적 복제	`.clone()`
`Copy`	암묵적 복사	자동
`Default`	기본값 생성	`::default()`
`PartialEq`/`Eq`	동등성 비교	`==`, `!=`
`PartialOrd`/`Ord`	순서 비교	`<`, `>`, `<=`, `>=`
`Hash`	해시 계산	`HashMap`/`HashSet` 키
`From`/`Into`	타입 변환	`.into()`
`Drop`	소멸자	스코프 종료 시

derive로 자동 구현

```
#[derive(Debug, Clone, PartialEq, Eq, Hash, Default)]
struct User {
    id: u64,
    name: String,
}
```

🚀 실습 과제

과제 1: Shape 트레이트

```
// TODO: Shape 트레이트 정의
// - area(&self) -> f64
// - perimeter(&self) -> f64

// TODO: Circle, Rectangle, Triangle 구현
```

과제 2: Serializable 트레이트

```
// TODO: Serializable 트레이트 정의
// - to_json(&self) -> String
// - from_json(json: &str) -> Result<Self, Error>
```

🚀 실습 가이드

`examples/` 폴더에서 확인:

1. `traits_basic.rs`: 트레이트 정의와 구현
2. `trait_bounds.rs`: 트레이트 바운드
3. `trait_objects.rs`: 동적 디스패치

📚 핵심 정리

개념	설명
트레이트	공유 동작 정의 (인터페이스)
기본 구현	트레이트에서 메서드 기본 동작 제공
트레이트 바운드	제네릭 타입의 기능 제약
`impl Trait`	간결한 트레이트 바운드/반환
`dyn Trait`	동적 디스패치 (트레이트 객체)
블랭킷 구현	조건부 자동 트레이트 구현
고아 규칙	트레이트/타입 중 하나는 로컬
수퍼트레이트	트레이트가 다른 트레이트 요구
객체 안전성	트레이트 객체 사용 조건

Chapter 11: 라이프타임 (Lifetimes)

> "라이프타임은 참조가 유효한 범위를 컴파일러에게 알려주는 주석입니다. 모든 검사는 컴파일 타임에 이루어지므로 **런타임 비용이 전혀 없습니다**."

왜 라이프타임이 필요한가?

댕글링 참조 (Dangling Reference) 문제

C/C++에서 흔히 발생하는 위험한 버그입니다:

```
// C: 위험한 코드! 해제된 메모리 접근
int* get_value() {
    int x = 42;
    return &x;  // x는 함수 종료 시 사라짐!
}

int main() {
    int* ptr = get_value();
    printf("%d\n", *ptr);  // 💥 정의되지 않은 동작!
}
```

Rust는 이를 **컴파일 타임에 완전히 방지**합니다:

```rust
fn main() {
    let r;              // 참조를 담을 변수
    {
        let x = 5;
        r = &x;         // ❌ 컴파일 에러!
    }                   // x는 여기서 소멸
    println!("{}", r);  // r은 댕글링 참조가 됨
}
```

컴파일러 출력:

```
error[E0597]: `x` does not live long enough
 --> src/main.rs:5:13
  |
4 |         let x = 5;
5 |         r = &x;
  |             ^^ borrowed value does not live long enough
6 |     }
  |     - `x` dropped here while still borrowed
7 |     println!("{}", r);
  |                    - borrow later used here
```

11.1 빌림 검사기 (Borrow Checker)

Rust 컴파일러의 핵심 컴포넌트가 **모든 참조의 유효성**을 검증합니다:

```rust
fn main() {
    let x = 5;              // ------------+-- 'a
                            //             |
    let r = &x;             // ---+-- 'b   |
                            //    |        |
    println!("{}", r);      // ---+        |
                            //             |
}                           // ------------+

// 'b(r의 생명주기) ⊂ 'a(x의 생명주기)
// r은 x가 유효한 동안에만 사용됨 ✅
```

유효하지 않은 경우

```rust
fn main() {
    let r;                  // ------------+-- 'a
    {                       //             |
        let x = 5;          // ---+-- 'b   |
        r = &x;             //    |        |  ❌ 'a가 'b보다 오래 삶
    }                       // ---+        |
                            //             |
    println!("{}", r);      // ------------+
}
```

11.2 함수에서의 라이프타임

문제 상황

```rust
// ❌ 컴파일 에러!
fn longest(x: &str, y: &str) -> &str {
    if x.len() > y.len() {
        x
    } else {
        y
    }
}
```

에러 메시지:

```
error[E0106]: missing lifetime specifier
 --> src/main.rs:1:33
  |
1 | fn longest(x: &str, y: &str) -> &str {
  |               ----     ----     ^ expected named lifetime
parameter
  |
  = help: this function's return type contains a borrowed
value, but the
          signature does not say whether it is borrowed from
`x` or `y`
```

문제: 컴파일러는 반환되는 참조가 `x`와 `y` 중 어느 것의 라이프타임을 따르는지 알 수 없습니다.

라이프타임 어노테이션

라이프타임 매개변수로 관계를 명시합니다:

```
fn longest<'a>(x: &'a str, y: &'a str) -> &'a str {
    if x.len() > y.len() {
        x
    } else {
        y
    }
}
```

의미: "x와 y는 같은 라이프타임 `'a`를 가지며, 반환값도 `'a` 동안 유효하다"

> 💡 중요: 라이프타임 어노테이션은 참조의 실제 수명을 바꾸지 않습니다. 컴파일러에게 관계를 알려줄 뿐입니다.

라이프타임 문법

```
&i32        // 참조
&'a i32     // 명시적 라이프타임이 있는 참조
&'a mut i32 // 명시적 라이프타임이 있는 가변 참조
```

라이프타임의 실제 의미

`'a`는 x와 y 중 더 짧은 라이프타임으로 결정됩니다:

```rust
fn main() {
    let string1 = String::from("long string is long");

    {
        let string2 = String::from("xyz");
        let result = longest(string1.as_str(),
string2.as_str());
        println!("가장 긴 문자열: {}", result);  // ✅ 블록 내에서 사
용
    }
    // result는 여기서 더 이상 유효하지 않음
    // (string2의 라이프타임에 의해 제한됨)
}
```

잘못된 사용

```rust
fn main() {
    let string1 = String::from("long string is long");
    let result;
    {
        let string2 = String::from("xyz");
        result = longest(string1.as_str(), string2.as_str());
    }
    // ❌ 컴파일 에러! string2가 소멸된 후 result 사용
    println!("가장 긴 문자열: {}", result);
}
```

함수 설계와 라이프타임

반환값의 라이프타임은 **매개변수의 라이프타임**과 연결되어야 합니다:

```rust
// ✅ OK: 반환값이 x와 연결됨
fn first<'a>(x: &'a str, y: &str) -> &'a str {
    x
}

// ❌ 에러: 반환값이 로컬 변수
fn wrong<'a>(x: &'a str, y: &'a str) -> &'a str {
    let result = String::from("really long string");
    result.as_str()  // 에러! 로컬 변수 참조 반환
}

// ✅ 해결: 소유권 이전
fn correct(x: &str, y: &str) -> String {
    let result = String::from("really long string");
    result  // 소유권 이전
}
```

11.3 구조체에서의 라이프타임

참조를 필드로 갖는 구조체는 라이프타임이 필요합니다:

```
struct ImportantExcerpt<'a> {
    part: &'a str,
}

fn main() {
    let novel = String::from("Call me Ishmael. Some years
ago...");
    let first_sentence = novel.split('.').next().expect("문장을
찾을 수 없음");

    let excerpt = ImportantExcerpt {
        part: first_sentence,
    };

    println!("{}", excerpt.part);
}
```

의미: `ImportantExcerpt` 인스턴스는 `part` 필드가 참조하는 데이터보다 오래 살 수 없습니다.

잘못된 사용

```
fn main() {
    let excerpt;
    {
        let novel = String::from("Call me Ishmael.");
        excerpt = ImportantExcerpt {
            part: &novel.split('.').next().unwrap(),
        };
    }
    // ❌ 컴파일 에러! novel이 소멸됨
    println!("{}", excerpt.part);
}
```

여러 라이프타임 매개변수

```
struct Context<'s, 'c> {
    source: &'s str,
    cache: &'c str,
}
```

11.4 라이프타임 생략 규칙 (Elision Rules)

모든 참조에 라이프타임을 명시하는 것은 번거롭습니다. Rust는 **세 가지 규칙**으로 라이프타임을 추론합니다:

규칙 1: 입력 라이프타임

각 참조 매개변수는 고유한 라이프타임을 받습니다:

```
fn foo(x: &str, y: &str)
// 해석: fn foo<'a, 'b>(x: &'a str, y: &'b str)
```

규칙 2: 단일 입력 라이프타임

입력 라이프타임이 하나뿐이면 출력에 자동 적용:

```
fn first_word(s: &str) -> &str
// 해석: fn first_word<'a>(s: &'a str) -> &'a str
```

규칙 3: 메서드의 self

메서드에서 `&self` 또는 `&mut self`가 있으면 그 라이프타임이 출력에 적용:

```
impl<'a> ImportantExcerpt<'a> {
    fn announce_and_return_part(&self, announcement: &str) ->
&str {
        println!("Attention: {}", announcement);
        self.part
    }
}
// 반환 타입은 self의 라이프타임('a)을 따름
```

생략 가능 vs 명시 필요

```
// ✅ 생략 가능 (규칙 2 적용)
fn first_word(s: &str) -> &str { ... }

// ✅ 생략 가능 (규칙 1만, 출력이 참조 아님)
fn print_both(s1: &str, s2: &str) { ... }

// ❌ 생략 불가 (규칙 적용 불가)
fn longest(x: &str, y: &str) -> &str { ... }
// 어떤 입력의 라이프타임을 따를지 모름!
```

11.5 메서드에서의 라이프타임

```
struct ImportantExcerpt<'a> {
    part: &'a str,
}

impl<'a> ImportantExcerpt<'a> {
    // 라이프타임 없음 (self 참조 안 함)
    fn level(&self) -> i32 {
        3
    }

    // 규칙 3 적용: self.part의 라이프타임 반환
    fn announce_and_return_part(&self, announcement: &str) ->
&str {
        println!("Attention please: {}", announcement);
        self.part
    }

    // 명시적 라이프타임: announcement와 같은 라이프타임 반환
    fn return_announcement<'b>(&self, announcement: &'b str) ->
&'b str {
        announcement
    }
}
```

11.6 정적 라이프타임 (`'static`)

프로그램 전체 기간 동안 유효한 참조입니다:

```rust
let s: &'static str = "I have a static lifetime.";
```

문자열 리터럴

모든 문자열 리터럴은 `'static`입니다:

```rust
// 바이너리에 직접 저장됨
let literal: &'static str = "hello";
```

static 변수

```rust
static HELLO_WORLD: &str = "Hello, world!";

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

⚠️ 'static 남용 주의

```rust
// ❌ 잘못된 'static 사용
fn bad_function(s: &str) -> &'static str {
    // 's'의 라이프타임을 'static으로 확장하려는 시도
    s  // 컴파일 에러!
}

// ✅ 올바른 방법: 소유권 반환
fn good_function(s: &str) -> String {
    s.to_string()
}
```

> 팁: `'static`을 쓰고 싶은 유혹이 들면, 대부분 설계를 재고해야 합니다.

11.7 제네릭, 트레이트, 라이프타임 종합

모든 것을 함께 사용하는 예제:

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

fn longest_with_an_announcement<'a, T>(
    x: &'a str,
    y: &'a str,
    ann: T,
) -> &'a str
where
    T: Display,
{
    println!("Announcement! {}", ann);
    if x.len() > y.len() {
        x
    } else {
        y
    }
}

fn main() {
    let string1 = String::from("abcd");
    let string2 = "xyz";

    let result = longest_with_an_announcement(
        string1.as_str(),
        string2,
        "Today is someone's birthday!",
    );

    println!("가장 긴 문자열: {}", result);
}
```

11.8 고급 라이프타임 패턴

라이프타임 바운드

```
// T는 'a보다 오래 살아야 함
struct Ref<'a, T: 'a> {
    r: &'a T,
}

// T: 'static = T는 참조를 포함하지 않거나, 'static 참조만 포함
fn print_static<T: 'static>(t: T) {
    println!("{:?}", t);
}
```

라이프타임 서브타이핑

```
// 'a: 'b = 'a는 'b보다 오래 산다
fn choose_first<'a: 'b, 'b>(first: &'a str, _: &'b str) -> &'b
str {
    first
}
```

Higher-Ranked Trait Bounds (HRTB)

```rust
// 모든 라이프타임 'a에 대해 작동
fn apply<F>(f: F, s: &str) -> String
where
    F: for<'a> Fn(&'a str) -> &'a str,
{
    f(s).to_string()
}
```

11.9 Non-Lexical Lifetimes (NLL)

Rust 2018부터 빌림 검사기가 더 스마트해졌습니다:

```rust
fn main() {
    let mut data = vec![1, 2, 3];

    let first = &data[0];
    println!("{}", first);
    // 구 버전: first가 스코프 끝까지 유효
    // NLL: first 마지막 사용 후 빌림 종료

    data.push(4);  // ✅ NLL 덕분에 작동!

    println!("{:?}", data);
}
```

NLL 이전 (Rust 2015)

```
fn main() {
    let mut data = vec![1, 2, 3];

    let first = &data[0];
    println!("{}", first);

    // ❌ Rust 2015에서는 에러!
    // data.push(4);  // first가 스코프 끝까지 유효하다고 판단
}
```

🚀 실습 과제

과제 1: 문자열 파서

```
// TODO: 구조체와 라이프타임 사용
// - InputParser<'a>: 입력 문자열 참조 저장
// - parse() -> Vec<&'a str>: 단어 분리
// - first_word() -> Option<&'a str>
```

과제 2: 캐시 시스템

```
// TODO: 라이프타임 제약 설계
// - Cache<'a, T>
// - get(&self, key: &str) -> Option<&'a T>
// - 저장된 값의 라이프타임 보장
```

🚀 실습 가이드

`examples/` 폴더에서 확인:

1. `lifetimes.rs`: 라이프타임 어노테이션 예제
2. `lifetime_struct.rs`: 구조체 라이프타임
3. `elision.rs`: 라이프타임 생략 규칙

📚 핵심 정리

개념	설명
라이프타임	참조가 유효한 범위
`'a`	라이프타임 매개변수
빌림 검사기	참조 유효성 검증 (컴파일 타임)
라이프타임 생략	세 가지 규칙으로 자동 추론
`'static`	프로그램 전체 기간 유효
NLL	더 스마트한 빌림 검사
`T: 'a`	T가 'a보다 오래 살아야 함

Chapter 12: [프로젝트] 파일 정리 CLI 도구 (File Organizer CLI)

> "이제까지 배운 소유권, 에러 처리, 모듈, 컬렉션을 총동원하여 실제 유용한 도구를 만들어 봅시다."

이 프로젝트는 **커맨드 라인 인자**를 받아 특정 디렉토리의 파일들을 정리하는 도구입니다.

12.1 목표

사용자가 `cargo run -- <path>`를 입력하면:

1. 해당 경로의 파일들을 읽습니다.
2. 확장자(extension)를 확인합니다.
3. 확장자별 폴더(예: `png`, `txt`)를 만들고 파일을 이동시킵니다.

12.2 주요 학습 포인트

- **`std::env`**: 커맨드 라인 인자 파싱
- **`std::fs`**, **`std::path`**: 파일 시스템 조작
- **에러 전파**: `Result`를 통한 견고한 실패 처리

🚀 실습 코드

`main.rs`를 확인하세요. 이 코드는 단순히 동작할 뿐만 아니라, `Result` 등을 사용하여 “Rust스러운(Idiomatic)” 방식으로 작성되었습니다.

Chapter 13: 함수형 프로그래밍 기능 (Iterators and Closures)

> "Rust는 함수형 언어의 강력한 특징—클로저와 이터레이터—를 시스템 프로그래밍에 가져왔습니다. 그리고 **런타임 비용 없이** 제공합니다."

왜 함수형 프로그래밍인가?

함수형 스타일은 **선언적**이고 **조합 가능**합니다:

```rust
// 명령형 스타일 (어떻게 할지 기술)
let mut result = Vec::new();
for i in 0..10 {
    if i % 2 == 0 {
        result.push(i * i);
    }
}

// 함수형 스타일 (무엇을 할지 기술)
let result: Vec<i32> = (0..10)
    .filter(|i| i % 2 == 0)
    .map(|i| i * i)
    .collect();
```

장점	설명
가독성	의도가 명확함
조합성	체이닝으로 복잡한 연산 표현
최적화	컴파일러가 효율적으로 최적화
버그 감소	불변성, 부작용 최소화

13.1 클로저 (Closures)

클로저는 **환경을 캡처**할 수 있는 익명 함수입니다.

기본 문법

```rust
fn main() {
    // 타입 추론
    let add_one = |x| x + 1;

    // 명시적 타입
    let add_two = |x: i32| -> i32 { x + 2 };

    // 여러 매개변수
    let add = |a, b| a + b;

    // 매개변수 없음
    let say_hello = || println!("Hello!");

    println!("{}", add_one(5));  // 6
    println!("{}", add_two(5));  // 7
    println!("{}", add(2, 3));   // 5
    say_hello();                 // Hello!
}
```

함수와 클로저 비교

```rust
// 함수
fn add_one_fn(x: i32) -> i32 {
    x + 1
}

// 클로저
let add_one_closure = |x: i32| -> i32 { x + 1 };

// 축약 가능한 형태들
let add_one_closure = |x: i32| { x + 1 };
let add_one_closure = |x| x + 1;
```

환경 캡처

클로저의 강력한 특징: **주변 환경의 변수를 캡처**할 수 있습니다.

```rust
fn main() {
    let x = 4;
    let y = 10;

    // x와 y를 캡처
    let equal_to_x = |z| z == x;
    let sum_with_y = |a| a + y;

    println!("4 == x? {}", equal_to_x(4));  // true
    println!("5 + y = {}", sum_with_y(5));  // 15
}
```

일반 함수는 환경 캡처가 불가능합니다:

```rust
fn main() {
    let x = 4;

    // ❌ 컴파일 에러! 함수는 환경 캡처 불가
    // fn equal_to_x(z: i32) -> bool {
    //     z == x
    // }
}
```

세 가지 Fn 트레이트

클로저가 환경을 캡처하는 방식에 따라 세 가지 트레이트 중 하나를 구현합니다:

트레이트	캡처 방식	설명	호출 횟수
`FnOnce`	값 이동 (Move)	환경 변수 소유권 가져감	한 번만
`FnMut`	가변 빌림 (`&mut`)	환경 수정 가능	여러 번
`Fn`	불변 빌림 (`&`)	읽기만	여러 번

```rust
fn main() {
    // Fn: 불변 빌림
    let x = vec![1, 2, 3];
    let print_x = || println!("{:?}", x);  // &x 캡처
    print_x();
    print_x();  // 여러 번 호출 가능
    println!("{:?}", x);  // x 여전히 사용 가능

    // FnMut: 가변 빌림
    let mut count = 0;
    let mut increment = || count += 1;  // &mut count 캡처
    increment();
    increment();
    println!("count: {}", count);  // 2

    // FnOnce: 소유권 이동
    let name = String::from("Alice");
    let consume = || {
        let s = name;  // name 소유권 이동
        println!("Consumed: {}", s);
    };
    consume();
    // consume();  // ❌ 두 번째 호출 불가
}
```

---

move 키워드

`move` 키워드로 소유권 이동을 강제합니다:

```rust
fn main() {
    let x = vec![1, 2, 3];

    // move: 소유권 강제 이동
    let equal_to_x = move |z: Vec<i32>| z == x;

    // println!("{:?}", x);  // ❌ x는 이미 이동됨

    println!("{}", equal_to_x(vec![1, 2, 3]));  // true
}
```

스레드에서 필수:

```rust
use std::thread;

fn main() {
    let v = vec![1, 2, 3];

    // move 없으면 컴파일 에러!
    // 스레드가 v보다 오래 살 수 있음
    let handle = thread::spawn(move || {
        println!("Here's a vector: {:?}", v);
    });

    handle.join().unwrap();
}
```

클로저를 매개변수로 받기

```rust
// 제네릭 + 트레이트 바운드
fn apply<F>(f: F, x: i32) -> i32
where
    F: Fn(i32) -> i32,
{
    f(x)
}

// impl Trait 문법
fn apply_impl(f: impl Fn(i32) -> i32, x: i32) -> i32 {
    f(x)
}

fn main() {
    let double = |x| x * 2;
    let triple = |x| x * 3;

    println!("{}", apply(double, 5));  // 10
    println!("{}", apply_impl(triple, 5));  // 15
}
```

클로저를 반환하기

```
fn make_adder(n: i32) -> impl Fn(i32) -> i32 {
    move |x| x + n
}

fn main() {
    let add_5 = make_adder(5);
    let add_10 = make_adder(10);

    println!("{}", add_5(3));   // 8
    println!("{}", add_10(3));  // 13
}
```

13.2 이터레이터 (Iterators)

이터레이터는 순차적 요소 접근을 추상화합니다.

기본 사용

```rust
fn main() {
    let v = vec![1, 2, 3];

    // for 루프는 내부적으로 이터레이터 사용
    for val in &v {
        println!("{}", val);
    }

    // 명시적 이터레이터
    let mut iter = v.iter();
    assert_eq!(iter.next(), Some(&1));
    assert_eq!(iter.next(), Some(&2));
    assert_eq!(iter.next(), Some(&3));
    assert_eq!(iter.next(), None);
}
```

세 가지 이터레이터

메서드	반환 타입	소유권
`iter()`	`Iterator<Item = &T>`	불변 참조
`iter_mut()`	`Iterator<Item = &mut T>`	가변 참조
`into_iter()`	`Iterator<Item = T>`	소유권 이동

```rust
fn main() {
    let v = vec![1, 2, 3];

    // iter(): 불변 참조
    for val in v.iter() {
        println!("{}", val);  // val: &i32
    }

    // iter_mut(): 가변 참조
    let mut v2 = vec![1, 2, 3];
    for val in v2.iter_mut() {
        *val *= 2;  // val: &mut i32
    }
    println!("{:?}", v2);  // [2, 4, 6]

    // into_iter(): 소유권 이동
    let v3 = vec![1, 2, 3];
    for val in v3.into_iter() {
        println!("{}", val);  // val: i32
    }
    // v3는 더 이상 사용 불가
}
```

Iterator 트레이트

```
pub trait Iterator {
    type Item;  // 연관 타입

    fn next(&mut self) -> Option<Self::Item>;

    // 많은 기본 메서드들 (next로부터 파생)
    // fn count(self) -> usize
    // fn last(self) -> Option<Self::Item>
    // fn nth(&mut self, n: usize) -> Option<Self::Item>
    // ... 등등
}
```

커스텀 이터레이터

```
struct Counter {
    count: u32,
    max: u32,
}

impl Counter {
    fn new(max: u32) -> Counter {
        Counter { count: 0, max }
    }
}

impl Iterator for Counter {
    type Item = u32;

    fn next(&mut self) -> Option<Self::Item> {
        if self.count < self.max {
            self.count += 1;
            Some(self.count)
        } else {
            None
        }
    }
}

fn main() {
    let counter = Counter::new(5);

    for n in counter {
        println!("{}", n);  // 1, 2, 3, 4, 5
    }
}
```

소비 어댑터 (Consuming Adaptors)

이터레이터를 소비하여 최종 값을 생성합니다:

```
fn main() {
    let v = vec![1, 2, 3, 4, 5];

    // sum: 합계
    let total: i32 = v.iter().sum();
    println!("합계: {}", total);  // 15

    // count: 개수
    let count = v.iter().count();
    println!("개수: {}", count);  // 5

    // max, min
    let max = v.iter().max();
    let min = v.iter().min();
    println!("최대: {:?}, 최소: {:?}", max, min);

    // collect: 컬렉션으로 변환
    let doubled: Vec<i32> = v.iter().map(|x| x * 2).collect();
    println!("{:?}", doubled);

    // find: 조건에 맞는 첫 요소
    let first_even = v.iter().find(|&&x| x % 2 == 0);
    println!("첫 짝수: {:?}", first_even);  // Some(2)

    // any, all
    let has_even = v.iter().any(|&x| x % 2 == 0);
    let all_positive = v.iter().all(|&x| x > 0);
    println!("짝수 있음: {}, 모두 양수: {}", has_even,
all_positive);
}
```

이터레이터 어댑터 (Iterator Adaptors)

이터레이터를 **다른 이터레이터로 변환**합니다. **지연 평가(Lazy)**!

```rust
fn main() {
    let v = vec![1, 2, 3, 4, 5];

    // map: 각 요소 변환
    let doubled: Vec<i32> = v.iter().map(|x| x * 2).collect();
    println!("map: {:?}", doubled);  // [2, 4, 6, 8, 10]

    // filter: 조건에 맞는 요소만
    let evens: Vec<&i32> = v.iter().filter(|x| *x % 2 ==
0).collect();
    println!("filter: {:?}", evens);  // [2, 4]

    // take: 처음 n개
    let first_three: Vec<&i32> = v.iter().take(3).collect();
    println!("take: {:?}", first_three);  // [1, 2, 3]

    // skip: 처음 n개 건너뛰기
    let after_two: Vec<&i32> = v.iter().skip(2).collect();
    println!("skip: {:?}", after_two);  // [3, 4, 5]

    // enumerate: 인덱스와 함께
    for (i, val) in v.iter().enumerate() {
        println!("[{}] = {}", i, val);
    }

    // zip: 두 이터레이터 병합
    let names = vec!["Alice", "Bob"];
    let ages = vec![25, 30];
    let combined: Vec<_> =
names.iter().zip(ages.iter()).collect();
    println!("zip: {:?}", combined);  // [("Alice", 25),
("Bob", 30)]
}
```

체이닝 (Chaining)

여러 어댑터를 연결하여 복잡한 연산을 표현합니다:

```rust
fn main() {
    let numbers = vec![1, 2, 3, 4, 5, 6, 7, 8, 9, 10];

    // 짝수만 필터 → 제곱 → 처음 3개 → 합계
    let result: i32 = numbers
        .iter()
        .filter(|n| *n % 2 == 0)  // 2, 4, 6, 8, 10
        .map(|n| n * n)           // 4, 16, 36, 64, 100
        .take(3)                   // 4, 16, 36
        .sum();                    // 56

    println!("결과: {}", result);
}
```

flat_map

중첩 컬렉션 평탄화:

```rust
fn main() {
    let nested = vec![vec![1, 2], vec![3, 4], vec![5]];

    let flat: Vec<&i32> = nested.iter()
        .flat_map(|inner| inner.iter())
        .collect();

    println!("{:?}", flat);  // [1, 2, 3, 4, 5]
}
```

---

fold: 가장 강력한 어댑터

모든 소비 어댑터는 `fold`로 구현할 수 있습니다:

```
fn main() {
    let v = vec![1, 2, 3, 4, 5];

    // sum을 fold로 구현
    let sum = v.iter().fold(0, |acc, x| acc + x);
    println!("합계: {}", sum);  // 15

    // max를 fold로 구현
    let max = v.iter().fold(i32::MIN, |acc, &x| {
        if x > acc { x } else { acc }
    });
    println!("최대: {}", max);  // 5

    // 문자열 연결
    let words = vec!["Hello", "World", "!"];
    let sentence = words.iter().fold(String::new(), |mut acc,
&word| {
        if !acc.is_empty() {
            acc.push(' ');
        }
        acc.push_str(word);
        acc
    });
    println!("{}", sentence);  // "Hello World !"
}
```

13.3 성능: Zero-cost Abstraction

> "이터레이터는 for 루프만큼 빠릅니다."

벤치마크 비교

```
// 이터레이터 버전
let sum: i32 = values.iter().sum();

// for 루프 버전
let mut sum = 0;
for val in &values {
    sum += val;
}

// 결과: 성능 동일! (또는 이터레이터가 약간 더 빠를 수도)
```

컴파일러 최적화

Rust 컴파일러는 이터레이터 체인을 **루프 언롤링**하고 인라인화합니다:

```rust
// 이 코드는...
numbers.iter()
    .filter(|n| *n % 2 == 0)
    .map(|n| n * n)
    .sum()

// 컴파일러에 의해 최적의 루프로 변환됨
// (대략적으로)
let mut sum = 0;
for n in &numbers {
    if n % 2 == 0 {
        sum += n * n;
    }
}
```

13.4 자주 쓰는 이터레이터 메서드 정리

어댑터 (지연 평가)

메서드	설명	예시
`map(f)`	각 요소에 함수 적용	`.map(
`filter(p)`	조건 만족 요소만	`.filter(
`take(n)`	처음 n개	`.take(5)`
`skip(n)`	처음 n개 건너뛰기	`.skip(2)`
`enumerate()`	인덱스 추가	`.enumerate()`

메서드	설명	예시
`zip(iter)`	두 이터레이터 병합	`.zip(other)`
`chain(iter)`	이터레이터 연결	`.chain(other)`
`flat_map(f)`	중첩 평탄화	`.flat_map(
`take_while(p)`	조건 만족하는 동안	`.take_while(
`skip_while(p)`	조건 만족하는 동안 건너뛰기	`.skip_while(
`rev()`	역순	`.rev()`
`cloned()`	참조를 복제	`.cloned()`

소비자

메서드	설명	반환 타입
`collect()`	컬렉션으로 변환	`Collection`
`sum()`	합계	숫자 타입
`product()`	곱	숫자 타입
`count()`	개수	`usize`
`max()` / `min()`	최대/최소	`Option<T>`
`find(p)`	조건 만족 첫 요소	`Option<T>`

메서드	설명	반환 타입
`position(p)`	조건 만족 첫 인덱스	`Option<usize>`
`any(p)` / `all(p)`	논리 검사	`bool`
`fold(init, f)`	누적 연산	임의
`for_each(f)`	각 요소에 실행	`()`

🚀 실습 과제

과제 1: 단어 빈도수 계산

```
// TODO: 주어진 텍스트에서 각 단어의 빈도수를 HashMap으로 반환
// fn word_frequencies(text: &str) -> HashMap<&str, usize>
```

과제 2: 피보나치 이터레이터

```
// TODO: 피보나치 수열 이터레이터 구현
// struct Fibonacci { ... }
// impl Iterator for Fibonacci { ... }
```

🚀 실습 가이드

`examples/` 폴더에서 확인:

1. `closures.rs`: 클로저 문법과 캡처
2. `iterators.rs`: 이터레이터 어댑터 활용

📚 핵심 정리

개념	설명
클로저	환경 캡처하는 익명 함수
`Fn` 트레이트	불변 캡처, 여러 번 호출
`FnMut` 트레이트	가변 캡처, 여러 번 호출
`FnOnce` 트레이트	소유권 이동, 한 번 호출
`move`	소유권 강제 이동
이터레이터	순차 요소 접근 추상화
어댑터	이터레이터 → 이터레이터 (지연)
소비자	이터레이터 → 값 (즉시)
Zero-cost	추상화 비용 없음

Chapter 14: 스마트 포인터 (Smart Pointers)

> "스마트 포인터는 일반 참조에 **추가 능력과 메타데이터**를 부여한 데이터 구조입니다. 일반 참조와 달리 데이터의 **소유권**을 가집니다."

포인터 vs 스마트 포인터

종류	예시	소유권	메타데이터
일반 참조	`&T`, `&mut T`	❌	❌
스마트 포인터	`Box<T>`, `Rc<T>`	✅	✅

스마트 포인터는 보통 구조체로 구현되며, `Deref`와 `Drop` 트레이트를 구현합니다:

- **Deref**: 참조처럼 동작 (`*` 연산자)
- **Drop**: 스코프 종료 시 정리 코드 실행

14.1 `Box<T>`: 힙 할당

가장 간단한 스마트 포인터입니다. 데이터를 **힙**에 저장합니다.

기본 사용법

```rust
fn main() {
    let b = Box::new(5);
    println!("b = {}", b);  // 자동 역참조
    println!("b = {}", *b);  // 명시적 역참조
}
```

사용 시점

1. **컴파일 타임에 크기를 알 수 없는 타입**
2. **큰 데이터의 소유권 이동 시 복사 방지**
3. **트레이트 객체 저장**

재귀 타입 정의

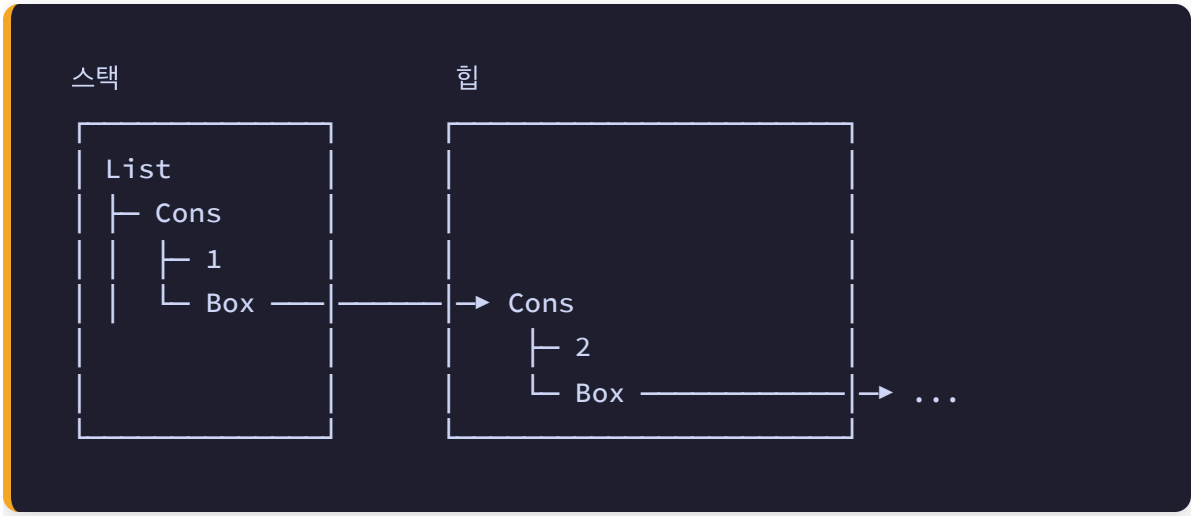
재귀 타입은 자신을 포함하므로 크기가 무한대가 됩니다:

```rust
// ❌ 컴파일 에러! 무한 크기
// enum List {
//     Cons(i32, List),  // List 안에 List...
//     Nil,
// }
```

`Box`로 해결:

```rust
enum List {
    Cons(i32, Box<List>),  // 포인터 크기는 고정
    Nil,
}

use List::{Cons, Nil};

fn main() {
    let list = Cons(1,
        Box::new(Cons(2,
            Box::new(Cons(3,
                Box::new(Nil))))));

    print_list(&list);
}

fn print_list(list: &List) {
    match list {
        Cons(val, next) => {
            println!("{}", val);
            print_list(next);
        }
        Nil => println!("끝"),
    }
}
```

메모리 레이아웃



14.2 `Deref` 트레이트

스마트 포인터를 **일반 참조처럼** 사용할 수 있게 합니다.

커스텀 스마트 포인터

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

struct MyBox<T>(T);

impl<T> MyBox<T> {
    fn new(x: T) -> MyBox<T> {
        MyBox(x)
    }
}

impl<T> Deref for MyBox<T> {
    type Target = T;  // 역참조 결과 타입

    fn deref(&self) -> &Self::Target {
        &self.0
    }
}

fn main() {
    let x = 5;
    let y = MyBox::new(x);

    assert_eq!(5, x);
    assert_eq!(5, *y);  // *y는 *(y.deref())로 변환됨
}
```

역참조 강제 (Deref Coercion)

`Deref`를 구현한 타입은 자동으로 변환됩니다:

```rust
fn hello(name: &str) {
    println!("Hello, {}!", name);
}

fn main() {
    let m = MyBox::new(String::from("Rust"));

    // 역참조 강제 체인:
    // &MyBox<String> → &String → &str
    hello(&m);

    // 역참조 강제 없이:
    // hello(&(*m)[..]);  // 훨씬 복잡!
}
```

Deref 강제 규칙

변환	조건
`&T` → `&U`	`T: Deref<Target=U>`
`&mut T` → `&mut U`	`T: DerefMut<Target=U>`
`&mut T` → `&U`	`T: Deref<Target=U>`

14.3 `Drop` 트레이트

값이 스코프를 벗어날 때 실행되는 정리 코드를 정의합니다.

기본 사용법

```
struct CustomSmartPointer {
    data: String,
}

impl Drop for CustomSmartPointer {
    fn drop(&mut self) {
        println!("Dropping CustomSmartPointer with data `{}`!",
self.data);
    }
}

fn main() {
    let c = CustomSmartPointer { data: String::from("my stuff")
};
    let d = CustomSmartPointer { data: String::from("other
stuff") };
    println!("CustomSmartPointers created.");
}
// 출력:
// CustomSmartPointers created.
// Dropping CustomSmartPointer with data `other stuff`!
// Dropping CustomSmartPointer with data `my stuff`!
```

순서: 생성 역순으로 drop됩니다 (스택처럼).

조기 해제: `std::mem::drop`

```
fn main() {
    let c = CustomSmartPointer { data: String::from("some
data") };
    println!("CustomSmartPointer created.");

    // c.drop();  // ❌ 직접 호출 불가!
    drop(c);      // ✅ std::mem::drop 사용

    println!("CustomSmartPointer dropped before end of main.");
}
```

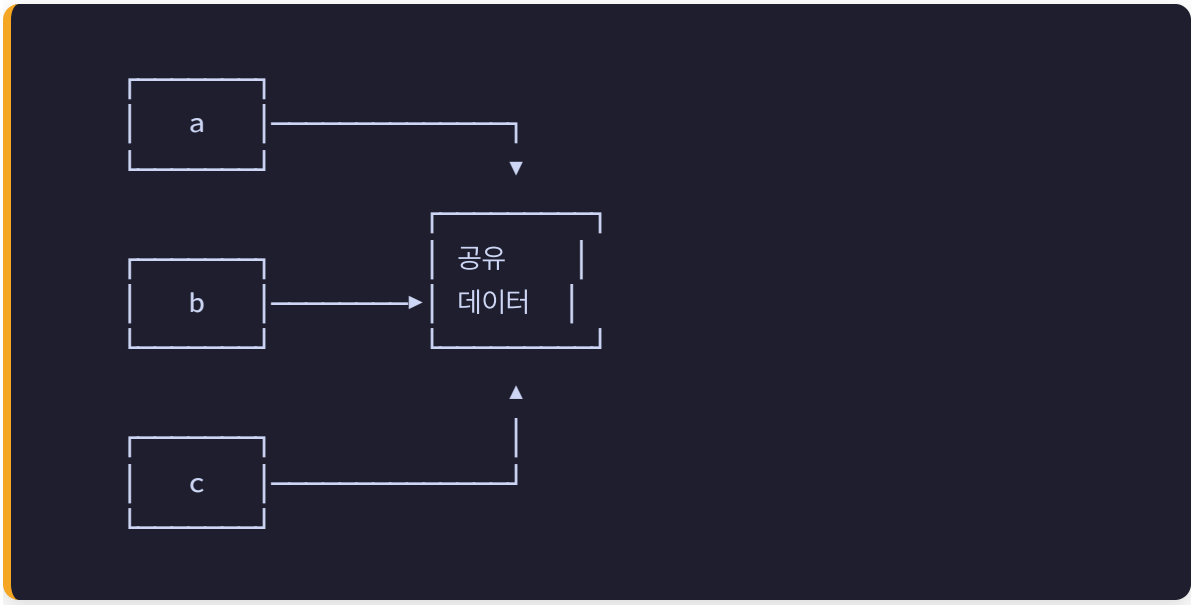
Drop 활용 예시

```
struct FileHandle {
    name: String,
}

impl FileHandle {
    fn new(name: &str) -> Self {
        println!("Opening file: {}", name);
        FileHandle { name: name.to_string() }
    }
}

impl Drop for FileHandle {
    fn drop(&mut self) {
        println!("Closing file: {}", self.name);
    }
}

fn main() {
    let _file = FileHandle::new("data.txt");
    println!("Working with file...");
}  // 자동으로 close됨
```

14.4 Rc<T>: 참조 카운팅

단일 값에 여러 소유자가 필요할 때 사용합니다.

사용 시나리오



기본 사용법

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

enum List {
    Cons(i32, Rc<List>),
    Nil,
}

use List::{Cons, Nil};

fn main() {
    let a = Rc::new(Cons(5, Rc::new(Cons(10, Rc::new(Nil)))));
    println!("a 생성 후 카운트: {}", Rc::strong_count(&a));  // 1

    let b = Cons(3, Rc::clone(&a));  // 카운트 증가 (깊은 복사 아님!)
    println!("b 생성 후 카운트: {}", Rc::strong_count(&a));  // 2

    {
        let c = Cons(4, Rc::clone(&a));
        println!("c 생성 후 카운트: {}", Rc::strong_count(&a));
// 3
    }

    println!("c 스코프 종료 후 카운트: {}", Rc::strong_count(&a));
// 2
}
```

Rc::clone vs .clone()

```
let a = Rc::new(5);

// Rc::clone: 참조 카운트만 증가 (O(1))
let b = Rc::clone(&a);

// .clone(): 동일 (관례적으로 Rc::clone 사용)
let c = a.clone();
```

⚠️ 주의: `Rc<T>`는 불변입니다. 내부 값을 수정할 수 없습니다!

14.5 `RefCell<T>`: 내부 가변성

런타임에 빌림 규칙을 검사합니다.

컴파일 타임 vs 런타임 빌림 검사

타입	검사 시점	규칙 위반 시
`&T`, `&mut T`	컴파일 타임	컴파일 에러
`RefCell<T>`	런타임	패닉!

기본 사용법

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

fn main() {
    let value = RefCell::new(5);

    // borrow_mut(): 가변 참조 (런타임 검사)
    *value.borrow_mut() += 1;

    // borrow(): 불변 참조 (런타임 검사)
    println!("value: {}", value.borrow());  // 6
}
```

런타임 패닉

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

fn main() {
    let value = RefCell::new(5);

    let a = value.borrow();       // 불변 빌림
    let b = value.borrow();       // ✅ 여러 불변 빌림 OK

    // ❌ 런타임 패닉! 불변 빌림 중 가변 빌림
    // let c = value.borrow_mut();

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

내부 가변성이 필요한 경우

```
trait Messenger {
    fn send(&self, msg: &str);  // &self: 불변 참조
}

struct MockMessenger {
    sent_messages: RefCell<Vec<String>>,  // 내부 가변성!
}

impl MockMessenger {
    fn new() -> MockMessenger {
        MockMessenger {
            sent_messages: RefCell::new(vec![]),
        }
    }
}

impl Messenger for MockMessenger {
    fn send(&self, message: &str) {
        // &self인데도 수정 가능!

self.sent_messages.borrow_mut().push(String::from(message));
    }
}
```

14.6 `Rc<RefCell<T>>` 패턴

여러 소유자 + 가변 접근이 필요할 때 조합합니다.

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

#[derive(Debug)]
enum List {
    Cons(Rc<RefCell<i32>>, Rc<List>),
    Nil,
}

use List::{Cons, Nil};

fn main() {
    let value = Rc::new(RefCell::new(5));

    let a = Rc::new(Cons(Rc::clone(&value), Rc::new(Nil)));
    let b = Cons(Rc::new(RefCell::new(3)), Rc::clone(&a));
    let c = Cons(Rc::new(RefCell::new(4)), Rc::clone(&a));

    // 모든 경로에서 공유하는 값 수정
    *value.borrow_mut() += 10;

    println!("a = {:?}", a);  // 값이 15로 변경됨
    println!("b = {:?}", b);
    println!("c = {:?}", c);
}
```

14.7 순환 참조와 `Weak<T>`

`Rc<T>`만 사용하면 순환 참조로 메모리 누수가 발생할 수 있습니다.

순환 참조 문제

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

#[derive(Debug)]
enum List {
    Cons(i32, RefCell<Rc<List>>),
    Nil,
}

fn main() {
    let a = Rc::new(Cons(5, RefCell::new(Rc::new(Nil))));
    let b = Rc::new(Cons(10, RefCell::new(Rc::clone(&a))));

    // a가 b를 가리키게 함 → 순환!
    if let Cons(_, ref link) = *a {
        *link.borrow_mut() = Rc::clone(&b);
    }

    // 스택 오버플로우 (순환 참조)
    // println!("{:?}", a);
}
```

Weak로 해결

`Weak<T>`는 소유권 없이 참조만 합니다:

```rust
use std::cell::RefCell;
use std::rc::{Rc, Weak};

#[derive(Debug)]
struct Node {
    value: i32,
    parent: RefCell<Weak<Node>>,      // 약한 참조
    children: RefCell<Vec<Rc<Node>>>, // 강한 참조
}

fn main() {
    let leaf = Rc::new(Node {
        value: 3,
        parent: RefCell::new(Weak::new()),
        children: RefCell::new(vec![]),
    });

    let branch = Rc::new(Node {
        value: 5,
        parent: RefCell::new(Weak::new()),
        children: RefCell::new(vec![Rc::clone(&leaf)]),
    });

    // leaf의 부모를 branch로 설정
    *leaf.parent.borrow_mut() = Rc::downgrade(&branch);

    // upgrade()로 접근 (Option<Rc<T>> 반환)
    println!("leaf parent = {:?}",
leaf.parent.borrow().upgrade());
}
```

Strong vs Weak 참조

타입	카운트	소유권	메모리 해제 영향
`Rc<T>`	strong_count	✅	strong_count=0이면 해제
`Weak<T>`	weak_count	❌	영향 없음

14.8 스마트 포인터 정리

타입	용도	스레드 안전
`Box<T>`	힙 할당, 재귀 타입	✅ (T가 Send면)
`Rc<T>`	단일 스레드 공유 소유권	❌
`Arc<T>`	멀티 스레드 공유 소유권	✅
`RefCell<T>`	단일 스레드 내부 가변성	❌
`Mutex<T>`	멀티 스레드 내부 가변성	✅
`Weak<T>`	순환 참조 방지	Rc용: ❌, Arc용: ✅

🚀 실습 과제

과제 1: 이진 트리

```
// TODO: Box<T>를 사용한 이진 트리 구현
// - 삽입, 검색, 순회 메서드
```

과제 2: 옵저버 패턴

```
// TODO: Rc<RefCell<T>>를 사용한 옵저버 패턴
// - 여러 옵저버가 한 subject 구독
```

🚀 실습 가이드

`examples/` 폴더에서 확인:

1. `box_example.rs`: Box와 재귀 타입
2. `rc_refcell.rs`: Rc와 RefCell 조합

📚 핵심 정리

개념	설명
`Box<T>`	힙 할당, 단일 소유권

개념	설명
`Deref`	역참조 연산자 오버로딩
`Drop`	소멸자 (스코프 종료 시 실행)
`Rc<T>`	참조 카운팅, 다중 소유권
`RefCell<T>`	런타임 빌림 검사, 내부 가변성
`Weak<T>`	순환 참조 방지
역참조 강제	`Deref` 체인 자동 적용

Chapter 15: 두려움 없는 동시성 (Fearless Concurrency)

> "Rust의 소유권 시스템은 **컴파일 타임에 데이터 레이스를 방지**합니다. 다른 언어에서는 런타임에야 발견되는 버그를 Rust에서는 컴파일 시점에 잡아냅니다."

왜 "두려움 없는" 동시성인가?

다른 언어에서 동시성 버그는 매우 찾기 어렵습니다:

- **데이터 레이스**: 두 스레드가 동시에 같은 데이터를 수정
- **데드락**: 두 스레드가 서로를 기다리며 영원히 멈춤
- **주문 의존성**: 실행 순서에 따라 결과가 달라짐

Rust는 **타입 시스템**으로 많은 동시성 버그를 방지합니다:

```rust
use std::thread;

fn main() {
    let v = vec![1, 2, 3];

    // ❌ 컴파일 에러! v의 소유권이 불분명
    // let handle = thread::spawn(|| {
    //     println!("{:?}", v);
    // });

    // ✅ move로 소유권 명확히
    let handle = thread::spawn(move || {
        println!("{:?}", v);
    });

    handle.join().unwrap();
}
```

15.1 스레드 (Threads)

새 스레드 생성

```rust
use std::thread;
use std::time::Duration;

fn main() {
    // 새 스레드 생성
    let handle = thread::spawn(|| {
        for i in 1..10 {
            println!("spawned thread: {}", i);
            thread::sleep(Duration::from_millis(1));
        }
    });

    // 메인 스레드
    for i in 1..5 {
        println!("main thread: {}", i);
        thread::sleep(Duration::from_millis(1));
    }

    // 스레드 완료 대기
    handle.join().unwrap();
}
```

JoinHandle

`thread::spawn`은 `JoinHandle`을 반환합니다:

```rust
use std::thread;

fn main() {
    let handle = thread::spawn(|| {
        // 스레드에서 값 반환
        let sum: i32 = (1..=100).sum();
        sum
    });

    // join()으로 결과 받기
    let result = handle.join().unwrap();
    println!("스레드 결과: {}", result);  // 5050
}
```

---

move 클로저

스레드에서 외부 데이터를 사용하려면 `move`가 필요합니다:

```rust
use std::thread;

fn main() {
    let v = vec![1, 2, 3];

    // move: 소유권을 스레드로 이동
    let handle = thread::spawn(move || {
        println!("Here's a vector: {:?}", v);
    });

    // ❌ v는 이미 이동됨
    // println!("{:?}", v);

    handle.join().unwrap();
}
```

왜 move가 필요한가?

```rust
use std::thread;

fn main() {
    let v = vec![1, 2, 3];

    // 만약 move 없이 &v를 캡처한다면...
    // let handle = thread::spawn(|| {
    //     println!("{:?}", v);  // &v 캡처
    // });

    drop(v);  // v가 여기서 삭제될 수 있음!

    // handle.join();  // 스레드가 삭제된 v에 접근 → 💥
}
```

스레드 빌더

스레드에 이름과 스택 크기를 지정할 수 있습니다:

```
use std::thread;

fn main() {
    let builder = thread::Builder::new()
        .name("worker-1".into())
        .stack_size(32 * 1024);  // 32KB

    let handle = builder.spawn(|| {
        let name =
thread::current().name().unwrap().to_string();
        println!("Running in thread: {}", name);
    }).unwrap();

    handle.join().unwrap();
}
```

15.2 메시지 패싱: 채널 (Channels)

> “메모리를 공유하여 통신하지 말고, 통신하여 메모리를 공유하라.” — Go 언어 철학

기본 사용

```rust
use std::sync::mpsc;  // Multiple Producer, Single Consumer
use std::thread;

fn main() {
    // 채널 생성: (송신자, 수신자)
    let (tx, rx) = mpsc::channel();

    thread::spawn(move || {
        let val = String::from("hi");
        tx.send(val).unwrap();
        // ❌ val은 send로 이동됨!
        // println!("{}", val);
    });

    // recv(): 블로킹 (메시지 올 때까지 대기)
    let received = rx.recv().unwrap();
    println!("Got: {}", received);
}
```

recv vs try_recv

메서드	동작	반환 타입
`recv()`	블로킹 (대기)	`Result<T, RecvError>`
`try_recv()`	논블로킹 (즉시 반환)	`Result<T, TryRecvError>`
`recv_timeout(d)`	타임아웃	`Result<T, RecvTimeoutError>`

```rust
use std::sync::mpsc;
use std::time::Duration;

fn main() {
    let (tx, rx) = mpsc::channel();

    // try_recv: 논블로킹
    match rx.try_recv() {
        Ok(msg) => println!("Got: {}", msg),
        Err(mpsc::TryRecvError::Empty) => println!("No message
yet"),
        Err(mpsc::TryRecvError::Disconnected) => println!
("Channel closed"),
    }

    // recv_timeout: 최대 1초 대기
    match rx.recv_timeout(Duration::from_secs(1)) {
        Ok(msg) => println!("Got: {}", msg),
        Err(_) => println!("Timeout or disconnected"),
    }
}
```

여러 값 전송

```rust
use std::sync::mpsc;
use std::thread;
use std::time::Duration;

fn main() {
    let (tx, rx) = mpsc::channel();

    thread::spawn(move || {
        let vals = vec![
            String::from("hi"),
            String::from("from"),
            String::from("the"),
            String::from("thread"),
        ];

        for val in vals {
            tx.send(val).unwrap();
            thread::sleep(Duration::from_millis(200));
        }
    });

    // 이터레이터로 수신 (채널 닫힐 때까지)
    for received in rx {
        println!("Got: {}", received);
    }
}
```

여러 송신자 (clone)

```rust
use std::sync::mpsc;
use std::thread;
use std::time::Duration;

fn main() {
    let (tx, rx) = mpsc::channel();

    // 첫 번째 송신자 복제
    let tx1 = tx.clone();

    // 스레드 1
    thread::spawn(move || {
        let vals = vec!["hi", "from", "thread 1"];
        for val in vals {
            tx1.send(val).unwrap();
            thread::sleep(Duration::from_millis(200));
        }
    });

    // 스레드 2
    thread::spawn(move || {
        let vals = vec!["more", "messages", "from thread 2"];
        for val in vals {
            tx.send(val).unwrap();
            thread::sleep(Duration::from_millis(200));
        }
    });

    for received in rx {
        println!("Got: {}", received);
    }
}
```

Sync 채널

버퍼 크기를 제한하여 **백프레셔**를 적용합니다:

```rust
use std::sync::mpsc;
use std::thread;

fn main() {
    // 버퍼 크기 2 (3번째 send부터 블로킹)
    let (tx, rx) = mpsc::sync_channel(2);

    thread::spawn(move || {
        for i in 0..5 {
            println!("Sending {}", i);
            tx.send(i).unwrap();  // 버퍼 가득 차면 블로킹
            println!("Sent {}", i);
        }
    });

    for received in rx {
        println!("Received: {}", received);
        thread::sleep(std::time::Duration::from_millis(500));
    }
}
```

15.3 공유 상태: `Mutex<T>`

상호 배제(Mutual Exclusion): 한 번에 하나의 스레드만 데이터에 접근.

단일 스레드 예제

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

fn main() {
    let m = Mutex::new(5);

    {
        // lock(): MutexGuard 반환
        let mut num = m.lock().unwrap();
        *num = 6;
    }  // MutexGuard drop → 락 자동 해제

    println!("m = {:?}", m);  // 6
}
```

Mutex API

메서드	동작	반환
`lock()`	락 획득 (블로킹)	`LockResult<MutexGuard>`
`try_lock()`	락 시도 (논블로킹)	`TryLockResult<MutexGuard>`

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

fn main() {
    let m = Mutex::new(5);

    // try_lock: 이미 잠겨있으면 즉시 실패
    if let Ok(mut guard) = m.try_lock() {
        *guard = 10;
    } else {
        println!("락 획득 실패");
    }
}
```

데드락 주의

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

fn main() {
    let m1 = Mutex::new(1);
    let m2 = Mutex::new(2);

    // ⚠️ 잠재적 데드락 위험!
    // 스레드 A: m1 락 → m2 락 시도
    // 스레드 B: m2 락 → m1 락 시도
    // → 서로 대기하며 영원히 멈춤
}
```

해결 방법:

1. 항상 같은 순서로 락 획득
2. try_lock과 재시도 로직 사용

3. 락 범위 최소화

15.4 여러 스레드에서 Mutex: `Arc<T>`

`Rc<T>`는 스레드 안전하지 않습니다. **Atomic Reference Counting** `Arc<T>`를 사용해야 합니다.

Rc vs Arc

타입	스레드 안전	성능
`Rc<T>`	❌	더 빠름 (원자적 연산 없음)
`Arc<T>`	✅	약간 느림 (원자적 연산)

멀티스레드 카운터

```rust
use std::sync::{Arc, Mutex};
use std::thread;

fn main() {
    // Arc<Mutex<T>>: 여러 스레드에서 공유 + 변경 가능
    let counter = Arc::new(Mutex::new(0));
    let mut handles = vec![];

    for _ in 0..10 {
        let counter = Arc::clone(&counter);
        let handle = thread::spawn(move || {
            let mut num = counter.lock().unwrap();
            *num += 1;
        });
        handles.push(handle);
    }

    for handle in handles {
        handle.join().unwrap();
    }

    println!("Result: {}", *counter.lock().unwrap());  // 10
}
```

패턴 비교

패턴	용도
`Rc<RefCell<T>>`	단일 스레드, 런타임 빌림 검사
`Arc<Mutex<T>>`	멀티 스레드, 상호 배제

패턴	용도
`Arc<RwLock<T>>`	멀티 스레드, 다중 읽기 / 단일 쓰기

RwLock: 다중 읽기 / 단일 쓰기

```rust
use std::sync::{Arc, RwLock};
use std::thread;

fn main() {
    let lock = Arc::new(RwLock::new(5));
    let mut handles = vec![];

    // 여러 읽기 스레드 (동시 실행 가능!)
    for _ in 0..3 {
        let lock = Arc::clone(&lock);
        handles.push(thread::spawn(move || {
            let r = lock.read().unwrap();
            println!("Read: {}", *r);
        }));
    }

    // 쓰기 스레드 (독점)
    {
        let lock = Arc::clone(&lock);
        handles.push(thread::spawn(move || {
            let mut w = lock.write().unwrap();
            *w += 1;
            println!("Written: {}", *w);
        }));
    }

    for handle in handles {
        handle.join().unwrap();
    }
}
```

15.5 `Send`와 `Sync` 트레이트

이 두 마커 트레이트가 Rust 동시성 안전의 핵심입니다.

Send: 스레드 간 소유권 이동

```
// 대부분의 타입은 Send
// Arc<T>: Send (T가 Send면)
// Rc<T>: Send 아님! (참조 카운트가 원자적이지 않음)

use std::thread;
use std::rc::Rc;

fn main() {
    let r = Rc::new(5);

    // ❌ 컴파일 에러! Rc<i32>는 Send가 아님
    // thread::spawn(move || {
    //     println!("{}", r);
    // });
}
```

Sync: 여러 스레드에서 참조 공유

```
// T: Sync ⟺ &T: Send
// 대부분의 기본 타입: Sync
// RefCell<T>: Sync 아님! (빌림 검사가 스레드 안전하지 않음)
```

타입별 Send/Sync

타입	Send	Sync
대부분의 기본 타입	✅	✅
`Rc<T>`	❌	❌
`Arc<T>`	✅ (T: Send)	✅ (T: Sync)
`RefCell<T>`	✅	❌
`Mutex<T>`	✅ (T: Send)	✅
`*const T`, `*mut T`	❌	❌

> **핵심**: 컴파일러가 Send/Sync를 체크하므로 **스레드 안전하지 않은 코드는 컴파일되지 않습니다**!

15.6 원자적 타입 (Atomics)

락 없이 원자적 연산을 수행합니다:

```rust
use std::sync::atomic::{AtomicUsize, Ordering};
use std::sync::Arc;
use std::thread;

fn main() {
    let counter = Arc::new(AtomicUsize::new(0));
    let mut handles = vec![];

    for _ in 0..10 {
        let counter = Arc::clone(&counter);
        handles.push(thread::spawn(move || {
            // 원자적 증가 (락 없음!)
            counter.fetch_add(1, Ordering::SeqCst);
        }));
    }

    for handle in handles {
        handle.join().unwrap();
    }

    println!("Counter: {}", counter.load(Ordering::SeqCst));
// 10
}
```

Ordering

Ordering	메모리 순서 보장	성능
`SeqCst`	가장 엄격 (순차 일관성)	가장 느림
`Acquire` / `Release`	중간	중간
`Relaxed`	최소	가장 빠름

15.7 스레드 풀 (Thread Pool)

반복적인 스레드 생성은 비용이 큽니다. 스레드 풀을 사용하세요:

```
// rayon 크레이트 사용 예시
// use rayon::prelude::*;
//
// fn main() {
//     let v: Vec<i32> = (1..1000000).collect();
//
//     // 병렬 이터레이터
//     let sum: i32 = v.par_iter().sum();
//
//     println!("Sum: {}", sum);
// }
```

🚀 실습 과제

과제 1: 병렬 다운로더

```
// TODO: 여러 URL을 병렬로 다운로드
// - 채널로 결과 수집
// - Arc<Mutex<T>>로 진행률 추적
```

과제 2: 생산자-소비자 패턴

```
// TODO: N개의 생산자, M개의 소비자
// - mpsc 채널 사용
// - 작업 완료 조건 처리
```

🚀 실습 가이드

`examples/` 폴더에서 확인:

1. `threads.rs`: 스레드 생성과 move
2. `channels.rs`: 채널로 메시지 전송
3. `shared_state.rs`: Arc + Mutex 패턴

📚 핵심 정리

개념	용도
`thread::spawn`	새 스레드 생성
`JoinHandle::join`	스레드 완료 대기
`move`	클로저에 소유권 이동
`mpsc::channel`	메시지 패싱 (다중 송신, 단일 수신)
`Mutex<T>`	상호 배제 락

개념	용도
`RwLock<T>`	다중 읽기 / 단일 쓰기 락
`Arc<T>`	멀티스레드용 참조 카운팅
`Send`	스레드 간 소유권 이동 가능
`Sync`	스레드 간 참조 공유 가능
`Atomic*`	락 없는 원자적 연산

Chapter 16: 객체지향 특징 (OOP Features)

> “Rust는 상속 대신 구성(Composition)과 트레이트(Trait)를 통해 더 유연한 설계를 유도합니다.”

16.1 상속의 문제와 Rust의 대안

전통적인 OOP(상속)는 부모 클래스의 불필요한 데이터까지 물려받는 “바나나/고릴라/정글” 문제를 야기할 수 있습니다. Rust는 **트레이트 객체(Trait Object)**를 통해 다형성만 취합니다.

```
// Box<dyn Draw>는 "Draw 트레이트를 구현한 어떤 것"을 의미합니다.
// 런타임에 vtable을 조회하여 메소드를 호출합니다 (Dynamic Dispatch).
let components: Vec<Box<dyn Draw>> = vec![
    Box::new(Button { ... }),
    Box::new(SelectBox { ... }),
];
```

이 방식은 코드를 유연하게 만들지만, 약간의 런타임 비용이 듭니다. 성능이 중요하다면 제네릭(Static Dispatch)을, 유연성이 중요하다면 트레이트 객체를 선택하면 됩니다. Rust는 이 선택권을 개발자에게 줍니다.

Chapter 17: 패턴 매칭 심화 (Advanced Patterns)

> “패턴 매칭은 단순히 값을 비교하는Switch case가 아닙니다. 데이터의 구조를 해체(Destructuring)하고 흐름을 제어하는 언어입니다.”

17.1 논박 가능성(Refutability)

- `let x = 5;`: 100% 성공합니다. (논박 불가능)
- `if let Some(x) = opt`: 실패할 수 있습니다. (논박 가능)

Rust는 이 둘을 엄격히 구분합니다. 실패할 수 있는 패턴을 `let`에 쓰면 컴파일 에러가 납니다.

17.2 매치 가드(Match Guard)와 @바인딩

패턴 매칭의 표현력은 끝이 없습니다.

```
match num {
    // x가 5보다 작을 때만 매칭 (Guard)
    Some(x) if x < 5 => println!("Less than five: {}", x),

    // 범위에 매칭되면서 동시에 id 변수에 값을 저장 (@ binding)
    Message::Hello { id: id @ 3..=7 } => println!("ID: {}",
id),

    _ => (),
}
```

Chapter 18: [프로젝트] 멀티스레드 웹 서버 (Multithreaded Web Server)

> “진정한 러스트 개발자로 거듭나는 마지막 관문입니다. 소유권, 트레이트, 스마트 포인터, 병행성을 모두 쏟아부어 안정적인 서버를 만듭니다.”

18.1 프로젝트 개요

우리는 `Hello`를 응답하는 아주 단순한 HTTP 서버를 만듭니다. 하지만 내부는 아주 견고하게 설계됩니다.

1. **TCP 리스너**: 포트를 열고 연결을 기다립니다.
2. **요청 파싱**: 들어온 바이트 스트림을 HTTP 요청으로 해석합니다.
3. **스레드 풀 (Thread Pool)**:
    - 무한정 스레드를 생성하면 서버가 터집니다(DoS 취약).
    - 고정된 개수(예: 4개)의 스레드를 미리 만들어두고 작업을 분배합니다.
4. **우아한 종료 (Graceful Shutdown)**:
    - Ctrl+C를 눌렀을 때, 하던 작업을 마저 끝내고 안전하게 종료합니다. `Drop` 트레이트의 진가를 볼 수 있습니다.

18.2 핵심 아키텍처: ThreadPool

```
pub struct ThreadPool {
    workers: Vec<Worker>,
    sender: mpsc::Sender<Job>,
}
```

- **Sender**: 메인 스레드가 작업을 내려보내는 채널입니다.
- **Worker**: 각 스레드를 감싸고 있는 래퍼입니다. 내부적으로 `Receiver`를 공유(`Arc<Mutex<..>>`)하여 경쟁적으로 작업을 가져갑니다.

이 패턴은 Rust의 “Shared-State Concurrency“와 “Message Passing“이 얼마나 조화롭게 섞일 수 있는지 보여주는 최고의 예제입니다.

🚀 마치며

이 책을 끝까지 읽으셨다면, 이미 Rustacean입니다. 더 이상 컴파일러와 싸우지 말고, 컴파일러를 믿고 든든한 동료로 삼아 더 안전한 시스템을 만드세요.