

# Rust 빠른 참조

소유권, 타입, 트레잇, 패턴 매칭 핵심

## 기본

### Hello World

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

### Cargo 명령어

```
cargo new my_project # create new project
cargo build          # compile (debug)
cargo build --release # compile (optimized)
cargo run            # build and run
cargo test           # run tests
```

### 프로젝트 구조

```
Cargo.toml 프로젝트 매니페스트 (의존성, 메타데이터)
src/main.rs 바이너리 크레이트 진입점
src/lib.rs 라이브러리 크레이트 루트
tests/ 통합 테스트 디렉터리
```

## 변수 및 가변성

### 바인딩 및 가변성

```
let x = 5; // immutable by default
let mut y = 10; // mutable
y += 1;
const MAX: u32 = 100; // compile-time constant
```

### 새도입

```
let x = 5;
let x = x + 1; // shadows previous x
let x = "now a string"; // can change type
```

### 스칼라 타입

```
i8..i128, isize 부호 있는 정수
u8..u128, usize 부호 없는 정수
f32, f64 부동소수점 (f64 기본값)
bool true / false
char 유니코드 스칼라 값 (4바이트)
```

### 복합 타입

```
let tup: (i32, f64, char) = (42, 6.4, 'z');
let (a, b, c) = tup; // destructure
let arr: [i32; 3] = [1, 2, 3];
let first = arr[0];
```

## 함수

### 정의

```
fn add(a: i32, b: i32) -> i32 {
    a + b // no semicolon = return expression
}
```

### 클로저

```
let double = |x: i32| x * 2;
let sum: i32 = vec![1, 2, 3]
    .iter()
    .map(|x| x * 2)
    .sum();
```

## 함수 포인터 및 트레잇

```
fn(T) -> U 함수 포인터 타입
Fn(T) -> U 빌리는 클로저
FnMut(T) -> U 가변 빌리는 클로저
FnOnce(T) -> U 소유권을 가져가는 클로저
```

## 제어 흐름

### If / Else

```
let status = if score >= 90 { "A" }
              else if score >= 80 { "B" }
              else { "C" }; // if is an expression
```

### 반복문

```
loop { break; } // infinite
while condition { } // while
for item in &vec { } // iterator
for i in 0..10 { } // range
for (i, v) in vec.iter().enumerate() { }
```

### 반복문 레이블

```
'outer: for i in 0..5 {
    for j in 0..5 {
        if i + j > 6 { break 'outer; }
    }
}
```

## 소유권 및 빌림

### 소유권 규칙

- 각 값에는 정확히 하나의 소유자가 있다.
- 소유자가 범위를 벗어나면 값이 삭제된다.
- 값은 이동하거나 복제할 수 있다.

### 이동 및 복제

```
let s1 = String::from("hello");
let s2 = s1; // s1 is moved, no longer valid
let s3 = s2.clone(); // deep copy, both valid
```

### 빌림

```
fn len(s: &String) -> usize { s.len() } // shared ref
fn push(s: &mut String) { s.push('!'); } // mutable ref
// Rule: many &T OR one &mut T, never both
```

### 수명

```
fn longest<'a>(a: &'a str, b: &'a str) -> &'a str {
    if a.len() > b.len() { a } else { b }
}
```

## 구조체 및 열거형

### 구조체

```
struct User {
    name: String,
    age: u32,
    active: bool,
}
let u = User { name: String::from("Alice"), age: 30, active: true };
```

## Impl 블록

```
impl User {
    fn new(name: &str, age: u32) -> Self {
        Self { name: name.to_string(), age, active: true }
    }
    fn greeting(&self) -> String {
        format!("Hi, {}", self.name)
    }
}
```

## 열거형

```
enum Shape {
    Circle(f64),
    Rect { w: f64, h: f64 },
    Point,
}
let s = Shape::Circle(5.0);
```

## 패턴 매칭

### Match 표현식

```
match shape {
    Shape::Circle(r) => std::f64::consts::PI * r * r,
    Shape::Rect { w, h } => w * h,
    Shape::Point => 0.0,
}
```

### If Let 및 While Let

```
if let Some(val) = optional {
    println!("{val}");
}
while let Some(top) = stack.pop() {
    println!("{top}");
}
```

### 패턴 문법

```
- 와일드카드, 모든 것 매칭
x @ 1..=5 매칭된 범위를 x에 바인딩
(a, b, ..) 튜플 구조 분해, 나머지 무시
Some(x) if x > 0 매치 가드
Foo { x, .. } 구조체, 다른 필드 무시
```

## 오류 처리

### Result 및 Option

```
enum Result<T, E> { Ok(T), Err(E) }
enum Option<T> { Some(T), None }
```

### ? 연산자

```
fn read_file(path: &str) -> Result<String, io::Error> {
    let mut s = String::new();
    File::open(path)?.read_to_string(&mut s);
    Ok(s)
}
```

### 오류 처리

```
match result {
    Ok(val) => println!("{val}"),
    Err(e) => eprintln!("Error: {e}"),
}
let val = result.unwrap_or(0);
let val = result.unwrap_or_else(|_| default());
```

# Rust 빠른 참조

## 일반 메서드

<code>.unwrap()</code>	값을 가져오거나 패닉
<code>.expect(msg)</code>	값을 가져오거나 메시지와 함께 패닉
<code>.unwrap_or(default)</code>	값을 가져오거나 기본값 사용
<code>.map(f)</code>	Ok/Some 값 변환
<code>.and_then(f)</code>	연산 체인 (flatmap)
<code>.is_ok() / .is_some()</code>	불리언 확인

## 트레이트

### 정의 및 구현

```
trait Summary {
    fn summarize(&self) -> String;
    fn preview(&self) -> String { // default impl
        format!("{}", ..., &self.summarize()[..20])
    }
}
impl Summary for User {
    fn summarize(&self) -> String { self.name.clone() }
}
```

### 트레이트 바운드

```
fn notify(item: &impl Summary) { }
fn notify<T: Summary + Display>(item: &T) { }
fn notify(item: &(impl Summary + Display)) { }
```

## 일반 트레이트

<b>Display</b>	사용자 대상 문자열 형식화
<b>Debug</b>	디버그 형식화 ({}:?)
<b>Clone, Copy</b>	복제 (깊은 복사 / 비트 복사)
<b>PartialEq, Eq</b>	동등 비교
<b>PartialOrd, Ord</b>	순서 비교
<b>Iterator</b>	반복을 위한 next()
<b>From, Into</b>	타입 변환
<b>Default</b>	기본값 생성자

## 컬렉션

### Vec

```
let mut v: Vec<i32> = vec![1, 2, 3];
v.push(4);
v.pop(); // returns Option<i32>
let first = &v[0]; // panics if empty
let first = v.get(0); // returns Option<&i32>
```

### HashMap

```
use std::collections::HashMap;
let mut m = HashMap::new();
m.insert("key", 42);
m.entry("key").or_insert(0);
if let Some(val) = m.get("key") { }
```

### String

```
let s = String::from("hello");
let s = "hello".to_string();
let combined = format!("{}", s, "world");
for c in s.chars() { } // iterate characters
```

### 이터레이터

```
let sum: i32 = vec![1, 2, 3].iter().sum();
let doubled: Vec<_> = v.iter().map(|x| x * 2).collect();
let evens: Vec<_> = v.iter().filter(|x| *x % 2 == 0).collect();
```

## 동시성

### 스레드

```
use std::thread;
let handle = thread::spawn(|| {
    println!("from spawned thread");
});
handle.join().unwrap();
```

### 채널

```
use std::sync::mpsc;
let (tx, rx) = mpsc::channel();
tx.send(42).unwrap();
let val = rx.recv().unwrap();
```

### 공유 상태

<b>Arc&lt;T&gt;</b>	원자적 참조 카운팅 (스레드 안전 Rc)
<b>Mutex&lt;T&gt;</b>	상호 배제, 내부 값 접근을 위해 잠금
<b>RwLock&lt;T&gt;</b>	여러 독자 또는 하나의 쓰기자
<b>Send</b>	트레이트: 스레드 간 전송 안전
<b>Sync</b>	트레이트: 스레드 간 참조 공유 안전

## 매크로 및 속성

### 일반 매크로

<b>println!()</b>	개행 포함 출력
<b>format!()</b>	형식화된 String 반환
<b>vec![]</b>	리터럴로 Vec 생성
<b>todo!()</b>	플레이스홀더, 런타임에 패닉
<b>assert!(expr)</b>	expr이 false면 패닉
<b>assert_eq!(a, b)</b>	a != b이면 패닉

### 파생 속성

```
#[derive(Debug, Clone, PartialEq)]
struct Point { x: f64, y: f64 }
// Auto-implements Debug, Clone, PartialEq
```

### 테스트 속성

```
#[cfg(test)]
mod tests {
    use super::*;
    #[test]
    fn it_works() { assert_eq!(add(2, 2), 4); }
    #[test]
    #[should_panic]
    fn it_panics() { panic!("boom"); }
}
```