Rust Programming Language(Part I)

- Rust Basics
- Ownership & Lifetime
- Error Handling(Option & Result)
- Object Oriented Programing

Slowboot
(chunghan.yi@gmail.com)
Doc. Revision: 0.92

목차(Table of Contents)

소유권(Ownership)

Error Handling
(Option<T>, Result<T, E>

Struct + Generics + Trait =>
OOP

Iterator + Closure =>
Functional Programming

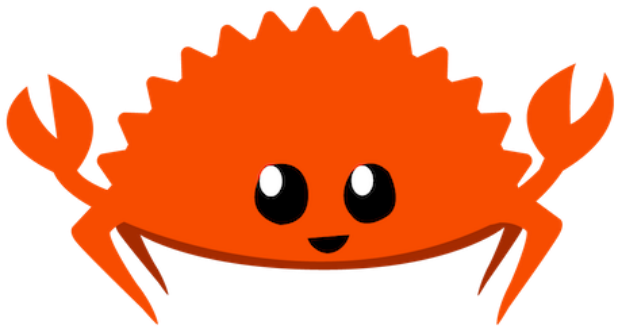
Smart Pointer

Thread, Async
Concurrent Programming

Part I

Part II

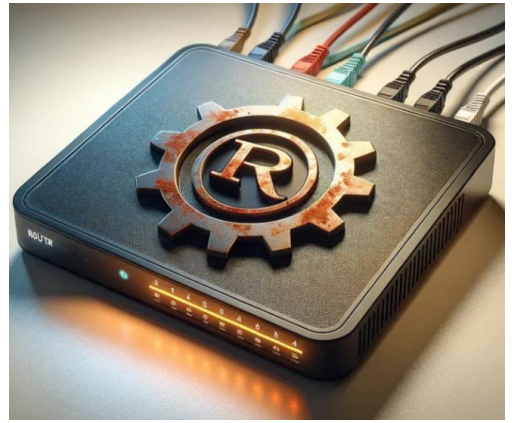
Rust의 6가지 주요 항목



취지(Purpose)

- [1] 어려운 Rust language의 개념을 쉽게 요약 & 정리한다.
- [2] Rust language의 주요 concept를 빠르게 훑어 봄으로써, 이후 관련 서적을 읽을 때 도움이 될 수 있도록 한다.
- [3] 따라서 Rust를 구성하는 모든 내용을 세세히 소개하는 것 보다는, 실제로 필드에서 자주 사용하게 되는 개념을 중심으로 예제와 함께 소개한다.

Let's make a Rust Router!

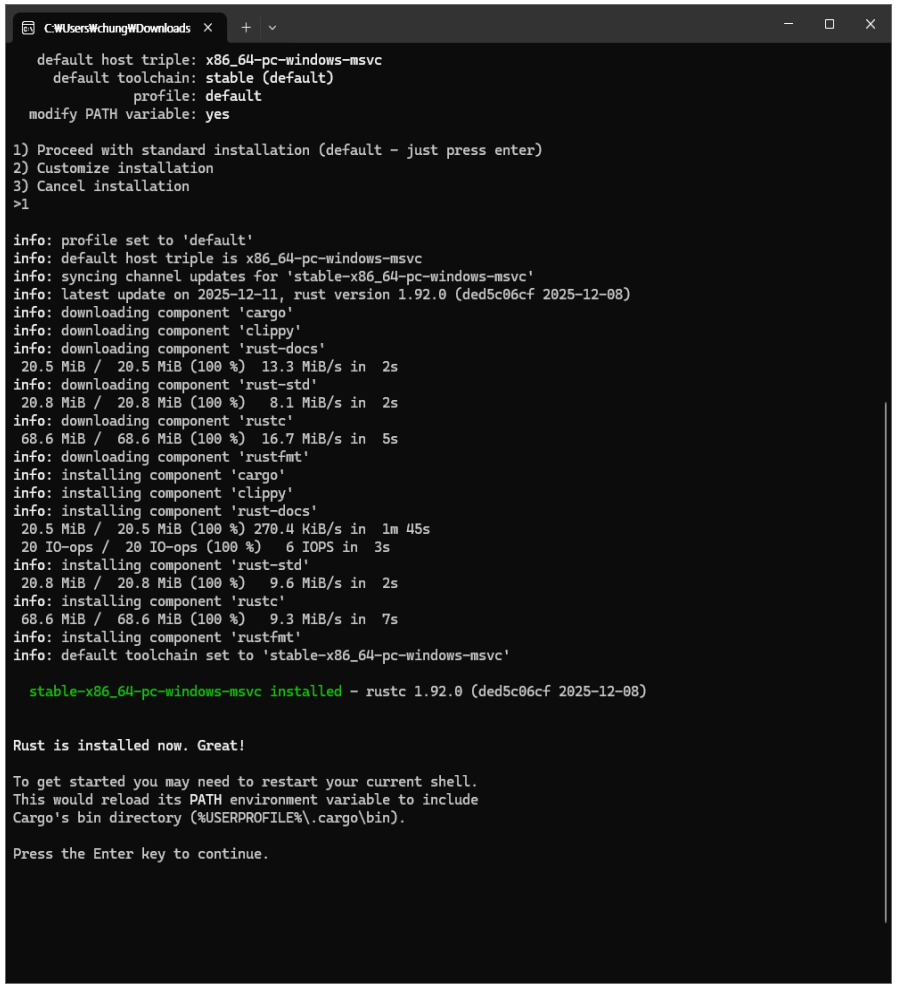


1. Rust 개발 환경 구축

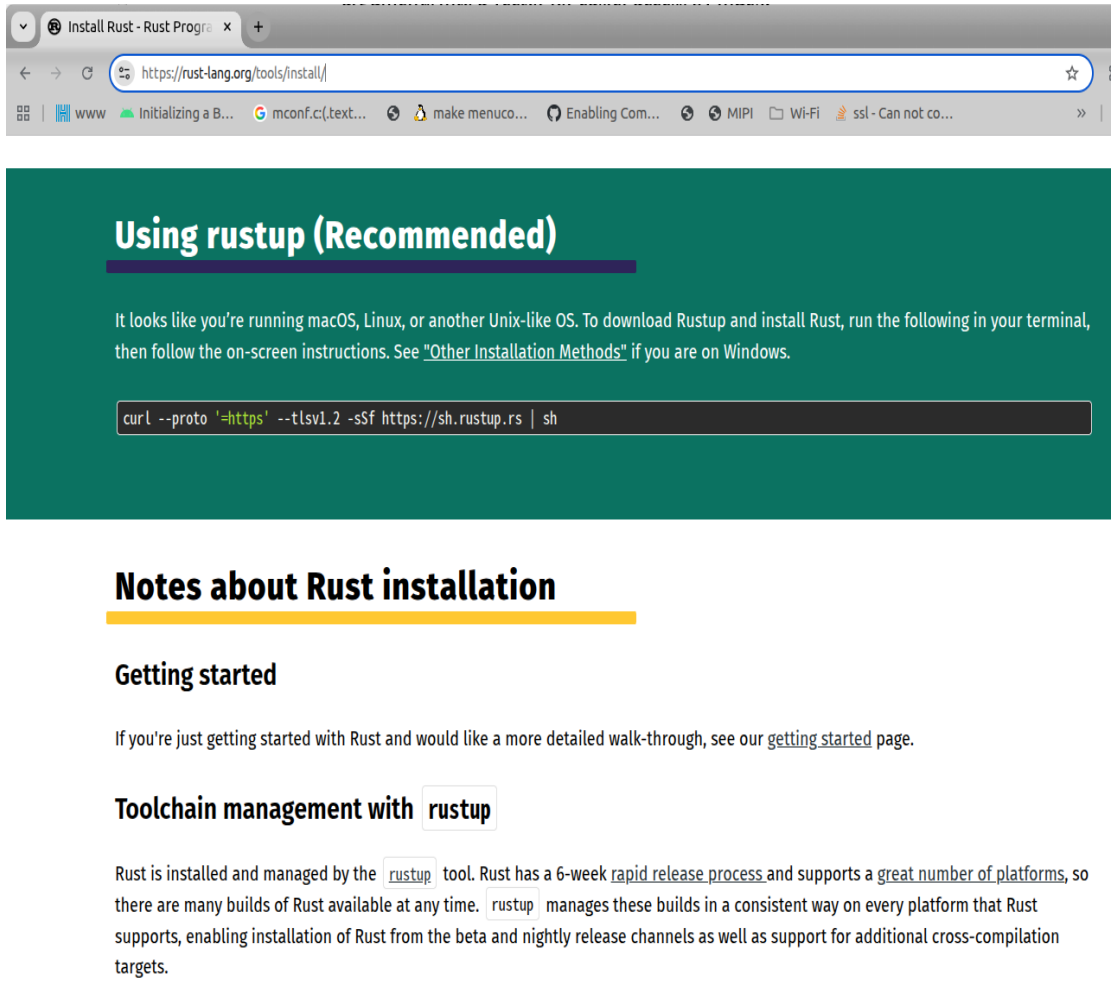
Windows, MacOS and Linux

1. Rust 개발 환경 구축(1) – Windows 환경

https://rustup.rs/
=> rustup-init.exe



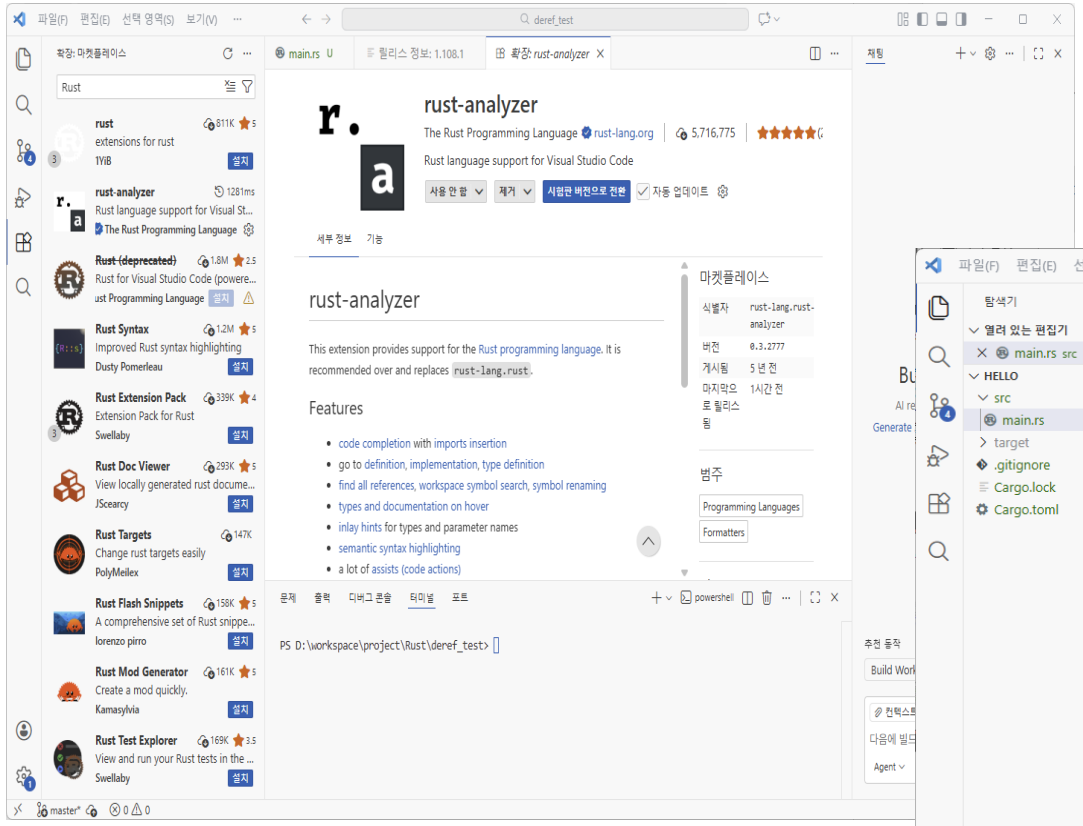
1. Rust 개발 환경 구축(2) – Linux 환경



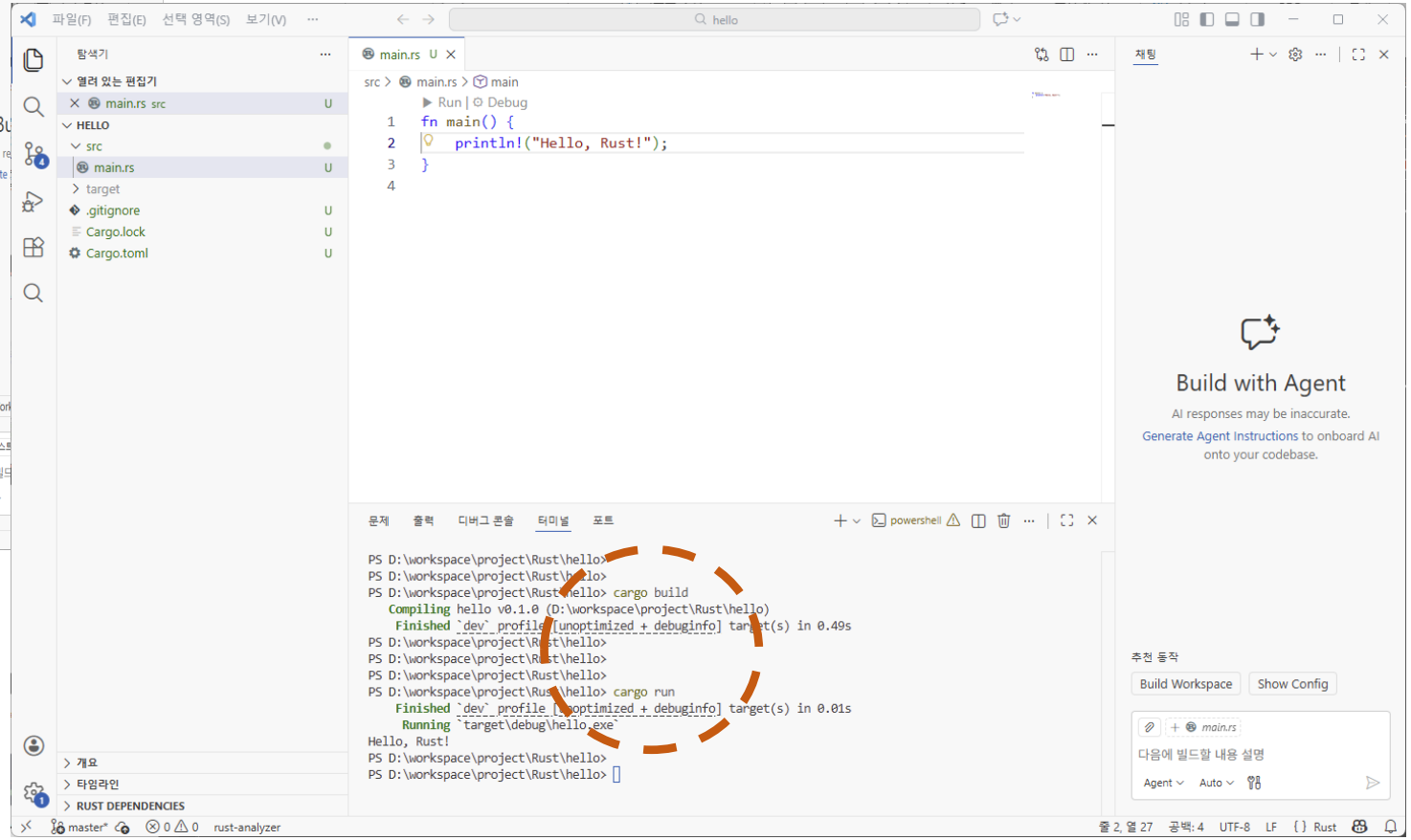
<How to install Rust on Ubuntu>

```
$ sudo apt update && sudo apt upgrade -y
$ sudo apt install build-essential –y
$ sudo apt install curl -y

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

$ source $HOME/.cargo/env
$ rustc –-version
$ cargo --version
```

1. Rust 개발 환경 구축(3) – Visual Studio Code



Hello, Rust example



- Visual Studio Code
- RustRover by IntelliJ IDEA
- Neovim

1. Rust 개발 환경 구축(4) – Hello Rust

main.rs

```
fn developer_mood(caffeine_level: u8) -> &'static str {
    match caffeine_level {
        0 => "I can't work without coffee! ☕",
        1..=2 => "Alright, I can write a few functions.",
        3..=5 => "Productivity mode: ON! 🚀",
        _ => "Too much coffee! I'm rewriting the entire project in Rust! 🔥"
    }
}

fn main() {
    println!("{}", developer_mood(4));
    // Output: Productivity mode: ON! 🚀
}
```

<How to create a project and build>

$ cargo new hello

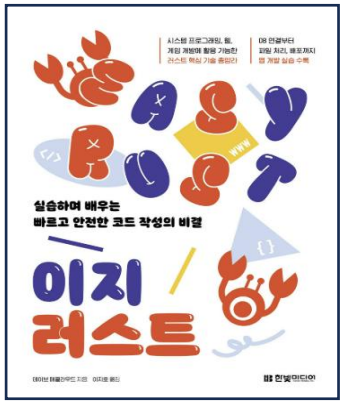
Edit src/main.rs

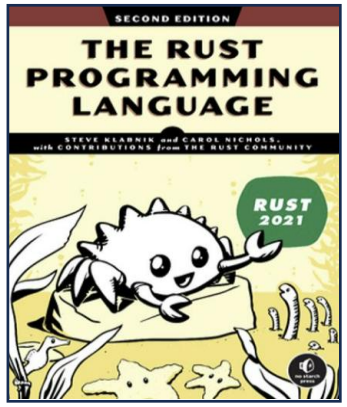
$ cargo build
or
$ cargo build –release

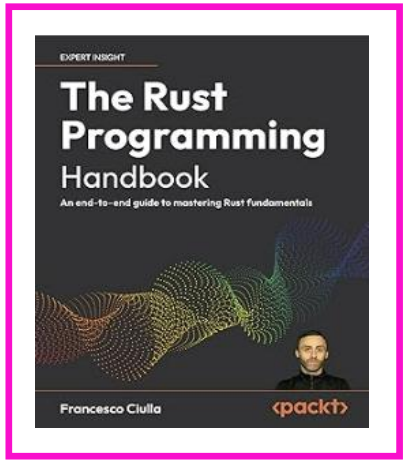
$ cargo run

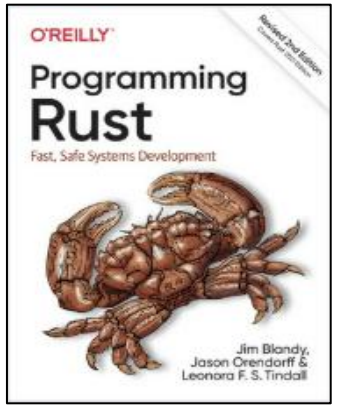
$ cd target/debug
$ ls –l hello

1. Rust 개발 환경 구축(5) – Reference Books









https://slowbootkernelhacks.blogspot.com/2026/02/howaboutlearningrustprogramminglanguage.html

2. Rust 기초

변수, 상수, 함수, 제어

2. Rust 기초(1) – Variable Types(1)

정수 변수 i32, u32 사용 예

```
fn main() {
    let signed_int: i32 = -42;
    let unsigned_int: u32 = 42;
    println!("Signed integer: {}, Unsigned integer: {}", signed_int,
unsigned_int);
}
```

실수 변수 사용 예

```
fn main() {
    let float_num: f64 = 3.14;
    println!("Floating-point number: {}", float_num);
}
```

Boolean 변수 사용 예

```
fn main() {
    let is_rust_fun: bool = true;
    println!("Is Rust fun? {}", is_rust_fun);
}
```

Character 변수 사용 예

```
fn main() {
    let letter: char = 'R';
    let emoji: char = '😊';
    println!("Letter: {}, Emoji: {}", letter, emoji);
}
```

2. Rust 기초(1) – Variable Types(2)

Tuple 변수 사용 예

```
fn main() {
    // A tuple holding an integer, a float, and a character.
    let my_tuple: (i32, f64, char) = (500, 6.4, 'R');

    // Method 1: Destructuring with a `let` binding.
    // This is a form of pattern matching that breaks the tuple into
separate variables.
    let (x, y, z) = my_tuple;
    let first_element = my_tuple.0;
    let second_element = my_tuple.1;
```

Array 변수 사용 예

```
fn main() {
    let array: [i32; 3] = [1, 2, 3];
    println!("Array values: {} {} {}", array[0], array[1], array[2]);
}
```

2. Rust 기초(1) – Variable Types(3)

Slice 변수 사용 예

```rust
fn main() {
    let array = [10, 20, 30, 40, 50];
    // Create a slice that references elements from index 1 up to (but not including) index 3.
    // The type of `slice` is `&[i32]`.
    let slice = &array[1..3];
    println!("Original array: {:?}", array);
    println!("Slice (a view into the array): {:?}", slice); // Output: [20, 30]
}
```

String 변수 사용 예

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

&str 변수 사용 예

```rust
fn main() {
    let s = "Hello, world!"; // string literal
    println!("{}", s);
}
```

2. Rust 기초(2) – Immutable & Mutable Variable

```rust
fn main() {
    let x = 5;
    println!("The value of x is: {}", x);
    // x = 6; // This line would cause a compile-time error because x is
immutable
}
```

Immutable 변수는 수정할 수 없음.

```rust
fn main() {
    let x = 5;
    let x = x + 1; // This shadows the previous x
    println!("The value of x is: {}", x);
}
```

변수 Shadowing

```rust
fn main() {
    let mut x = 5;
    println!("The value of x is: {}", x);
    x = 6; // This is allowed because x is mutable
    println!("The value of x is: {}", x);
}
```

Mutable 변수는 수정할 수 있음.

2. Rust 기초(3) – 구조체 Struct

struct 변수 사용 예

```rust
struct User {
    username: String,
    email: String,
    sign_in_count: u64,
    active: bool,
}

fn main() {
    let user1 = User {
        username: String::from("someusername123"),
        email: String::from("someone@example.com"),
        sign_in_count: 1,
        active: true,
    };

    println!("Username: {}", user1.username);
    println!("Email: {}", user1.email);
    println!("Sign in count: {}", user1.sign_in_count);
    println!("Active: {}", user1.active);
}
```

2. Rust 기초(4) – 열거형 Enum

enum 변수 사용 예

```rust
#[derive(Debug)]
enum Color {
    // A variant that holds a tuple of three 8-bit unsigned integers
    Rgb(u8, u8, u8),
    // A variant that holds a single String
    Named(String),
}

fn main() {
    let red = Color::Rgb(255, 0, 0);
    let custom_color = Color::Named(String::from("Forest Green"));

    println!("An RGB color: {:?}", red);
    println!("A named color: {:?}", custom_color);
}
```

Rust의 enum의 각 variant는 type을 가질 수 있음.

2. Rust 기초(5) – 함수 Function(1)

Function 사용 예

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

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

fn add(a: i32, b: i32) -> i32 {
    a + b
}
```

fn 함수명 (인수 선언) -> 반환 값 타입 {

}

Call by value 예

```rust
fn takes_value_copy(mut some_integer: i32) {
    some_integer += 1;
    println!("Value inside function: {}", some_integer);
}

fn main() {
    let x = 5;
    takes_value_copy(x);
    println!("Original value of x after function call: {}", x); // x is still 5
}
```

2. Rust 기초(5) – 함수 Function(2)

Call by reference 예

```rust
// This function borrows a String and calculates its length.
fn calculate_length(s: &String) -> usize {
    s.len()
} // `s` goes out of scope here, but since it doesn't have ownership, the data is not dropped.

fn main() {
    let s1 = String::from("hello");

    // We pass a reference to s1 using the `&` operator.
    // s1 is borrowed, not moved.
    let len = calculate_length(&s1);

    println!("The length of '{}' is {}.", s1, len);
    // `s1` is still valid here because its ownership was never transferred.
}
```

Call by mutable reference 예

```rust
fn main() {
    let mut s = String::from("hello");
    takes_mutable_reference(&mut s);
    println!("s in main: {}", s); // s is modified by the function
}

fn takes_mutable_reference(some_string: &mut String) {
    some_string.push_str(", world");
}
```

2. Rust 기초(7) – if else 문과 loop 문

if and else문 예

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

    if number < 5 {
        println!("The number is less than 5");
    } else if number > 5 {
        println!("The number is greater than 5");
    } else {
        println!("The number is exactly 5");
    }
}
```

loop문 예

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

    loop {
        counter += 1;
        println!("Counter is now: {}", counter);
        if counter == 5 {
            break; // Exits the loop
        }
    }
    println!("Loop finished.");
}
```

(*) while loop문은 C와 동일하여 생략함.

2. Rust 기초(8) – for loop 문

for loop문 예

```rust
fn main() {
    for number in 1..=10 {
        // If the number is odd, skip the println! and go to the next
iteration.
        if number % 2 != 0 {
            continue;
        }
        // This line only runs for even numbers.
        println!("Found an even number: {}", number);
    }
}
```

for keyword문 예

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

    for element in a.iter() {
        println!("The value is: {}", element);
    }
}
```

2. Rust 기초(9) – match 문(1)

match 문 예#1

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

    match x {
        1 | 2 => println!("The number is one or two"),
        3 => println!("The number is three"),
        _ => println!("It's some other number"),
    }
}
```

match 문 예#2

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

fn main() {
    let msg = Message::Move { x: 10, y: 20 };

    match msg {
        Message::Quit => println!("Quit message"),
        Message::Move { x, y } => println!("Move to x: {}, y: {}", x, y),
        Message::Write(text) => println!("Write message: {}", text),
        Message::ChangeColor(r, g, b) => println!("Change color to red:
{}, green: {}, blue: {}", r, g, b),
    }
}
```

Rust에는 Switch case문이 없음.

2. Rust 기초(9) – match 문(2)

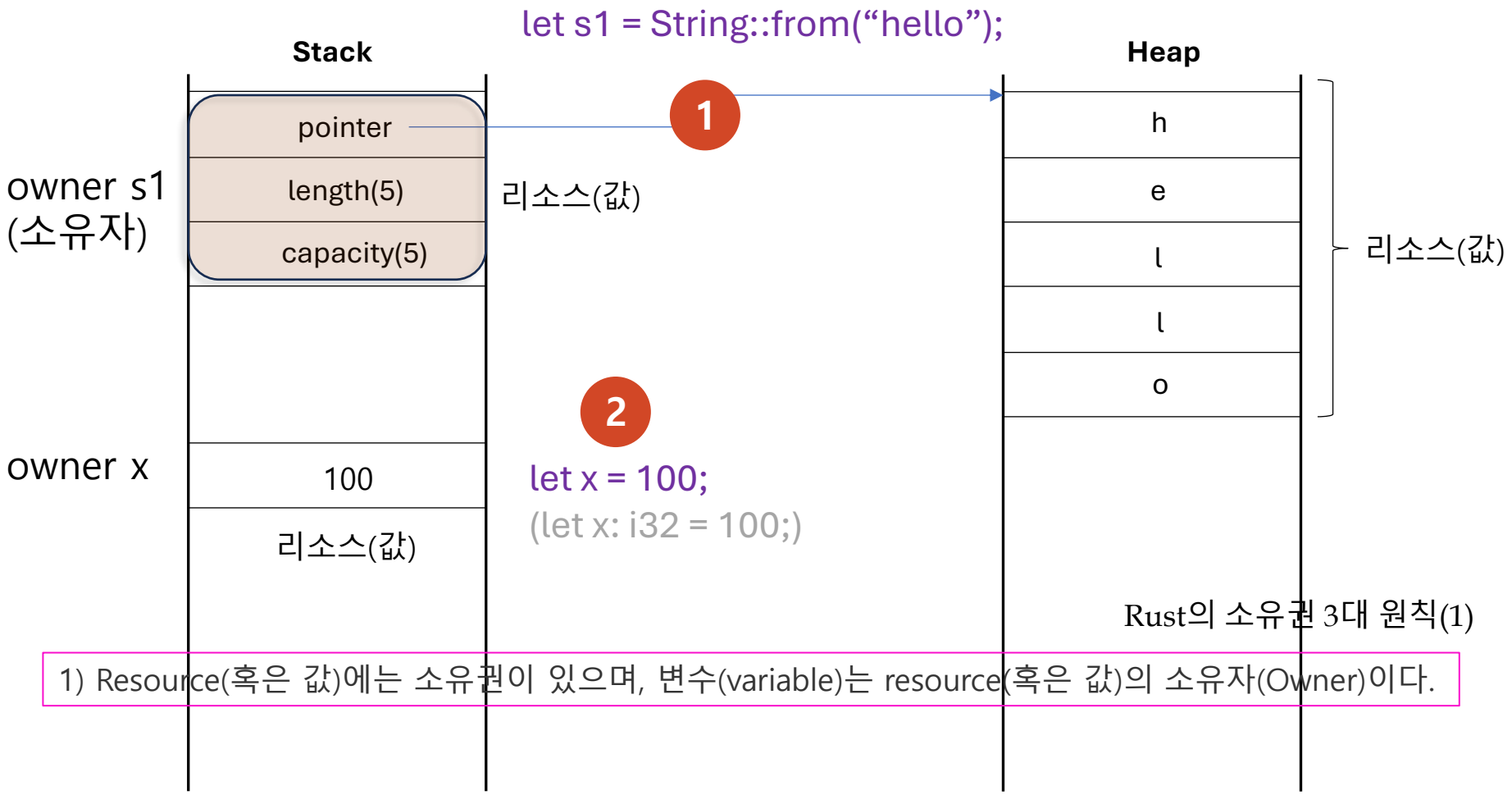
match 문 예#3

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

    match some_number {
        Some(x) => println!("The number is: {}", x),
        None => println!("No number"),
    }

    match absent_number {
        Some(x) => println!("The number is: {}", x),
        None => println!("No number"),
    }
}
```

3. 소유권(Ownership)

이동(Move), 복사(Copy), 빌림(Borrowing) & 참조(Reference)

3. 소유권(1) – 소유권 3대 원칙(1)



3. 소유권(1) – 소유권 3대 원칙(2)

Rust의 소유권 3대 원칙(2)

2) 소유권(Ownership)은 이동(move)할 수 있으며, 특정 시점에서의 소유자(Owner)는 오직 1개(1개의 변수) 뿐이다.

소유권 이동

```
let pasta = String::from("Carbonara");
let dinner = pasta; // ownership moves from `pasta` to `dinner`

// println!("{}", pasta); // ✕ Error! `pasta` no longer owns the data
println!("{}", dinner); // ☑ "Carbonara"
```

3. 소유권(1) – 소유권 3대 원칙(3)

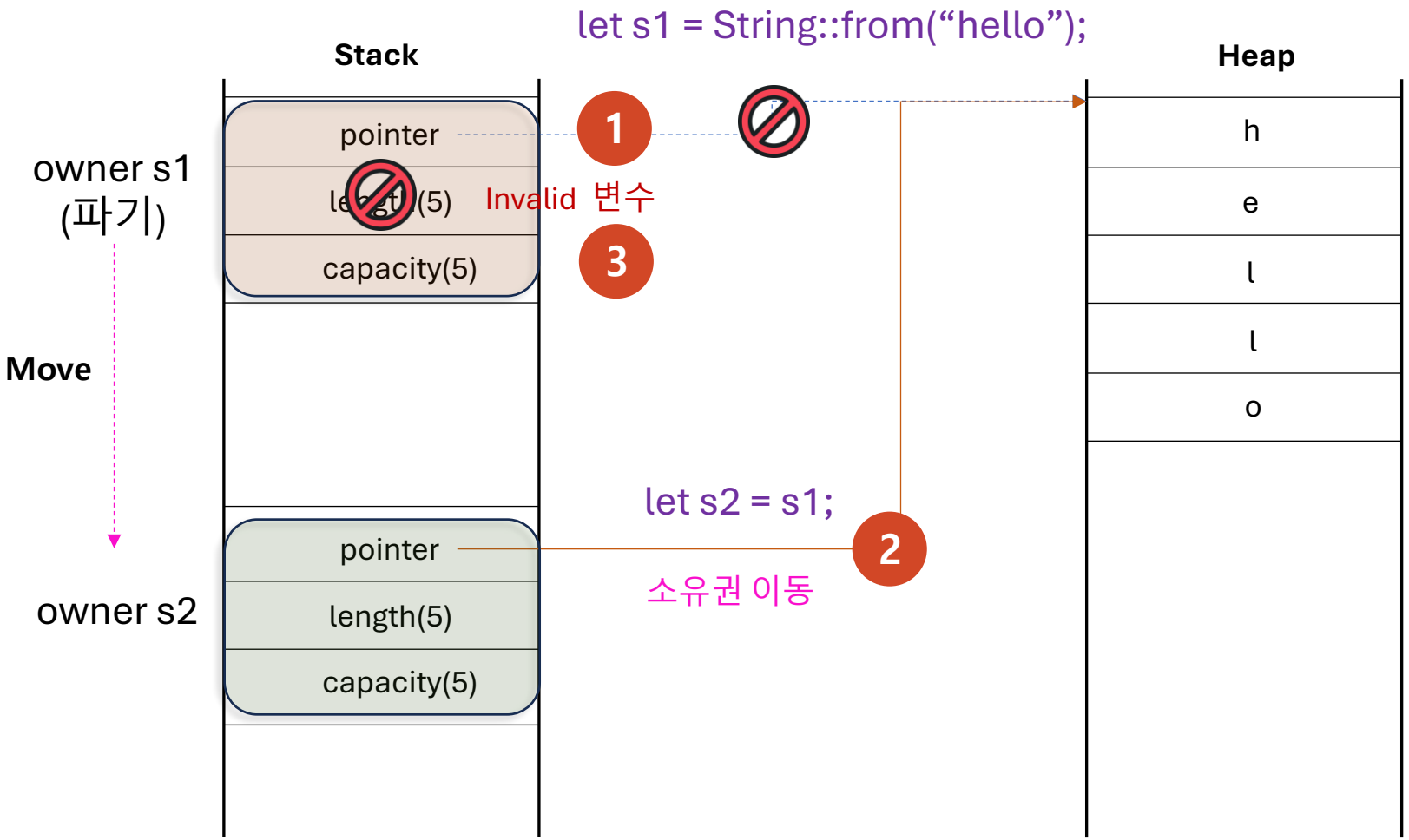
소유자가 소유한 리소스 자동 파기

```
fn main() {
    {
        let espresso = String::from("Delicious");
        println!("{}", espresso); // ☑ "Delicious"
    } // `espresso` goes out of scope here; Rust automatically drops its value

    // println!("{}", espresso); // ✕ Error! `espresso` doesn't exist anymore
}
```

Rust의 소유권 3대 원칙(3)

3) 소유자가 유효한 범위(scope)를 벗어나면, 소유자가 소유한 값은 자동으로 파기된다. 이는 memory leak을 방지해주는 효과가 있다.

3. 소유권(2) – 이동 Move(1)

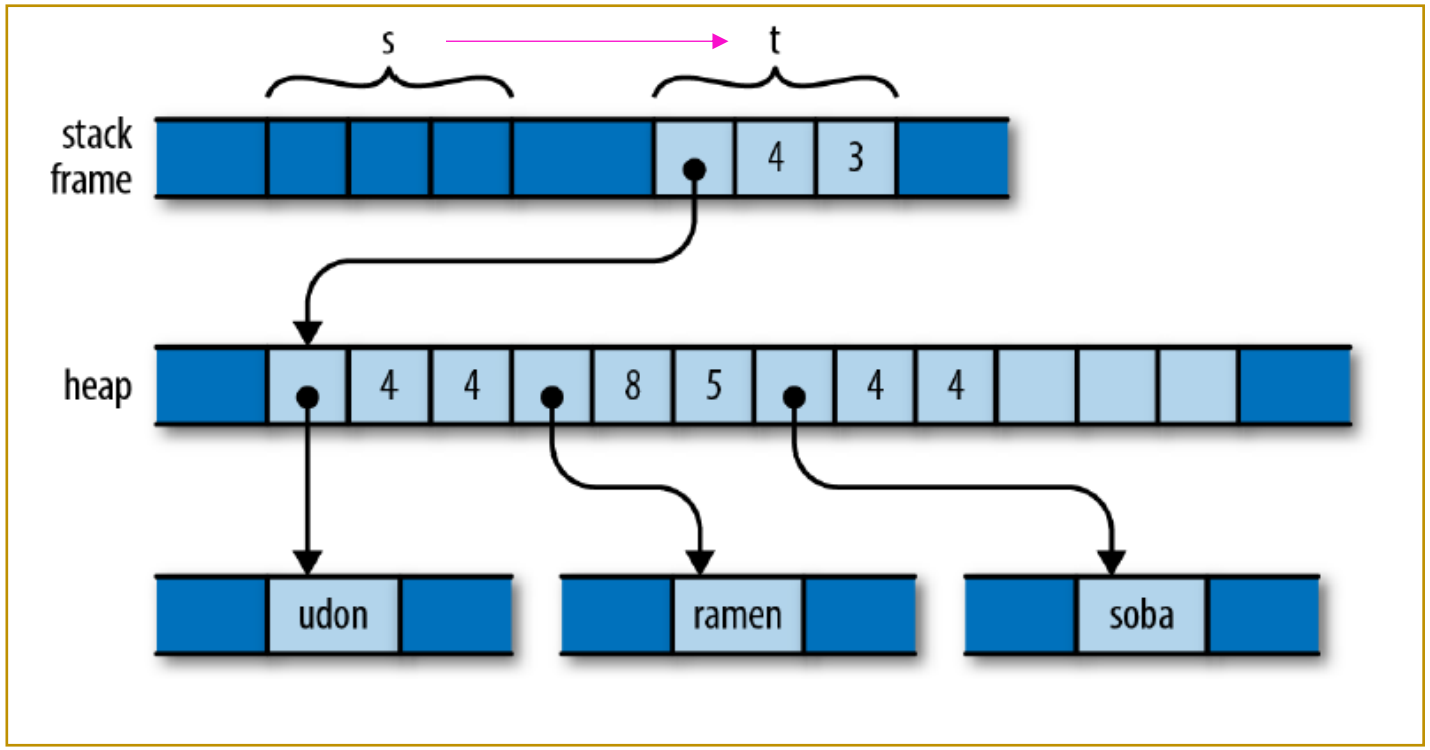


- Heap 영역에 resource가 할당되는 type의 경우, 대입문을 수행하면 값이 복사되지 않고 이동한다.
- Move는 Stack 정보만 이동하므로 빠르다.

3. 소유권(2) – 이동 Move(2)

```
let s = vec!["udon".to_string(), "ramen".to_string(), "soba".to_string()];
let t = s;
```

Vector 변수의 이동 예



3. 소유권(2) – 이동 Move(3)

변수 대입문 사용시 이동

```rust
fn main() {
    // s1 is on the stack and owns the "hello" data on the heap.
    let s1 = String::from("hello");

    // The `let s2 = s1;` line performs a move.
    // The pointer, length, and capacity from s1 are copied to s2.
    // Ownership of the heap data is transferred to s2.
    let s2 = s1;

    // After the move, s1 is no longer considered valid by the compiler.
    // Uncommenting the line below would cause a compile-time error.
    // println!("This will not compile: {}", s1);

    println!("s2 holds the value: {}", s2);
}
```

함수 파라미터 전달 시 이동

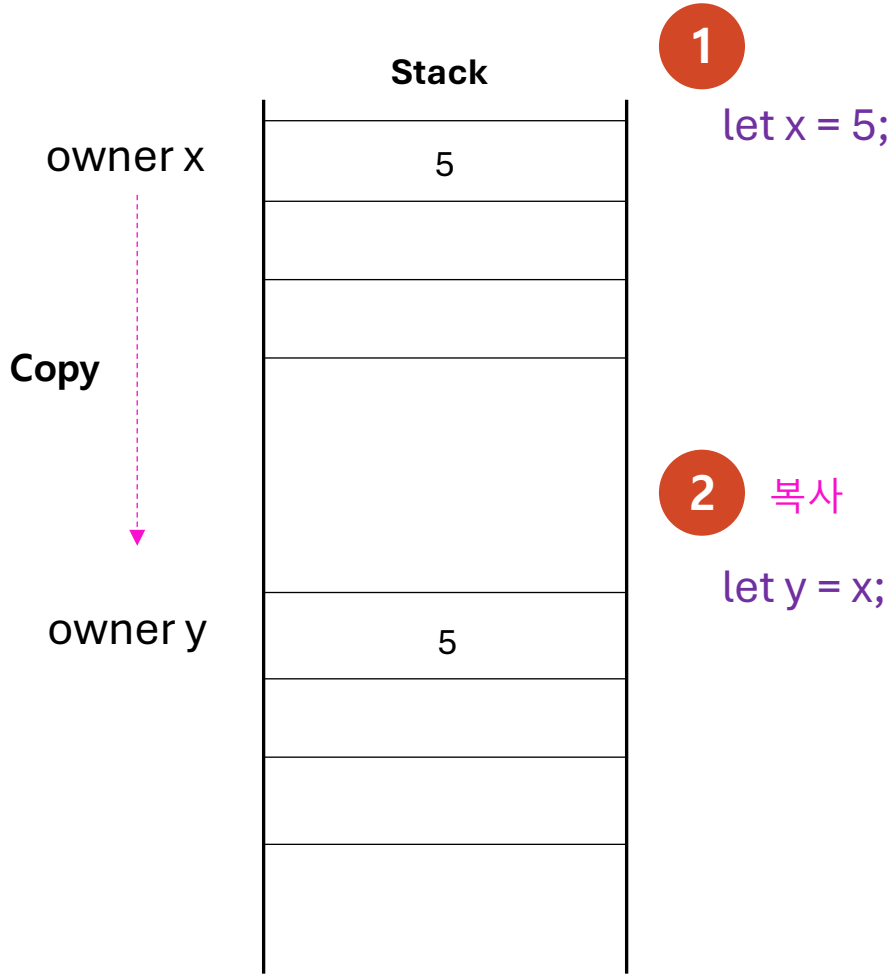
```rust
fn main() {
    let my_string = String::from("Hello, world!");

    take_ownership(my_string);

    // This line would cause a compile-time error
    // println!("{}", my_string);
}

fn take_ownership(s: String) {
    println!("Taking ownership: {}", s);
}
```

3. 소유권(2) – 이동 Move(4)

함수 결과 전달 시 이동

```
fn main() {
    let my_string = String::from("Hello, Rust!");

    let my_string = return_ownership(my_string);

    println!("{}", my_string); // Now this is valid
}

fn return_ownership(s: String) -> String {
    s
}
```

3. 소유권(3) – 복사 Copy

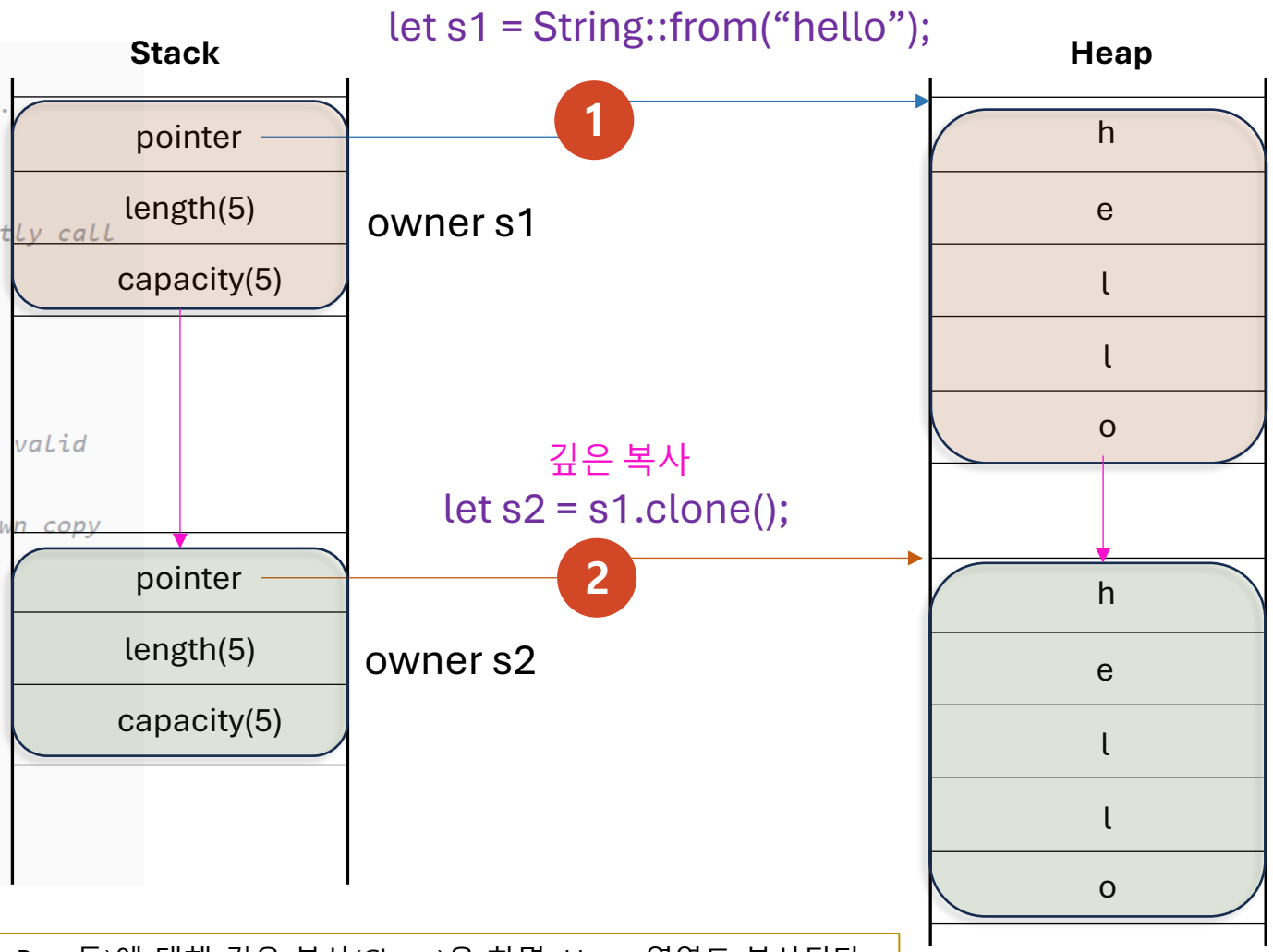


Stack에 할당되는 기본 Type(정수, 실수, Boolean 등)은 이동 대신 복사가 일어남.

```rust
fn main() {
    // `x` is an i32, which implements the `Copy` trait.
    let x = 5;

    // Because `x`'s type is `Copy`, a bit-for-bit copy of the value 5
    // is made and assigned to `y`. `x` is not moved or invalidated.
    let y = x;

    println!("x = {}, y = {}", x, y); // Both x and y are valid and can be used.

    // Let's look at a non-Copy type for contrast.
    let s1 = String::from("hello");
    // `String` does not implement `Copy`, so this is a move.
    let s2 = s1;

    // The line below would cause a compile-time error because s1 was moved.
    // println!("s1 = {}, s2 = {}", s1, s2);
    // error[E0382]: borrow of moved value: `s1`
}
```

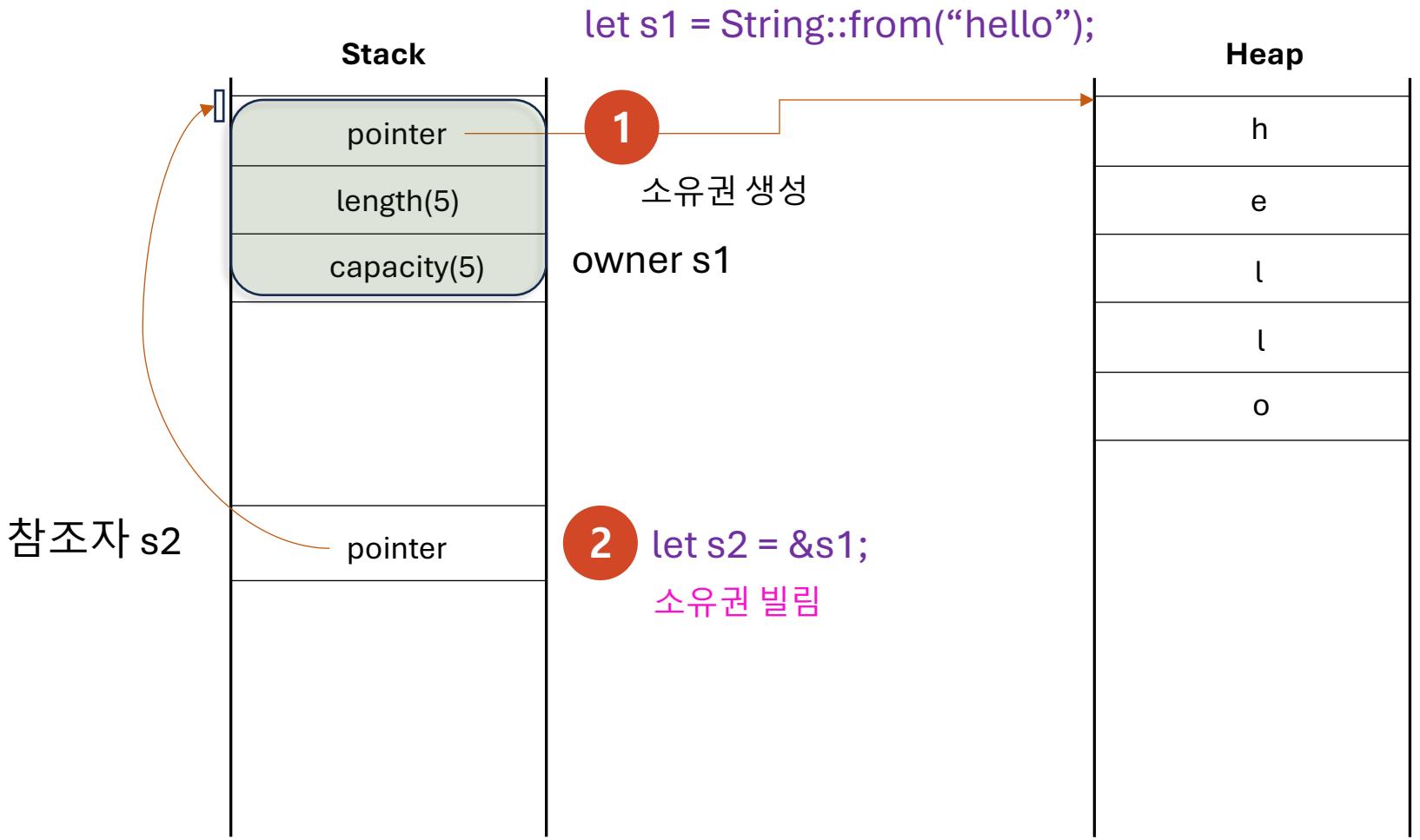
3. 소유권(4) – 깊은 복사 Clone

```rust
fn main() {
    // `s1` is a String, which does not implement the `Copy` trait.
    let s1 = String::from("hello");

    // To create an independent duplicate of `s1`, we must explicitly call
.clone().
    // This performs a deep copy of the string data on the heap.
    let s2 = s1.clone();

    // Because we cloned `s1`, the original variable `s1` is still valid
and retains ownership
    // of its own data. `s2` is a brand new String that owns its own copy
of the data.
    println!("s1 = {}, s2 = {}", s1, s2);

    // We can modify one without affecting the other.
    // let mut s3 = s1.clone();
    // s3.push_str(", world!");
    // println!("s1 = {}, s3 = {}", s1, s3);
}
```



Heap에 할당되는 Type(String, Vector, Box 등)에 대해 깊은 복사(Clone)을 하면, Heap 영역도 복사된다.

3. 소유권(5) – 빌림 Borrowing(1)



빌림(Borrwoing)은 소유권을 이동시키지 않고, 해당 Resource를 사용하는 방법으로, 참조자(Reference)를 이용하여 실현한다.

3. 소유권(5) – 빌림 Borrowing(2)

Immutable Borrowing

```rust
fn main() {
    let book = String::from("Rust Programming");

    let len = calculate_length(&book); // Borrow book immutably
    println!("The length of '{}' is {}.", book, len);

    // book is still valid here
}

fn calculate_length(s: &String) -> usize {
    s.len()
}
```

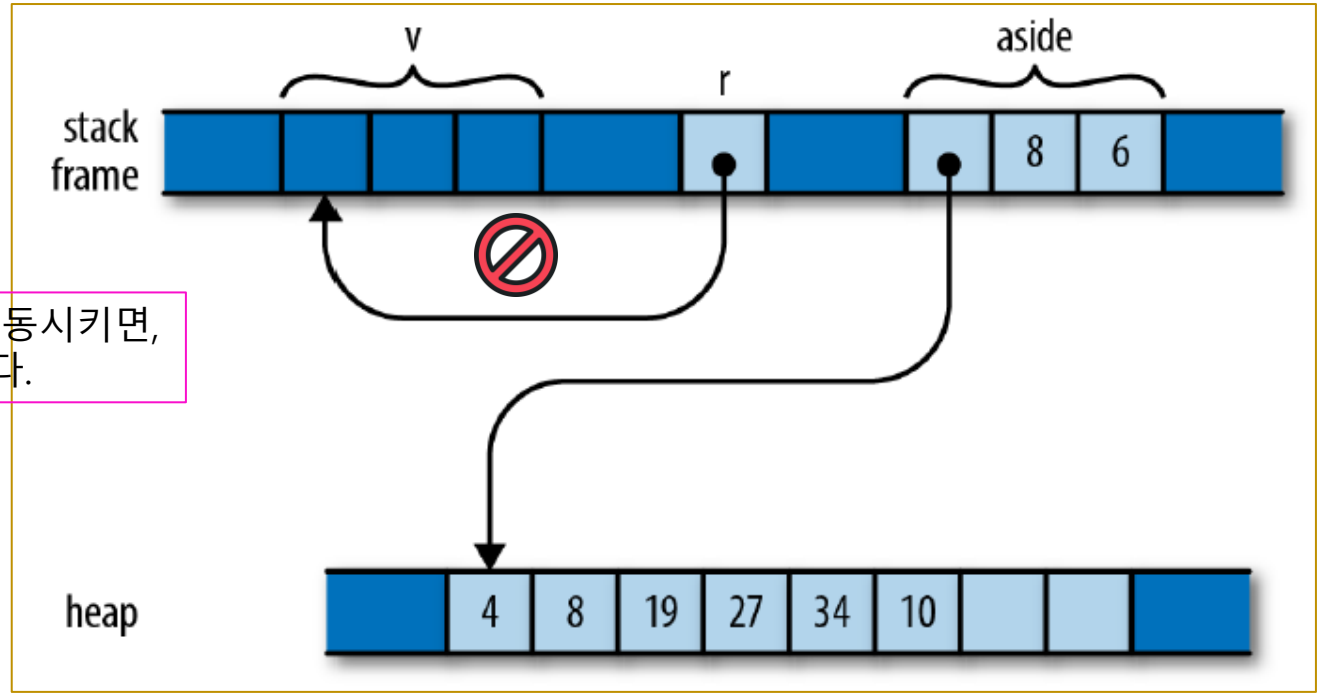
Mutable Borrowing

```rust
fn main() {
    let mut article = String::from("Rust is awesome");
    update_article(&mut article); // Borrow article mutably
    println!("Updated article: {}", article);
    // article is still valid here
}

fn update_article(s: &mut String) {
    s.push_str(" for system programming!");
}
```

3. 소유권(5) – 빌림 Borrowing(3)

```
let v = vec![4, 8, 19, 27, 34, 10];
let r = &v;
let aside = v; // move vector to aside
r[0]; // bad: uses `v`, which is now uninitialized
```

Vector 변수의 참조 예



참조하고 있는 중에, 다른 변수로 이동시키면,
Dangling Reference 문제가 발생한다.

3. 소유권(6) – 빌림 규칙(1)

1) 특정 시점에 1개의 mutable reference(&mut T)와 여러 개의 immutable reference(&T)를 동시에 사용해서는 안된다.
즉, 한 놈이 수정하고 있는데, 다른 녀석들이 그 값을 읽어가는 상황이 벌어져서는 안된다.
이는 data races, dangling pointers, unsafe memory access 등을 (compile 시점에) 찾아내기 위해 반드시 필요하다.

```rust
fn main() {
    let mut note = String::from("Rust is fast");

    let r1 = &note; // Immutable borrow
    let r2 = &note; // Immutable borrow
    // let r3 = &mut note; // Error: cannot borrow as mutable
because it is also borrowed as immutable

    println!("Note: {}, {}", r1, r2);

    let r3 = &mut note; // Mutable borrow is allowed here because no
immutable borrows are active
    r3.push_str(" and safe");
    println!("Updated note: {}", r3);
}
```

3. 소유권(6) – 빌림 규칙(2)

2) 모든 reference는 유효(valid)해야 한다. 즉, 모든 reference는 할당 해제된 resource나 scope를 벗어난 resource를 참조하면 안된다.
이는 dangling pointer or reference issue를 막기 위해 반드시 필요하다.

```rust
// This function attempts to return a reference to data that will be deallocated.
// Rust's compiler will prevent this with a lifetime error.
fn get_dangling_reference() -> &String {
    let s = String::from("hello");

    &s // We are trying to return a reference to `s`.
} // But `s` goes out of scope and is dropped here, so the memory is freed.
  // The returned reference would be "dangling" - pointing to invalid memory.

fn main() {
    // let reference_to_nothing = get_dangling_reference();
    // If the code above were allowed to compile, `reference_to_nothing`
    // would be a dangling reference, and using it would be undefined behavior.
}
```

3. 소유권(7) – Lifetime(1)

Reference Lifetime의 정의

- Rust의 라이프타임(Lifetime)은 참조자(&T)가 유효한 범위를 의미하며, 댕글링 참조(dangling reference)를 방지하여 메모리 안전성을 보장하는 핵심 개념이다.
- 컴파일러가 유효성을 분석하며, 대부분 암묵적으로 추론되지만, 복잡한 경우 'a와 같은 제네릭 라이프타임 파라미터를 사용해 명시해야 한다.

댕글링 참조 방지: 소유권이 해제된 데이터에 대한 참조를 컴파일 시점에 막아 에러를 발생시킨다.

컴파일 타임 검증: 라이프타임 체크는 런타임 성능에 영향을 주지 않으며, 모든 참조자는 컴파일러가 수명을 분석한다.

Lifetime Annotation: 함수의 입력 참조자와 반환 참조자의 라이프타임이 연관될 때,
fn longest<'a>(x: &'a str, y: &'a str) -> &'a str와 같이 어퍼스트로피(')를 사용하여 명시적으로 표시한다.

3. 소유권(7) – Lifetime(2)

Dangling reference issue

```
fn main() {
    let reference_to_nothing;
    {
        let x = 5;
        // Error: 'x' only lives inside this block
        reference_to_nothing = &x;
    } // 'x' is dropped here
    // This would crash if Rust allowed it!
    println!("r: {}", reference_to_nothing);
}
```

Lifetime은 Reference의 유효기간을 지정하는 기능

Lifetime의 정의

```
// src/main.rs
fn main() {
    let r;                // ---------+-- 'a (Outer scope: Long duration)
    {                     //          |
        let x = 5;        // -+-- 'b  | (Inner scope: Short duration)
        r = &x;           //  |       |
    }                     // -+       | <--- 'b ends here (x is dropped)

    println!("r: {}", r); //          |
}                         // ---------+ <--- 'a ends here

// Error: The reference 'r' has lifetime 'a, but refers to 'x' which has
lifetime 'b.
// Because the scope 'b is shorter (ends earlier) than 'a, 'r' is left
pointing to invalid memory.
```

```
&i32        // a reference
&'a i32     // a reference with an explicit lifetime 'a
&'a mut i32 // a mutable reference with an explicit lifetime 'a
```

3. 소유권(7) – Lifetime(3)

<조건>
- 2개의 입력 reference
- 1개의 출력 reference

<대응>
- 동일한 'a lifetime annotation 입력
  - 2개의 입력 reference의 intersection(교집합) 즉, 더 작은 lifetime으로 지정함을 의미

```rust
// We declare generic lifetime 'a.
// We tell the compiler: "x, y, and the return value must all live at least as long as 'a".
// Practically, 'a becomes the intersection (the smaller) of the lifetimes of x and y.
fn longest<'a>(x: &'a str, y: &'a str) -> &'a str {
    if x.len() > y.len() {
        x
    } else {
        y
    }
}

fn main() {
    let string1 = String::from("long string");
    let string2 = "xyz";

    // We pass references to both strings.
    // The compiler sees that both live until the end of main, so the result is valid.
    let result = longest(string1.as_str(), string2);
    println!("The longest string is {}", result);

    // DEMO: Nested scopes
    {
        let string3 = String::from("tiny");
        // Here, 'result2' is valid because 'string3' is still alive inside this block.
        let result2 = longest(string1.as_str(), string3.as_str());
        println!("The longest inner string is {}", result2);
    }
}
```

3. 소유권(7) – Lifetime(4)

구조체와 Lifetime

```rust
// src/main.rs
// This struct holds a reference, so it needs a lifetime 'a
struct ImportantExcerpt<'a> {
    part: &'a str,
}

fn main() {
    let novel = String::from("Call me Ishmael. Some years ago...");

    // The instance 'first_sentence' cannot outlive 'novel'
    let first_sentence = ImportantExcerpt {
        part: &novel[0..15],
    };

    // If 'novel' were dropped here, 'first_sentence' would become invalid.
}
```

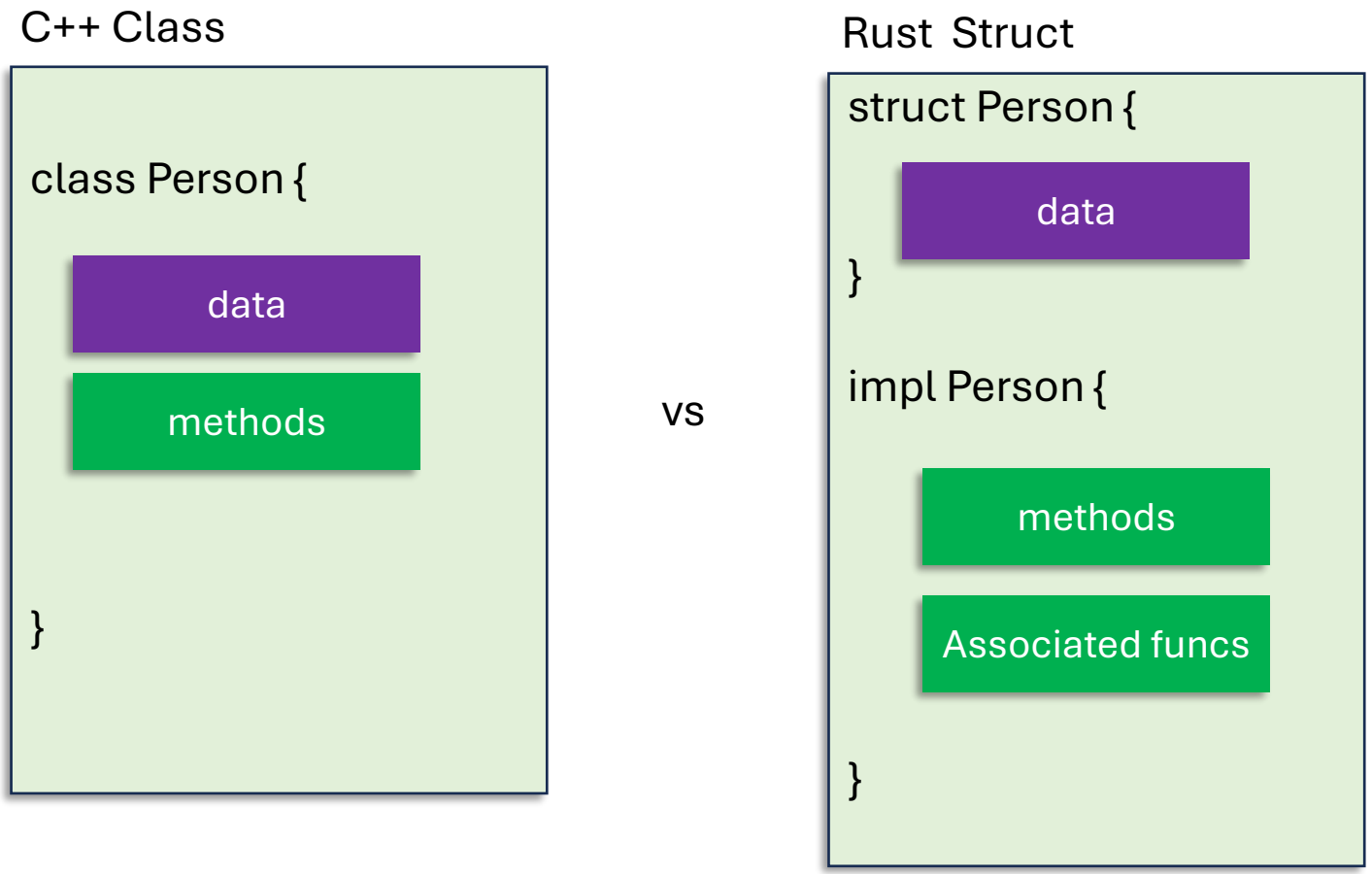
Struct내에 reference field가 있는 경우, Lifetime을 표시해 주어야 함.

```rust
struct Highlight<'a> {
    part: &'a str,
}

// 1. impl<'a>: We DECLARE the lifetime parameter here so the compiler knows it exists.
// 2. Highlight<'a>: We USE it here to select the specific struct type.
impl<'a> Highlight<'a> {
    fn announce(&self) {
        println!("Attention to: {}", self.part);
    }
}
```

4. Composite Type과 Collections

구조체(struct), 열거형(enum), 튜플
and
Vector, HashMap

4. Composite Type과 Collections(1) - 구조체 Struct(1)



4. Composite Type과 Collections(1) – 구조체 Struct(2)

struct 변수 사용 예

```rust
struct User {
    username: String,
    email: String,
    sign_in_count: u64,
    active: bool,
}

fn main() {
    let user1 = User {
        username: String::from("someusername123"),
        email: String::from("someone@example.com"),
        sign_in_count: 1,
        active: true,
    };

    println!("Username: {}", user1.username);
    println!("Email: {}", user1.email);
    println!("Sign in count: {}", user1.sign_in_count);
    println!("Active: {}", user1.active);
}
```

Tuple struct 변수 사용 예

```rust
struct Color(i32, i32, i32);
fn main() {
    let black = Color(0, 0, 0);

    println!("Black: ({}, {}, {})", black.0, black.1, black.2);
}
```

Unit struct 변수 사용 예

```rust
struct AlwaysEqual;

fn main() {
    let _subject = AlwaysEqual;
}
```

4. Composite Type과 Collections(1) – 구조체 Struct(3)

```rust
struct Rectangle {
    width: u32,
    height: u32,
}
```

struct와 method 구현 예

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

    fn can_hold(&self, other: &Rectangle) -> bool {
        self.width > other.width && self.height > other.height
    }
}
```

```rust
fn main() {
    let rect1 = Rectangle {
        width: 30,
        height: 50,
    };

    let rect2 = Rectangle {
        width: 10,
        height: 40,
    };

    let rect3 = Rectangle {
        width: 60,
        height: 45,
    };

    println!("The area of rect1 is {} square pixels.", rect1.area());
    println!("Can rect1 hold rect2? {}", rect1.can_hold(&rect2));
    println!("Can rect1 hold rect3? {}", rect1.can_hold(&rect3));
}
```

4. Composite Type과 Collections(1) – 구조체 Struct(4)

self가 없음.

struct와 연관 함수(associated function) 구현 예

```
impl Rectangle {
    fn square(size: u32) -> Rectangle {
        Rectangle {
            width: size,
            height: size,
        }
    }
}

fn main() {
    let sq = Rectangle::square(3);
    println!("The area of the square is {} square pixels.", sq.area());
}
```

4. Composite Type과 Collections(1) – 구조체 Struct(5)

Struct로 Tuple을 구현한 예

```
// Define tuple structs
struct Color(u8, u8, u8); // RGB
struct Point(i32, i32);   // 2D coordinates

fn main() {
    // Instantiate like tuples, but with the type name
    let red = Color(255, 0, 0);
    let origin = Point(0, 0);

    // Access fields using dot notation and index
    println!("Red's green component: {}", red.1); // Accesses the second
field (index 1)

    // They define distinct types
    // let point_tuple: (i32, i32) = origin; // Error: mismatched types
Point != (i32, i32)

    // Can be destructured
    let Point(x, y) = origin;
    println!("Origin coordinates: x={}, y={}", x, y);
}
```

Tuple type을 사용한 예

```
fn main() {
    let tuple: (i32, f64, bool) = (42, 6.7, true);

    let int_value = tuple.0;
    let float_value = tuple.1;
    let bool_value = tuple.2;

    println!("Integer value: {}", int_value);
    println!("Float value: {}", float_value);
    println!("Boolean value: {}", bool_value);
}
```

4. Composite Type과 Collections(1) – 구조체 Struct(6)

구조체의 소유권 이동 구현 예

```
struct User {
    username: String,
    email: String,
    sign_in_count: u64,
    active: bool,
}

fn main() {
    let user1 = User {
        username: String::from("someusername123"),
        email: String::from("someone@example.com"),
        sign_in_count: 1,
        active: true,
    };

    let user2 = user1; // Moves ownership of user1 to user2

    // println!("Username: {}", user1.username); // This would cause a
compile-time error
    println!("Username: {}", user2.username); // Accessing user2 is fine
}
```

소유권 이동(Move) 발생

4. Composite Type과 Collections(1) – 구조체 Struct(7)

구조체의 소유권 빌림 구현 예

```
fn main() {
    let user1 = User {
        username: String::from("someusername123"),
        email: String::from("someone@example.com"),
        sign_in_count: 1,
        active: true,
    };

    let user_ref = &user1; // Borrowing user1
    println!("Username: {}", user_ref.username); // Reading borrowed data
    // user1 can still be used because user_ref is just a reference
    println!("Email: {}", user1.email);
}
```

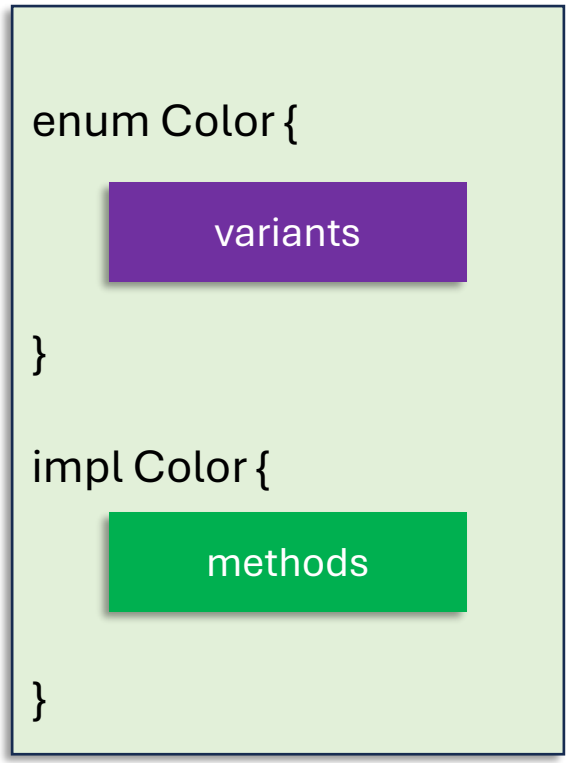
4. Composite Type과 Collections(1) – 구조체 Struct(8)

```
#[derive(Debug)]
struct User {
    username: String,
    email: String,
    sign_in_count: u64,
    active: bool,
}

fn main() {
    let user1 = User {
        username: String::from("someusername123"),
        email: String::from("someone@example.com"),
        sign_in_count: 1,
        active: true,
    };
    println!("{:?}", user1);
}
```

구조체의 Debug 트레잇 사용 예

4. Composite Type과 Collections(2) – 열거형 Enum(1)

Rust Enum



enum 변수 사용 예

```rust
#[derive(Debug)]
enum Color {
    // A variant that holds a tuple of three 8-bit unsigned integers
    Rgb(u8, u8, u8),
    // A variant that holds a single String
    Named(String),
}

fn main() {
    let red = Color::Rgb(255, 0, 0);
    let custom_color = Color::Named(String::from("Forest Green"));

    println!("An RGB color: {:?}", red);
    println!("A named color: {:?}", custom_color);
}
```

Rust의 enum의 각 variant는 type을 가질 수 있음.

4. Composite Type과 Collections(2) – 열거형 Enum(2)

enum method 구현 예

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

impl Message {
    fn call(&self) {
        match self {
            Message::Quit => println!("Quit message"),
            Message::Move { x, y } => println!("Move to x: {}, y: {}", x,
y),
            Message::Write(text) => println!("Write message: {}", text),
            Message::ChangeColor(r, g, b) => println!("Change color to
red: {}, green: {}, blue: {}", r, g, b),
        }
    }
}
```

4. Composite Type과 Collections(3) – Vector(1)

```rust
fn main() {
    let mut i32_vec = Vec::<i32>::new();  // 자료형 명시

    i32_vec.push(1); // 자료형 추가
    i32_vec.push(2);
    i32_vec.push(3);

    println!("{:?}", i32_vec);

    let mut float_vec = Vec::new(); // 자료형 추론

    float_vec.push(1.3);
    float_vec.push(2.3);
    float_vec.push(3.4);

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

    float_vec.pop();    // 마지막 요소 제거

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

    let string_vec = vec\![String::from("Hello"), String::from("World")\];

    for word in string_vec.iter() { // 반복자 사용
        println!("{}", word);
    }
}
```

Vec<T>

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

    for n in &v {
        println!("{}", n);
    }

    let mut v = vec![1,2,3];

    for n in &mut v {
        *n = *n + 1;
        println!("{}", n);
    }
}
```

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

    for (index, number) in numbers.iter().enumerate() {
        println!("Index: {}, Value: {}", index, number);
    }
}
```

4. Composite Type과 Collections(3) – Vector(2)

```rust
fn main() {
    // 3x3 2차원 벡터 생성
    let rows = 3;
    let cols = 3;
    let mut matrix: Vec<Vec<i32>> = vec![vec![0; cols]; rows];

    // 값 할당
    for i in 0..rows {
        for j in 0..cols {
            matrix[i][j] = i as i32 + j as i32; // 간단한 값 할당
        }
    }

    // 출력
    for row in &matrix {
        for value in row {
            print!("{} ", value);
        }
        println!();
    }
}
```

```rust
fn main() {
    // 2x2x2 크기의 3차원 벡터 생성
    let x_size = 2;
    let y_size = 2;
    let z_size = 2;
    let mut cube: Vec<Vec<Vec<i32>>> = vec![vec![vec![0; z_size]; y_size]; x_size];

    // 값 할당
    for x in 0..x_size {
        for y in 0..y_size {
            for z in 0..z_size {
                cube[x][y][z] = x as i32 + y as i32 + z as i32; // 간단한 값 할당
            }
        }
    }

    // 출력
    for x in &cube {
        for y in x {
            for value in y {
                print!("{} ", value);
            }
            println!();
        }

        println!("---"); // 층 구분
    }
}
```

4. Composite Type과 Collections(4) – HashMap

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

fn main() {
    let mut scores = HashMap::new();
    let name = String::from("Alice");

    scores.insert(name, 10); // name의 소유권이 HashMap으로 이동

    //println!("{}", name); // ❌오류! name의 소유권이 이동되었음

    // 키가 없으면 기본값 삽입
    scores.insert(String::from("Bob"), 30);

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

HashMap<T>

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

fn main() {
    let mut scores = HashMap::new();

    scores.insert("Alice", 10);
    scores.insert("Bob", 20);
    scores.insert("Charlie", 30);

    // HashMap 순회
    for (key, value) in &scores {
        println!("{}: {}", key, value);
    }
}
```

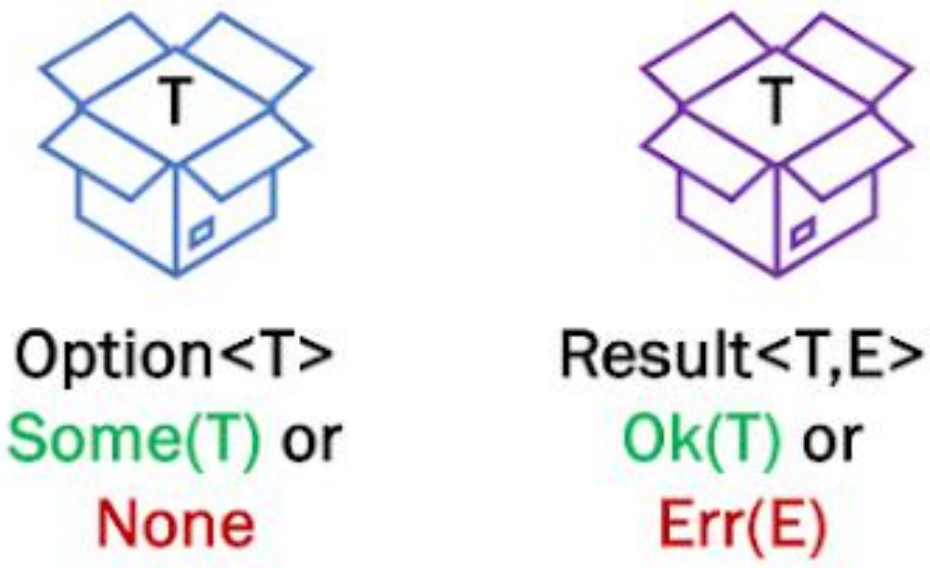
```rust
use std::collections::HashMap;

fn main() {
    let mut scores = HashMap::new();

    scores.insert("Alice", 10);

    // 기존 키가 존재하면 값을 변경하지 않음
    scores.entry("Alice").or_insert(20); // 기존 값 유지
    scores.entry("Bob").or_insert(30);   // 새로운 값 추가

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

    // 키가 없을 때 기본값을 삽입하고, 값을 변경
    let charlie = scores.entry("Charlie").or_insert(30);

    *charlie += 10; // Charlie의 값 증가
    println!("{:?}", scores);

    // 값을 계산해서 넣는다.
    scores.entry("Dave").or_insert_with(|| {
        let computed_value = 50;
        computed_value
    });

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

5. Error Handling

열거형(Enum) Option<T> and Result<T, E>

5. Error Handling(1) – Option과 Result 열거형(Enum)

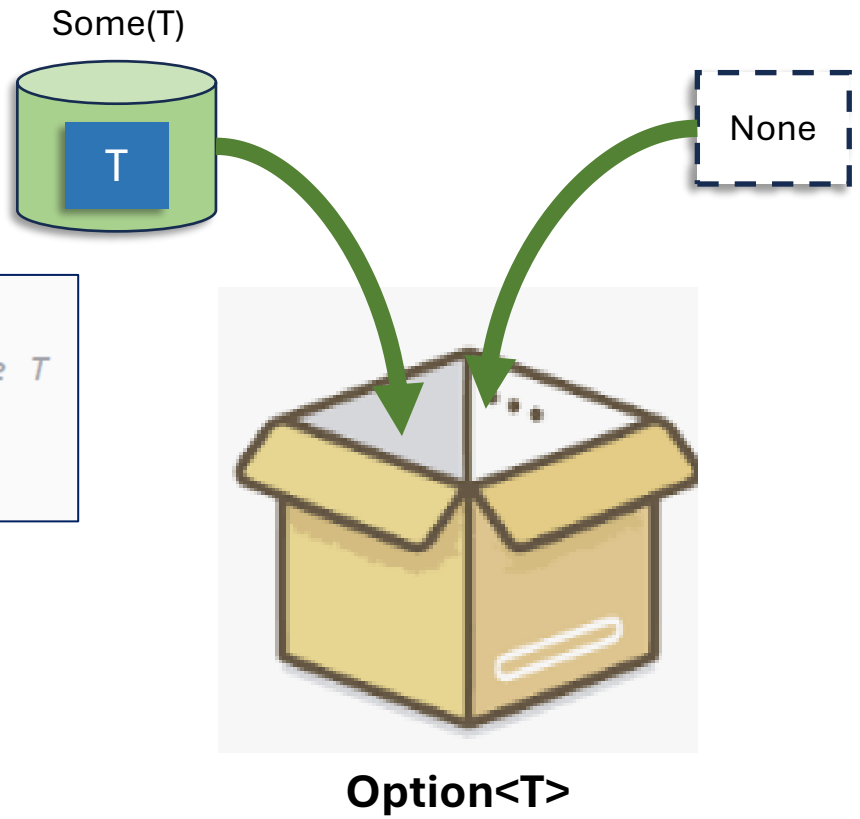
- Option<T>는 함수 실행 결과, 원하는 값이 없을 수도 있는 경우에 주로 사용함.
- 이 경우, 값이 존재하는 않는 이유는 중요하지 않을 경우에 주로 사용함.



- Result<T, E>는 함수 실행 결과, 성공과 실패로 나뉘는 경우에 주로 사용함.
- 또한, 함수 호출자에서 실패의 이유를 알리고자 할 경우에 주로 사용함.

5. Error Handling(2) – Option<T> (1)

```
enum Option<T> {
    Some(T), // Value is present: contains a value of type T
    None,    // Value is absent
}
```



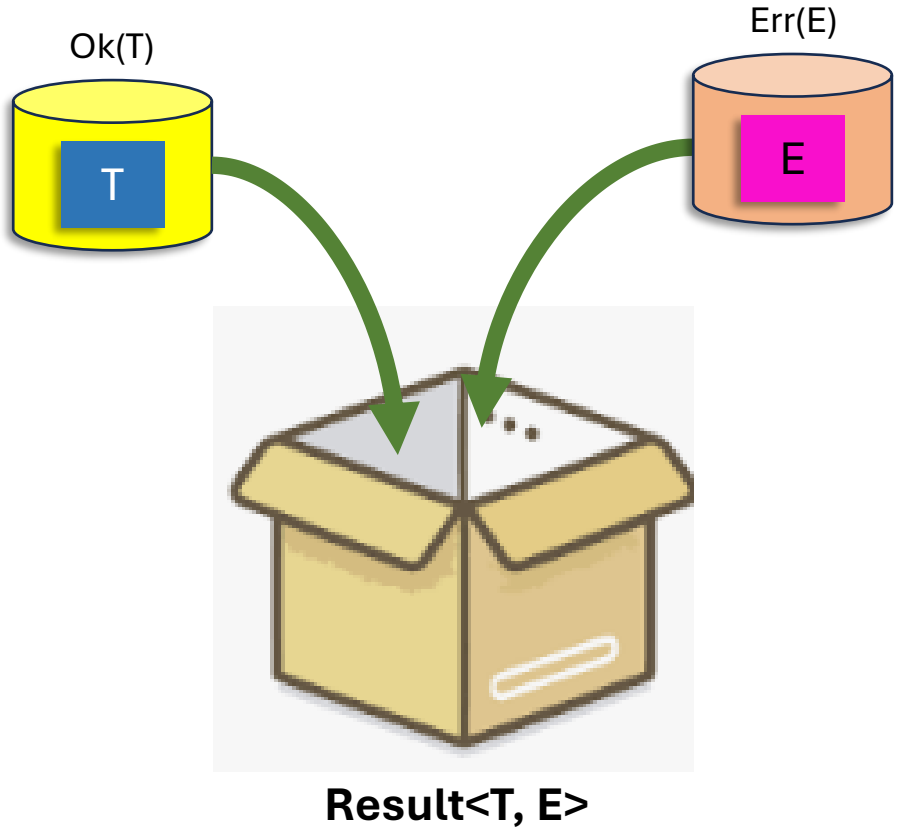
T는 제네릭(Generics) 타입을 의미함.

5. Error Handling(2) – Option<T> (2)

```rust
// This function searches for a word in a sentence.
// It returns Option<usize>: Some(index) if the word is found, or None otherwise.
fn find_word_index(haystack: &str, needle: &str) -> Option<usize> {
    // The built-in .find() method on string slices already returns Option<usize>,
    // which is very convenient.
    haystack.find(needle)
}

fn main() {
    let famous_phrase = "The quick brown fox jumps over the lazy dog.";
    let word_to_find = "fox";
    let word_not_present = "cat";

    // --- Test Case 1: Word is found ---
    println!("Searching for '{}'...", word_to_find);
    match find_word_index(famous_phrase, word_to_find) {
        Some(index) => println!("Success! Found at index {}.", index),
        None => println!("Failure: Word not found."),
    }

    println!("---");

    // --- Test Case 2: Word is NOT found ---
    println!("Searching for '{}'...", word_not_present);
    match find_word_index(famous_phrase, word_not_present) {
        Some(index) => println!("Success! Found at index {}.", index),
        None => println!("Correctly determined that the word was not found."),
    }
}
```

5. Error Handling(3) – Result<T, E> (1)

```rust
enum Result<T, E> {
    Ok(T),  // Success: contains a value of type T
    Err(E), // Failure: contains an error value of type E
}
```



T와 E는 제네릭(Generics) 타입을 의미함.

5. Error Handling(3) – Result<T, E> (2)

```rust
// Function returning a Result: Ok with an f64 result, or Err with a
String error message.
fn divide(numerator: f64, denominator: f64) -> Result<f64, String> {
    if denominator == 0.0 {
        // If denominator is zero, return an error (Err)
        Err(String::from("Error: division by zero!"))
    } else {
        // Otherwise, return the result successfully (Ok)
        Ok(numerator / denominator)
    }
}

fn main() {
    let result1 = divide(10.0, 2.0); // Successful call
    let result2 = divide(5.0, 0.0);  // Failing call

    // Use 'match' to handle both cases of the Result
    match result1 {
        Ok(value) => println!("Division 10/2 succeeded: {}", value), //
Output: 5.0
        Err(message) => println!("Error in division 10/2: {}", message),
    }
```

5. Error Handling(4) – Unwrapping(1)

```rust
// A simple function that returns a Result, for context.
fn divide(a: f64, b: f64) -> Result<f64, String> {
    if b == 0.0 {
        Err("Division by zero".to_string())
    } else {
        Ok(a / b)
    }
}
```

.unwrap( ) method 사용 예

```rust
fn main() {
    // --- Success Case ---
    let ok_result = divide(10.0, 5.0);

    // Print the Result before unwrapping to show its Ok(value)
state.
    println!("The Result before unwrap: {:?}", ok_result); //
Prints: Ok(2.0)

    // .unwrap() extracts the value from the Ok variant.
    let value = ok_result.unwrap();
    println!("The value after unwrap: {}", value); // Prints: 2.0

    // --- Panic Case ---
    // let err_result = divide(10.0, 0.0); // This would be
Err("Division by zero")
    // println!("The Result before panic: {:?}", err_result);
    //
    // // Calling .unwrap() on an Err variant will cause the program
to panic.
    // let value_panic = err_result.unwrap(); // This line would
PANIC!
}
```

.unwrap( )은 실패(Err)시, panic이 발생한다.

5. Error Handling(4) – Unwrapping(2)

.expect( ) method 사용 예

```
let non_existent_file: Result<String, std::io::Error> =
std::fs::read_to_string("file_no_exist.txt");
// This will panic with the specified message
// non_existent_file.expect("Expected the file to definitely
exist!");
```

.expect( )는 실패(Err)시, 파라미터 값을 출력하면서, panic이 발생한다.

.unwrap_or( ) method 사용 예

```
let err_result = divide(10.0, 0.0); // This is Err(...)
let value_or_default = err_result.unwrap_or(0.0); // Doesn't panic,
returns 0.0
println!("Value or default: {}", value_or_default);

let none_option: Option<i32> = None;
let option_value = none_option.unwrap_or(100); // Returns 100
println!("Option or default: {}", option_value);
```

.unwrap_or( )는 실패(Err)시, 파라미터 전달한다(panic은 발생하지 않음).

5. Error Handling(4) – Unwrapping(3)

.unwrap_or_else( ) method 사용 예

```
let err_result = divide(10.0, 0.0); // Err(...)
let value_or_computed = err_result.unwrap_or_else(|err_msg| {
    println!("Error during division: {}. Using fallback value.",
err_msg);
    -1.0 // Value computed/returned by the closure
});
println!("Value or computed: {}", value_or_computed);
```

.unwrap_or_else( )는 실패(Err)시, 파라미터로 넘긴 closur를 실행한다.

5. Error Handling(4) – Unwrapping(4-1)

? Operator 사용 예(1)

```rust
use std::fs;
use std::io;
use std::num::ParseFloatError;

// 1. Define a custom error type for our function
// Suppress the spurious 'dead_code' warning related to the derived Debug implementation
#[allow(dead_code)]
#[derive(Debug)]
enum ReadDivideError {
    Io(io::Error),
    Format(ParseFloatError),
    Math(String),
}

// 2. Implement 'From' trait to allow '?' to automatically convert io::Error
impl From<io::Error> for ReadDivideError {
    fn from(err: io::Error) -> ReadDivideError {
        ReadDivideError::Io(err)
    }
}

// 3. Implement 'From' trait to allow '?' to automatically convert ParseFloatError
impl From<ParseFloatError> for ReadDivideError {
    fn from(err: ParseFloatError) -> ReadDivideError {
        ReadDivideError::Format(err)
    }
}

// 4. Implement 'From' trait to allow '?' to automatically convert String (from divide)
impl From<String> for ReadDivideError {
    fn from(err: String) -> ReadDivideError {
        ReadDivideError::Math(err)
    }
}
```

```rust
// 5. Our original 'divide' function
fn divide(numerator: f64, denominator: f64) -> Result<f64, String> {
    if denominator == 0.0 {
        Err(String::from("Division by zero!"))
    } else {
        Ok(numerator / denominator)
    }
}

// Function using '?'
fn read_then_divide_with_qmark(file_path: &str, divisor: f64) -> Result<f64, ReadDivideError> {
    // The '?' operator automatically converts and propagates errors
    let content = fs::read_to_string(file_path)?;
    let number = content.trim().parse::<f64>()?;
    let result = divide(number, divisor)?;

    Ok(result)
}

fn main() {
    // --- Setup and Tests ---
    let filename = "number.txt";
    fs::write(filename, "100.5").expect("Cannot write file");

    println!("--- 1. Success Case ---");
    match read_then_divide_with_qmark(filename, 2.0) {
        Ok(r) => println!("Result with '?': {}", r),
        Err(e) => println!("Error with '?': {:?}", e),
    }

    println!("\n--- 2. Division by Zero Error ---");
    match read_then_divide_with_qmark(filename, 0.0) {
        Ok(_) => (),
        Err(e) => println!("Error with '?': {:?}", e),
    }

    // Clean up dummy file
    fs::remove_file(filename).ok();
}
```

5. Error Handling(4) – Unwrapping(4-2)

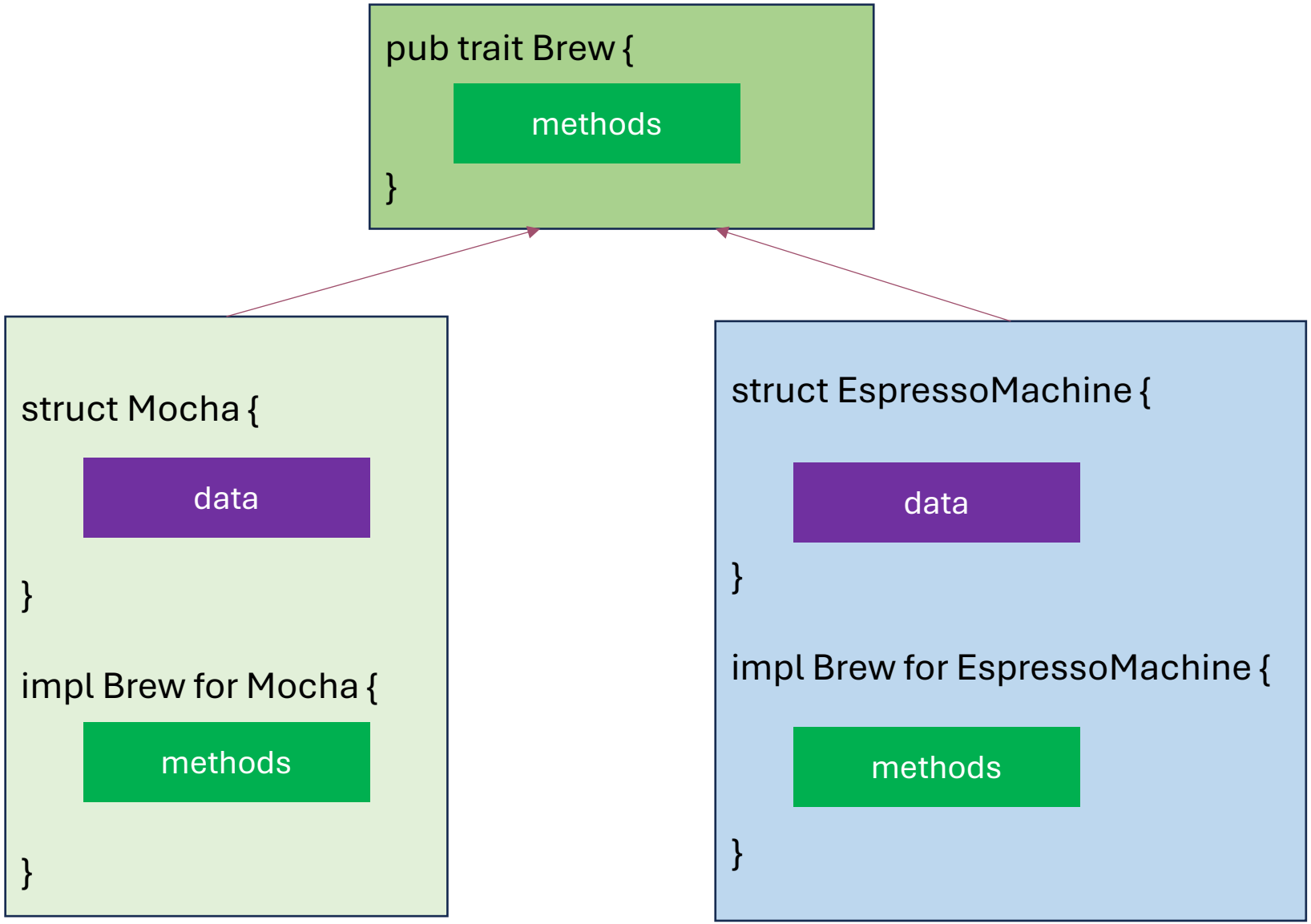
? Operator 사용 예(2)

```rust
// Function using '?' - Note it now returns our custom error type
fn read_then_divide_with_qmark(file_path: &str, divisor: f64) ->
Result<f64, ReadDivideError> {
    // '?' handles io::Error, converting it via From into
ReadDivideError::Io
    let content = fs::read_to_string(file_path)?;

    // '?' handles ParseFloatError, converting it via From into
ReadDivideError::Format
    let number = content.trim().parse::<f64>()?;

    // '?' handles the String error from divide, converting via From into
ReadDivideError::Math
    let result = divide(number, divisor)?;

    Ok(result) // If all OK, return Ok(...)
}
```

6. 다형성(Polymorphism)과 OOP

구조체(Struct), 트레잇(Trait) and 제네릭스(Generics)

6. 다형성과 OOP(1) – Trait(1)



6. 다형성과 OOP(1) – Trait(2)

```rust
// src/lib.rs
pub trait Brew {
    fn extract(&self) -> String;

    // This is a default implementation.
    // If a type doesn't define this method, it gets this version automatically.
    fn clean(&self) -> String {
        String::from("Cleaning with a simple hot water rinse.")
    }
}
```

```rust
pub struct Moka;
pub struct EspressoMachine;

impl Brew for Moka {
    fn extract(&self) -> String {
        String::from("Bubbling up some coffee...")
    }
    // We don't implement clean() here.
    // Moka automatically uses the default "hot water rinse".
}

impl Brew for EspressoMachine {
    fn extract(&self) -> String {
        String::from("Pressurizing water...")
    }

    // We override the default because this machine is complex.
    fn clean(&self) -> String {
        String::from("Running automatic descaling program.")
    }
}
```

6. 다형성과 OOP(1) – Trait(3)

```rust
pub trait SimpleIterator {
    type Item;
    fn next(&mut self) -> Option<Self::Item>;
}

struct Counter {
    current: u32,
    max: u32,
}

impl SimpleIterator for Counter {
    type Item = u32;
    fn next(&mut self) -> Option<Self::Item> {
        if self.current < self.max {
            let val = self.current;
            self.current += 1;
            Some(val)
        } else {
            None
        }
    }
}

fn main() {
    let mut counter = Counter {
        current: 0,
        max: 3
    };
    for _i in 0..=3 {
        println!("{:?}", counter.next());
    }
}
```

Associated Type

6. 다형성과 OOP(2) – Debug and Display Trait

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

trait PrintableSummary: Display {
    fn print_summary(&self);
}

#[derive(Debug)]
struct Report {
    title: String,
    content: String,
}

impl Display for Report {
    fn fmt(&self, f: &mut std::fmt::Formatter<'_>) -> std::fmt::Result {
        write!(f, "Report: '{}'", self.title)
    }
}
```

```rust
impl PrintableSummary for Report {
    fn print_summary(&self) {
        println!("--- Summary ---");
        println!("###1 ----> {:?}", self);
        println!("###2 ----> {}", self);
        println!("Content length: {}", self.content.len());
        println!("-----------------");
    }
}

fn main() {
    let report = Report {
        title: "Q1 Results".into(),
        content: "Sales were strong...".into(),
    };
    report.print_summary();
}
```

6. 다형성과 OOP(3) – Trait Orphan Rule

```rust
// src/lib.rs
use std::fmt;

// --- ALLOWED SCENARIOS ---
// 1. Implementing a LOCAL trait (Brew) on an EXTERNAL type (String).
// This is allowed because we own the trait "Brew".
impl Brew for String {
    fn extract(&self) -> String {
        String::from("Pour-over coffee")
    }
}

// 2. Implementing an EXTERNAL trait (Display) on a LOCAL type (Moka).
// This is allowed because we own the type "Moka".
impl fmt::Display for Moka {
    fn fmt(&self, f: &mut fmt::Formatter) -> fmt::Result {
        write!(f, "Moka pot size {}", self.size)
    }
}
```

외부 type에 대해 사용자 정의 Trait를 적용하는 것은 허용

사용자 정의 type에 대해 외부 Trait를 적용하는 것은 허용

```rust
// --- FORBIDDEN SCENARIO (The Orphan Rule) ---
/*
// ⚠ ERROR: Implementing an EXTERNAL trait on an EXTERNAL type.
// You cannot implement Display for Vec<i32> because you own neither!
impl fmt::Display for Vec<i32> {
    fn fmt(&self, f: &mut fmt::Formatter) -> fmt::Result {
        write!(f, "Vector content...")
    }
}
*/

// --- THE SOLUTION (Newtype Pattern) ---
// We wrap the external type in a local struct.
pub struct MyIntList(pub Vec<i32>);

// Now MyIntList is LOCAL, so we can implement Display!
impl fmt::Display for MyIntList {
    fn fmt(&self, f: &mut fmt::Formatter) -> fmt::Result {
        // We access the wrapped Vec using .0
        write!(f, "List: {:?}", self.0)
    }
}
```

외부 type에 대해 외부 Trait를 적용하는 것은 금지!!

외부 type에 대해 외부 Trait를 적용하기 위한 Trick!!

6. 다형성과 OOP(4) – 제네릭스(Generic)(1)

```rust
// We are forced to write two functions with identical logic
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 {
        if item > largest {
            largest = item;
        }
    }
    largest
}
```

```rust
// We read this as: "largest is a function generic over some type T"
fn largest<T>(list: &[T]) -> T {
    let mut largest = list[0];
    for &item in list {
        // NOTE: This specific line will cause a compiler error right now!
        // We will explain why and fix it in the next section.
        if item > largest {
            largest = item;
        }
    }
    largest
}
```

6. 다형성과 OOP(4) – 제네릭스(Generic)(2)

```
pub struct Queue<T> {
    older: Vec<T>,
    younger: Vec<T>
}

impl<T> Queue<T> {
    pub fn new() -> Queue<T> {
        Queue { older: Vec::new(), younger: Vec::new() }
    }

    pub fn push(&mut self, t: T) {
        self.younger.push(t);
    }

    pub fn is_empty(&self) -> bool {
        self.older.is_empty() && self.younger.is_empty()
    }

    ...
}
```

6. 다형성과 OOP(5) – Trait Bounds(1)

```rust
// src/main.rs
// We restrict T: It must be comparable (PartialOrd) and copyable (Copy)
// We return Option<T> to handle the case where the list is empty.
fn largest<T: PartialOrd + Copy>(list: &[T]) -> Option<T> {
    if list.is_empty() {
        return None;
    }
    let mut largest = list[0];

    for &item in list {
        if item > largest {
            largest = item;
        }
    }
    Some(largest)
}
```

PartialOrd 트레잇 사용

Copy 트레잇 사용

6. 다형성과 OOP(5) – Trait Bounds(2)

```rust
use std::fmt::{Debug, Display};

// Hard to read: The bounds clutter the function name
fn compare_prints<T: Display + Clone, U: Clone + Debug>(t: &T, u: &U) { }

// Easier to read: The bounds are moved to the 'where' clause
fn compare_prints<T, U>(t: &T, u: &U)
where
    T: Display + Clone,
    U: Clone + Debug,
{
    // Function body...
}
```

6. 다형성과 OOP(6) – 정적 Binding: Impl Trait

```rust
trait TreasureBox {
    fn open(&self, key_no: i32) -> bool;
    fn check(&self);
}

struct JewelryBox {
    price: i32,
    key_no: i32,
}

struct TrapBox {
    damage: i32,
}

impl TreasureBox for JewelryBox {
    fn open(&self, key_no: i32) -> bool {
        self.key_no == key_no
    }

    fn check(&self) {
        println!("보석 상자였다. {} 골드 입수.", self.price);
    }
}

impl TreasureBox for TrapBox {
    fn open(&self, _key_no: i32) -> bool {
        return true;
    }

    fn check(&self) {
        println!("함정이었다. HP가 {} 감소했다.", self.damage);
    }
}
```

```rust
fn open_box(tbox: &impl TreasureBox, key_no: i32) {
    if !tbox.open(key_no) {
        println!("열쇠가 맞지 않아 상자가 열리지 않는다.");
        return;
    }
    tbox.check();
}

fn main() {
    let box1 = JewelryBox {
        price: 30,
        key_no: 1,
    };

    let box2 = TrapBox { damage: 3 };

    let box3 = JewelryBox {
        price: 20,
        key_no: 2,
    };

    let my_key = 2;
    open_box(&box1, my_key);
    open_box(&box2, my_key);
    open_box(&box3, my_key);
}
```

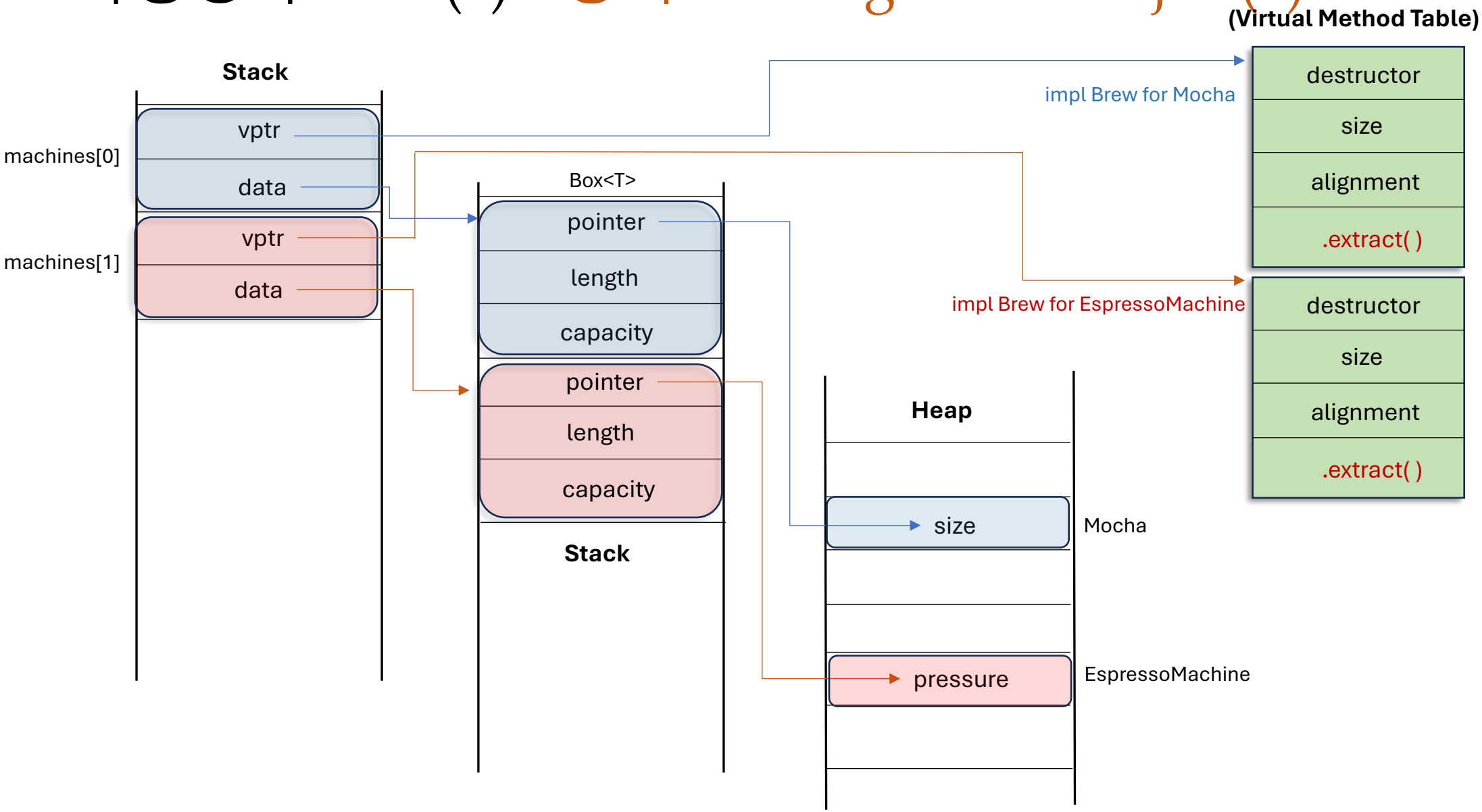
6. 다형성과 OOP(7) – 동적 Binding : Trait Object(1)

```rust
// 1. Setup: Let's define the types and implementations first.
// (Moka was defined previously, but we define EspressoMachine here).

pub struct Moka { pub size: u8 }

impl Brew for Moka {
    fn extract(&self) -> String {
        format!("Bubbling up {} cups.", self.size)
    }
}

pub struct EspressoMachine { pub pressure: u8 }

impl Brew for EspressoMachine {
    fn extract(&self) -> String {
        format!("Extracting at {} bars of pressure.", self.pressure)
    }
}
```

Dynamic Dispatch

```rust
// 2. The Main Event: Storing mixed types using Trait Objects.
fn main() {
    // We can store mixed types because they both satisfy 'dyn Brew'.
    // 'Box' puts the data on the heap, so the vector just stores pointers
    // of the same size.
    let machines: Vec<Box<dyn Brew>> = vec![
        Box::new(Moka { size: 6 }),
        Box::new(EspressoMachine { pressure: 9 }),
    ];

    for machine in machines {
        println!("Coffee shop says: {}", machine.extract());
    }
}
```

6. 다형성과 OOP(7) – 동적 Binding : Trait Object(2)



6. 다형성과 OOP(8) – 표준 Trait(1)

```rust
use std::borrow::Borrow;
use std::convert::TryFrom;
use std::ops::{Deref, DerefMut};
use std::rc::Rc;

// ========================================
// 1. Default
// 기본값을 정의하는 트레이트입니다.
// ========================================
#[derive(Debug)]
struct GameConfig {
    volume: i32,
    resolution: String,
}

impl Default for GameConfig {
    fn default() -> Self {
        Self {
            volume: 50,
            resolution: String::from("1920x1080"),
        }
    }
}

// ========================================
// 2. Clone & Copy
// Clone: 명시적인 복사 (.clone()) / 힙 메모리 데이터 복사 가능
// Copy: 암묵적인 비트 단위 복사 (stack-only), 소유권 이동이 일어나지 않음
// ========================================
#[derive(Debug, Clone, Copy)]
struct Point {
    x: i32,
    y: i32,
}

#[derive(Debug, Clone)] // String을 포함하므로 Copy 불가능
struct Person {
    name: String,
}
```

```rust
// 3. Drop
// 스코프를 벗어날 때 실행되는 소멸자 로직을 정의합니다.
// ========================================
struct CustomSmartPointer {
    data: String,
}

impl Drop for CustomSmartPointer {
    fn drop(&mut self) {
        println!(">>> [Drop] 메모리 해제됨: {}", self.data);
    }
}

// ========================================
// 4. Deref & DerefMut
// 스마트 포인터가 내부 데이터의 메서드에 접근할 수 있게 해줍니다 (* 연산자 오버로딩).
// ========================================
struct MyBox<T>(T);

impl<T> Deref for MyBox<T> {
    type Target = T;
    fn deref(&self) -> &Self::Target {
        &self.0
    }
}

impl<T> DerefMut for MyBox<T> {
    fn deref_mut(&mut self) -> &mut Self::Target {
        &mut self.0
    }
}
```

```rust
// ========================================
// 5. From & Into
// 타입 변환을 위한 트레이트입니다. From을 구현하면 Into는 자동으로 구현됩니다.
// ========================================
#[derive(Debug)]
struct NumberWrapper(i32);

impl From<i32> for NumberWrapper {
    fn from(item: i32) -> Self {
        NumberWrapper(item)
    }
}

// ========================================
// 6. TryFrom & TryInto
// 실패할 수 있는 타입 변환입니다. Result를 반환합니다.
// ========================================
#[derive(Debug)]
struct EvenNumber(i32);

impl TryFrom<i32> for EvenNumber {
    type Error = &'static str;

    fn try_from(value: i32) -> Result<Self, Self::Error> {
        if value % 2 == 0 {
            Ok(EvenNumber(value))
        } else {
            Err("짝수가 아닙니다!")
        }
    }
}
```

6. 다형성과 OOP(8) – 표준 Trait(2)

```rust
// ========================================
// 7. AsRef & AsMut
// 비용이 적게 드는 참조 변환(Reference-to-Reference conversion)을 수행합니다.
// ========================================
fn print_length<T: AsRef<str>>(s: T) {
    // T가 String이든 &str이든 상관없이 &str로 취급
    println!("길이: {}", s.as_ref().len());
}

// ========================================
// 8. Borrow & BorrowMut
// 데이터 구조(HashMap 등)에서 키를 조회할 때 유용합니다.
// 소유한 데이터(String)를 참조 형태(&str)로 빌릴 수 있게 해줍니다.
// Eq, Hash, Ord가 원본과 빌린 형태에서 동일하게 동작해야 한다는 계약이 있습니다.
// ========================================
fn check_key<K, Q>(key: &K, query: &Q)
where
    K: Borrow<Q>, // K 타입을 Q 타입으로 빌릴 수 있어야 함
    Q: PartialEq + ?Sized,
{
    if key.borrow() == query {
        println!("키가 일치합니다!");
    } else {
        println!("키가 다릅니다.");
    }
}
```

```rust
// ========================================
// 9. ToOwned
// Borrow된 데이터(예: &str)에서 소유권이 있는 데이터(예: String)를 생성합니다.
// Clone의 일반화된 형태입니다.
// ========================================
fn make_owned(s: &str) -> String {
    s.to_owned() // &str -> String 변환 (내부적으로 복사 발생)
}

// ========================================
// 10. Sized
// 컴파일 타임에 크기가 알려진 타입을 나타내는 마커 트레이트입니다.
// ?Sized는 크기가 알려지지 않을 수도 있음(DST)을 나타냅니다.
// ========================================
fn generic_sized<T: Sized>(t: T) {
    println!("이 타입은 컴파일 타임에 크기가 정해져 있습니다.");
    // std::mem::size_of_val(&t); // 가능
}

fn generic_maybe_unsized<T: ?Sized>(t: &T) {
    println!("이 타입은 크기가 동적일 수 있으므로 참조로만 다룹니다.");
}
```

6. 다형성과 OOP(8) – 표준 Trait(3)

```rust
fn main() {
    println!("=== Rust Utility Traits Demo ===\n");

    // 1. Default
    let conf = GameConfig::default();
    println!("[Default] 기본 설정: {:?}", conf);

    // 2. Clone & Copy
    let p1 = Point { x: 10, y: 20 };
    let p2 = p1; // Copy 발생 (p1 여전히 사용 가능)
    println!("[Copy] p1: {:?}, p2: {:?}", p1, p2);

    let person1 = Person { name: "Alice".into() };
    let person2 = person1.clone(); // Clone (깊은 복사)
    // let person3 = person1; // 이 줄을 활성화하면 person1은 소유권 이동으로 사용 불가
    println!("[Clone] {:?} 복제됨", person2);

    // 3. Drop
    {
        let _ptr = CustomSmartPointer { data: String::from("중요한 데이터") };
        println!("[Drop] 스코프 내부");
    } // 여기서 Drop 호출됨
    println!("[Drop] 스코프 외부");

    // 4. Deref & DerefMut
    let mut my_box = MyBox(String::from("Hello"));
    // MyBox에는 len()이 없지만, Deref를 통해 String의 len() 호출 가능
    println!("[Deref] 길이: {}", my_box.len());
    // DerefMut을 통해 내부 String 수정 가능
    my_box.push_str(" World");
    println!("[DerefMut] 내용: {}", *my_box);

    // 5. From & Into
    let num = NumberWrapper::from(100);
    let num2: NumberWrapper = 200.into();
    println!("[From/Into] {:?}, {:?}", num, num2);

    // 6. TryFrom & TryInto
    let even = EvenNumber::try_from(4);
    let odd: Result<EvenNumber, _> = 5.try_into();
    println!("[TryFrom] 짝수 성공: {:?}", even);
    println!("[TryInto] 홀수 실패: {:?}", odd);

    // 7. AsRef
    println!("[AsRef] String 전달:");
    print_length(String::from("Rust"));
    println!("[AsRef] &str 전달:");
    print_length("Programming");

    // 8. Borrow
    let owner = String::from("key_value");
    // String을 소유하고 있지만 &str로 비교 가능 (Borrow 트레이트 덕분)
    print!("[Borrow] ");
    check_key(&owner, "key_value");

    // 9. ToOwned
    let borrowed_str: &str = "im borrowed";
    let owned_string: String = make_owned(borrowed_str);
    println!("[ToOwned] 소유권 생성: {:?}", owned_string);

    // 10. Sized vs ?Sized
    let x = 10;
    generic_sized(x);

    let slice: &str = "Dynamic Size";
    generic_maybe_unsized(slice); // str은 Sized가 아니지만 ?Sized로 허용
}
```

Thank You