

Rust Programming Language(Part II)

- Functional Programming(Iterator & Closure)
- Smart Pointer
- Concurrent Programming
- Package, Crate, Module

Slowboot
[\(chunghan.yi@gmail.com\)](mailto:chunghan.yi@gmail.com)
Doc. Revision: 0.93

목차(Table of Contents)

Part I.

1. Rust 개발 환경 구축
2. Rust 기초 - 변수, 상수, 제어 흐름, 함수 등
3. 소유권(Ownership) - 이동(Move), 복사(Copy), 빌림(Borrowing)과 참조(Reference), 수명(Lifetime)
4. Composite Type과 Collections - 구조체, 열거형, 튜플, Vector, HashMap
5. Error Handling - Option과 Result
6. 다형성과 OOP - 구조체(Struct), 트레이트(Trait)과 제네릭스(Generics)

Part II.

7. Functional Programming - 반복자(Iterator)와 클로저(Closure)
8. 스마트 포인터
9. Concurrency - 쓰레드(Thread)와 채널(Channel)
10. Package, Crate, Module
11. File and Network I/O
12. 고급 기능 - unsafe, macro, FFI 등

소유권(Ownership)

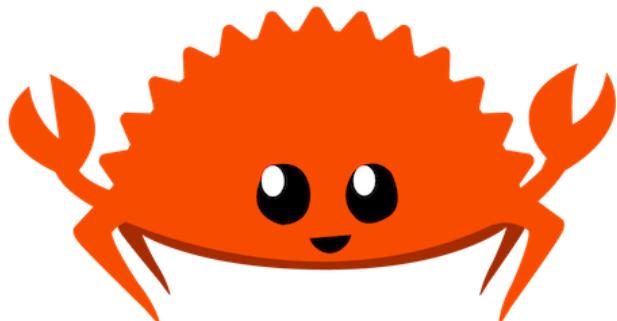
Error Handling
(Option<T>, Result<T, E>)

Part I

Struct + Generics + Trait =>
OOP

Iterator + Closure =>
Functional Programming

Rust의 6가지 주요 항목



Part II

Smart Pointer

Thread, Async
Concurrent Programming

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.

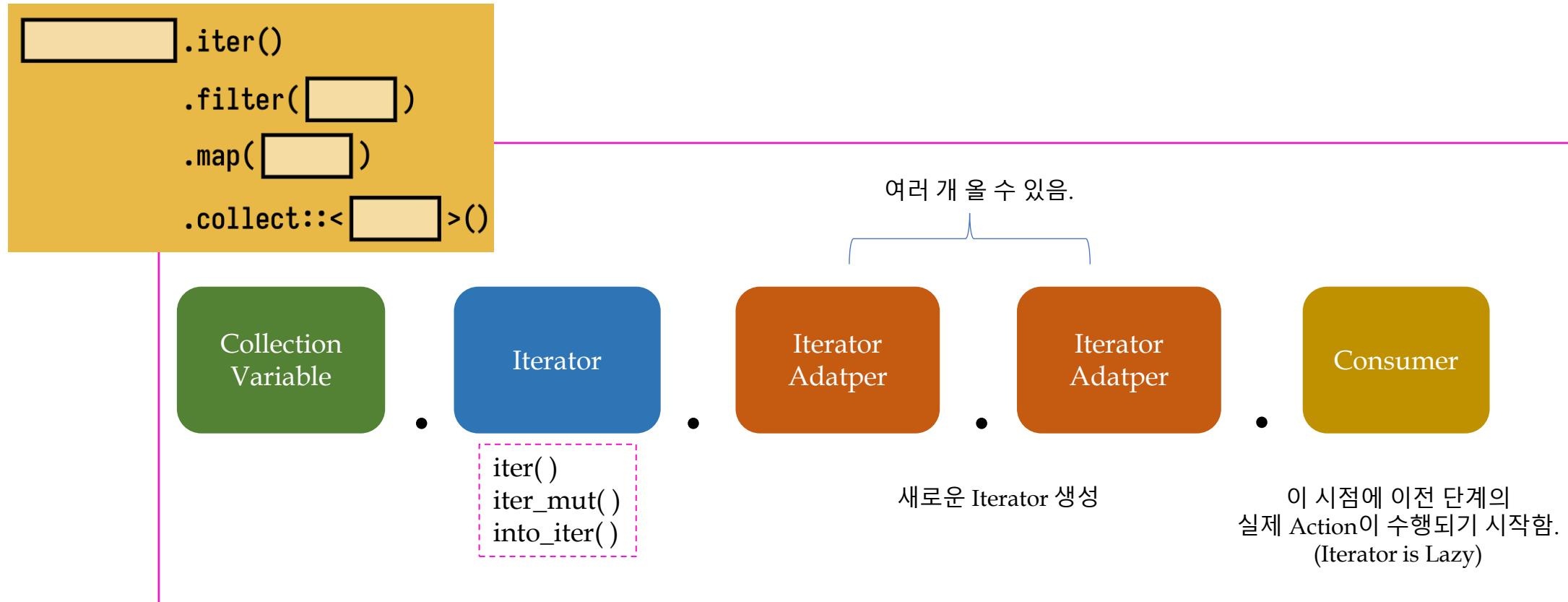


Let's make a Rust Router!

7. Functional Programming

반복자(Iterator)와 클로저(Closure)

7. Functional Programming(1) - Function Call Chain



```
let numbers = vec![1, 2, 3, 4, 5, 6, 7, 8, 9, 10];  
let evens: Vec<i32> = numbers.iter().filter(|&&x| x % 2 == 0).copied().collect();
```

7. Functional Programming(2) - Iterator(1)

<Iterator의 정의>

이터레이터(Iterator, 반복자)는 컬렉션(배열, 벡터 등)의 요소를 순차적으로 처리하는 std::iter::Iterator 트레이트(trait)을 구현한 객체이다.

지연 평가(Lazy Evaluation)

- iter(), iter_mut()나 into_iter()로 이터레이터를 생성해도 즉시 요소를 순회하지 않는다. collect(), sum() 같은 소비(consumer) 메서드가 호출될 때만 비로소 동작한다.

Iterator/Intoliterator Trait

- 핵심은 next() 메서드이다. 요소를 차례로 반환하고 끝에 도달하면 None을 반환하여 반복을 멈춘다.

반복자 어댑터(Iterator Adapter)

- map, filter 등으로, 또 다른 이터레이터를 반환하여 체이닝(chaining)을 가능하게 한다

소비자(Consumer)

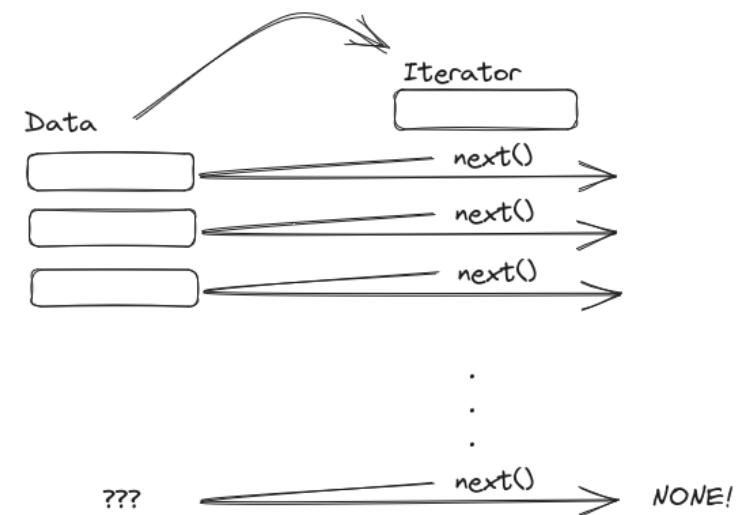
- collect(), sum(), fold() 등으로, 데이터를 처리하여 최종 결과를 반환한다(lazy 방식).

7. Functional Programming(2) - Iterator(2)

Iterator & IntoIterator 트레이

```
pub trait Iterator {  
    type Item;  
  
    // Required method  
    fn next(&mut self) -> Option<Self::Item>;  
  
    // Provided methods  
    fn next_chunk<const N: usize>(  
        &mut self,  
    ) -> Result<[Self::Item; N], IntoIter<Self::Item, N>>  
        where Self: Sized { ... }  
  
    fn size_hint(&self) -> (usize, Option<usize>) { ... }  
  
    fn count(self) -> usize  
        where Self: Sized { ... }  
  
    fn last(self) -> Option<Self::Item>  
        where Self: Sized { ... }  
}
```

```
pub trait IntoIterator {  
    type Item;  
    type IntoIter: Iterator<Item = Self::Item>;  
  
    // Required method  
    fn into_iter(self) -> Self::IntoIter;  
}
```



iter(), iter_mut() method는 Iterator Trait의 method가 아니라, Collection type의 method이다.

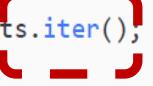
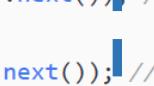
??? next() NONE!

7. Functional Programming(2) - Iterator(3)

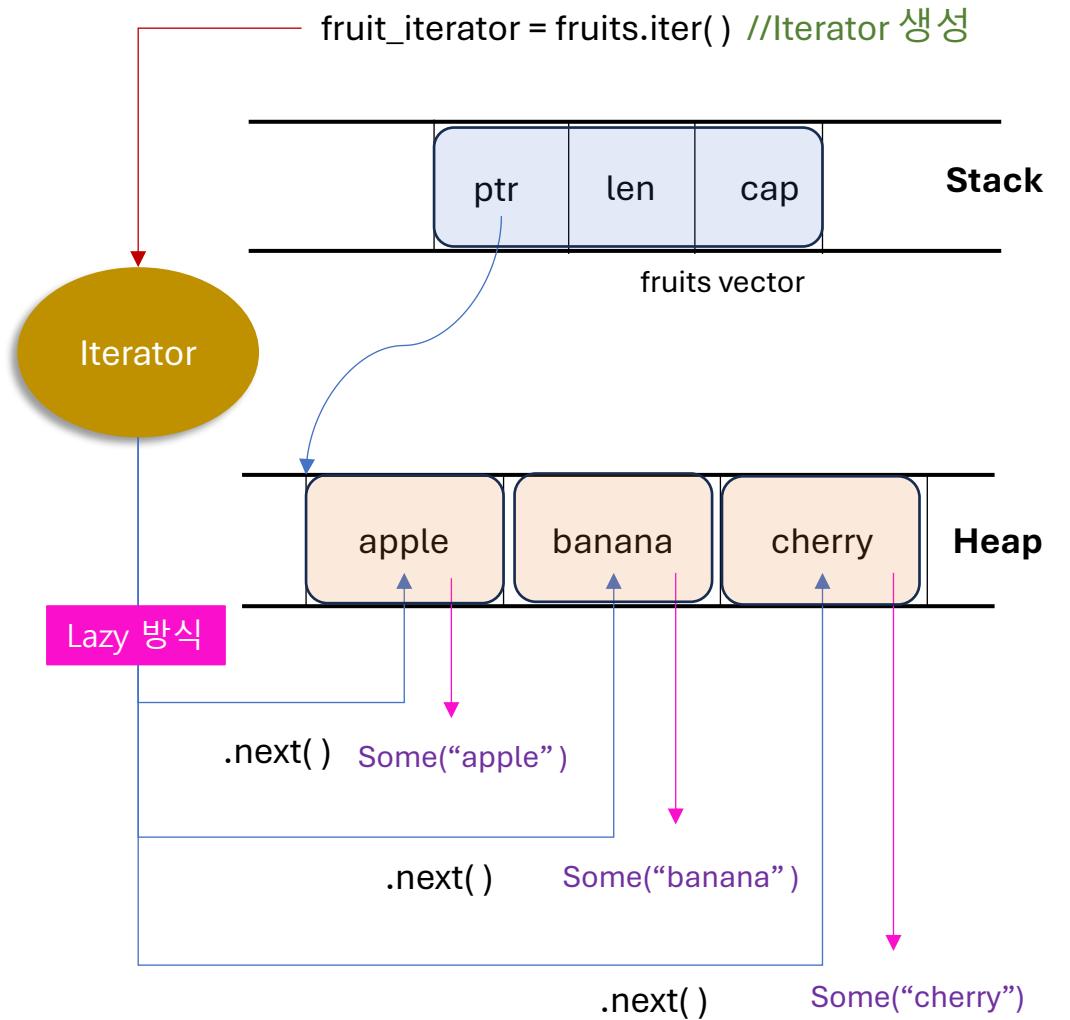
Iterator와 next() method

```
fn main() {
    let fruits = vec!["apple", "banana", "cherry"];

    // Create an iterator over references to the elements in the vector.
    // The iterator variable must be `mut` because calling `next()`
advances

    // the iterator and changes its internal state.
    let mut fruit_iterator = fruits.iter(),
        
        
        
        
        
        
    // Call next() manually
    println!("First call: {:?}", fruit_iterator.next()); // Some("apple")
    println!("Second call: {:?}", fruit_iterator.next()); // Some("banana")
    println!("Third call: {:?}", fruit_iterator.next()); // Some("cherry")
    println!("Fourth call: {:?}", fruit_iterator.next()); // None
    (sequence finished)

    // Note: Once an iterator returns None, it will always return None
    // afterwards.
    println!("Fifth call: {:?}", fruit_iterator.next()); // None
}
```



7. Functional Programming(2) - Iterator(4-1)

iter()

```
fn iter(&self);
```

<Iterator를 만드는 3가지 방법>

- Collection의 각 항목에 대해 immutable reference를 결과로 던져주는 Iterator 생성
- Collection(예: Vector)의 각 항목을 빌려와 사용하지만, 각 항목을 수정하지는 않는다. 따라서 원본 Collection은 그대로 보존된다.

iter_mut()

```
fn iter_mut(&mut self);
```

- Collection의 각 항목에 대해 Mutable reference를 결과로 던져주는 Iterator 생성
- Collection(예: Vector)의 각 항목을 mutable 속성으로 빌려와 수정한다. 따라서 원본 Collection은 수정된 상태로 보존된다.

into_iter()

```
fn into_iter(self);
```

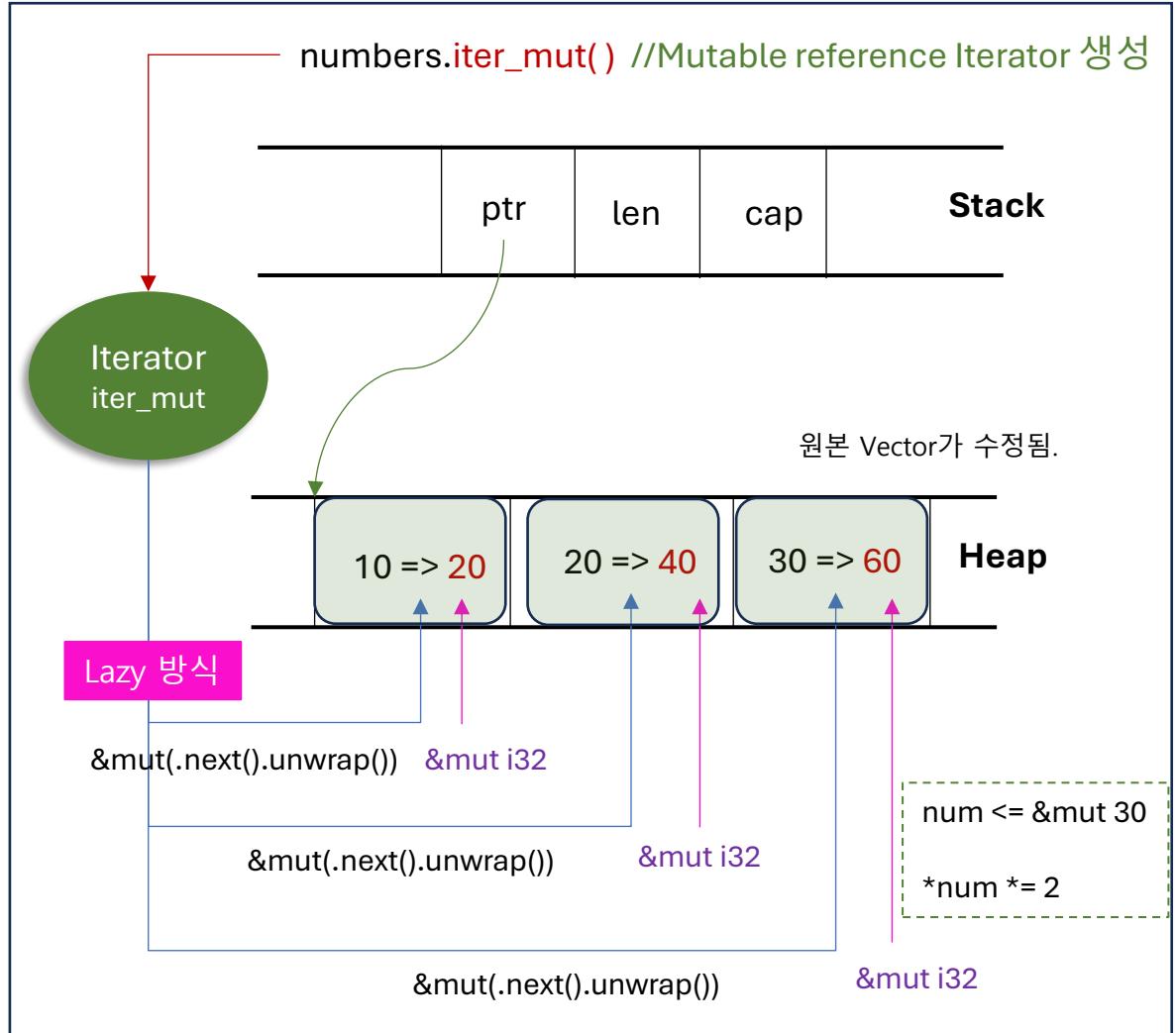
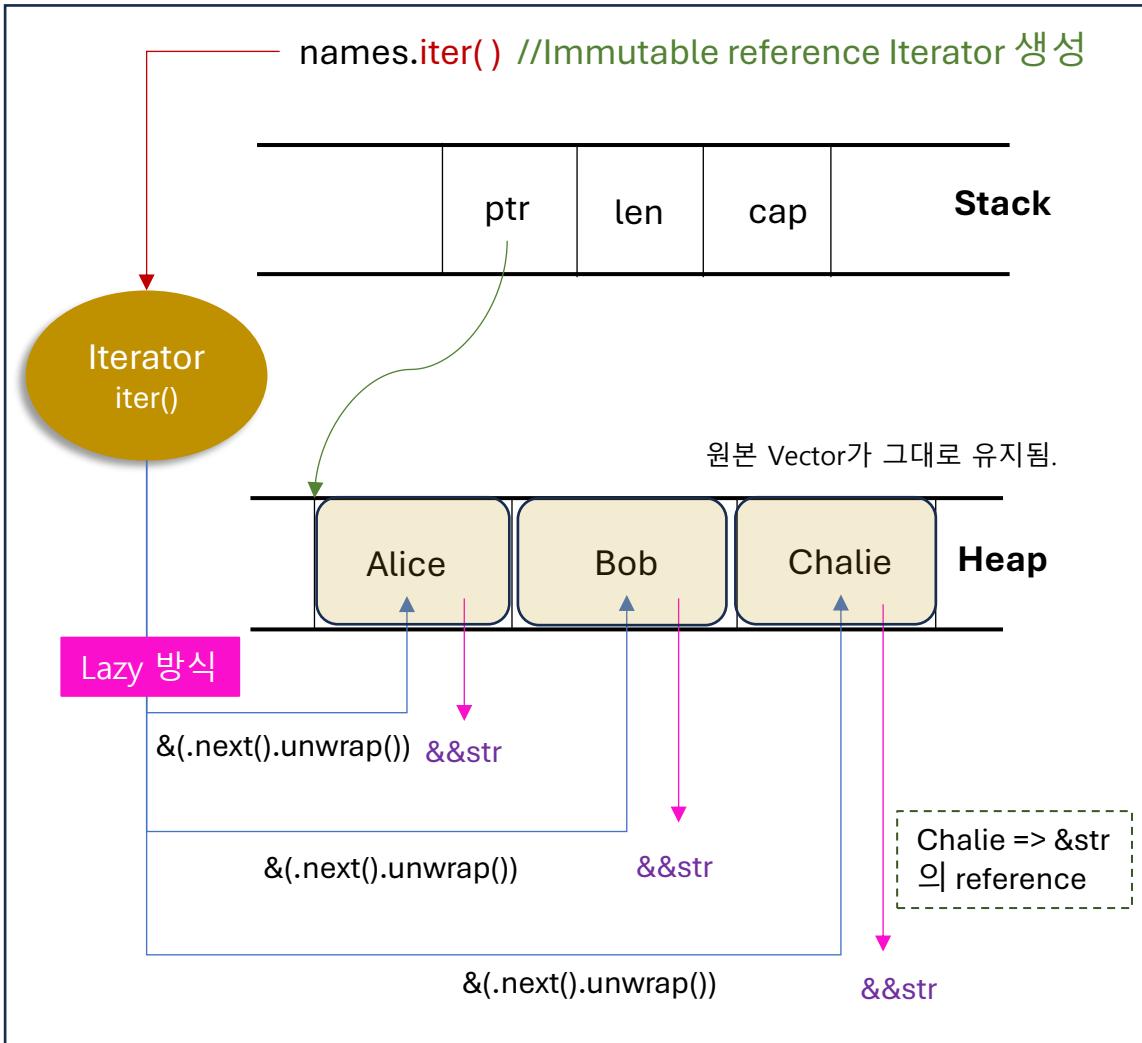
- Collection의 각 항목에 대해 소유권 이동(Move)이 발생하도록 하는 Iterator 생성
- Collection(예: Vector)의 각 항목의 소유권을 가져와 사용한다. 따라서 반복 후 원본은 사라진다.

7. Functional Programming(2) - Iterator(4-2)

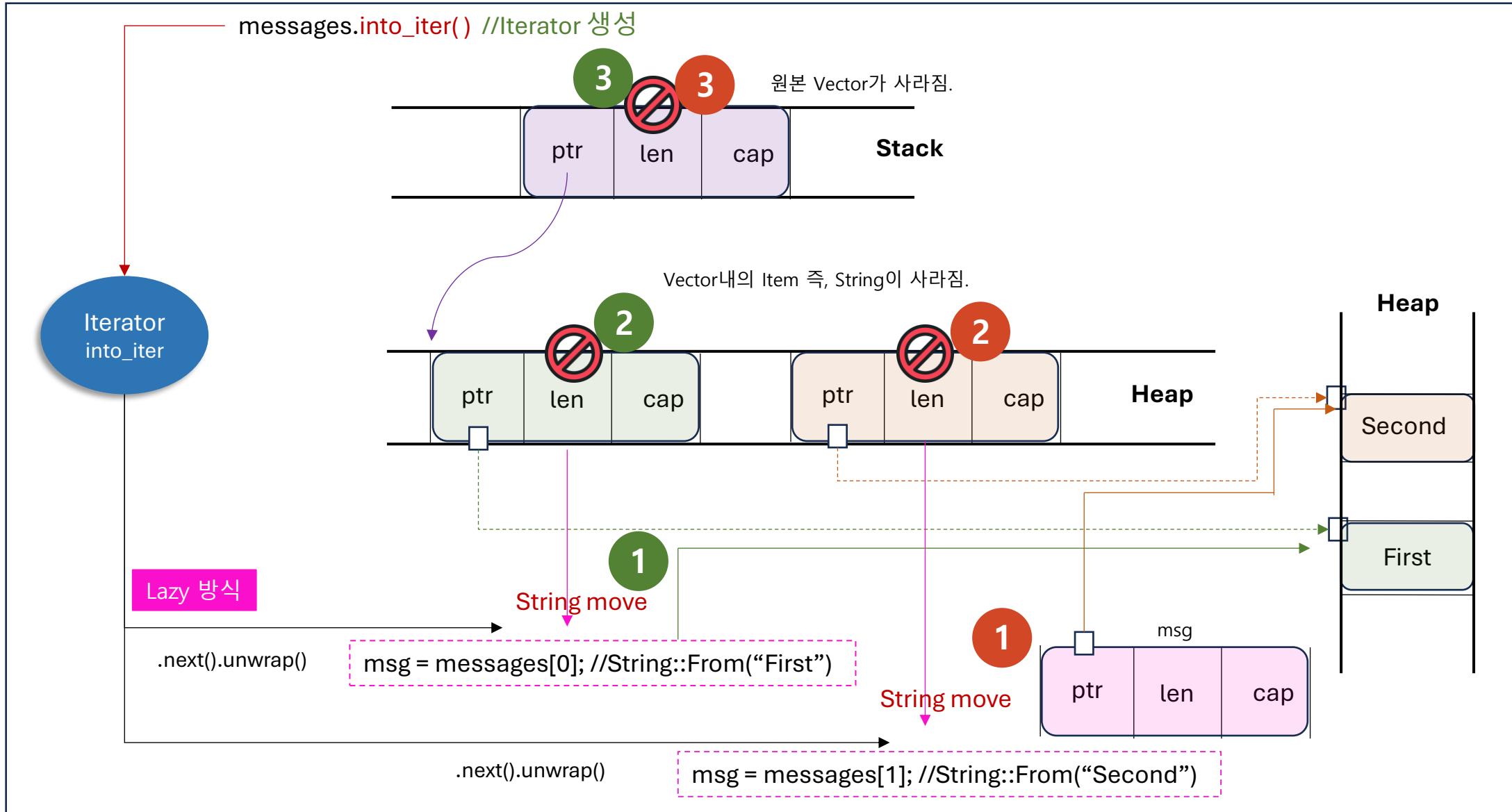
```
fn main() {  
    // --- 1. iter() - Immutable Borrows ---  
    let names = vec!["Alice", "Bob", "Charlie"];  
    for name in names.iter() {  
        // `name` here is of type `&&str` (a reference to a string slice)  
        println!("Hello, {}!", name);  
    }  
    // `names` is still valid and can be used here.  
    println!("The names vector is still available: {:?}", names);  
  
    // --- 2. iter_mut() - Mutable Borrows ---  
    let mut numbers = vec![10, 20, 30];  
    for num in numbers.iter_mut() {  
        // `num` here is of type `&mut i32` (a mutable reference)  
        *num *= 2; // We dereference `num` to modify the value it points  
        to  
    }  
    // `numbers` has been modified in place.
```

```
println!("The numbers vector has been modified: {:?}", numbers);  
  
    // --- 3. into_iter() - Taking Ownership ---  
    let messages = vec![String::from("First"), String::from("Second")];  
    for msg in messages.into_iter() {  
        // `msg` here is of type `String` (the owned value)  
        println!("Processing message: {}", msg);  
    }  
    // The `messages` vector has been moved and is no longer valid.  
    // The line below would cause a compile-time error.  
    // println!("Can we use messages again? No: {:?}", messages);
```

7. Functional Programming(2) - Iterator(4-3)



7. Functional Programming(2) - Iterator(4-4)



7. Functional Programming(3) - Consumer

Collect(), sum(), fold() consumer method

```
fn main() {  
    let numbers = vec![1, 2, 3, 4, 5];  
  
    // --- Using collect() ---  
  
    // Here, we filter for numbers greater than 2 and collect them into a  
    // new Vec.  
    // We use a type annotation `Vec<_>` to tell collect what to build.  
    let greater_than_two: Vec<_> = numbers  
        .iter()  
        .filter(|&n| n > 2)  
        .collect();  
  
    println!("Numbers greater than 2: {:?}", greater_than_two); // Output:  
    [3, 4, 5]
```

```
// --- Using sum() ---  
// We can sum the numbers in a range.  
let total: i32 = (1..=10).sum(); // The iterator is the range (1..=10)  
println!("The sum of numbers from 1 to 10 is: {}", total); // Output:  
55  
  
// --- Using fold() ---  
// Let's calculate the product of the numbers in our vector.  
// We start with an initial accumulator value of 1.  
let product = numbers  
    .iter()  
    .fold(1, |accumulator, &item| accumulator * item);  
println!("The product of the numbers is: {}", product); // Output: 120
```

Consumer는 최종 단계에서 실행되는 method로 Iterator로부터 소유권을 가져간다.
Iterator는 lazy해서, consumer method가 호출되는 시점에야 비로소 실제 원하는 action이 이루어진다.

7. Functional Programming(4) - Iterator Adapter(1)

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

    // Create an iterator, map a closure to square each number,
    // and then collect the results.

    let squares: Vec<i32> = numbers.iter() // Iterator yields &i32
        .map(|x| x * x) // Closure takes &i32,
    squares it, returns i32
        .collect(); // Collects the i32 results

    println!("Original: {:?}", numbers); // Output: [1, 2, 3, 4]
    println!("Squares: {:?}", squares); // Output: [1, 4, 9, 16]

    // Example with Strings
    let names = vec!["alice", "bob", "charlie"];
    let upper_names: Vec<String> = names.iter() // Iterator yields &str
        .map(|name| name.to_uppercase())
    // Closure returns String
        .collect(); // Collects the

    println!("Upper names: {:?}", upper_names); // Output: ["ALICE",
    "BOB", "CHARLIE"]
}
```

map() Iterator Adapter
(또 다른 Iterator를 만드는 method)

실제로 이 closure 내부의 동작은
아래 .collect() consumer method가
호출되는 시점에 이루어짐.

7. Functional Programming(4) - Iterator Adapter(2)

```
fn main() {
    let numbers = vec![1, 2, 3, 4, 5, 6, 7, 8, 9, 10];

    // Create an iterator, filter for even numbers, and collect.
    let evens: Vec<i32> = numbers.iter() // Iterator yields &i32
        // Closure takes &i32, returns true if
even.

        // We need *x because filter gets a
reference (&i32)
        .filter(|&&x| x % 2 == 0) // Note the
double reference `&&x`
        .copied() // Convert the iterator of &i32
to i32
        .collect(); // Collects the i32 results

    println!("Original: {:?}", numbers); // Output: [1, 2, ..., 10]
    println!("Evens: {:?}", evens); // Output: [2, 4, 6, 8, 10]
    // Chaining map and filter
    let scores = vec![85, 42, 95, 60, 77];
    // Get scores above 70, and add a 5-point bonus
    let adjusted_high_scores: Vec<i32> = scores.iter()
        .filter(|&&score| score >
70) // Keep scores > 70
        Add 5 to the filtered scores
        .map(|&score| score + 5) //

        .collect();

    println!("Adjusted high scores: {:?}", adjusted_high_scores); //
Output: [90, 100, 82]
}
```

filter() Iterator Adapter
(또 다른 Iterator를 만드는 method)

<TODO>

이 밖에도 유용한 iterator adapter 존재함
▪ take, skip, rev, zip, enumerate, flat_map

7. Functional Programming(4) - Iterator Adapter(3)

filter()의 Closure 파라미터 Reference 사용 방법에 대한 분석



case#1) numbers.iter().filter(|&&x| { })

$\&\&x = n: \&\&i32 \Rightarrow x$

case#2) numbers.iter().filter(|x| { })

$x = n: \&\&i32 \Rightarrow **x$

[1] iter() method는 Collection 변수 각 항목(예: i32 type)에 대해 Reference 값(예: n: &i32)을 return 한다. 그리고, iterator adapter인 filter method는 argument type으로 Reference를 전달 받도록 정의하고 있다. 따라서, case1과 같이 closure 파라미터로 &&x를 지정한 상태에서 argument n: &&i32를 넘기게 되면,

$\&\&x = n: \&\&i32 \Rightarrow x = n: i32$ //양변의 && 상쇄

와 같은 형태가 되어, 실제 closure 내부에서는 x를 곧 바로 사용할 수가 있게 된다.

[2] 한편, case2와 같이 closure 파라미터로 x를 지정하게 될 경우에는

$x = n: \&\&i32 \Rightarrow **x$

형태가 되어, closure 내부에서 실제 값에 접근하려면 두번의 역참조 즉, **x를 해 주어야만 한다.

7. Functional Programming(5) - Closure(1)

<Closure의 정의>

클로저(Closure)는 주변 환경(scope)의 변수를 캡처하여 사용할 수 있는 익명 함수이다. 변수에 저장하거나 다른 함수의 인자로 전달 가능하며, 불변(Immutable)/가변 참조(Mutable) 또는 소유권 이동 방식(Move)으로 외부 값을 캡처할 수 있다.

환경 캡처 (Capture Environment): 정의된 스코프 내의 변수를 클로저 내부에서 자유롭게 사용할 수 있다.

익명성 (Anonymous): 이름을 정의하지 않고 바로 변수에 대입하거나 인자로 넘긴다.

타입 추론 (Type Inference): 대부분의 경우 컴파일러가 매개변수와 반환 타입을 추론하므로 명시하지 않아도 된다.

Closure는 환경 capture 방식에 따라 아래와 같이 3가지 Closure 트레잇(Fn, FnMut, FnOnce)을 지원한다.

```
fn main() {
    // The compiler infers that `x` is an i32 and the return type is i32.
    let add_one = |x| x + 1;
    println!("5 + 1 = {}", add_one(5));

    // You can also add explicit type annotations for clarity.
    let multiply = |a: i32, b: i32| -> i32 {
        a * b
    };
    println!("3 * 4 = {}", multiply(3, 4));
}
```

7. Functional Programming(5) - Closure(2)

```
fn main() {
    let my_name = String::from("Alice");
    let mut counter = 0;
    let data = vec![1, 2, 3];

    // 1. Captures `my_name` by immutable reference (&String)
    // because it only reads it.
    let greet = || println!("Hello, {}!", my_name);
    greet();
    // `my_name` is still valid here.
    println!("`my_name` can still be used: {}", my_name);

    // 2. Captures `counter` by mutable reference (&mut i32) because
    // it modifies it.
    let mut increment = || {
```

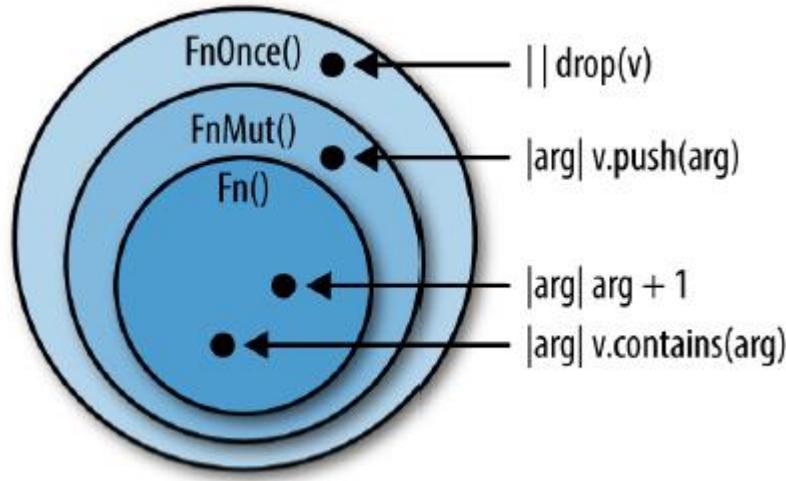
Capture & Move

```
        counter += 1;
        println!("Counter is now: {}", counter);
    };
    increment();
    increment();
    // `counter` has been modified.
    println!("Final counter value: {}", counter);

    // 3. Captures `data` by taking ownership (Vec<i32>) because of
    // the `move` keyword.
    // We'll discuss `move` next.
    let consume_data = move || {
        println!("Consumed data: {:?}", data);
        // `data` is dropped when this closure ends.
    };
    consume_data();
    // The line below would cause a compile error because `data` was
    // moved.
    // println!("Can we use data after move? No: {:?}", data);
}
```

7. Functional Programming(5) – Closure(3-1)

Closure 관련 3가지 트레이트



Trait	How it captures variables	How many times it can be called
Fn	Borrows captured variables immutably (<code>&self</code>)	Multiple times, potentially concurrently
FnMut	Borrows captured variables mutably (<code>&mut self</code>)	Multiple times, but not concurrently
FnOnce	Takes ownership of captured variables (<code>self</code>)	Exactly once, consuming the closure

7. Functional Programming(5) - Closure(3-2)

```
// This function accepts closures that only need immutable access (`Fn`).
fn call_reporter<F>(reporter: F)
where
    F: Fn() -> String, // Trait bound: must implement Fn
{
    println!("Report: {}", reporter());
}

// This function accepts closures that might mutate their environment
(`FnMut`).
fn call_mutator<F>(mut mutator: F) // Note the `mut` here
where
    F: FnMut(), // Trait bound: must implement FnMut
{
    // We can call it multiple times.
    mutator();
    mutator();
}

// This function accepts any closure but consumes it (`FnOnce`).
fn call_once<F>(consumer: F)
where
    F: FnOnce(), // Trait bound: must implement FnOnce
{
    consumer();
    // Calling `consumer()` again here would cause a compile error.
}
```

```
fn main() {
    let message = String::from("System status OK");
    // This closure captures `message` by reference, so it implements
    `Fn`.
    let report_closure = || message.clone(); // Fn trait 사용
    call_reporter(report_closure);

    let mut counter = 0;
    // This closure captures `counter` by mutable reference, so it
    implements `FnMut`.
    let mut increment_closure = || {
        counter += 1;
        println!("Counter is now: {}", counter);
    };
    // We pass ownership of the closure to `call_mutator`.
    call_mutator(increment_closure);

    let data = String::from("Consume me");
    // This closure moves `data`, so it implements `FnOnce`.
    let consume_closure = || {
        println!("Consumed: {}", data);
    };
    call_once(consume_closure);
}
```

8. 스마트 포인터(Smart Pointer)

Box<T>, Rc<T>/Arc<T>, RefCell<T>/Mutex<T>

8. 스마트 포인터(1) - What is Smart Pointer ?(1)

“Smart pointers are essentially structs that implement the Deref and Drop traits.”

(스마트 포인터는 Deref 트레이트와 Drop 트레이트를 구현한 struct 이다)

```
pub trait Deref {  
    fn deref(&self) -> &Self::Target;  
}
```

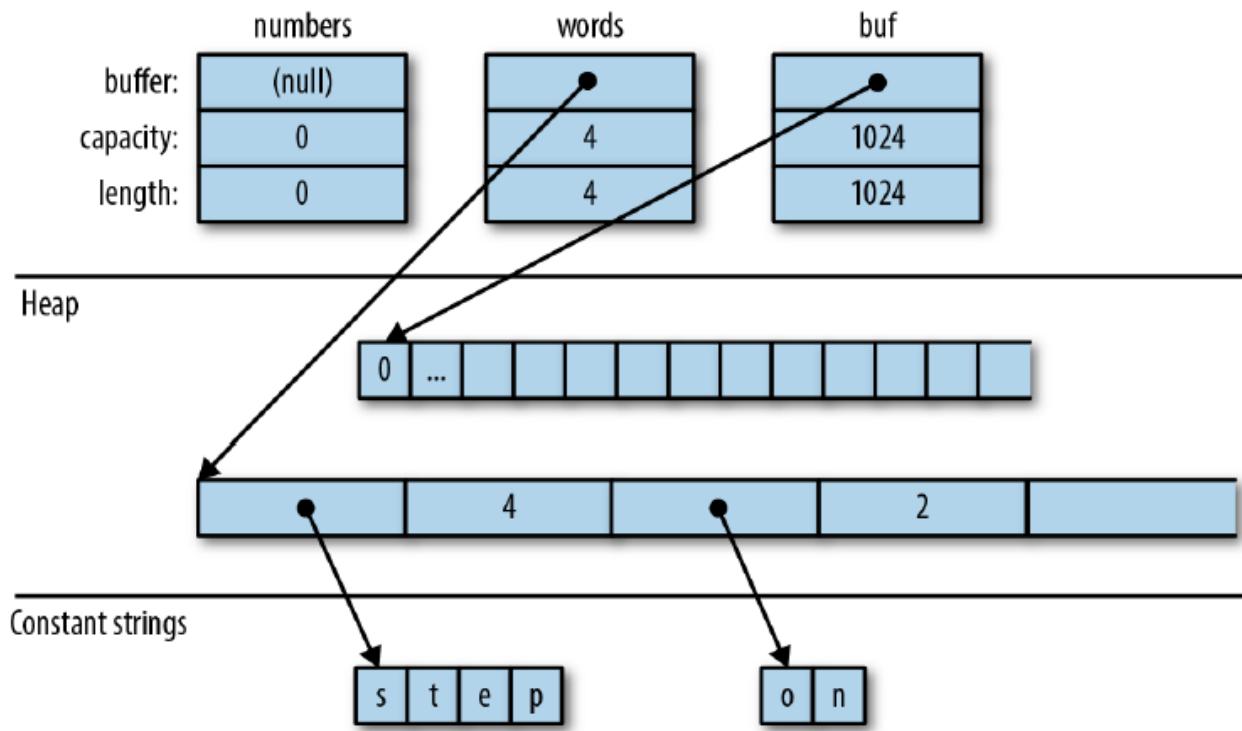
```
pub trait Drop {  
    fn drop(&mut self);  
}
```

```
struct SmartPointer {  
    pointer: usize,  
    length: usize,  
    capacity: usize,  
}  
  
impl Deref for SmartPointer {  
    fn deref(&self) -> &Self::Target {  
        ...  
    }  
}  
  
impl Drop for SmartPointer {  
    fn drop(&mut self) {  
        ...  
    }  
}
```

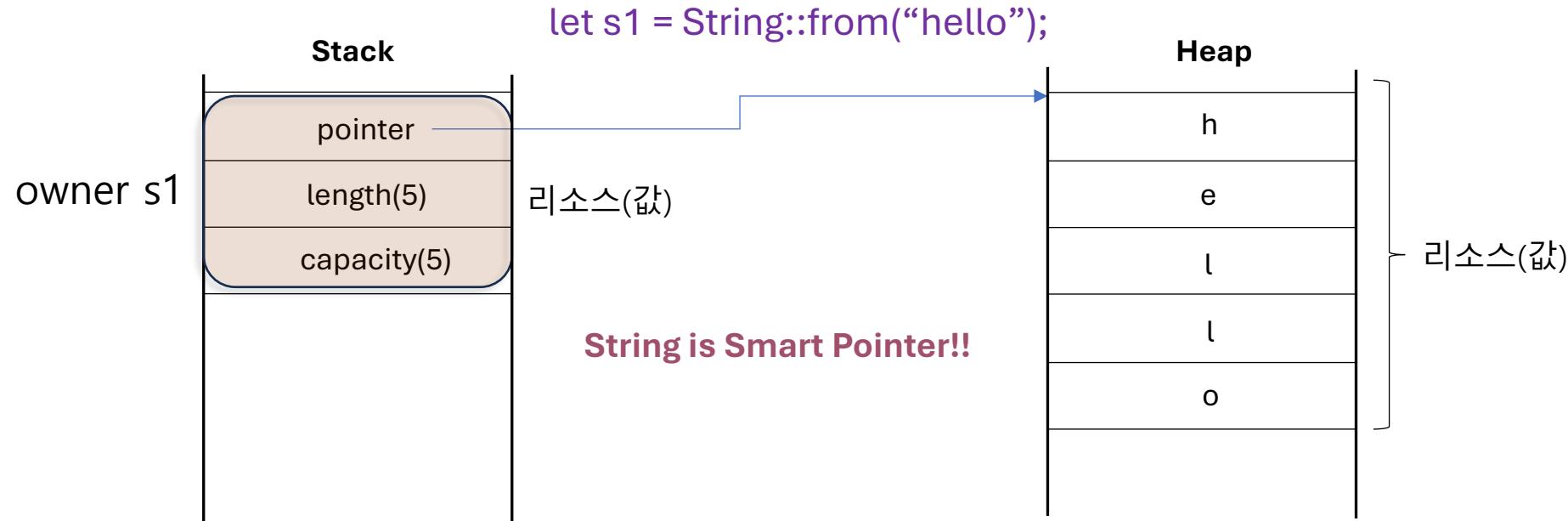
8. 스마트 포인터(1) – What is Smart Pointer ?(2)

```
// Create an empty vector
let mut numbers: Vec<i32> = vec![];
// Create a vector with given contents
let words = vec!["step", "on", "no", "pets"];
let mut buffer = vec![0u8; 1024]; // 1024 zeroed-out bytes
```

Vector is Smart Pointer!!



8. 스마트 포인터(1) - What is Smart Pointer ?(3)



Rust의 소유권 3대 원칙

1) Resource(혹은 값)에는 소유권이 있으며, 변수(variable)는 resource(혹은 값)의 소유자(Owner)이다.

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

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

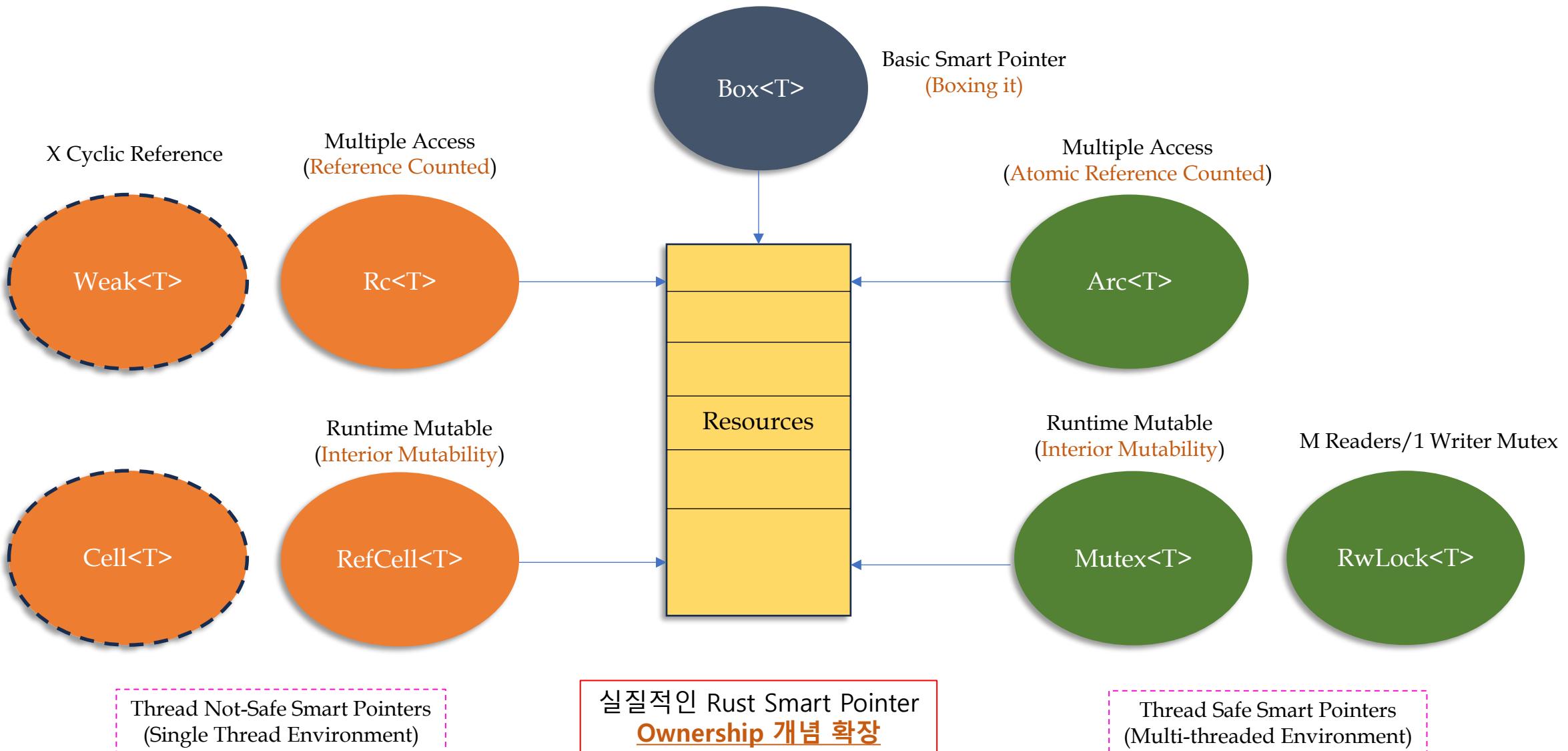
매우 제한적임

8. 스마트 포인터(1) - What is Smart Pointer ?(4)

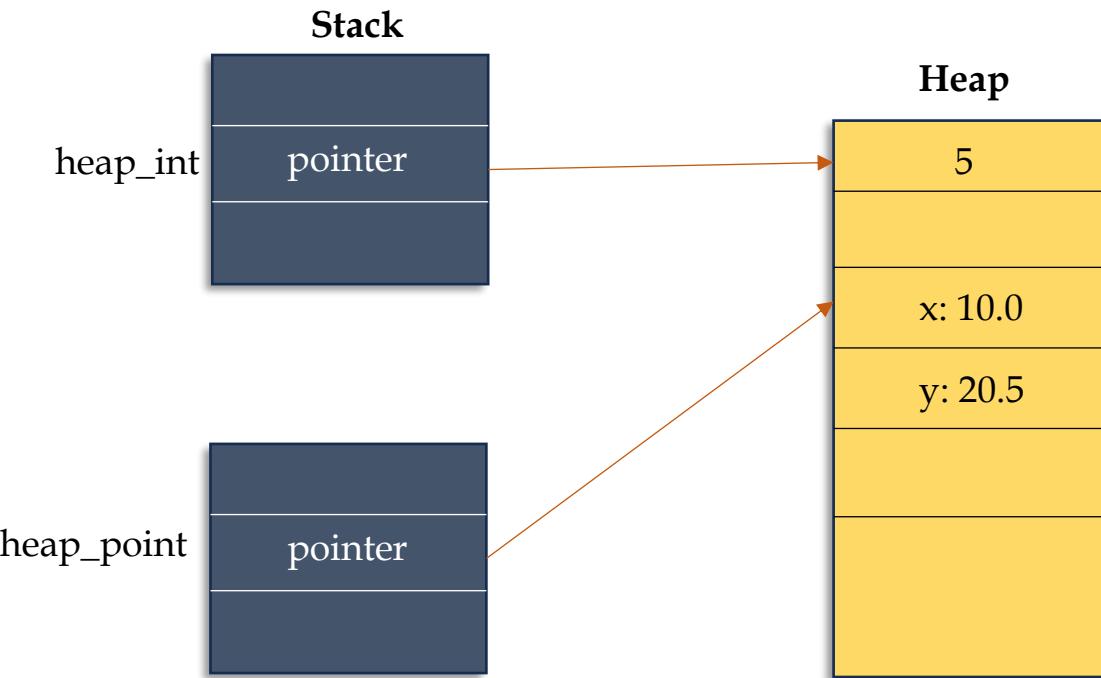
Rust 소유권 개념 확장

- [1] 동시에 여럿이 1개의 Resource에 접근하고 싶다(Reference로도 가능하나, 소유자가 memory 해제 시, dangling reference 상황 발생할 수 있다). `Rc<T>`
- [2] 전체적으로는 Immutable type로 정의되었으나, 특정 내부 필드에 대해서는 (run-time에) Mutable 처리를 하고 싶다. `RefCell<T>`
- [3] Multi-thread 환경에서도 문제가 없이 동작하고 싶다. `Arc<T>, Mutex<T>`

8. 스마트 포인터(1) - What is Smart Pointer ?(5)



8. 스마트 포인터(2) - Box<T>(1)



Box<T>가 주로 사용되는 예

- Trait Object 구현 시 - Box<dyn MyTrait>
- Recursive Type 선언 시
- Large data를 전달하는 경우 - Box<MyLargeStruct>

```
// It's standard practice to define structs outside the main function.
#[derive(Debug)]
struct Point {
    x: f64,
    y: f64,
}

fn main() {
    // Create a Box<i32> to store an integer on the heap.
    let heap_int = Box::new(5);
    println!("Value in a Box: {}", heap_int);

    // Create a Box<Point> to store a struct on the heap.
    let heap_point = Box::new(Point { x: 10.0, y: 20.5 });
    println!("Struct in a Box: {:?}", heap_point);

    // Explicitly dereference the Box to access the value.
    let value_from_box = *heap_int;
    println!("Explicitly dereferenced value: {}", value_from_box);

    // Access a field directly thanks to automatic dereferencing (Deref coercion).
    println!("Accessing field via deref coercion: heap_point.x = {}", heap_point.x);

    // When `heap_int` and `heap_point` go out of scope here, the memory they manage
    // on the heap is automatically freed via the Drop trait.
}
```

8. 스마트 포인터(2) - Box<T>(2)

```
use std::fmt;

// A recursive enum representing a simple arithmetic expression.
// `Box<T>` is used to give the recursive variants a known size.
#[derive(Debug)]
enum Expression {
    Value(i32),
    Add(Box<Expression>, Box<Expression>),
    Multiply(Box<Expression>, Box<Expression>),
    Negate(Box<Expression>),
}

// A custom error type for our evaluation function.
#[derive(Debug)]
pub enum EvaluationError {
    Overflow,
}

// Implement Display for user-friendly error messages.
impl fmt::Display for EvaluationError {
    fn fmt(&self, f: &mut fmt::Formatter) -> fmt::Result {
        match self {
            EvaluationError::Overflow => write!(f, "Arithmetic overflow
occurred during evaluation"),
        }
    }
}
```

Recursive type 선언 예

```
// Helper functions for building the expression tree ergonomically.
fn val(v: i32) -> Box<Expression> {
    Box::new(Expression::Value(v))
}
fn add(left: Box<Expression>, right: Box<Expression>) -> Box<Expression> {
    Box::new(Expression::Add(left, right))
}
fn multiply(left: Box<Expression>, right: Box<Expression>) ->
Box<Expression> {
    Box::new(Expression::Multiply(left, right))
}
fn negate(expr: Box<Expression>) -> Box<Expression> {
    Box::new(Expression::Negate(expr))
}
```

8. 스마트 포인터(3) - Rc<T>(1)

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

#[derive(Debug)]
struct SharedConfig {
    version: String,
    api_key: String,
}

struct ServiceA {
    id: u32,
    config: Rc<SharedConfig>,
}

struct ServiceB {
    name: String,
    config: Rc<SharedConfig>,
}

fn main() {
    // Create the shared configuration data wrapped in an Rc.
    let shared_config = Rc::new(SharedConfig {
        version: "v1.2.3".to_string(),
        api_key: "ABC123XYZ789".to_string(),
    });

    let service_a = ServiceA {
        id: 101,
        config: Rc::clone(&shared_config),
    };

    let service_b = ServiceB {
        name: "LoggerService".to_string(),
        config: Rc::clone(&shared_config),
    };
}
```

```
api_key: "ABC123XYZ789".to_string(),
});

println!("Initial strong count: {}", Rc::strong_count(&shared_config));
// Create clones of the Rc to share ownership.
// This only increments the reference count; it does not deep copy the data.

let service_a = ServiceA {
    id: 101,
    config: Rc::clone(&shared_config),
};

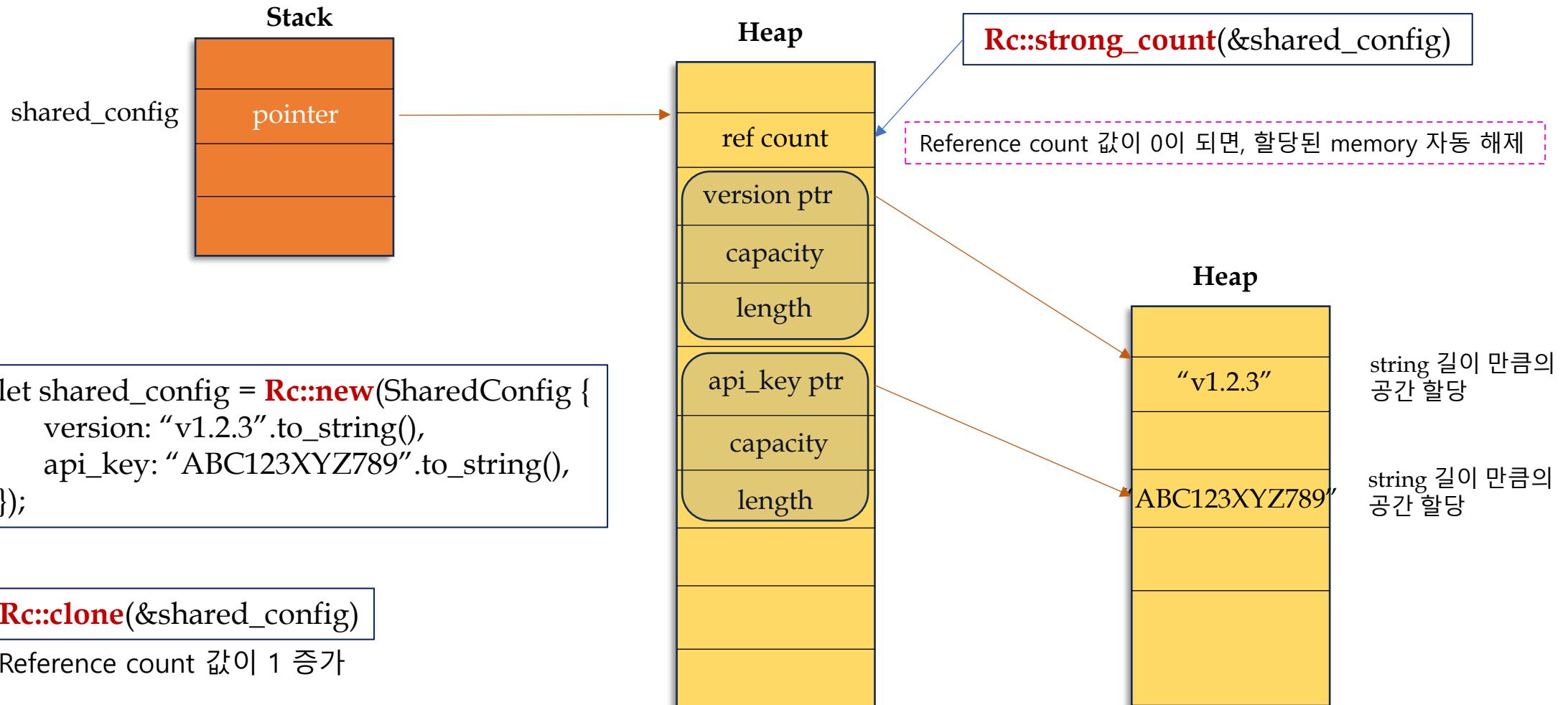
println!("After ServiceA created: strong count = {}", Rc::strong_count(&shared_config));
let service_b = ServiceB {
    name: "LoggerService".to_string(),
    config: Rc::clone(&shared_config),
};

println!("After ServiceB created: strong count = {}", Rc::strong_count(&shared_config));
println!("---");
println!("Service A accesses config version: {}", service_a.config.version);
println!("Service B accesses API key: {}", service_b.config.api_key);
println!("---");

// Explicitly drop service_a to see the reference count decrease.
drop(service_a);
println!("After ServiceA is dropped: strong count = {}", Rc::strong_count(&shared_config)); // Output: 2
// Explicitly drop service_b.
drop(service_b);
println!("After ServiceB is dropped: strong count = {}", Rc::strong_count(&shared_config)); // Output: 1
// The final Rc (`shared_config`) will be dropped at the end of main's scope.

// At that point, the count will become 0, and the SharedConfig data
// will be deallocated.
}
```

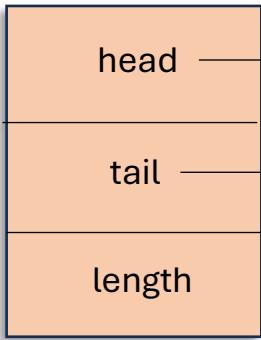
8. 스마트 포인터(3) - Rc<T>(2)



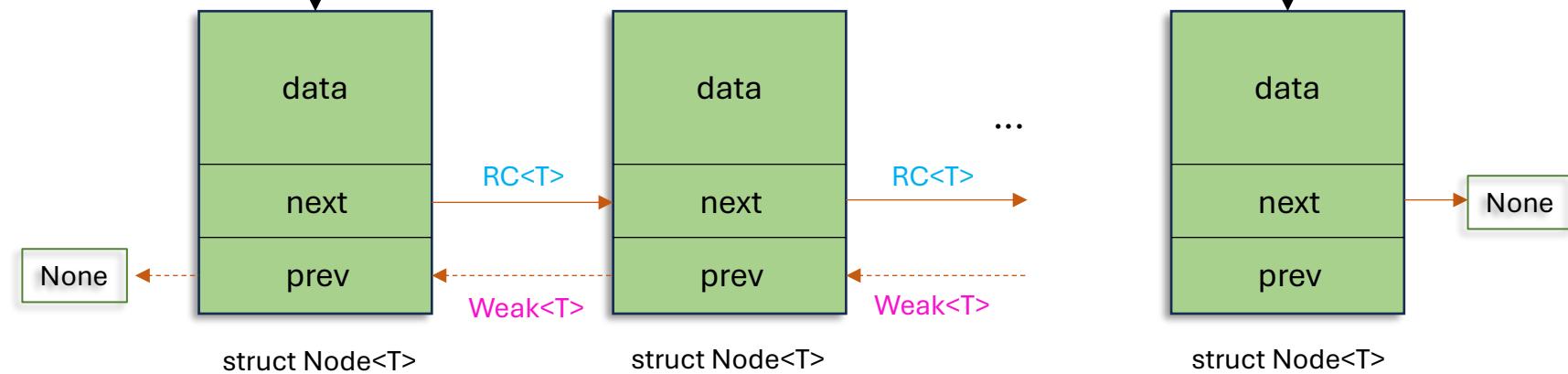
`Rc<T>`와 multi-thread 환경에서는 사용할 수가 없다. 즉, 여러 thread에서 사용 시, reference count 값이 뒤죽박죽 된다.

8. 스마트 포인터(3) - Rc<T>(3)

struct DoublyLinkedList<T>



Rc<T>는 Weak<T>와 함께 Doubly Linked List 구현 시 사용됨.



8. 스마트 포인터(4) - Arc<T>(1)

```
use std::sync::Arc;
use std::thread;
use std::time::Duration;
#[derive(Debug)]
struct SharedResource {
    id: u32,
    data: String,
}

fn main() {
    // Create shared data wrapped in an Arc
    let shared_resource_main = Arc::new(SharedResource {
        id: 1001,
        data: "This is some important data shared across threads.".to_string(),
    });
    println!("Main thread: Initial strong count = {}", Arc::strong_count(&shared_resource_main)); // Output: 1

    let mut thread_handles = vec![];
    for i in 0..3 { // Spawn 3 threads
        // Clone the Arc for each thread. This is crucial.
        // The cloned Arc is moved into the thread's closure.
        let shared_resource_for_thread = Arc::clone(&shared_resource_main);
        println!("Main thread: Count before spawning thread {} = {}", i, Arc::strong_count(&shared_resource_main));
        let handle = thread::spawn(move || {
            // Thread code here
        });
        thread_handles.push(handle);
    }
}
```

```
// This thread now has its own Arc pointing to the shared data
    println!("Thread {}: Started. Accessing resource ID: {}, Data:
'{}'. Strong count here: {}",

        i,
        shared_resource_for_thread.id,
        shared_resource_for_thread.data,
        Arc::strong_count(&shared_resource_for_thread) //
```

Count might be higher due to other clones

```
    );
    thread::sleep(Duration::from_millis(50)); // Simulate some
work
```

println!("Thread {}: Finished.", i);

// When shared_resource_for_thread goes out of scope here, the
count is decremented.

```
});
```

thread_handles.push(handle);

```
}
```

// The main thread still has its reference

```
println!("Main thread: After spawning threads, strong count = {}",
```

Arc::strong_count(&shared_resource_main));

```
println!(main_thread. Resource data: {}", shared_resource_main.data);
```

// Wait for all threads to complete

```
for handle in thread_handles {
    handle.join().unwrap();
}
```

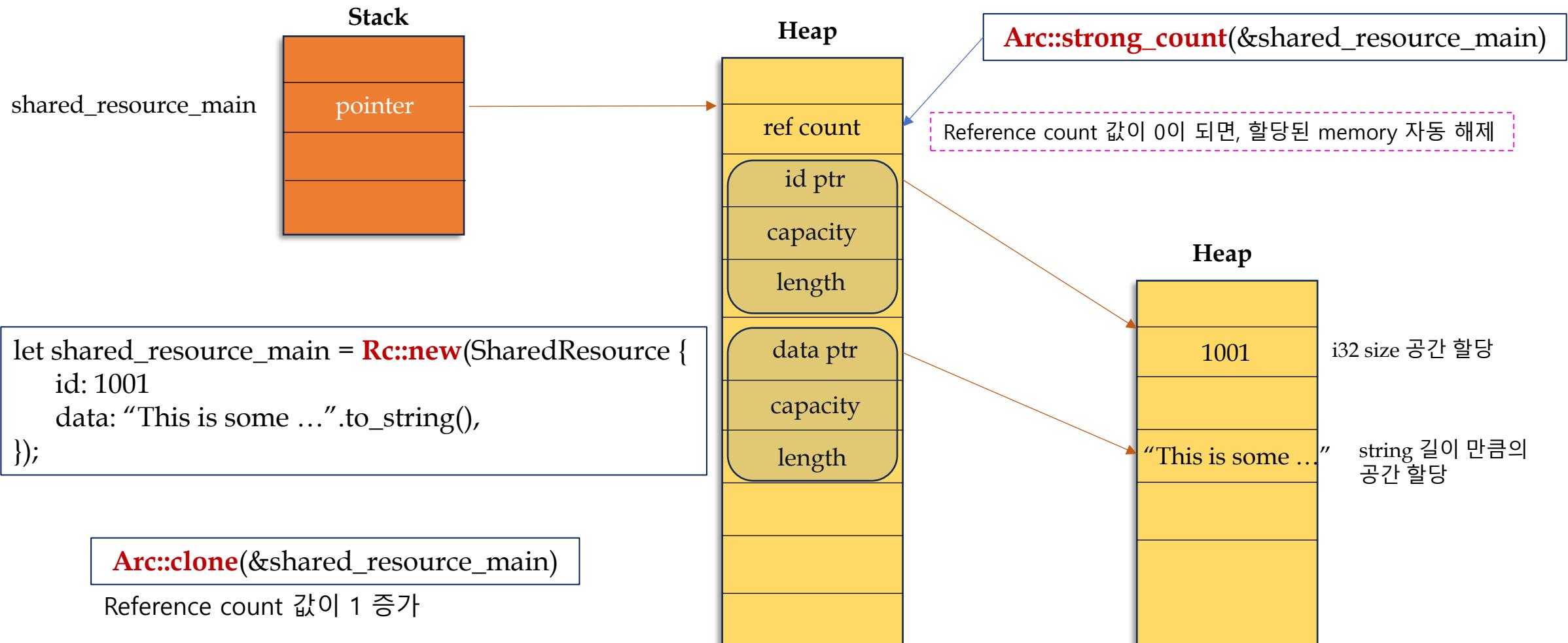
println!("Main thread: All threads finished. Final strong count
(before shared_resource_main drops): {}", Arc::strong_count(&shared_

resource_main)); // Should be 1

// When shared_resource_main drops, the count goes to 0, and
SharedResource is deallocated.

```
}
```

8. 스마트 포인터(4) – Arc<T>(2)



`Arc<T>`는 `Rc<T>`와 달리 Atomic 동작을 하므로, multi-thread 환경에 적합하다(즉, reference count가 일관되게 유지된다).

8. 스마트 포인터(5) - RefCell<T>(1)

Rc<RefCell<T>> 예제

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

// A Logger that uses interior mutability to track its state.
struct MessageLogger {
    message_count: RefCell<usize>,
    history: RefCell<Vec<String>>,
}

impl MessageLogger {
    fn new() -> Self {
        MessageLogger {
            message_count: RefCell::new(0),
            history: RefCell::new(Vec::new()),
        }
    }

    // This method takes &self but can modify internal state via RefCell.
    fn log(&self, message: &str) {
        let mut count = self.message_count.borrow_mut();
        *count += 1;
        self.history
            .borrow_mut()
            .push(format!("{}: {}", *count, message));
    }
}
```

```
*count += 1;

self.history
    .borrow_mut()
    .push(format!("{}: {}", *count, message));
}

fn get_count(&self) -> usize {
    *self.message_count.borrow()
}

fn print_history(&self) {
    println!("--- Log History ---");
    for entry in self.history.borrow().iter() {
        println!("{}", entry);
    }
    println!("-----");
}

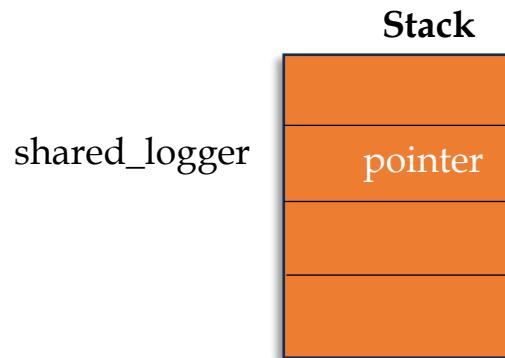
fn main() {
    // Wrap the MessageLogger in Rc to allow shared ownership.
    let shared_logger: Rc<MessageLogger> = Rc::new(MessageLogger::new());
    // Create multiple references to the same logger instance.
    let logger_clone1 = Rc::clone(&shared_logger);
    let logger_clone2 = Rc::clone(&shared_logger);

    // Log messages from different references.
    // Each call will mutate the same underlying MessageLogger instance.
    shared_logger.log("Main logger event.");
    logger_clone1.log("Event from clone 1.");
    logger_clone2.log("Event from clone 2.");
    // Check the final state from any of the references.
    println!("\nTotal messages logged: {}", shared_logger.get_count());
    shared_logger.print_history();
}
```

RefCell<T>는 대개 Rc<T>와 함께 사용(Rc<RefCell<T>>)되므로, 코드가 매우 복잡해 보임.

8. 스마트 포인터(5) - RefCell<T>(2)

borrow_mut()를 사용하면 RefMut<T>가 return됨.
이를 역참조하여 사용함.



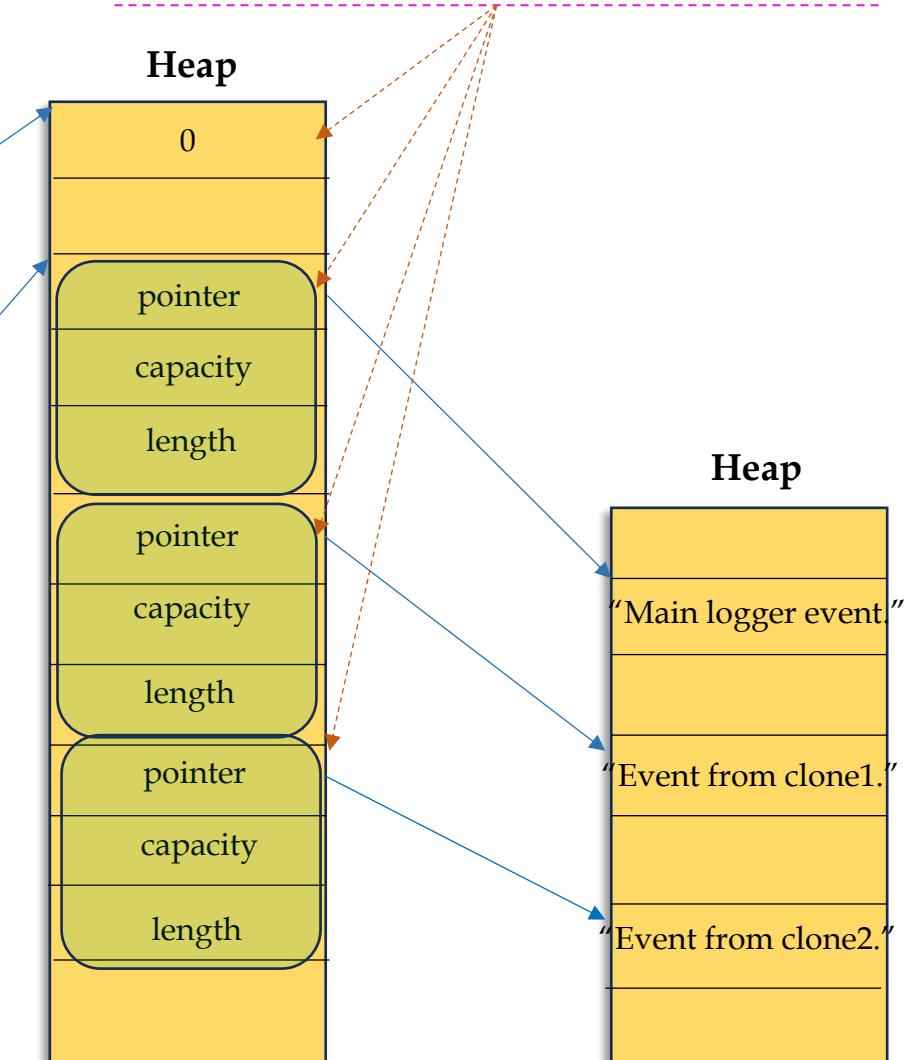
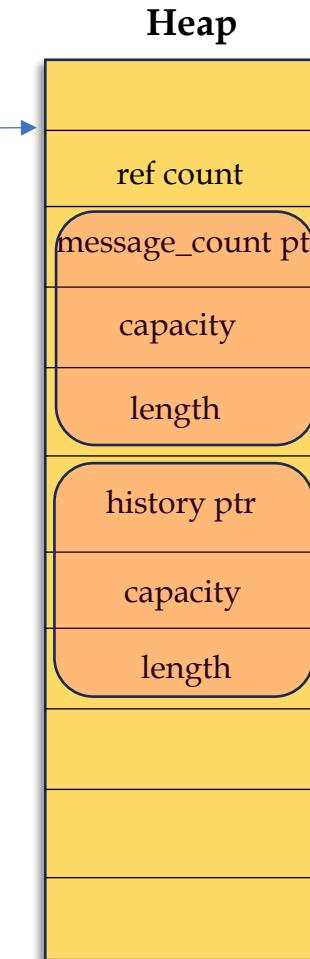
```
let shared_logger = Rc::new(MessageLogger::new());

MessageLogger {
    message_count: RefCell::new(0),
    history: RefCell::new(Vec::new()),
}
```

Rc::clone(&shared_logger)

let mut count = self.message_count.borrow_mut();

self.history
.borrow_mut()
.push(format!("{}: {}", *count, message));



RefCell<T>로 선언된 field는 runtime에 borrow_mut() 함수로 빌려와 수정 가능함.

8. 스마트 포인터(5) - RefCell<T>(3)

RefCell<T> 사용 방법 요약

```
struct MessageLogger {  
    message_count: RefCell<usize>,  
    history: RefCell<Vec<String>>,  
}
```

```
let m = MessageLogger {  
    message_count: RefCell::new(0),  
    history: RefCell::new(Vec::new()),  
}
```

//MessageLogger 자체는 Immutable로 선언되었으나, 내부 field를 수정하기 위하여, RefCell을 이용함.
let mut count = m.message_count.borrow_mut(); //RefMut<usize>가 return 된다.

```
*count += 1;
```

```
...
```

8. 스마트 포인터(6) - Mutex<T>(1)

```
use std::sync::{Arc, Mutex};
use std::thread;
use std::time::Duration;

struct GlobalCounter {
    count: Arc<Mutex<u32>>, // Shared, mutable counter
}

impl GlobalCounter {
    fn new() -> Self {
        GlobalCounter { count: Arc::new(Mutex::new(0)) }
    }

    fn increment(&self) {
        // Acquire the Lock. This blocks if another thread has the Lock.
        // unwrap() here will panic if the mutex is poisoned.
        let mut num_guard = self.count.lock().unwrap();
        *num_guard += 1; // Mutate the data through the MutexGuard
        // Lock is released when num_guard goes out of scope
    }

    fn get_value(&self) -> u32 {
        *self.count.lock().unwrap() // Lock, get value, unlock
    }
}
```

```
fn main() {
    let global_counter = GlobalCounter::new();
    let mut handles = vec![];

    println!("Initial count: {}", global_counter.get_value()); // Output:
    0

    // Spawn multiple threads that all increment the same counter
    for i in 0..5 {
        // Create the Arc to give ownership to the new thread
        let counter_clone = Arc::clone(&global_counter.count);

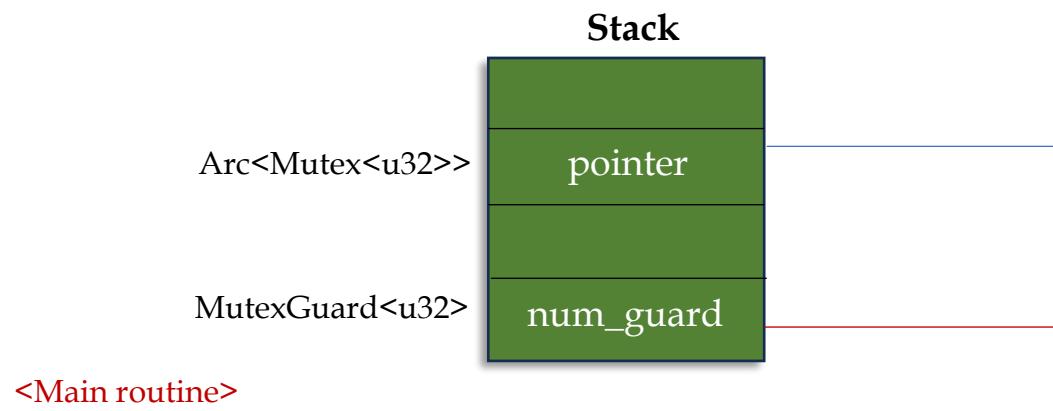
        let handle = thread::spawn(move || {
            // This is a different way to use the Arc<Mutex<T>> directly
            // without the GlobalCounter struct, for illustration.
            for _ in 0..100 {
                let mut num = counter_clone.lock().unwrap();
                *num += 1;
                // Slight delay to make interleaving more likely
                thread::sleep(Duration::from_micros(1));
            }
            println!("Thread {} finished incrementing.", i);
        });
        handles.push(handle);
    }

    // Using the methods on our GlobalCounter struct from the main thread
    // (This is a bit contrived here as the struct methods would also
    // contend for the same lock
    // if called concurrently with the threads above, but demonstrates
    // method usage)
    global_counter.increment(); // Main thread also increments

    // Wait for all spawned threads to finish
    for handle in handles {
        handle.join().unwrap();
    }

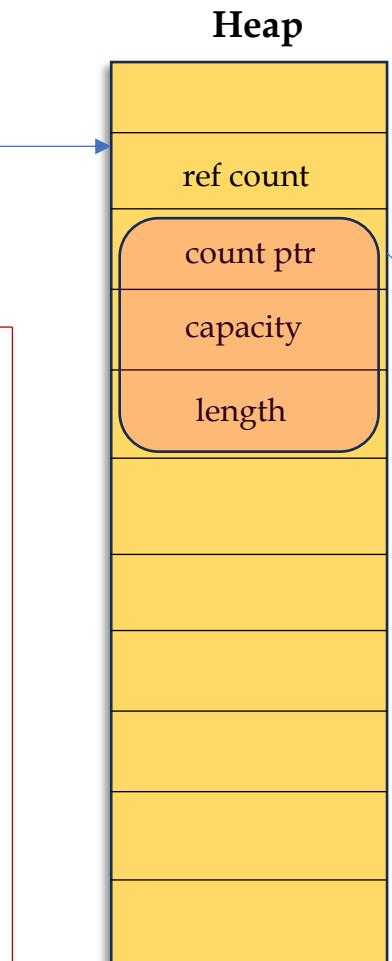
    println!("Final count: {}", global_counter.get_value()); // Expected:
    5 * 100 + 1 = 501
}
```

8. 스마트 포인터(6) - Mutex<T>(2)



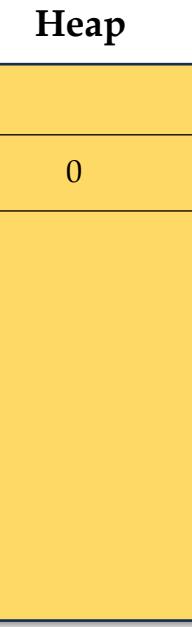
<Main routine>

```
struct GlobalCounter {  
    count: Arc<Mutex<u32>>,  
}  
  
fn new() -> Self {  
    GlobalCounter { count: Arc::new(Mutex::new(0)) }  
}  
  
let mut num_guard = self.count.lock().unwrap();  
    //MutexGuard<u32>를 return 한다.  
  
*num_guard += 1;  
  
*self.count.lock().unwrap()  
    //MutexGuard<u32>를 return 한다.
```

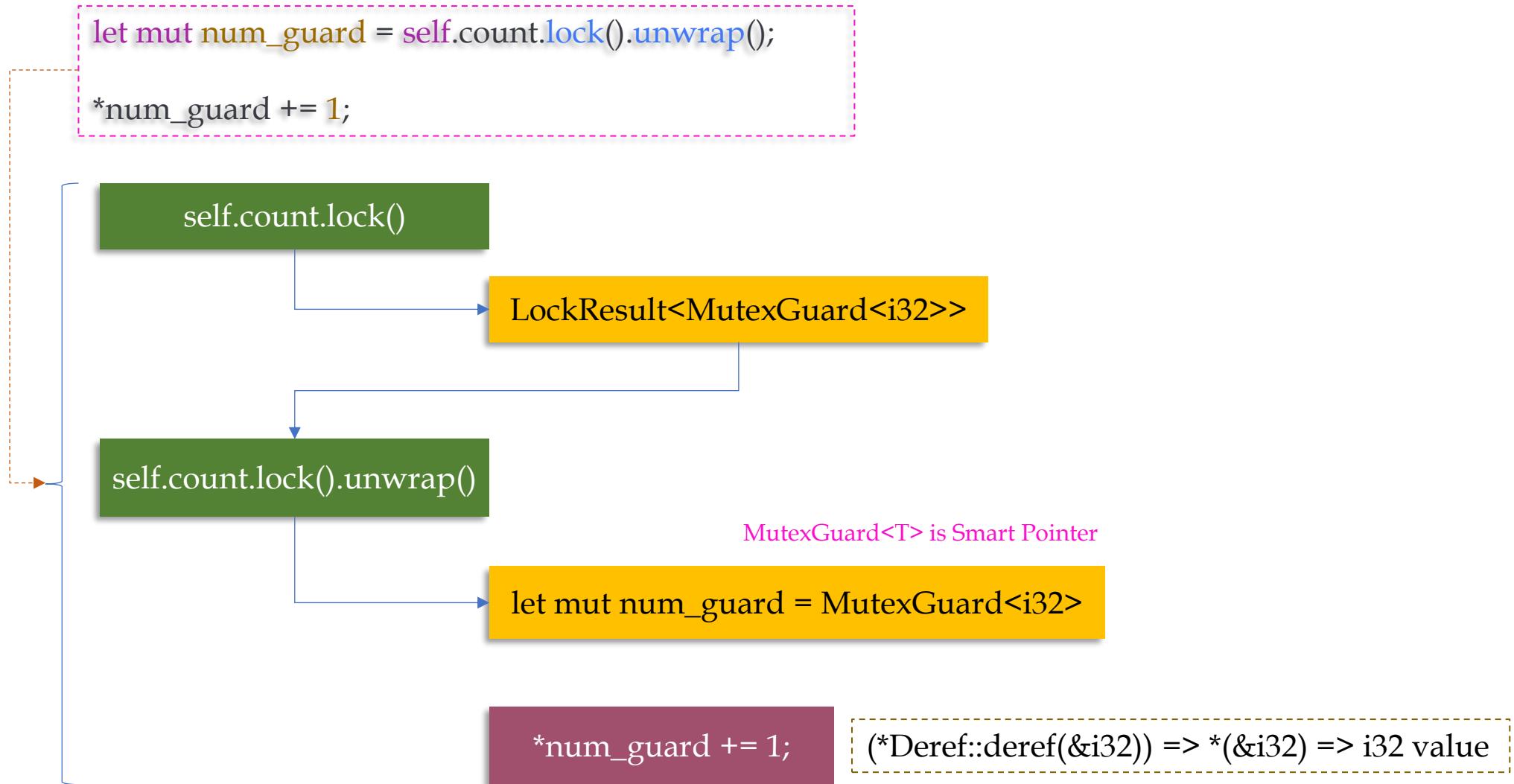


<Thread routine>

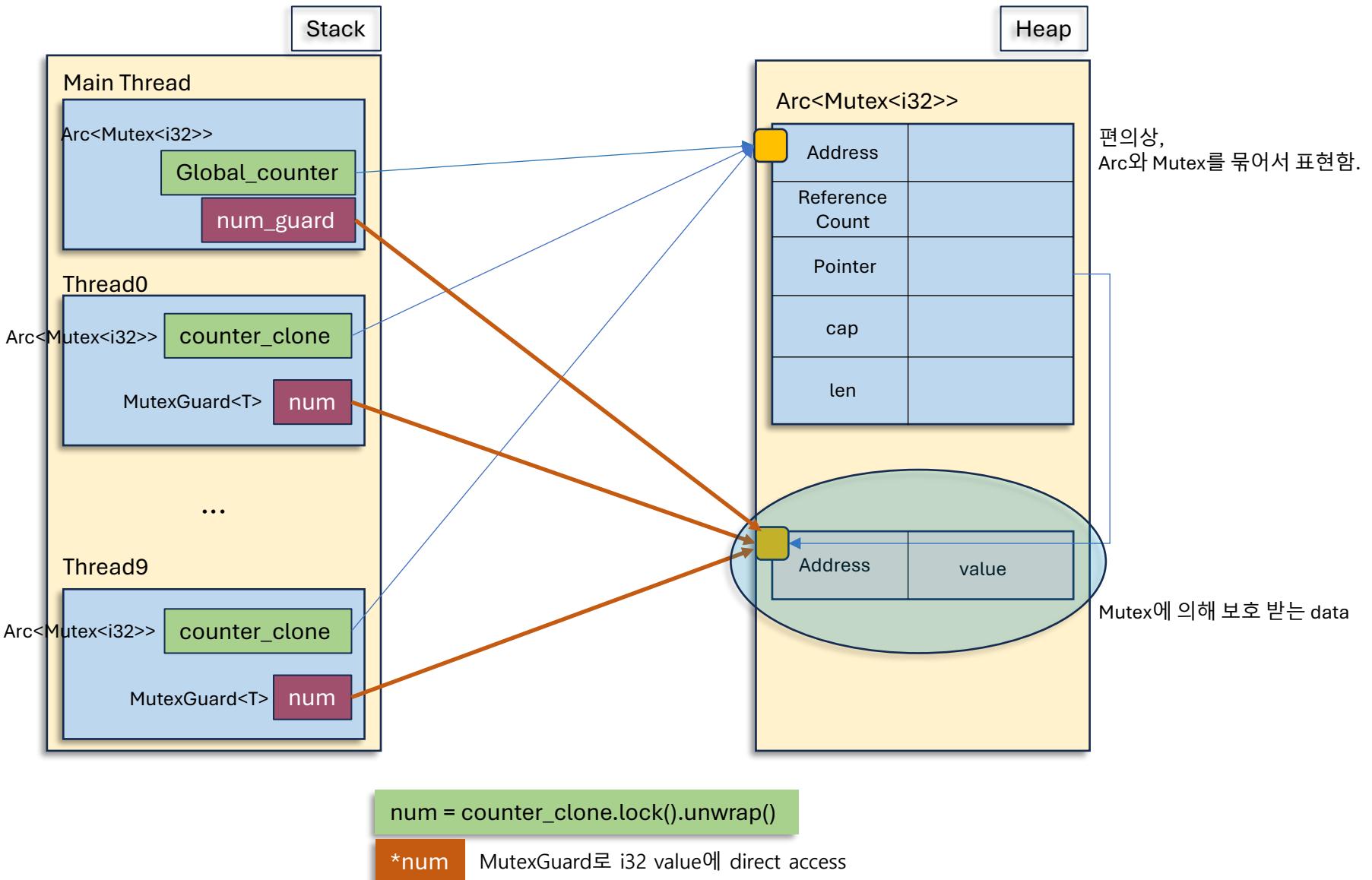
```
let counter_clone = Arc::clone(&global_counter.count);  
  
let mut num = counter_clone.lock().unwrap();  
*num += 1;
```



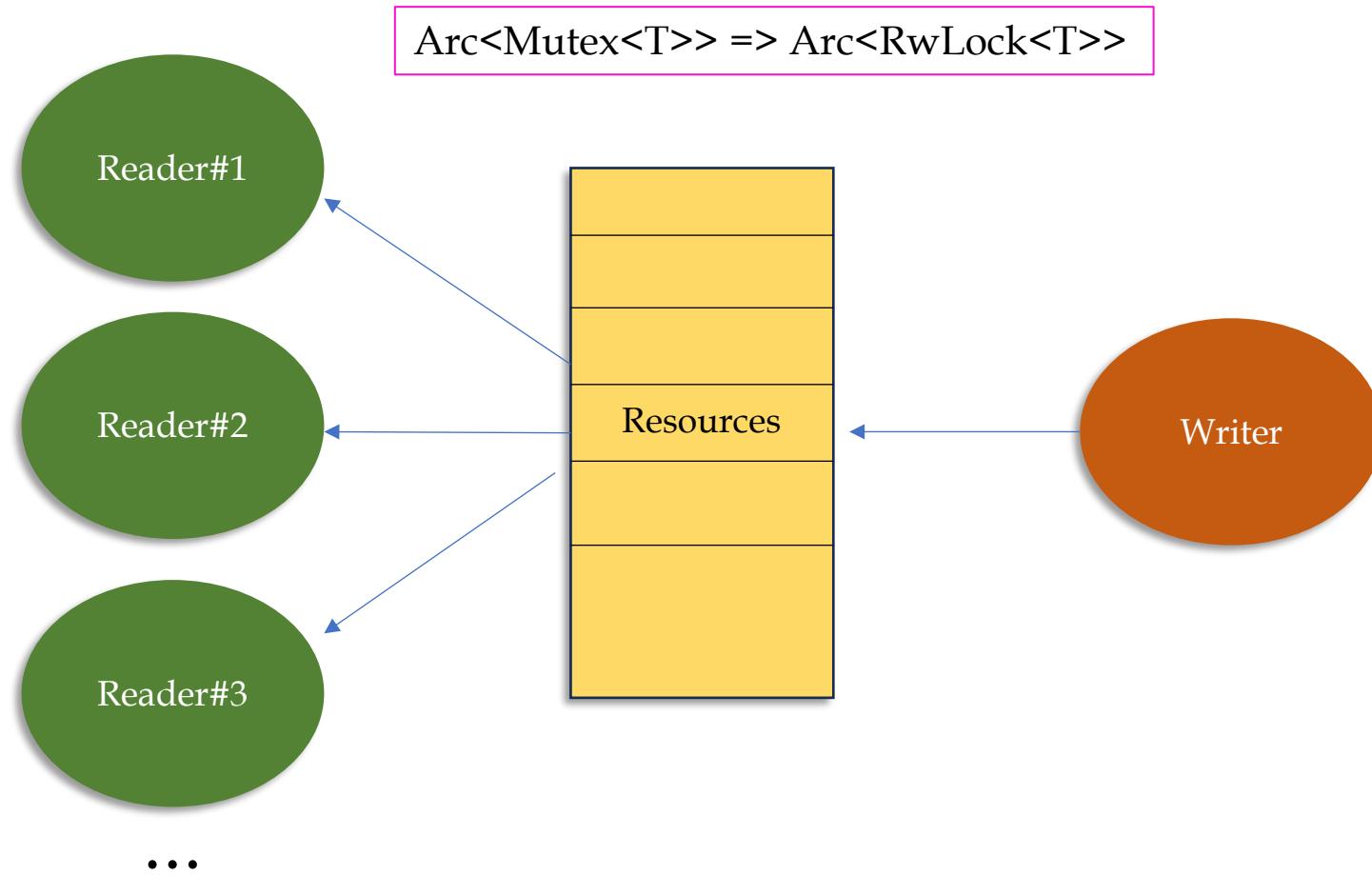
8. 스마트 포인터(6) - Mutex<T>(3)



8. 스마트 포인터(6) - Mutex<T>(4)



8. 스마트 포인터(7) - $\text{RwLock}\langle T \rangle(1)$



8. 스마트 포인터(7) - RwLock<T>(2)

```
use std::sync::{Arc, RwLock};
use std::thread;

fn main() {
    // A shared configuration that many threads will read, but one will
    // update.

    let config = Arc::new(RwLock::new("Initial Config".to_string()));

    let mut handles = vec![];

    // --- Spawn multiple READER threads ---
    for i in 0..5 {
        let config_clone = Arc::clone(&config);
        handles.push(thread::spawn(move || {
            // Acquire a read Lock. Multiple threads can hold this at the
            // same time.
            let config_guard = config_clone.read().unwrap();
            println!("Reader {}: sees config: '{}'", i, *config_guard);
        }));
    }
}
```

```
// --- Spawn one WRITER thread ---
let config_clone = Arc::clone(&config);
handles.push(thread::spawn(move || {
    // Simulate some work before writing
    thread::sleep(std::time::Duration::from_millis(10));

    // Acquire a write Lock. This will wait until all readers are
    // done.
    // Once held, no new readers or writers can get a lock.
    let mut config_guard = config_clone.write().unwrap();
    *config_guard = "Updated Config".to_string();
    println!("--- Writer: Updated config! ---");

    // Wait for all threads to finish
    for handle in handles {
        handle.join().unwrap();
    }

    println!("\nFinal config value: '{}'", *config.read().unwrap());
})
```

8. 스마트 포인터(8) - Todo!

- `Cow<'a T>`
- `NonNull<T>`
- `OnceCell<T>`
- `OnceLock<T>`
- `LazyLock<T>`
- `UnsafeCell<T>*`

9. Concurrency

Thread, Channel and Async Programming

9. Concurrency(1) - Threads(1)

Thread Spawn & Join

```
use std::thread;
use std::time::Duration;

fn main() {
    println!("Main thread: Starting up!");

    // Spawn a new thread
    let handle = thread::spawn(|| {
        // This code runs in the new thread
        for i in 1..=5 {
            println!("New thread: count {}", i);
            thread::sleep(Duration::from_millis(500)); // Pause for 0.5
seconds
        }
        println!("New thread: I'm done!");
    });
    // The main thread continues its work immediately
    for i in 1..=3 {
        println!("Main thread: working... {}", i);
        thread::sleep(Duration::from_millis(300)); // Pause for 0.3
seconds
    }
}
```

```
println!("Main thread: Waiting for the new thread to finish...");
// We'll see how to properly wait for the handle next.
// For now, if main exits, the spawned thread might be killed.
// To ensure the spawned thread finishes in this example, we can add a
longer sleep here,
// but using join() is the correct way.
// thread::sleep(Duration::from_secs(3)); // Temporary, to see spawned
thread output

// The correct way to wait for the spawned thread:
handle.join().unwrap(); // We'll explain join() shortly

println!("Main thread: All done!");
}
```

9. Concurrency(1) - Threads(2)

Thread Spawn & Join

```
use std::thread;
use std::time::Duration;

fn main() {
    let message = String::from("Hello from the main thread!");
    let important_number = 42;

    // The `move` keyword forces the closure to take ownership of
    // `message` and `important_number`
    let handle = thread::spawn(move || {
        println!("Spawned thread received message: '{}'", message);
        println!("Spawned thread received number: {}", important_number);
        // `message` and `important_number` are now owned by this
        // closure's environment.

        // The original variables in main are no longer accessible if they
        // were moved (like String).

        // For Copy types like i32, a copy is moved.
    });
}
```

```
// Attempting to use `message` here would cause a compile error:
// println!("Main thread still has message: {}", message); // ERROR!
// value borrowed here after move

// `important_number` was an i32, which is Copy, so a copy was moved.
// The original `important_number` in main is still valid.
println!("Main thread still has important_number: {}", important_number);

// Wait for the thread to finish
handle.join().unwrap();
println!("Main thread: Spawning thread finished.");
}
```

9. Concurrency(1) - Threads(3)

Thread function

```
use std::thread;

fn thread_function(start: u32, end: u32) {
    for i in start..end {
        println!("스레드: {}", i);
        std::thread::sleep(std::time::Duration::from_millis(100));
    }
}

fn main() {
    // thread::spawn 호출 시, 함수 포인터를 전달
    let handle = thread::spawn(|| thread_function(1, 10));

    // 메인 스레드 작업
    for i in 1..5 {
        println!("메인 스레드: {}", i);
        std::thread::sleep(std::time::Duration::from_millis(100));
    }

    // 스레드 종료 대기
    handle.join().unwrap();
}
```

```
use std::thread;

// 스레드에서 실행할 함수
fn thread_function() {
    for i in 1..10 {
        println!("스레드: {}", i);
        std::thread::sleep(std::time::Duration::from_millis(100));
    }
}

fn main() {
    // thread::spawn에 함수 포인터 전달
    let handle = thread::spawn(thread_function);

    // 메인 스레드 작업
    for i in 1..5 {
        println!("메인 스레드: {}", i);
        std::thread::sleep(std::time::Duration::from_millis(100));
    }

    // 스레드 종료 대기
    handle.join().unwrap();
}
```

9. Concurrency(1) - Threads(4)

std::thread::Builder

```
use std::thread;

fn main() {
    let builder = thread::Builder::new()
        .stack_size(4 * 1024 * 1024) // 4MB 스택 설정
        .name("CustomThread".into());
    let handle = builder.spawn(|| {
        println!("스레드 이름: {:?}", thread::current().name());
    }).unwrap();
    handle.join().unwrap();
}
```

```
use std::thread;
use std::time::Duration;

fn main() {
    let handle = thread::spawn(|| {
        println!("스레드 대기 중...");
        thread::park(); // 여기서 중단됨
        println!("스레드가 다시 실행됨!");
    });

    thread::sleep(Duration::from_secs(1));
    handle.thread().unpark(); // 중단된 스레드를 다시 실행
    handle.join().unwrap();
}
```

9. Concurrency(2) - Arc<Mutex<T>>

```
use std::sync::{Arc, Mutex};
use std::thread;

fn main() {
    // Create a counter, protected by a Mutex, and wrapped in an Arc for
    // sharing
    let counter = Arc::new(Mutex::new(0u32)); // Start with 0

    let mut handles = vec![];

    println!("Main: Initial counter value = {}", *counter.lock().unwrap());

    for i in 0..5 { // Spawn 5 threads
        let counter_clone_for_thread = Arc::clone(&counter); // Clone Arc
        // for the thread

        let handle = thread::spawn(move || {
            // Each thread will try to increment the counter 10 times
            for _ in 0..10 {
                // Acquire the lock. This blocks if another thread has it.
                let mut num_guard = counter_clone_for_thread.lock().unwrap();

```

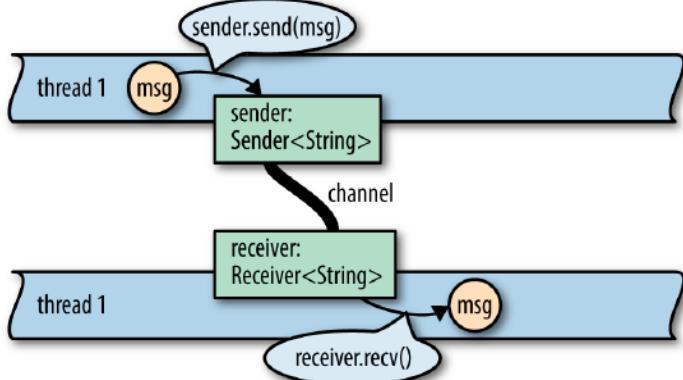
```
                // We now have exclusive mutable access to the u32 inside
                // the Mutex.

                *num_guard += 1;
                // The Lock is released automatically when num_guard goes
                // out of scope here.
            }
            println!("Thread {}: Finished incrementing.", i);
        });
        handles.push(handle);
    }

    // Wait for all threads to complete their work
    for handle in handles {
        handle.join().unwrap();
    }

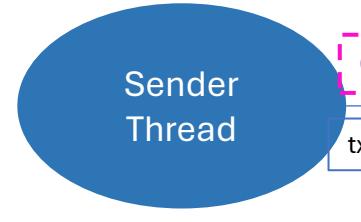
    // Lock the mutex in the main thread to read the final value
    let final_value = *counter.lock().unwrap();
    println!("Main: All threads finished. Final counter value = {}", final_value); // Expected: 50
}
```

9. Concurrency(3) - Thread Channel(1)



```
let (tx, rx) = (mpsc::Sender<String>, mpsc::Receiver<String>) = mpsc::channel();
```

Multiple Senders



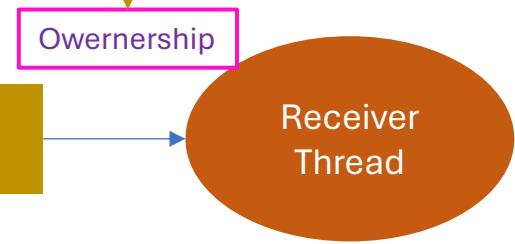
...

Sender Thread

```
tx.send(String::from"helloN").unwrap();
```

send(), recv() 결과도 Result<T, E>임
보내는 data(변수)에 대한 소유권이 이동함

Single Receiver



9. Concurrency(3) - Thread Channel(2)

```
use std::sync::mpsc;
use std::thread;
use std::time::Duration;

fn main() {
    let (tx, rx): (mpsc::Sender<String>, mpsc::Receiver<String>) =
        mpsc::channel();

    // Spawn a thread that will send messages
    let sender_thread_handle = thread::spawn(move || {
        let messages_to_send = vec![
            String::from("Greetings"),
            String::from("from"),
            String::from("the producer"),
            String::from("thread!"),
        ];

        for msg_content in messages_to_send {
            println!("Sender Thread: Preparing to send '{}'", msg_content);
            // Send the message. send() takes ownership of msg_content.
            if tx.send(msg_content).is_err() {
                eprintln!("Sender Thread: Error sending message.");
                break;
            }
        }
    });
}
```

```
    eprintln!("Sender Thread: Receiver has disconnected,
unable to send further messages.");
    break; // Exit the Loop if we can't send
}
println!("Sender Thread: Message sent successfully.");
thread::sleep(Duration::from_millis(200)); // Simulate some
work
}
println!("Sender Thread: All messages dispatched or receiver
gone.");
});

// Main thread will now try to receive.
// The receiver (rx) is still in scope here.
println!("Main Thread: Waiting for messages from sender thread...");
for received_message in rx { // rx can be used as an iterator
    println!("Main Thread: Received: '{}', received_message");
}
println!("Main Thread: Channel disconnected (all senders dropped.");

// Wait for the sender thread to finish its execution completely
sender_thread_handle.join().expect("Sender thread panicked!");
println!("Main Thread: Sender thread has joined.");
}
```

9. Concurrency(3) - Thread Channel(3)

```
use std::sync::mpsc;
use std::thread;
use std::time::Duration;

fn main() {
    // Channel for the main thread to signal the worker to start
    let (start_tx, start_rx) = mpsc::channel::<()>(); // Using unit
    // type for signaling
    // Channel for the worker to send its result back to the main
    // thread
    let (result_tx, result_rx) = mpsc::channel::<String>();
    let worker_handle = thread::spawn(move || {
        println!("Worker Thread: Initialized and waiting for the
green light...");
        // Block until a () signal is received on start_rx
        start_rx.recv().expect("Failed to receive start signal from
main thread.");
        println!("Worker Thread: Green light received! Performing
complex task...");
        thread::sleep(Duration::from_secs(1)); // Simulate some work
        let computation_result = "Task completed successfully by
worker!".to_string();
        // Send the result back to the main thread
        result_tx.send(computation_result).expect("Failed to send
result to main thread.");
        println!("Worker Thread: Result sent, finishing up.");
    });
}
```

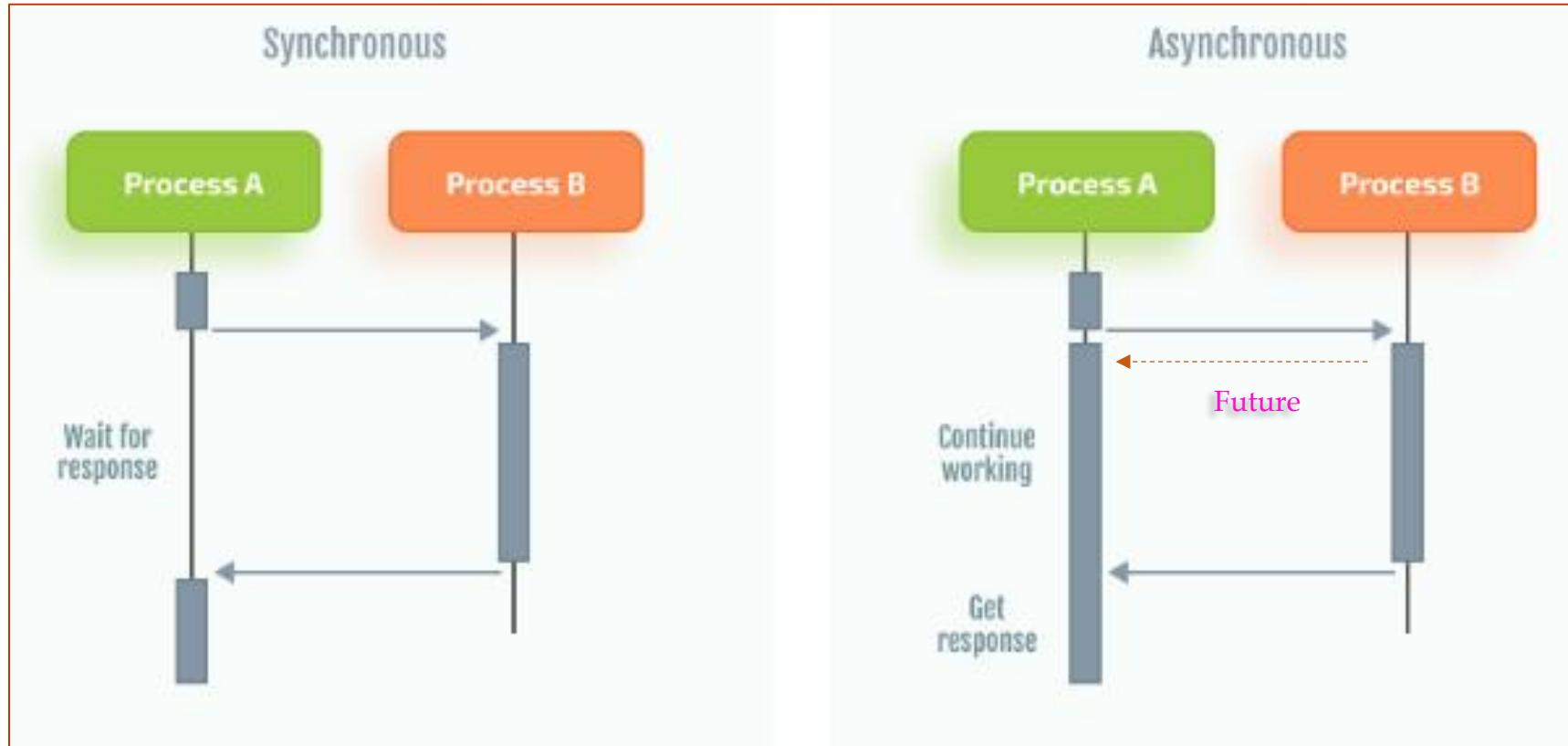
```
println!("Main Thread: Performing some setup before signaling
worker...");
thread::sleep(Duration::from_millis(500)); // Simulate setup
work

println!("Main Thread: Setup complete. Sending start signal to
worker...");
start_tx.send(()).expect("Failed to send start signal to
worker."); // Send the () signal

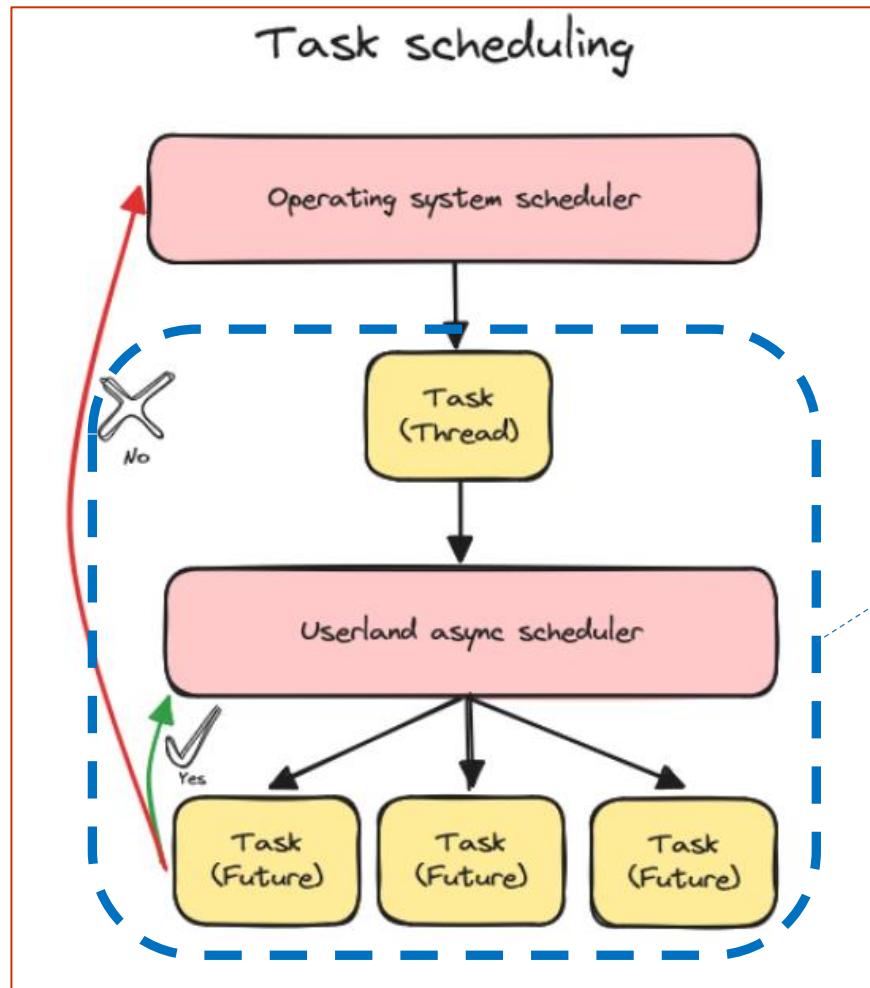
// Block and wait for the worker thread to send back its result
println!("Main Thread: Waiting for result from worker...");
let worker_output = result_rx.recv().expect("Failed to receive
result from worker.");
println!("Main Thread: Received from worker: '{}', worker_"
output);

// Ensure the worker thread has fully completed its execution
worker_handle.join().expect("Worker thread panicked during
execution!");
println!("Main Thread: Worker thread has joined. Program
exiting.");
}
```

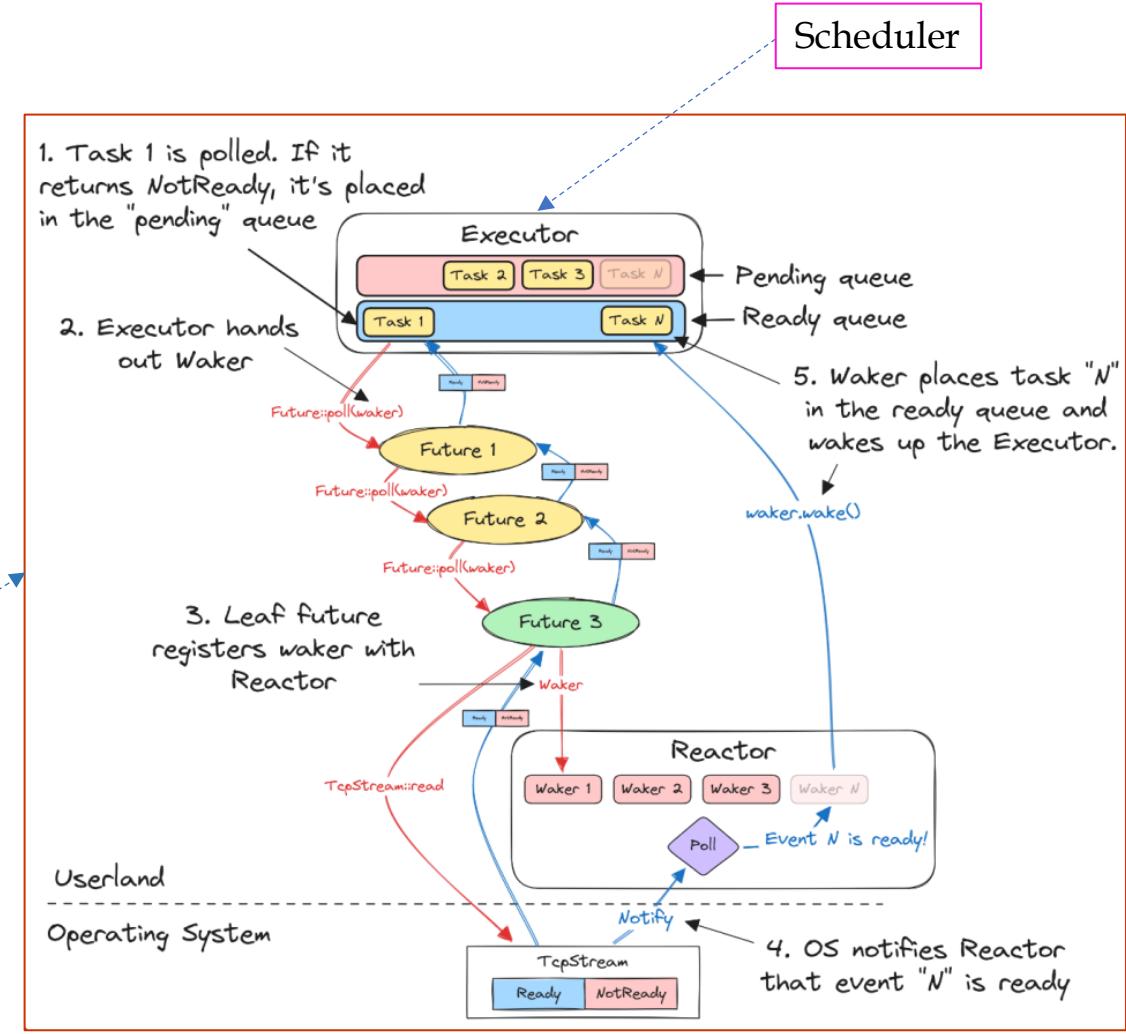
9. Concurrency(4) - Async Programming(1)



9. Concurrency(4) - Async Programming(2)



OS Thread를 직접 사용하는 방식과 달리, Green Thread를 사용한다고 함



Executor, Reactor, Future를 합하여 Runtime이라고 부른다.

9. Concurrency(4) - Async Programming(3)

Tokio(runtime) async/await

```
use tokio::time::{sleep, Duration};  
  
/// This is an async function. When called, it returns a `Future`  
/// that will resolve to a String.  
async fn fetch_simulated_data(task_id: u32) -> String {  
    println!("Task {}: Starting fetch...", task_id);  
  
    // This is an async-aware sleep.  
    // Unlike `std::thread::sleep`, this does NOT block the whole thread.  
    // It yields control back to the tokio runtime,  
    // allowing other async tasks to run.  
    sleep(Duration::from_secs(1)).await;  
  
    println!("Task {}: Finished fetch.", task_id);  
    format!("Data from task {}", task_id)  
}
```

Tokio Crate는 비동기 Programming을 위한 Runtime이다.

```
/// This function contains our main async logic.  
async fn process_tasks_sequentially() {  
    println!("Starting sequential processing...");  
  
    // We call and .await the first task.  
    // Our function's execution pauses here (non-blockingly)  
    // until `fetch_simulated_data(1)` completes.  
    let data1 = fetch_simulated_data(1).await;  
    println!("Main: Received first data: '{}', data1);  
  
    // Only *after* the first task is complete, we call and .await the  
    // second.  
    let data2 = fetch_simulated_data(2).await;  
    println!("Main: Received second data: '{}', data2);  
  
    println!("Sequential processing finished.");  
}  
  
/// The #[tokio::main] macro automatically:  
/// 1. Creates a new Tokio runtime instance.  
/// 2. Runs the `async fn main` on that runtime.  
#[tokio::main]  
async fn main() {  
    // We .await the future returned by our main logic function.  
    process_tasks_sequentially().await;  
}
```

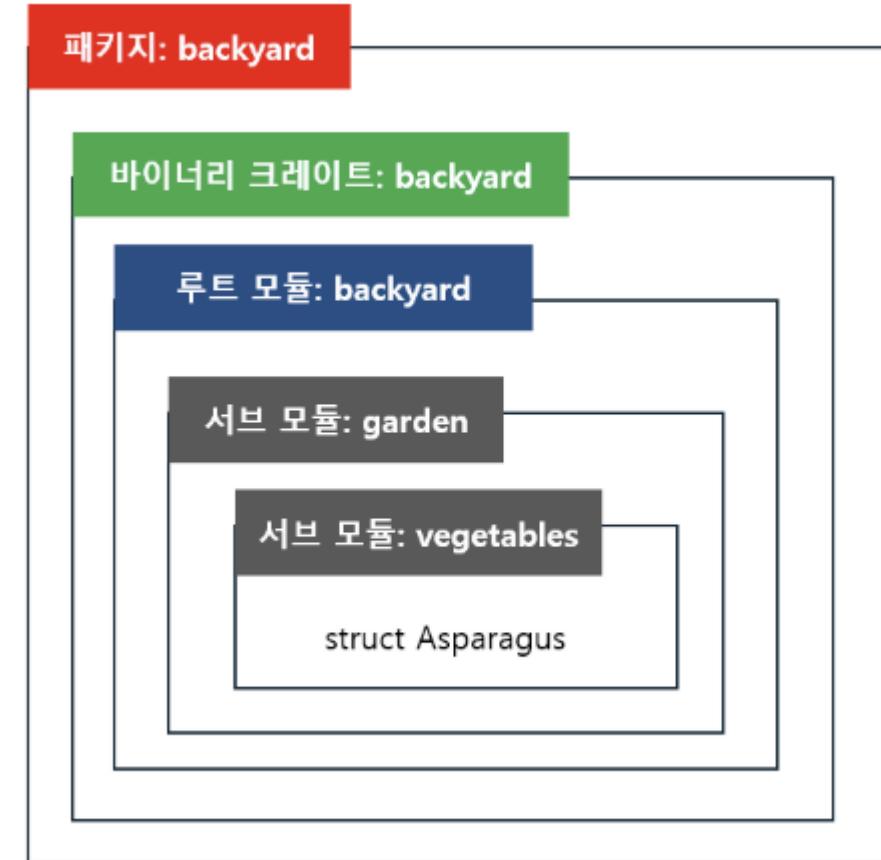
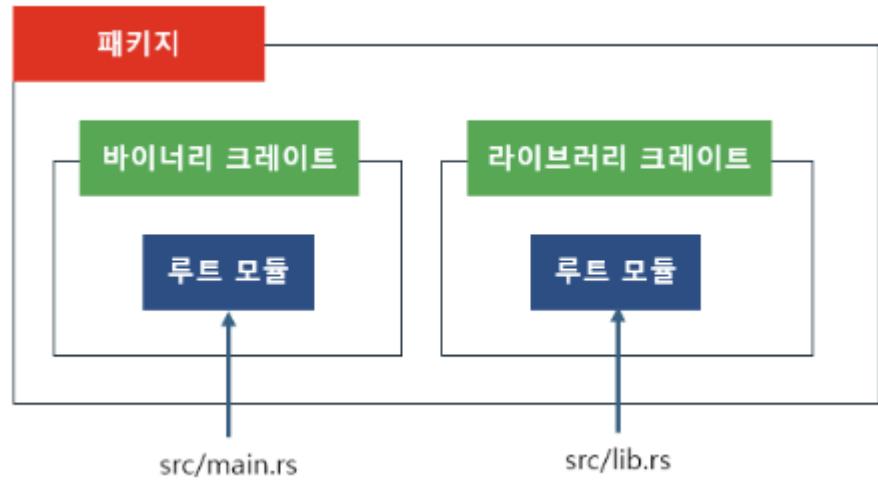
9. Concurrency(4) - Async Programming(4)

```
// This is a "handler" function.  
// It's an async function that returns something that can be  
// converted into an HTTP response. A simple &str works!  
async fn hello_world_handler() -> &'static str {  
    "Hello, Web!"  
}  
  
// We need the tokio::main macro to run our async main function  
#[tokio::main]  
async fn main() {  
    // Build our application with a single route  
    let app = Router::new().route("/", get(hello_world_handler));  
  
    // Run it  
    let listener = tokio::net::TcpListener::bind("127.0.0.1:8080")  
        .await  
        .expect("Failed to bind to address 127.0.0.1:8080");  
  
    println!("🚀 Server listening on http://{}", listener.local_addr().  
unwrap());  
  
    axum::serve(listener, app)  
        .await  
        .expect("Failed to start server");  
}
```

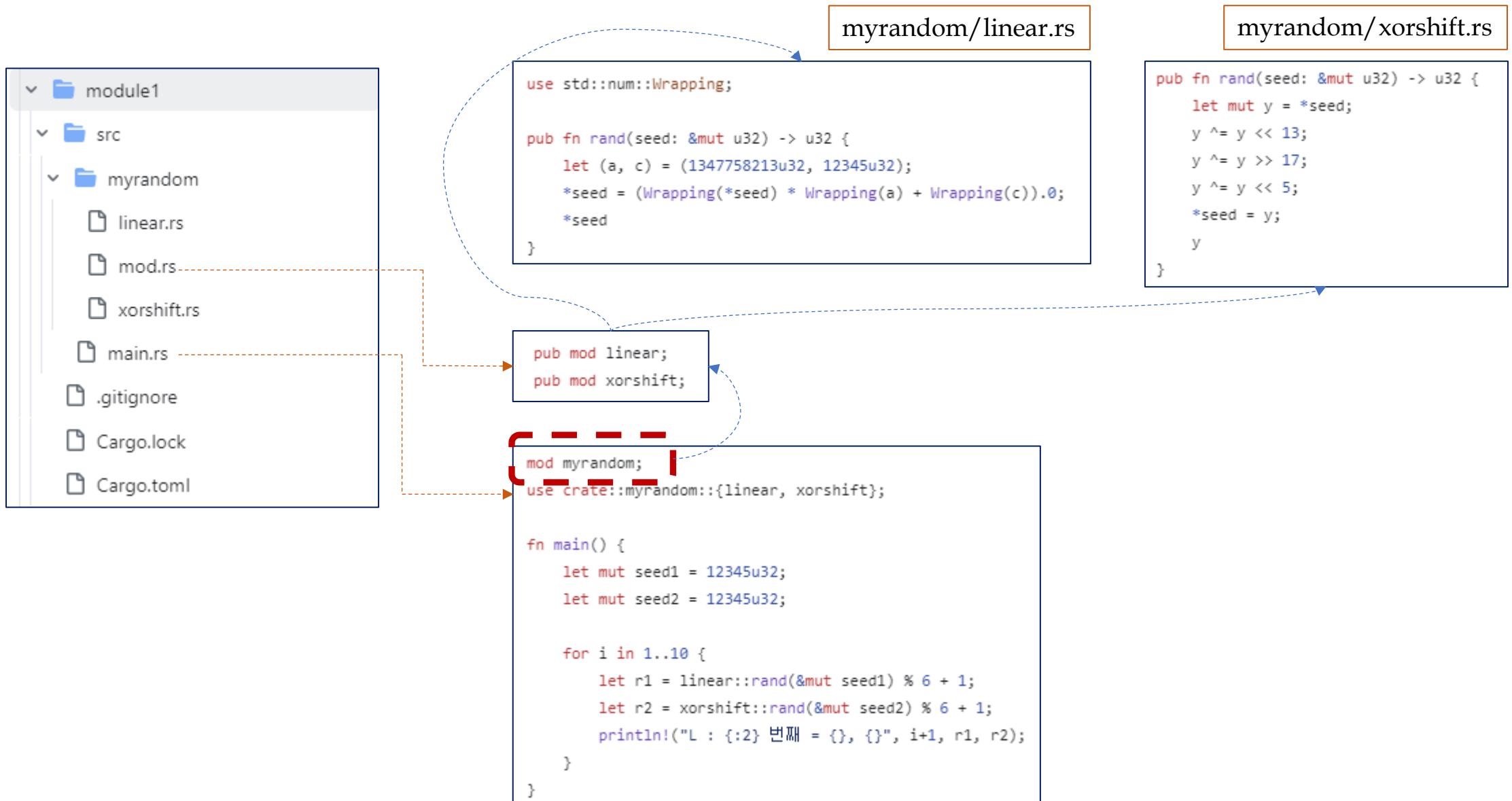
10. Package, Crate, Module

Package, Crate and Module

10. Package, Crate, Module(1)



10. Package, Crate, Module(2)



10. Package, Crate, Module(3-1)

utils/mod.rs

```
// 이 파일 자체가 'utils' 모듈의 본체가 됩니다.

// 하위 모듈 1: 문자열 도구
pub mod string_tools {
    pub fn to_uppercase(s: &str) {
        println!("[StringTools] Result: {}", s.to_uppercase());
    }
}

// 하위 모듈 2: 수학 도구
pub mod math_tools {
    pub fn add(a: i32, b: i32) -> i32 {
        a + b
    }
}
```

```
mod communication;
mod physics;
mod utils;

// =====
// [2] Main 함수
// 사용 코드는 이전과 거의 동일합니다.
// =====

fn main() {
    println!("--- Rust Multi-File Module Demo ---\n");

    // 1. 절대 경로 사용
    crate::communication::radio::broadcast("Hello via Absolute Path!");
    crate::physics::gravity::calculate_drop();

    // 2. 상대 경로 사용 (main과 communication은 같은 레벨)
    communication::network::connect();
    communication::network::check_status();

    // 3. use 키워드 사용
    use crate::utils::string_tools;
    string_tools::to_uppercase("hello use keyword");

    // 4. 별칭(Alias) 사용
    use crate::utils::math_tools::add as sum;
    println!("[Math alias] 10 + 20 = {}", sum(10, 20));

    // 5. 구조체 사용
    use crate::physics::thermodynamics::Temperature;
    let mut temp = Temperature::new(36.5);
    temp.degrees = 37.2;
    temp.show();

    // 6. 중첩 경로 import
    use crate::communication::radio::{self, broadcast};
    radio::broadcast("Nested path works!");
    broadcast("Direct function works!");
}
```

10. Package, Crate, Module(3-2)

physics/mod.rs

communication/mod.rs

```
// 이 파일 자체가 'communication' 모듈의 본체가 됩니다.

// 하위 모듈 1: 라디오
pub mod radio {
    pub fn broadcast(message: &str) {
        println!("[Radio] Broadcasting: {}", message);
    }
}

// 하위 모듈 2: 네트워크
pub mod network {
    pub fn connect() {
        println!("[Network] Connected to server.");
    }

    // 상대 경로 테스트
    pub fn check_status() {
        print!("[Network Status] ");
        // 여기서 super는 이 파일(mod.rs), 즉 communication 모듈을 가리킵니다.
        // 따라서 같은 파일 내에 있는 radio 모듈에 접근 가능합니다.
        super::radio::broadcast("Internal Check OK");
    }
}
```

```
// 이 파일 자체가 'physics' 모듈의 본체가 됩니다.

// 하위 모듈 1: 중력
pub mod gravity {
    pub fn calculate_drop() {
        println!("[Gravity] Calculating drop speed... ");
    }
}

// 하위 모듈 2: 열역학
pub mod thermodynamics {
    pub struct Temperature {
        pub degrees: f64,
        unit: String,
    }

    impl Temperature {
        pub fn new(degrees: f64) -> Temperature {
            Temperature {
                degrees,
                unit: String::from("Celsius"),
            }
        }

        pub fn show(&self) {
            println!("[Thermodynamics] Temp: {} {}", self.degrees, self.unit);
        }
    }
}
```

10. Package, Crate, Module(4)

```
mod outer {
    pub mod inner {
        pub fn greet() {
            println!("Hello from inner module");
        }

        pub fn call_self_greet() {
            // self 키워드 사용하여 현재 모듈의 함수 호출
            self::greet();
        }
    }

    pub fn call_super_greet() {
        // super 키워드를 사용하여 부모 모듈로 이동
        super::outer_greet();
    }

    pub fn outer_greet() {
        println!("Hello from outer module");
    }
}

fn main() {
    // 내부 모듈에서 자신과 부모 모듈의 함수 호출
    outer::inner::call_self_greet();
    outer::inner::call_super_greet();
}
```

10. Package, Crate, Module(5) - Test

```
// In src/lib.rs or your relevant module file
pub fn capitalize_first_letter(s: &str) -> String {
    if s.is_empty() {
        return String::new();
    }
    let mut chars = s.chars();
    match chars.next() {
        None => String::new(), // Should be covered by is_empty, but good
        for robustness
        Some(first_char) => {
            first_char.to_uppercase().to_string() + chars.as_str().to_
lowercase().as_str()
        }
    }
}
```

\$ cargo test

```
#[cfg(test)] // Only compile this module when running tests
mod tests {
    // Import items from the parent module (where capitalize_first_letter
    is)
    use super::*;

    #[test] // Marks this function as a test case
    fn test_capitalize_basic_string() {
        let input = "hello";
        let expected = "Hello";
        assert_eq!(capitalize_first_letter(input), expected);
    }

    #[test]
    fn test_capitalize_already_capitalized() {
        assert_eq!(capitalize_first_letter("World"), "World");
    }

    #[test]
    fn test_capitalize_mixed_case() {
        assert_eq!(capitalize_first_letter("rUsT"), "Rust");
    }

    #[test]
    fn test_capitalize_empty_string() {
        assert_eq!(capitalize_first_letter(""), "");
    }

    #[test]
    fn test_capitalize_single_char_string() {
        assert_eq!(capitalize_first_letter("a"), "A");
        assert_eq!(capitalize_first_letter("Z"), "Z");
    }

    #[test]
    fn test_capitalize_with_numbers_and_symbols() {
        assert_eq!(capitalize_first_letter("1st place!"), "1st Place!");
        assert_eq!(capitalize_first_letter("!leadingSymbol"),
        "!LeadingSymbol");
    }
}
```

11. File and Network I/O

File Read/Write, Socket Programming

11. File and Network I/O(1) - File Read/Write(1)

```
use std::fs::File;
use std::io::{self, BufRead, BufReader};
use std::path::Path;
use std::process; // For process::exit

// This function attempts to read and print lines from a file.
// It's similar to what our mini_grep will need to do before searching.
fn read_and_print_file_lines(file_path_str: &str) {
    let path = Path::new(file_path_str);

    // Attempt to open the file
    let file = match File::open(&path) {
        Ok(f) => f,
        Err(e) => {
            eprintln!("Error: Could not open file '{}': {}", path.
display(), e);
            process::exit(1); // Exit with error code
        }
    };
}
```

```
// Use BufReader for efficient line-by-line reading
let reader = BufReader::new(file);

println!("--- Contents of '{}' ---", path.display());
for (index, line_result) in reader.lines().enumerate() {
    match line_result {
        Ok(line_content) => {
            println!("Line {}: {}", index + 1, line_content);
        }
        Err(e) => {
            // Log error for a specific line but continue if possible,
            // or decide to exit if line read errors are critical.
            eprintln!("Error reading line {} from '{}': {}", index +
1, path.display(), e);
            // For a grep tool, we might want to skip unreadable lines
            // or halt. For now, we'll just report and continue.
        }
    }
}
println!("--- End of '{}' ---", path.display());
}
```

11. File and Network I/O(1) - File Read/Write(2)

```
fn main() {
    // Simulate getting a file path from command-line arguments
    // In a real CLI, this would come from std::env::args() or a parsing
    // crate.

    let args: Vec<String> = std::env::args().collect();

    if args.len() < 2 {
        eprintln!("Usage: {} <file_path>", args.get(0).unwrap_
        or(&"program_name".into()));
        process::exit(1);
    }
    let file_to_read = &args[1];
}
```

```
// Create a dummy file for testing if it doesn't exist
if !Path::new(file_to_read).exists() {
    if file_to_read == "sample_cli_read.txt" { // Only create if it's
        our expected test file
        std::fs::write(file_to_read, "First line for CLI test.\nSecond
        line, with a keyword.\nThird and final line.").expect("Failed to create
        sample file.");
        println!("Created sample file: {}", file_to_read);
    } else {
        eprintln!("Specified file '{}' does not exist and won't be
        auto-created for this generic example.", file_to_read);
        process::exit(1);
    }
}

read_and_print_file_lines(file_to_read);

// Clean up the dummy file if we created it for the test
if file_to_read == "sample_cli_read.txt" {
    std::fs::remove_file(file_to_read).ok();
}
```

11. File and Network I/O(2) - Socket Programming(1-1)

TCP Socket Server(1/2)

```
use std::net::{TcpListener, TcpStream};  
use std::io::{Read, Write};  
  
/// Handles a single client connection by reading from the stream and  
echoing back.  
fn handle_client(mut stream: TcpStream) -> std::io::Result<()> {  
    println!("Accepted connection from: {}", stream.peer_addr()?);  
  
    // A buffer to hold incoming data.  
    // 1024 bytes (1KB) is a common size for simple examples. In a real  
    // application, buffer size is a trade-off:  
    // - Too small (e.g., 64 bytes) can lead to many system calls, which  
is inefficient.  
    // - Too Large (e.g., 1MB) wastes memory, especially if you have many  
    // concurrent connections.  
    // Common sizes for I/O buffers are often 4KB (4096) or 8KB (8192).  
    let mut buffer = [0u8; 1024]; // A buffer to hold incoming data
```

```
// Loop to read data and echo it back  
loop {  
    // Read data from the client into the buffer  
    let bytes_read = stream.read(&mut buffer)?;  
  
    // If read() returns 0 bytes, the client has closed the connection  
    if bytes_read == 0 {  
        println!("Client disconnected.");  
        return Ok(());  
    }  
  
    // Echo the received data back to the client  
    stream.write_all(&buffer[..bytes_read])?;  
    println!("Echoed {} bytes.", bytes_read);  
}
```

11. File and Network I/O(2) - Socket Programming(1-2)

TCP Socket Server(2/2)

```
fn main() -> std::io::Result<()> {
    let listener_address = "127.0.0.1:8080";
    let listener = TcpListener::bind(listener_address)?;

    println!("Simple Echo Server listening on {}", listener_address);
    println!("Waiting for connections...");

    // listener.incoming() is an iterator that blocks until a new
    // connection arrives.

    // This loop processes one client connection fully before accepting
    // the next.
}
```

```
for stream_result in listener.incoming() {
    match stream_result {
        Ok(stream) => {
            // A new client has connected successfully.

            if let Err(e) = handle_client(stream) {
                eprintln!("Error handling client: {}", e);
            }
        }
        Err(e) => {
            // An error occurred while accepting a new connection.

            eprintln!("Failed to accept incoming connection: {}", e);
        }
    }
    ok(())
}
```

11. File and Network I/O(2) – Socket Programming(2)

TCP Socket Client

```
use std::net::TcpStream;
use std::io::{self, Write, BufRead, BufReader};

fn main() -> io::Result<()> {
    let server_address = "127.0.0.1:8080";
    println!("Connecting to echo server at {}...", server_address);

    // 1. Connect to the server. The '?' operator handles connection
    // errors concisely.
    let mut stream = TcpStream::connect(server_address)?;
    println!("Connected! Type a message and press Enter. Type 'quit' to
exit.");

    // 2. Prepare for reading and writing.
    // We can clone the stream to have separate handles for reading and
    // writing.
    // This is a common pattern for more complex I/O.
    let mut reader = BufReader::new(stream.try_clone()?);
```

```
loop {
    // Read a line of input from the user's keyboard.
    let mut input_line = String::new();
    io::stdin().read_line(&mut input_line)?;

    let message_to_send = input_line.trim(); // Trim whitespace and
newline
    if message_to_send == "quit" || message_to_send.is_empty() {
        break; // Exit loop if user types 'quit' or just presses Enter
    }

    // 3. Send the message to the server.
    // We add a newline so the server's `read_line` can process it.
    writeln!(stream, "{}", message_to_send)?;
    stream.flush()?; // Ensure the buffered data is sent immediately.

    // 4. Read the echo back from the server.
    let mut echoed_response = String::new();
    reader.read_line(&mut echoed_response)?;

    print!("Server echoed: {}", echoed_response); // `read_line`
includes the newline
}

println!("Disconnecting from server.");
Ok(())
}
```

11. File and Network I/O(3) - Command Line Parser

```
use clap::Parser; // Import the Parser trait

/// A simple program to greet a person, demonstrating clap.

#[derive(Parser, Debug)]
#[command(author = "Your Name", version = "0.1.0", about = "Greets a
person - clap example", long_about = None)]
struct CliArgs {
    /// The name of the person to greet
    #[arg(short, long)] // Allows -n <NAME> or --name <NAME>
    name: String,

    /// Number of times to greet
    #[arg(short, long, default_value_t = 1)] // Allows -c <COUNT> or
--count <COUNT>, defaults to 1
    count: u8,

    /// Optional message to include in the greeting
    #[arg(long)] // Allows --message <MESSAGE>
    message: Option<String>,

    // For our mini_grep example, it might look more like:
    // query: String,
    // file_path: std::path::PathBuf,
    // #[arg(short, long, action = clap::ArgAction::SetTrue)] // for a
flag like -i
    // ignore_case: bool,
}
```

```
[dependencies]
clap = { version = "4.5", features = ["derive"] } # Check crates.io for
the latest version
```

```
fn main() {
    // CliArgs::parse() will parse arguments from std::env::args(),
    // handle errors, and provide help/version messages automatically.
    let args = CliArgs::parse();

    for _ in 0..args.count {
        if let Some(ref msg) = args.message {
            println!("Hello, {}! Here's your message: {}", args.name,
msg);
        } else {
            println!("Hello, {}!", args.name);
        }
    }

    // If --help or --version was passed, clap handles it and exits before
this point.

    // If parsing failed, clap prints an error and exits.
}
```

12. 고급 기능

Unsafe, Macro 등

12. 고급 기능(1) - Unsafe Raw Pointer(1)

```
fn main() {
    let mut num = 5;

    // Create raw pointers from references. This part is safe.
    // r1 is an immutable raw pointer to num.
    let r1: *const i32 = &num as *const i32;
    // r2 is a mutable raw pointer to num.
    let r2: *mut i32 = &mut num as *mut i32;

    // --- WARNING: DANGEROUS OPERATION ---
    // The following code creates a raw pointer from an arbitrary memory
    // address.
    // This is NOT something you should do in normal application code.
    // Accessing random memory addresses is undefined behavior and will
    // almost certainly crash your program with a segmentation fault.
    // This is only done in very specific Low-Level programming, like
    // interacting with known, fixed hardware addresses.
    let arbitrary_address = 0x012345usize;
    let r3_arbitrary_ptr = arbitrary_address as *const i32;
    // --- END WARNING ---
```

```
unsafe {
    // Dereferencing r1 to read the value of num
    // This is safe because we know r1 was created from a valid
    reference.
    println!("Value via r1 (immutable raw pointer): {}", *r1); // Output: 5

    // Dereferencing r2 to write a new value to the memory location of
    num.
    // This is safe because r2 was also created from a valid
    reference.
    *r2 = 10; // Modifies `num` through the raw pointer
    println!("`num` has been changed via r2 to: {}", num); // Output:
    10

    // r1 still points to `num`, so it will now see the new value.
    println!("Value via r1 after change via r2: {}", *r1); // Output:
    10

    // --- DANGER: DO NOT DO THIS ---
    // Uncommenting the line below would attempt to dereference r3_
    arbitrary_ptr.
    // This is EXTREMELY DANGEROUS because r3_arbitrary_ptr points to
    an arbitrary,
    // likely invalid, memory location. It would almost certainly
    crash your program.
    // println!("Attempting to read from arbitrary address r3: {}", *
    r3_arbitrary_ptr);
    // --- END DANGER ---
}
```

12. 고급 기능(1) - Unsafe Raw Pointer(2)

```
// Creating a null pointer.
let null_pointer: *const i32 = std::ptr::null();
let mut_null_pointer: *mut i32 = std::ptr::null_mut();

// It's crucial to check if a raw pointer is null before attempting to
derefence it.
if !null_pointer.is_null() {
    // This block will not execute because null_pointer is indeed
null.

    unsafe {
        println!("This line should not be reached: {}", *null_
pointer);
    }
} else {
    println!("null_pointer is confirmed to be null, not
derefencing.");
}
```

12. 고급 기능(2) - Macro(1)

Standard macro

format!(fmt, args...)

fmt 문자열을 이용해 String 객체를 생성한다.

```
let s = format!("Hello, {}!", "world");
println!("{}", s); // 출력: Hello, world!
```

print!(fmt, args...)

표준 출력에 포맷팅 된 문자열을 줄 바꿈 없이 출력한다.

```
print!("Hello, {}!", "world"); // 출력: Hello, world!
```

println!(fmt, args...)

표준 출력에 포맷팅 된 문자열을 출력하고 줄 바꿈을 추가한다.

```
println!("Hello, {}!", "world"); // 출력: Hello, world!\n
```

<https://wikidocs.net/blog/@laniakea/1939/>

eprint!(fmt, args...)

표준 오류 스트림에 포맷팅 된 문자열을 줄 바꿈 없이 출력한다.

```
eprint!("Error: {}", "Something went wrong!"); // 표준 오류 출력
```

eprintln!(fmt, args...)

표준 오류 스트림에 포맷팅 된 문자열을 출력하고 줄 바꿈을 추가한다.

```
eprintln!("Error: {}", "Something went wrong!"); // 표준 오류 출력 + 줄바꿈
```

write!(dst, fmt, args...)

지정된 출력 버퍼(예: 파일, 문자열 버퍼)에 포맷팅 된 문자열을 쓴다. std::io::Write 트레이트가 구현된 대상이 필요하다.

```
use std::io::Write;

let mut buf = Vec::new();

write!(&mut buf, "Hello, {}!", "world").unwrap();
println!("{:?}", buf);
// 출력: [72, 101, 108, 108, 111, 44, 32, 119, 111, 114, 108, 100, 33]
```

12. 고급 기능(2) - Macro(2)

Derived macro

```
// enable `Foo` to be Debug-printed and
// implicitly copied if needed
#[derive(Debug, Clone, Copy)]
struct Foo(usize);

fn hi(f: Foo) {}
let foo = Foo(1);

hi(foo); // moved
hi(foo); // implicit copy
hi(foo); // implicit copy
hi(foo); // implicit copy

// enable `Name` to be used as a key in a HashMap
#[derive(Eq, PartialEq, Hash)]
struct Name(String);

let name = Name("cheat sheet".into());
let mut names = std::collections::HashMap::new();

names.insert(name, ());

// enable `Letters` to be used with comparison operators
#[derive(PartialEq, PartialOrd)]
enum Letters {
    A,
    B,
    C,
}

if Letters::A < Letters::B { /* ... */ }
```

12. 고급 기능(2) - Macro(3)

```
macro_rules! my_macro {
    // 매번 => 결과
    ($x:expr) => {
        println!("Value: {}", $x);
    };
}

fn main() {
    my_macro!(5);
    my_macro!("Hello, world!");
}
```

```
[features]
default = []      # 기본적으로 활성화할 기능 (없으면 빈 배열)
feature_flag = [] # 새로운 기능 추가
```

```
$ cargo run -bin example1 -features feature_flag
```

사용자 정의 macro

```
macro_rules! multiple_print {
    ($($x:expr),*) => {
        $($println!("Value: {}", $x));*
    };
}

fn main() {
    multiple_print!(1, 2, 3);
    multiple_print!("Rust", "is", "awesome!");
}
```

선택적 코드 Compile 하기

```
macro_rules! conditional_compile {
    () => {
        #[cfg(feature = "feature_flag")]
        fn conditional_function_enabled() {
            println!("Feature flag is enabled!");
        }
    }

    #[cfg(not(feature = "feature_flag"))]
    fn conditional_function_disabled() {
        println!("Feature flag is not enabled!");
    }
}

fn main() {
    conditional_compile!();

    // 기능 플래그가 활성화되었을 때 호출되는 함수
    #[cfg(feature = "feature_flag")]
    conditional_function_enabled();

    // 기능 플래그가 비활성화되었을 때 호출되는 함수
    #[cfg(not(feature = "feature_flag"))]
    conditional_function_disabled();
}
```

12. 고급 기능(3) - Todo!

- FFI(Foreign Function Interface)
- Linux kernel programming

Thank You

©opyright 2026 Chunghan Yi, All rights reserved.

