

# Rust 프로그래밍 언어

# 강사 소개

## 성재용



- 카이스트 전산학부 21학번 재학중
- 경기과학고등학교 졸업
- 한국정보올림피아드 금상
- Rust 프로젝트 다수 진행
- Rust 오픈소스 기여

# Rust의 짧은 역사

## 기존 프로그래밍 언어의 Memory 관리 방법

프로그래머가 직접 allocate 및 deallocate



장점

- 프로그램의 성능을 정확히 예측할 수 있음

단점

- 프로그래밍이 어려워짐

Garbage Collector를 사용



장점

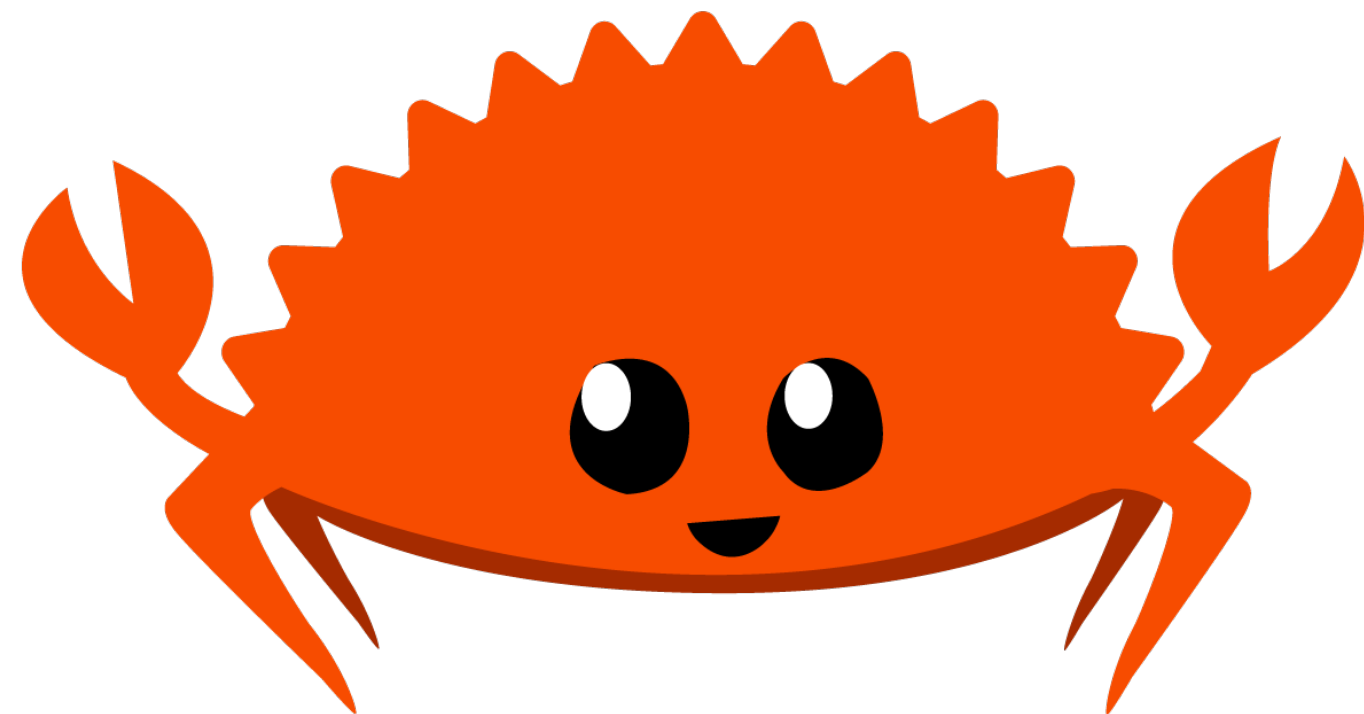
- 프로그래밍이 쉬워짐

단점

- GC가 돌아가는 만큼 프로그램이 무거워짐

# Rust의 짧은 역사

## Rust의 새로운 Memory 관리 방법



Rust는 기존의 두 방식이 아닌 다른 방식을 선택함

→ 소유권 (Ownership)

Rust에서 만나게 될 새로운 프로그래밍 개념들은 대부분 소유권에서부터 시작

Rust 프로그램은 소유권이라는 개념을 통해 컴파일에 성공하면 Memory Safety 및 Thread Safety가 보장됨

# Rust 설치

# Cargo

## Cargo란?



- Rust의 패키지 매니저
- 기본적인 기능
  - Dependency 다운로드
  - 패키지 컴파일
  - 배포 가능한 패키지 빌드
  - [crates.io](https://crates.io)에 업로드

# Rust 기본 문법

# Rust 기본 문법

## Variable

```
fn main() {  
    let x: u32 = 5;  
    x = 6; // compile error  
}
```

```
fn main() {  
    let mut x = 5;  
    x = 6; // ok  
}
```

```
fn main() {  
    let x = 5;  
    let x = x + 1; // shadowing  
    {  
        let x = x + 5;  
        println!("{x}"); // prints 11  
    }  
    println!("{x}"); // prints 6  
}
```



# Rust 기본 문법

## Primitive Types

### Integer Types (정수형)

Length	Unsigned	Signed
8-bit	u8	i8
16-bit	u16	i16
32-bit	u32	i32
64-bit	u64	i64
128-bit	u128	i128
arch	usize	isize

### Floating-Point Types (실수형)

Length	Type
32-bit	f32
64-bit	f64

### Boolean Type (불형)

Length	Type
32-bit	f32
64-bit	f64

### Character Type (문자형)

char

# Rust 기본 문법

## Strongly Typed

### Rust의 Strongly Typed

```
fn main() {  
    let a = 10u32;  
    let b: u16 = a; // compile error  
    let c = a as u64; // type conversion  
}
```

### C의 Weakly Typed

```
int main() {  
    int a = 10;  
    long b = a; // type conversion  
    return b;  
}
```

# Rust 기본 문법

## Compound Types

### Tuple Type (튜플형)

```
fn main() {  
    let tup: (i32, f64, u8) = (500, 6.4, 1);  
}
```

### Array Type (배열형)

```
fn main() {  
    let a: [i32; 5] = [1, 2, 3, 4, 5];  
}
```

# Rust 기본 문법

## Functions

### Function

```
fn main() {  
    println!("Hello, world!");  
    let x = another_function(5);  
    println!("x = {x}");  
}  
  
fn another_function(x: u32) → u32 {  
    println!("x = {x}");  
    x + 1  
}
```

### Statement

```
fn main() {  
    let x = {  
        let y = 1;  
        y  
    };  
    println!("x = {x}");  
}
```

# Rust 기본 문법 - Control Flow

## Rust 프로그래밍 언어

### if/else

```
fn main() {  
    let x = if true {  
        1  
    } else if false {  
        2  
    } else {  
        3  
    };  
    println!("x = {x}");  
}
```

### for

```
fn main() {  
    for i in 0..10 {  
        println!("i = {i}");  
    }  
}
```

```
fn main() {  
    let a = [1, 2, 3];  
  
    for i in a {  
        println!("{i}");  
    }  
}
```

# Rust 기본 문법

## Control Flow - while & loop

### while

```
fn main() {  
    let mut i = 3;  
  
    while i > 0 {  
        println!("{}", i);  
        i -= 1;  
    }  
}
```

### loop

```
fn main() {  
    let mut count = 0;  
    'counting_up: loop {  
        let mut remaining = 10;  
        loop {  
            if remaining == 9 {  
                break;  
            }  
            if count == 2 {  
                break 'counting_up;  
            }  
            remaining -= 1;  
        }  
        count += 1;  
    }  
    println!("End count = {count}");  
}
```

```
fn main() {  
    let a = loop {  
        break 1;  
    };  
}
```

# Ownership

## Ownership의 기본 개념

```
{  
    let a = String::from("hello");  
    // created here  
    ..  
} // dropped here
```

```
let b = {  
    let a = String::from("hello");  
    ..  
    a // a passes ownership to b  
}; // now b owns "hello"
```

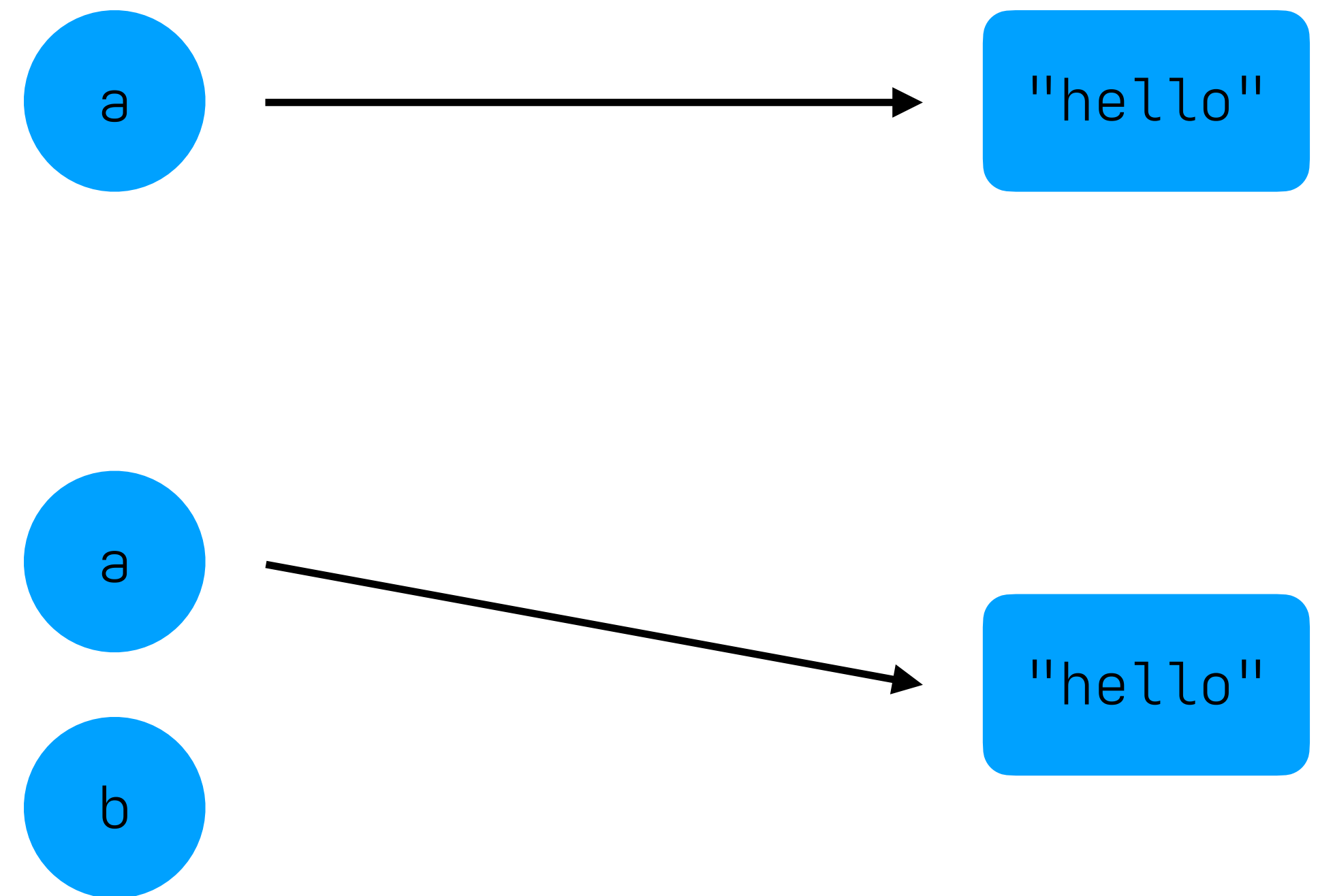
- 한 변수가 값을 소유(own)
- 각 값은 한 번에 한 소유자(owner)만 소유
- 그 변수가 scope를 벗어나면 drop (dealloc)
- 값을 넘겨주면 넘겨준 변수가 그 값을 소유
- primitive 타입은 복사

# Ownership

## Ownership의 작동 과정

```
{  
    let a = String::from("hello");  
    // created here  
    ..  
} // dropped here
```

```
let b = {  
    let a = String::from("hello");  
    ..  
    a // a passes ownership to b  
}; // now b owns "hello"
```



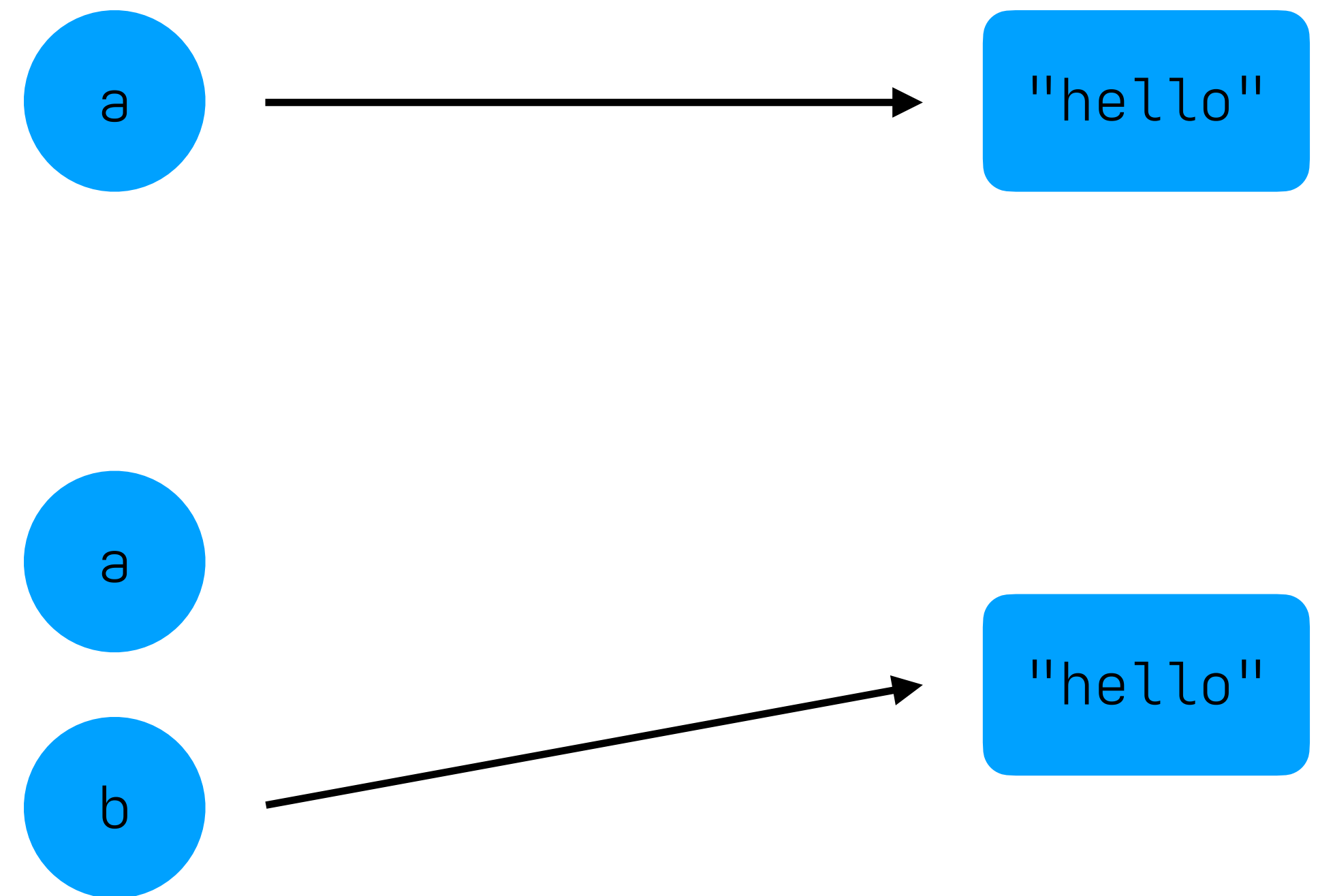


# Ownership

## Ownership의 작동 과정

```
{  
    let a = String::from("hello");  
    // created here  
    ..  
} // dropped here
```

```
let b = {  
    let a = String::from("hello");  
    ..  
    a // a passes ownership to b  
}; // now b owns "hello"
```



# Ownership

## 다른 프로그래밍 언어와의 차이점 - C & C++

### C의 변수 대입

- 모든 변수는 최대 width가 8 byte
- 모든 변수의 값은 대입될 때 copy
- 모든 변수가 Rust의 primitive type 처럼 작동

### C++의 변수 대입

- 모든 변수의 대입은 기본적으로 copy
- 비효율적인 경우가 존재하여 다른 방법들이 존재
  - `std::move`
  - l-value reference
  - r-value reference

# Ownership

## 다른 프로그래밍 언어와의 차이점 - Python 및 기타 GC 언어

### Python의 변수 대입

- 모든 변수는 포인터
- 변수 대입은 포인터 복사로 구현
- 값을 복사하기 위해 다른 방법들이 존재
  - `copy.copy`
  - `copy.deepcopy`

# Ownership

실제 메모리에서의 작동 과정



# Ownership

## Ownership 예제

```
fn main() {  
    let s1 = String::from("hello");  
    let s2 = s1;           // s1 dropped here  
    let s3 = s2.clone();   // deep copy  
  
    println!("{s1}, world!"); // compile error  
    println!("{s2}, world!"); // ok  
}
```

```
fn main() {  
    let s1 = 5;  
    let s2 = s1;           // s1 copied here (primitive type)  
  
    println!("{s1}"); // ok  
}
```

# Ownership

## Ownership 예제

```
fn main() {  
    let s1 = String::from("hello");  
    takes_ownership(s1); // s1 is now moved  
    let s2 = String::from("hello");  
    let s3 = takes_ownership_and_gives_back(s2);  
    println!("{s3}");  
}  
  
fn takes_ownership(s: String) {  
    println!("{s}");  
}  
  
fn takes_ownership_and_gives_back(s: String) → String {  
    println!("{s}");  
    s  
}
```

# Reference

## Reference 개념

```
fn main() {  
    let s1 = String::from("hello");  
    let len = calculate_length(&s1);  
    println!("{}", len);  
  
    let mut s = String::from("hello");  
    change(&mut s);  
}  
  
fn calculate_length(s: &String) -> usize {  
    s.len()  
}  
  
fn change(some_string: &mut String) {  
    some_string.push_str(", world");  
}
```

- 소유권의 개념만으로는 한 번에 한 곳에서만 접근이 가능
- 이를 위해 reference 기능 존재
- 값을 읽기만 하는 immutable reference와 값을 수정 가능한 mutable reference 두 가지 존재

# Reference

## Reference 개념

```
fn main() {  
    let mut a = String::from("hello");  
    let b = &mut a;  
    let c = a.len(); // compile error  
    b.push_str(", world");  
    println!("{}", c);  
}  
  
fn main() {  
    let mut a = String::from("hello");  
    let b = &mut a;  
    b.push_str(", world"); // b is dropped  
    let c = a.len(); // ok  
    println!("{}", c);  
}
```

- immutable reference는 제한 없이 빌려주기 가능
- 메모리 값이 변경되지 않으면 어떤 스레드에서 아무리 읽어도 memory-safety 보장
- mutable reference는 최대 1개만 빌려 주기 가능
- 메모리 값을 변경시키는 것은 최대 한 곳에서만 가능해야 memory-safety 보장
- immutable/mutable 중 하나만 빌려주기 가능



# Rust 기본 문법

## Structs

### struct 정의

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

```
struct Email(String);
```

### struct 인스턴스(instance)

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

# Rust 기본 문법

## Structs

### struct 예제

```
fn main() {  
    let email = String::from("someone@example.com");  
    let user1 = User {  
        email,  
        username: String::from("someusername123"),  
        active: true,  
        sign_in_count: 1,  
    };  
    let user2 = User {  
        email: String::from("someone@example.com"),  
        ..user1  
    };  
}
```

# Rust 기본 문법

## Derive in Struct

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

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

    println!("rect1 is {:?}", rect1);
}
```

- 자동으로 코드를 생성해주는 매크로
- Debug, Clone 등 여러 가지 존재

# Rust 기본 문법

## Methods

```
impl Rectangle {  
    fn new(width: u32, height: u32) → Rectangle {  
        Rectangle { width, height }  
    }  
  
    fn change_width(&mut self, width: u32) {  
        self.width = width;  
    }  
  
    fn area(&self) → u32 {  
        self.width * self.height  
    }  
}
```

# Rust 기본 문법

## Methods

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

```
fn main() {  
    let rect1 = Rectangle {  
        width: 30,  
        height: 50,  
    };  
  
    println!(  
        "The area of the rectangle is {} square pixels.",  
        rect1.area()  
    );  
}
```

# Rust 기본 문법

## Enums

```
enum Error {  
    Simple,  
    Description(String),  
    Detailed {  
        location: usize,  
        description: String,  
    },  
}
```

- C언어의 enum에서 발전된 기능
- 여러 종류 중 하나의 값만을 가짐
- 각 값 자체도 추가적으로 필드가 있을 수 있음

# Rust 기본 문법

## Enums

### enum 예제

```
fn main() {  
    let err1 = Error::Simple;  
    let err2 = Error::Description("division by zero".to_string());  
    let err3 = Error::Detailed {  
        location: 123,  
        description: "division by zero".to_string(),  
    };  
}
```

# Rust 기본 문법

## Control Flow - match

```
enum Coin {  
    Penny,  
    Nickel,  
    Dime,  
    Quarter,  
}  
  
fn value_in_cents(coin: Coin) → u8 {  
    match coin {  
        Coin::Penny ⇒ 1,  
        Coin::Nickel ⇒ 5,  
        Coin::Dime ⇒ 10,  
        Coin::Quarter ⇒ 25,  
    }  
}
```

- match는 어떠한 값이 맞는 지를 검사할 때 if-else chain보다 좋은 방법
- match pattern이 매우 강력  
→ 이 [문서](#)를 읽는 것을 추천
- 모든 경우에 대해 처리를 하는 코드 작성해야 함



# Rust 기본 문법

## Control Flow - if let

```
fn main() {  
    let config_max = Some(3u8);  
    match config_max {  
        Some(max) => println!("The maximum is configured to be {}", max),  
        _ => (),  
    }  
}
```

```
fn main() {  
    let config_max = Some(3u8);  
    if let Some(max) = config_max {  
        println!("The maximum is configured to be {}", max);  
    }  
}
```

# Rust 기본 문법

## Trait

```
pub trait Summary {  
    fn summarize(&self) → String;  
}  
  
pub struct NewsArticle {  
    pub headline: String,  
    pub location: String,  
    pub author: String,  
    pub content: String,  
}  
  
impl Summary for NewsArticle {  
    fn summarize(&self) → String {  
        format!("{}", by {} ({}), self.headline,  
            self.author, self.location)  
    }  
}
```

```
pub struct Tweet {  
    pub username: String,  
    pub content: String,  
    pub reply: bool,  
    pub retweet: bool,  
}  
  
impl Summary for Tweet {  
    fn summarize(&self) → String {  
        format!("{}", self.username,  
            self.content)  
    }  
}
```

# Rust 기본 문법

## Generics

```
enum Option<T> {  
    Some(T),  
    None,  
}
```

```
fn drop<T>(t: T) {}
```

```
fn hello<T: Display>(to: T) {  
    println!("Hello, {to}");  
}
```

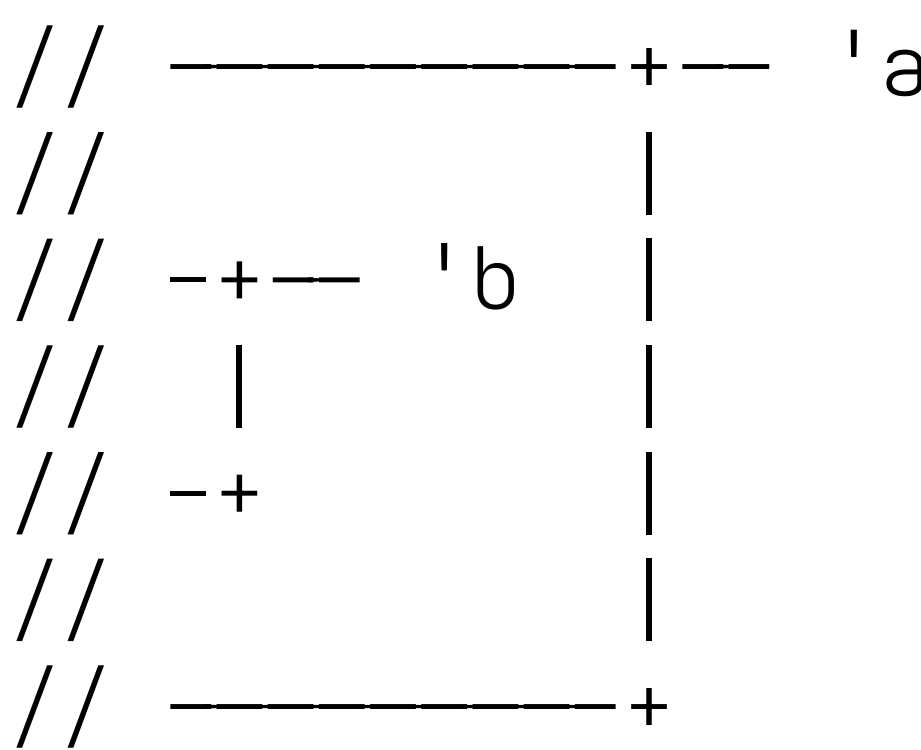
- 모던 프로그래밍 언어에서 나온 개념
- 반복되는 것을 줄여 하나의 코드로 만듦
- 데이터 구조에도 들어갈 수 있고 함수에도 들어갈 수 있음
- 특정 trait들을 만족하는 타입으로 한정 가능

# Rust 기본 문법

## Lifetimes

```
fn longest<'a>(x: &'a str, y: &'a str) → &'a str {  
    if x.len() > y.len() {  
        x  
    } else {  
        y  
    }  
}
```

```
fn main() {  
    {  
        let r;  
        {  
            let x = 5;  
            r = &x;  
        }  
        println!("r: {}", r);  
    }  
}
```



- lifetime은 reference가 살아있는 시간
- 명시를 하지 않으면 컴파일러가 알아서 구함

# Rust 기본 문법

## Closure

```
fn main() {  
    fn add_one_v1 (x: u32) → u32 { x + 1 }  
    let add_one_v2 = |x: u32| → u32 { x + 1 };  
    let add_one_v3 = |x| { x + 1 };  
    let add_one_v4 = |x| x + 1 ;  
}
```

```
fn main() {  
    let mut list = vec![1, 2, 3];  
    println!("Before defining closure: {:?}", list);  
  
    let mut borrows_mutably = || list.push(7);  
  
    borrows_mutably();  
    println!("After calling closure: {:?}", list);  
}
```

- 익명 함수 (lambda function)
- 바깥의 환경을 capture 함
- closure 안에 들어가는 것도 borrow로 생각

# Smart Pointer

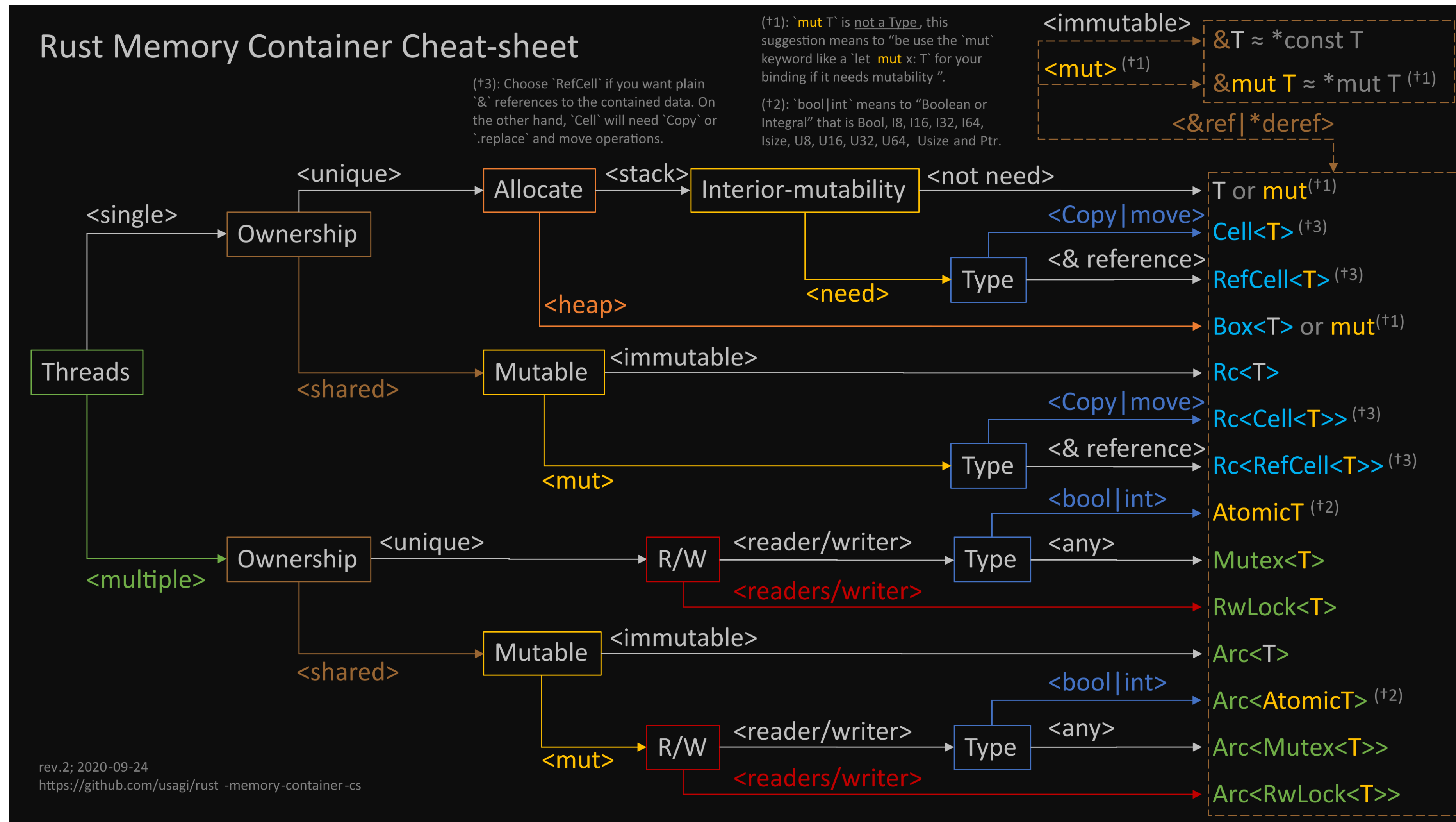
# Smart Pointer

## Smart Pointer란?

- 포인터를 직접 관리하는 것은 어려움
- 스마트 포인터: 자동으로 포인터를 안쓰면 deallocate 하는 포인터
- 예시) C++의 `unique_ptr`, `shared_ptr`, `weak_ptr`
- 각각의 방법을 통해 포인터를 언제 deallocate 할 지 결정

# Smart Pointer

## 다양한 메모리 컨테이너





# Smart Pointer

## Smart Pointer - Box

### Box 구현

```
pub struct Box<
    T: ?Sized,
    A: Allocator = Global,
>(Unique<T>, A);
```

### Unique 구현

```
pub struct Unique<T: ?Sized> {
    pointer: NonNull<T>,
    _marker: PhantomData<T>,
}
```

### NonNull 구현

```
pub struct NonNull<T: ?Sized> {
    pointer: *const T,
}
```

- 유일하게 존재하는 포인터
- Box가 drop되면 deallocate
- 그냥 값과 Box의 차이?
  - stack/heap
  - size가 고정 (포인터 크기)

# Smart Pointer

## Smart Pointer - Box

### Box 예제

```
fn main() {  
    let mut a = Box::new(1);  
    *a = 10;  
    let b = a.clone();  
  
    let pa = Box::into_raw(a);  
    let pb = Box::into_raw(b);  
  
    println!("a = {pa:p}, b = {pb:p}");  
}
```

### 예시 출력

a = 0x6000038ec040, b = 0x6000038ec050

# Smart Pointer

## Smart Pointer - Rc

### Rc 구현

```
pub struct Rc<T: ?Sized> {  
    ptr: NonNull<RcBox<T>>,  
    phantom: PhantomData<RcBox<T>>,  
}
```

### RcBox 구현

```
struct RcBox<T: ?Sized> {  
    strong: Cell<usize>,  
    weak: Cell<usize>,  
    value: T,  
}
```

- Rc는 Reference Counting의 약자
- Strong Count와 Weak Count를 통해 Strong Count가 0이 되면 deallocate
- Rc를 clone하면 Strong Count가 1 증가
- Weak Pointer를 통해 접근은 가능
- Cyclic Reference가 있을 때 Weak를 사용

# Smart Pointer

## Smart Pointer - Rc

### Rc 예시

```
fn main() {  
    let a = Rc::new(1);  
    let b = a.clone();  
    let c = Rc::downgrade(&a);  
  
    let pa = Rc::as_ptr(&a);  
    let pb = Rc::as_ptr(&a);  
    let pc = c.as_ptr();  
  
    println!("a = {pa:p}, b = {pb:p}, c = {pc:p}");  
}
```

### 예시 출력

```
a = 0x600002cbd1b0,  
b = 0x600002cbd1b0,  
c = 0x600002cbd1b0
```

# Smart Pointer

## Smart Pointer - RefCell

### RefCell 구현

```
pub struct RefCell<T: ?Sized> {  
    borrow: Cell<BorrowFlag>,  
    value: UnsafeCell<T>,  
}
```

### BorrowFlag 구현

```
type BorrowFlag = isize;  
const UNUSED: BorrowFlag = 0;
```

```
fn is_writing(x: BorrowFlag) → bool {  
    x < UNUSED  
}
```

```
fn is_reading(x: BorrowFlag) → bool {  
    x > UNUSED  
}
```

- RefCell은 Borrow Check를 런타임에 진행
- Mutable borrow를 최대 1번, Immutable borrow를 무제한으로 가능

# Smart Pointer

## Smart Pointer - Rc

### RefCell 예시

```
fn main() {  
    let a = RefCell::new(10);  
    let mut b = a.borrow_mut();  
    let c = a.borrow(); // error  
  
    *b = 10;  
}
```

# Macro

# Macro

## 개념

```
fn main() {  
    let a = 1;  
    println!("a = {a}");  
}
```

```
#![feature(prelude_import)]  
#[prelude_import]  
use std::prelude::rust_2021::*;  
#[macro_use]  
extern crate std;  
fn main() {  
    let a = 1;  
    {  
        ::std::io::_print(::core::fmt::Arguments::new_v1(  
            &["a = ", "\n"],  
            &[::core::fmt::ArgumentV1::new_display(&a)]  
        ));  
    };  
}
```

- 매크로는 코드를 컴파일 하기 전, 코드를 바꿔놓는 역할
- cargo expand를 통해 매크로가 어떻게 코드를 바꾸는지 확인 가능
- 매크로 뒤에 !가 붙어서 함수처럼 사용



# Macro

## macro\_rules!

```
fn main() {  
    macro_rules! add {  
        ($lhs:expr, $rhs:expr) => {{  
            $lhs + $rhs  
        }};  
    }  
  
    println!("{}", add!(1, 2));  
}
```

- 자신만의 macro를 만들 수 있음
- 변수는 앞에 \$가 붙음
- 변수는 타입이 존재

# File I/O

# File I/O

```
use std::fs::File;
use std::io::prelude::*;

fn main() → std::io::Result<()> {
    let mut file = File::create("foo.txt");
    file.write_all(b"Hello, world!");
    Ok(())
}
```

```
use std::fs::File;
use std::io::prelude::*;

fn main() → std::io::Result<()> {
    let mut file = File::open("foo.txt");
    let mut contents = String::new();
    file.read_to_string(&mut contents);
    assert_eq!(contents, "Hello, world!");
    Ok(())
}
```

- Rust에서의 파일 입출력은 매우 단순
- 추후에 Concurrency 파트에서 async로 입출력 하는 방법 배움