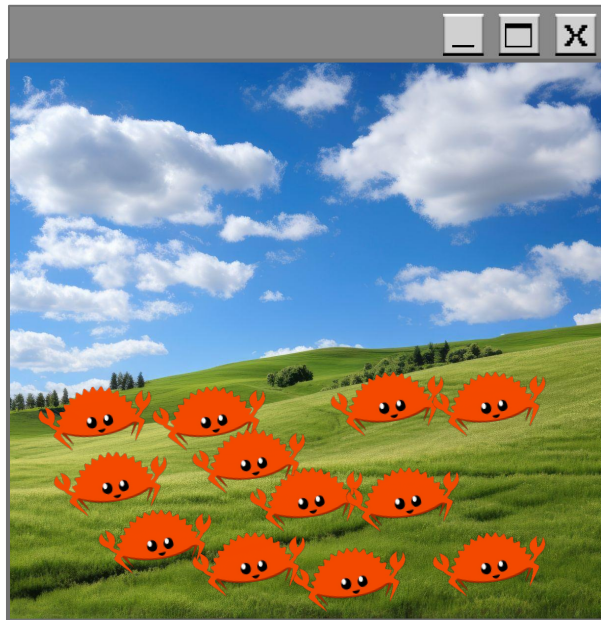


Memory management And Smart pointer In Rust

#Onership #Gc #Runtime #Atomic
#Memory #Heap #Stack
#AH_Study
#l01062506145@gmail.com #b"==" #dongjuLim



Contents

01 서론

02 메모리 관리의 기본 개념

03 Rust 기초 문법

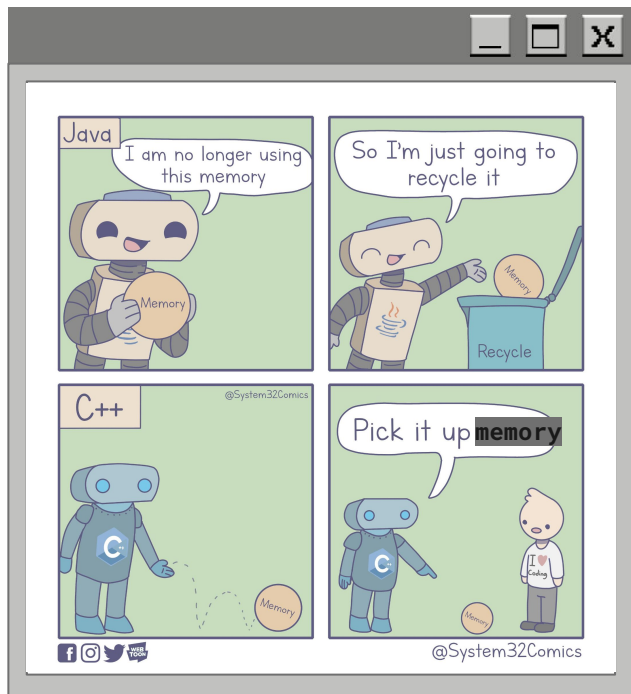
04 Rust의 소유권 시스템

05 Smart Pointer

06 벤치마크 (Big struct, Thread)

07 Bad Example

08 Q & A



1. 서론

- 메모리 관리는 소프트웨어의 성능, 안정성 및 보안의 기초
- 전통적으로 메모리 관리는 두 가지 패러다임으로 나뉨
 - Pointer 기반 수동 관리(C/C++)
 - GC 기반 자동 관리(Java, Python)
- Rust는 두 기법을 새롭게 재배치 하여 설계 됨
- 메모리 관리 메커니즘인 Pointer를 더 잘 써보자!

Memory Management's history



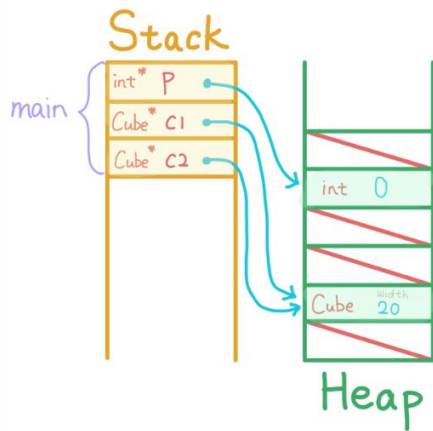
02 메모리 관리의 기본개념



#stack #heap #address #memory_leak #lifetime

2. 메모리 관리의 기본 개념

- C와 Rust에서 메모리 할당은 두 가지 영역에서 발생
 - Stack 영역 : 컴파일 시간에 정해짐
 - Last in First Out 방식
 - 함수 호출이 종료되면 해제
 - Heap 영역 : 런타임 시간에 유동적으로 변함
 - 프로그래머가 임의로 Life-time을 정할 수 있음
 - 해제 전까지는 용량을 가지고 있음
= 메모리 누수의 주원인



```
int main() {  
    int *p = new int;  
    Cube *c1 = new Cube();  
    Cube *c2 = c1;  
    c2->setLength( 10 );  
    return 0;  
}
```

2. 메모리 관리의 기본 개념

LifeCycle Example

```
fn function1() {  
    println!("function1");  
    선언 let f_memory : &str = "DATA1";  
    해제 println!("value of memory is {:?}", f_memory);  
}  
  
1개 사용 위치 신규 *  
fn function2() {  
    println!("function2");  
    선언 let mut f_memory : Box<&str> = Box::new( x: "DATA2");  
    해제 drop(f_memory);  
    // !!ERROR HERE!!  
    //println!("value of memory is {:?}", f_memory);  
}
```

Function1

- 선언과 동시에 초기화
- 사용
- (스코프에 의해) 해제

Function2

- 선언과 동시에 초기화
- 사용
- (명시적으로) 해제
- 해제가 됨으로써 사용 시 에러 발생

2. 메모리 관리의 기본 개념

Lifetime Example

```
fn lifetime() {  
    let r : &i32 ;           // 아직 초기화 X  
    {  
        let x : i32 = 5;  
        r = &x;              // 오류! r이 x를 빌리려 했으나 r보다 lifetime이 짧음  
    }                        // x는 여기서 drop됨  
    println!("{}", r);      // ??  
}
```


2. 메모리 관리의 기본 개념

Lifetime Example

```
PS C:\Users\l0106\codes\study\AH_Seminar_251026> cargo run --bin life_cycle
Compiling AH_Seminar_251026 v0.1.0 (C:\Users\l0106\codes\study\AH_Seminar_251026)
error[E0597]: `x` does not live long enough
  --> src\bin\life_cycle.rs:7:13
   |
6 |         let x:i32 = 5;
   |         - binding `x` declared here
7 |         r = &x;           //오류! r이 x를 빌리려 했으나 r보다 lifetime이 짧음
   |         ^^ borrowed value does not live long enough
8 |         // r = x;         //그래서 복사
9 |     }                     // x는 여기서 drop됨
   |     - `x` dropped here while still borrowed
10 |     println!("{}", r);    // 복사된 r이 출력됨
   |         - borrow later used here
```

2. 메모리 관리의 기본 개념

Lifetime Example

```
fn lifetime() {  
    let r:i32 ;           // 아직 초기화 X  
    {  
        let x:i32 = 5;  
        // r = &x;        //오류! r이 x를 빌려려 했으나 r보다 lifetime이 짧음  
        r = x;            //그래서 복사  
    }                     // x는 여기서 drop됨  
    println!("{}", r);    // ??  
}
```

2. 메모리 관리의 기본 개념

Lifetime Example

```
fn lifetime() {  
    let r : i32 ;           // 아직 초기화 X  
    {  
PS C:\Users\l0106\codes\study\AH_Seminar_251026> cargo run --bin life_cycle  
Compiling AH_Seminar_251026 v0.1.0 (C:\Users\l0106\codes\study\AH_Seminar_251026)  
Finished `dev` profile [unoptimized + debuginfo] target(s) in 0.23s  
Running `target\debug\life_cycle.exe`  
5  
    }                       // x는 여기서 drop됨  
    println!("{}", r);     // ??  
}
```

03 Rust 기초 문법



```
#mut_is_mutable #data_type #Option<>  
#function_and_return
```

3. Rust 기초 문법 - 선언

```
let intager:i32 = 1; // int 32bit
//intager = 2;
let unsigned_int:u32 = 10; // unsigned int 32bit
let long_int:i64; // int 64bit
let float:f32; //float 32bit

let str:&str = "some String"; //static_string
let string:String = "hello".to_string(); //dynamic_string
let char:char = 'c'; //char
let boolean:bool = true; //boolean
```

규칙

let <변수명>:<타입>;

or

let mut <변수명>:<타입>;

3. Rust 기초 문법 - mut 선언

```
let intager:i32 = 1; // int 32bit  
intager = 2;
```

3. Rust 기초 문법 - mut 선언

```
let intager:i32 = 1; // int 32bit  
intager = 2;
```

```
error[E0384]: cannot assign twice to immutable variable `intager`  
--> src\bin\basic_rust.rs:4:5  
3 |   let intager:i32 = 1; // int 32bit  
  |   ----- first assignment to `intager`  
4 |   intager = 2;  
  |   ^^^^^^^^^ cannot assign twice to immutable variable  
help: consider making this binding mutable  
3 |   let mut intager:i32 = 1; // int 32bit  
  |   +++
```

3. Rust 기초 문법 - mut 선언

```
let intager:i32 = 1; // int 32bit  
intager = 2;
```



```
let mut intager:i32 = 1; // int 32bit  
intager = 2;
```

```
error[E0384]: cannot assign twice to immutable variable `intager`  
--> src\bin\basic_rust.rs:4:5  
3 |   let intager:i32 = 1; // int 32bit  
  |   ----- first assignment to `intager`  
4 |   intager = 2;  
  |   ^^^^^^^^^^ cannot assign twice to immutable variable  
help: consider making this binding mutable  
3 |   let mut intager:i32 = 1; // int 32bit  
  |   +++
```


3. Rust 기초 문법 - Option<>

```
let option1: Option<i32> = Some(5);
let option2: Option<i32> = None;

println!("{:?}", option1);
println!("{:?}", option2);

match option1 {
    Some(x : i32 ) => println!("{}", x),
    None => println!("nothing"),
}

match option2 {
    Some(x : i32 ) => println!("{}", x),
    None => println!("nothing"),
}
```

```
Running 'target\debug\basic_rust.exe'
Some(5)
None
5
nothing
```

- Rust에는 null이 존재하지 않음
- null과 같이 존재하지 않는 값,
존재하는 값을 위한 명시적 제네릭 타입이 존재

3. Rust 기초 문법 - 함수

```
fn return_function(foo:i32) -> String {  
    // return format!("input number is : {}",foo);  
    format!("input number is : {}",foo)  
}
```

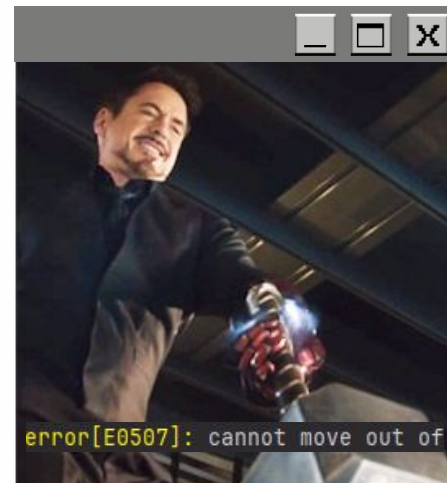
·
·
·

```
fn main() {  
    println!("{}",return_function( foo: 32));  
}
```

```
Running `target\debug\basic_rust.exe`  
input number is :32
```

```
fn <함수명>() -> <리턴 자료형> {  
    //return  
}
```

04 Rust의 소유권 시스템



#mut #move #value #ownership

4. Rust 소유권 시스템 (도입배경)

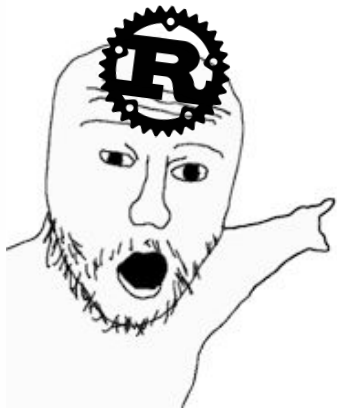
Memory safe



- C /C++ : memory 이중 해제 or 해제 누락이 발생하는 순간 예상 불가 버그 발생
- Java : 안전하지만 더 많은 자원 소모, 개발자가 필요에 따라 제어가 불가능

-> “그럼 메모리 관리에 대해 강제성을 부여하면 되는거 아닌가?”

4. Rust 소유권 시스템



Rule 1 : 모든 값은 하나의 소유자(owner) 를 가진다.

Rule 2 : 소유자가 범위를 벗어나면(drops), 값은 자동으로 해제된다

Rule 3 : 동시에 두 개의 가변 참조를 가질 수 없다. (데이터 경합 방지)

4. Rust 소유권 시스템


Rule : 모든 값은 하나의 소유자(owner) 를 가진다.

```
fn main1() {  
    let s1 : String = String::from( s: "hello");  
    let s2 : String = s1; // s1의 소유권이 s2로 이동  
    println!("{}", s1); // 에러! s1은 더 이상 유효하지 않음  
}
```

4. Rust 소유권 시스템

Rule : 모든 값은 하나의 소유자(owner) 를 가진다.

```
fn main1() {  
    let s1 : String =  
    let s2 : String = String::from( s: "hello");  
    println!("{}", s1); // 에러! s1은 더 이상 유효하지 않음  
}
```



4. Rust 소유권 시스템

```
fn main2() {  
    let s :String = String::from( s: "hi");  
    print_length(&s); // 불변 참조(대여)  
    println!("{}", s); // 여전히 사용 가능  
}
```

1개 사용 위치 신규 *

```
fn print_length(str_ref: &String) {  
    println!("len = {}", str_ref.len());  
}
```

- &는 명시적 참조(borrow)의 의미를 가짐
- print_length() 가 s를 참조만 하였기 때문에
에러가 발생 X

4. Rust 소유권 시스템

```
fn main2() {  
    let s :String = String::from( s: "hi");  
    print_length(&s); // 불변 참조(대여)  
    println!("{}", s); // 여전히 사용 가능  
}
```

1개 사용 위치 신규 *

```
fn print_length(str_ref: &String) {  
    println!("len = {}", str_ref.len());  
}
```

- &는 명시적 참조(borrow)의 의미를 가짐
- print_length() 가 s를 참조만 하였기 때문에
에러가 발생 X

```
Running `target\debug\basic_ownership.exe`  
len = 2  
hi
```

4. Rust 소유권 시스템

Rule : 소유자가 범위를 벗어나면(drops), 값은 자동으로 해제된다..?

```
fn main2() {  
    let s :String = String::from( s: "hi");  
    print_length(&s); // 불변 참조(대여)  
    println!("{}", s); // 여전히 사용 가능  
}
```

1개 사용 위치 신규 *

```
fn print_length(str_ref: &String) {  
    println!("len = {}", str_ref.len());  
}
```

4. Rust 소유권 시스템

Rule :소유자가 범위를 벗어나면(drops), 값은 자동으로 해제된다..?

```
fn main2() {  
    let s :String = String::from( s: "hi");  
    print_length(&s); // 불변 참조(대여)  
    println!("{}", s); // 여전히 사용 가능  
}
```

1개 사용 위치 신규 *

```
fn print_length(str_ref: &String) {  
    println!("len = {}", str_ref.len());  
}
```



아하! “대여” 는 해제를 하지 못하고
값이 반환 되는구나! 포인터랑
비슷할지도!?

4. Rust 소유권 시스템

```
fn main3() {
    let mut s :String = String::from( s: "hi");
    add_and_print_length1(&mut s); // 불변 참조(대여)
    println!("{}", s); // 여전히 사용 가능
    let result :String =add_and_print_length2(s); //s 는 소유권이 넘어갔으므로 사용 불가
    println!("{}", result);
}

1개 사용 위치 신규 *
fn add_and_print_length1(str_ref: &mut String) {
    (&mut *str_ref).push_str( string: " and bye");
    println!("len = {}", str_ref.len());
}

1개 사용 위치 신규 *
fn add_and_print_length2(str_ref:String) -> String {
    let mut result_str :String = str_ref;
    (&mut result_str).push_str( string: " and bye");
    println!("len = {}", result_str.len());
    result_str
}
```

/src/bin/basic_ownership



빌려갈 때 말
이라도 해줘..
아님 새로 사주던지..

- 빌려간 소유권(변수)을 조작하기 위해 &mut를 통해 명시가 필요
-> 리턴 관계가 명확히 표기되지 않아 권장X
- 허나 새 소유권을 함수가 할당하여 주는 방식도 존재함
-> 다시 할당하는 오버헤드의 가능성

05 스마트 포인터



```
#smart_pointer  
#Box<> #Rc<> #Arc #Mutex<>
```

5. 스마트 포인터



Smart pointer는 왜 필요할까?

- + 자동 해제 기능
- + 이외의 정보 저장...?

5. 스마트 포인터 in C++

```
1  #include <memory>
2  #include <iostream>
3  using namespace std;
4
5  struct Test {
6      int value;
7      Test() { cout << "malloc\n"; }
8      ~Test() { cout << "free\n"; }
9  };
10
11 int main() {
12     cout<<"start\n";
13     {
14         Test* p = new Test();
15     }                                     <- 스코프를 벗어났지만 해제X
16     cout<<"end\n";
17 }
```

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL PORTS



```
PS C:\Users\l10106\codes\study\AH_Seminar_251026> .\normal_pointer.exe
start
malloc
end
```


5. 스마트 포인터 in C++

```
1  #include <memory>
2  #include <iostream>
3  using namespace std;
4
5  struct Test {
6      int value;
7      Test() { cout << "malloc\n"; }
8      ~Test() { cout << "free\n"; }
9  };
10
11 int main() {
12     cout<<"start\n";
13     {
14         Test* p = new Test();
15     }                                     <- 스코프를 벗어났지만 해제X
16     cout<<"end\n";
17 }
```

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL PORTS

```
PS C:\Users\l0106\codes\study\AH_Seminar_251026> .\normal_pointer.exe
start
malloc
end
```

```
1  #include <memory>
2  #include <iostream>
3  using namespace std;
4
5  struct Test {
6      int value;
7      Test() { cout << "malloc\n"; }
8      ~Test() { cout << "free\n"; }
9  };
10
11 int main() {
12     cout<<"start\n";
13     {
14         Test* p = new Test();
15         delete p;                                     <- 메모리 해제 명시
16     }
17     cout<<"end\n";
18 }
```

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL PORTS

```
PS C:\Users\l0106\codes\study\AH_Seminar_251026> .\normal_pointer.exe
start
malloc
free
end
```

<- 소멸자가 호출되어 free가 출력됨.

5. 스마트 포인터 in C++

/normal_pointer.cpp

/smart_pointer.cpp

```
1  #include <memory>
2  #include <iostream>
3  using namespace std;
4
5  struct Test {
6      Test() { cout << "malloc\n"; }
7      ~Test() { cout << "free\n"; }
8  };
9
10 int main() {
11     cout<<"start\n";
12     {
13         unique_ptr<Test> p = make_unique<Test>();
14     } // 여기서 p가 스코프를 벗어나며 자동으로 delete 호출됨
15     cout<<"end\n";
16 }
17
```

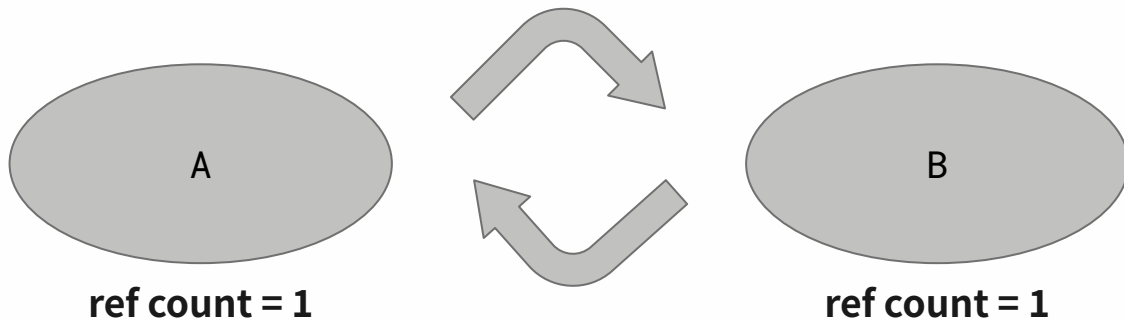
PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL PORTS powershell + - [] [X]

```
PS C:\Users\l0106\codes\study\AH_Seminar_251026> g++ -o smart_pointer .\smart_pointer.cpp
PS C:\Users\l0106\codes\study\AH_Seminar_251026> .\smart_pointer.exe
start
malloc
free
end
```

- **unique_ptr**
-> 독점 소유가 필요할 때
- **shared_ptr**
-> 참조 횟수 표기
- **weak_ptr**
-> shared_ptr를 안전하게 참조

5. 스마트 포인터

순환 참조 문제?

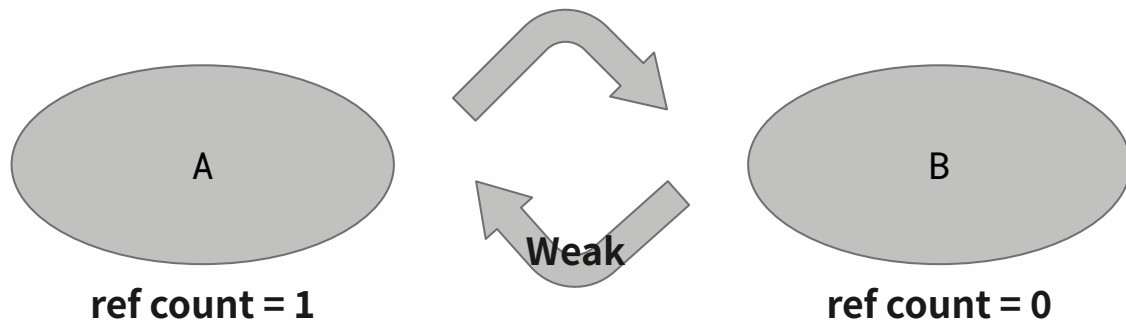


=

서로 순환되어 메모리 해제를 하지 못함 = 누수

5. 스마트 포인터

참조의 우선순위를 정하면 되는 일!



5. 스마트 포인터



- rust 코드에서는 스마트 포인터만 쓰도록 권장
- C++ 에 대비 제공되는 meta data가 많음
- 포인터 참조 또한 컴파일 단계에서 검사

5. 스마트 포인터



Smart pointer는 Rust에서 왜??? 필요할까?

5. 스마트 포인터

문제 상황: 여러 곳에서 효율적으로 **공동 데이터를 공유**하고 싶다~

```
{  
    let data :String = String::from( s: "important data");  
    let user1 :String = data; // data의 소유권이 user1으로 이동  
    let user2 :String = data; // 컴파일 에러! data는 이미 이동됨  
}
```

5. 스마트 포인터

```
// 메모리를 추가로 사용하는 경우
{
    let data :String = String::from( s: "important data");
    let user1 :String = data.clone();
    let user2 :String = data.clone();
    println!("total memory usage = {:?}",
        mem::size_of_val(&data) + mem::size_of_val(&user1) + mem::size_of_val(&user2)
    );
}
```

total memory usage = 72

5. 스마트 포인터

```
// 포인터를 사용하는 경우
{
    let data : Box<String> = Box::new(String::from( s: "important data"));
    let user1 : Box<String> = data.clone();
    let user2 : Box<String> = data.clone();
    println!("total memory usage = {:?}",
        mem::size_of_val(&data) + mem::size_of_val(&user1) + mem::size_of_val(&user2)
    );
}
```

total memory usage = 24

5. 스마트 포인터

연결 리스트를 작성할 때?

```
struct List1 {  
    value:i32, // 컴파일 에러. 무한 크기  
    next: List1  
}
```



```
struct List2 {  
    value:i32,  
    next: Box<List2> // 포인터는 고정 크기  
}
```

```
error[E0072]: recursive type `List1` has infinite size  
--> src\bin\smart_pointer.rs:98:1  
98 | struct List1 {  
   | ~~~~~  
99 |     value:i32, // 컴파일 에러. 무한 크기  
100 |     next: List1  
    |           ~---- recursive without indirection
```



5. 스마트 포인터

잠깐.. 그럼 Rust에서는 Smart 하지 않은 pointer는 못쓰나요?

5. 스마트 포인터

선언은 가능하나 역참조는 불가능

```
let mut num : i32 = 10;  
let po: *mut i32 = &mut num;  
*po = 20;  
println!("num = {}", num);
```

```
error[E0133]: dereference of raw pointer is unsafe and requires unsafe block  
--> src\bin\smart_pointer.rs:94:5
```

```
94 |     *po = 20;  
    |     ^^^ dereference of raw pointer
```

5. 스마트 포인터

하지만 unsafe로 wrapping 하면 가능은 하다 (권장 x)

```
//unsafe pointer
let mut num : i32 = 10;
let po: *mut i32 = &mut num;
unsafe {
    *po = 20;
}
println!("num = {}", num);
```

5. 스마트 포인터

5. 스마트 포인터

```
fn main_box() {  
    println!("-----Box<T>-----");  
    let mut x : Box<i32> = Box::new( x: 10); // 힙에 10 저장  
    *x +=10;  
    println!("x = {}", x); //x = 20  
    println!("-----");  
} // 자동 해제
```

```
-----Box<T>-----  
x = 20  
-----
```

Box<T>

- 가장 기본적인 스마트 포인터
- Box<T> 형태의 제네릭으로 제공
- *를 통해 값에 접근 가능 (C와 유사)

5. 스마트 포인터

```
fn main_rc() {
    println!("-----Rc<T>-----");
    // 공유 데이터 생성
    let shared_data : Rc<Vec<i32>> = Rc::new( value: vec![1, 2, 3]);
    println!("Reference count: {}", Rc::strong_count(&shared_data)); // 1

    {
        let _clone1 = Rc::clone(&shared_data);
        println!("Reference count: {}", Rc::strong_count(&shared_data)); // 2
        {
            let _clone2 = Rc::clone(&shared_data);
            println!("Reference count: {}", Rc::strong_count(&shared_data)); // 3
        }
        println!("Reference count: {}", Rc::strong_count(&shared_data)); // 2
    } // _clone1과 _clone2가 스코프를 벗어나면서 참조 카운트가 감소

    println!("Reference count: {}", Rc::strong_count(&shared_data)); // 1
    println!("-----");
}
```

Rc<T>

- 하나의 데이터를 여러곳에서 참조
- 불변 데이터를 여러 소유권에 공유하거나, 회수할 때 적합
- 단일 스레드로 제한됨

```
-----Rc<T>-----
Reference count: 1
Reference count: 2
Reference count: 3
Reference count: 2
Reference count: 1
-----
```


5. 스마트 포인터

```
fn main_arc() {
    println!("-----Arc<T>-----");
    let counter : Arc<Mutex<i32>> = Arc::new(Mutex::new(0));
    let mut handles : Vec<JoinHandle<>> = vec![];

    for i : i32 in 0..10 {
        let counter_clone : Arc<Mutex<i32>> = Arc::clone(&counter);
        let handle : JoinHandle<> = thread::spawn(move || {
            let mut num : MutexGuard<i32> = counter_clone.lock().unwrap();
            *num += 1;
            println!("Thread {i}, total = {}", *num);
        });
        (&mut handles).push(handle);
    }

    for h : JoinHandle<> in handles {
        h.join().unwrap();
    }

    println!("\ncounter = {:?}", counter.lock().unwrap());
    println!("-----");
}
```

Arc<T>

- Rc<T>의 멀티 스레드 버전
- Arc<T> 형태의 제네릭으로 제공
- Mutex<T>와 함께 사용하면 멀티 스레드에서 가변(mut) 참조 구현 가능

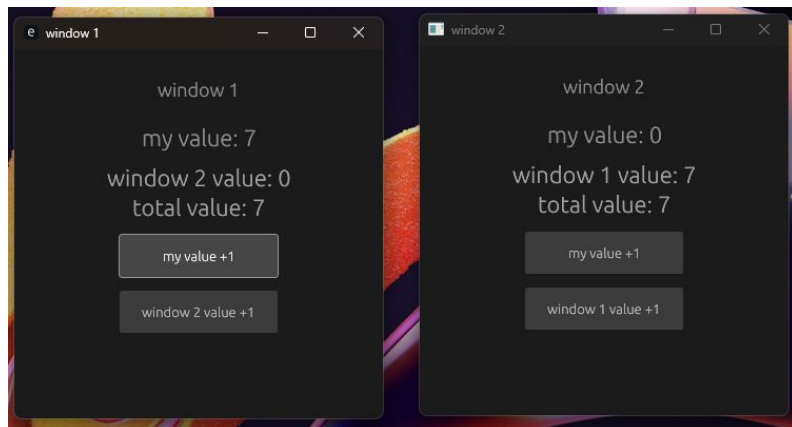
```
-----Arc<T>-----
Thread 0, total = 1
Thread 1, total = 2
Thread 5, total = 3
Thread 2, total = 4
Thread 6, total = 5
Thread 3, total = 6
Thread 4, total = 7
Thread 7, total = 8
Thread 9, total = 9
Thread 8, total = 10

counter = 10
-----
```

5. 스마트 포인터

/src/bin/value_share

ex) 1 process 2 thread share value



5. 벤치마크 (Big struct, Thread)

- 시나리오 1 : Size가 큰 Struct를 공유하기
- 시나리오 2 : 멀티 스레드로 불변의 설정을 읽기
- 시나리오 3 : Struct를 담는 캐시 활용

5. 벤치마크 (Big struct, Thread)

/src/bin/bench

5. 벤치마크 (Big struct, Thread)

시나리오 1: 큰 데이터 구조체 (1MB) - 1000회 복사

Copy: 651ms

Rc: 0ms

성능 향상: 651.00x

시나리오 2: 멀티스레드 설정 읽기 (50개 스레드)

Copy: 4ms

Arc: 2ms

성능 향상: 2.00x

시나리오 3: 캐시 조회 (10KB 엔트리, 10000회)

Copy: 8ms

Rc: 2ms

성능 향상: 4.00x

X. Result

- 스마트 포인터를 잘 쓴다면..
 - 포인터에 비해 Human error에 대해 자유롭다
 - 타입에 맞는 관리 전략을 제시한다
ex) Rc<> & Refcell<> : 명시된 메모리 해제 시점
 - Gc보다 더 적은 자원 소모

X. Result

- 스마트 포인터의 단점은?
 - 추가적인 정보에 대해 선언이 발생하여 오버헤드 발생
 - 순환 참조, 지연 해제에 대해 주의 필요
 - Mutex, Refcell 의 경우에는 완전히 안전하지 않다
(위 두 타입은 런타임 패닉을 발생 할 수 있다)
 - 가독성을 해칠 수 있는 가능성이 있다

새로운 Q&A