Rust의 소유권 이야기

Jin Hyung Park "Sigrid Jin" ('22. 4. 26.)

메모리 관리의 두 종류

명시적 메모리 관리

- 개발자가 사용한 메모리를 직접 해제하는 방식
- C/C++이 대표적으로 명시적 메모리 관리를 하는 프로그래밍 언어
- 개발자가 직접 `malloc` 와 `free` 함수를 사용해서 메모리를 동적으로 할당하고 해제

하지만...

- 명시적으로 메모리 관리를 하는 건 매우 어려운 일
- 값이 없는 변수를 참조하면 댕글링 포인터라 부르고, 메모리 해제를 두 번 하면 에러가 발생하고 개발자가 까먹고 해제를 하지 않으면 메모리 릭이 발생

자동 메모리 관리

- 메모리는 유한한 데 반해 마치 무한한 메모리가 존재하는 것 처럼 코딩할 수 있도록 한다.
- 하기 Java 코드는 마치 메모리를 무한정 사용할 수 있는 것처럼 코딩되어 있다.

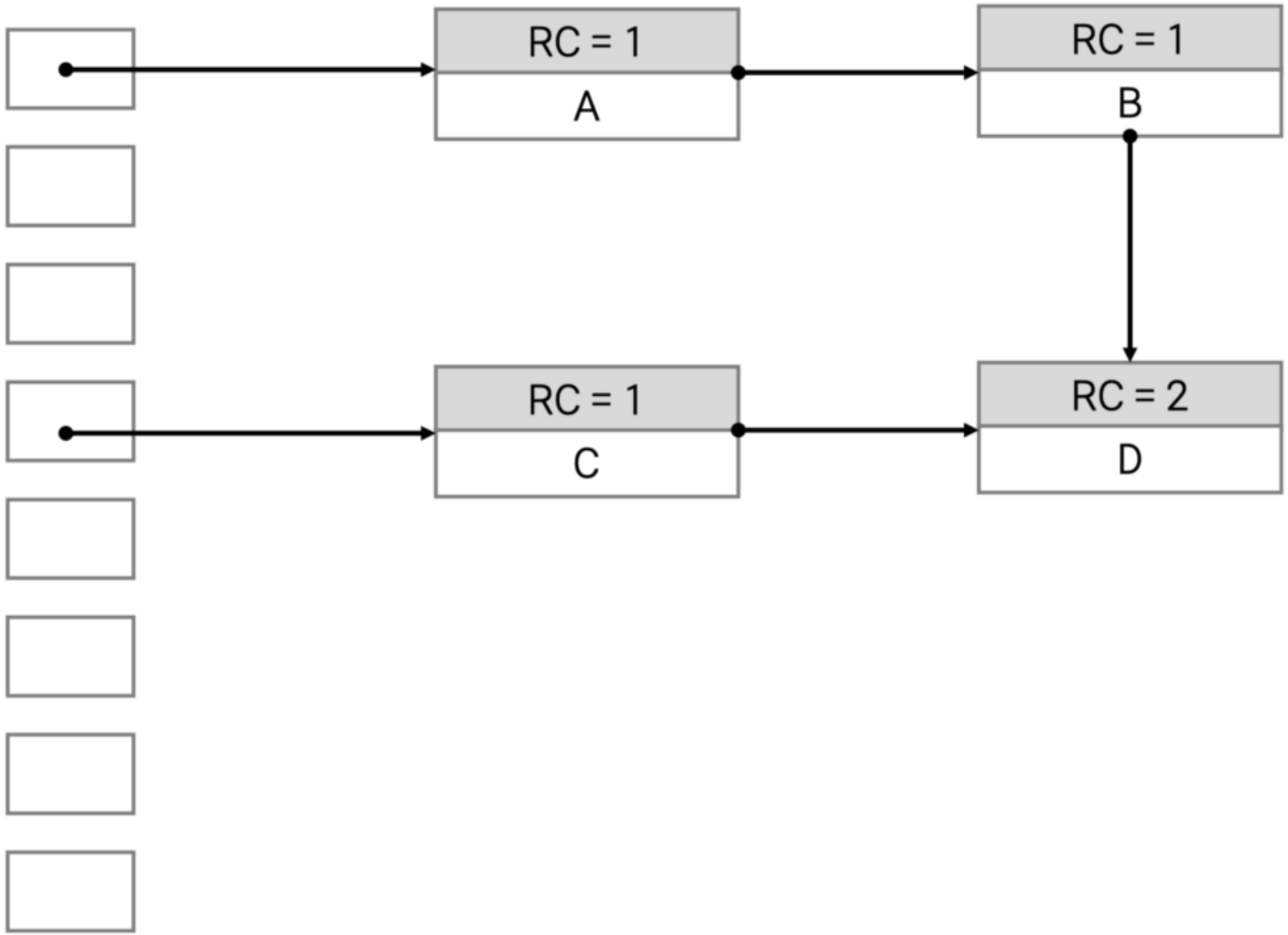
```
import java.util.UUID;
public class A {
    public static void main(String []args) {
        while (true) {
            System.out.println(new String(UUID.randomUUID().toString()));
        }
    }
}
```

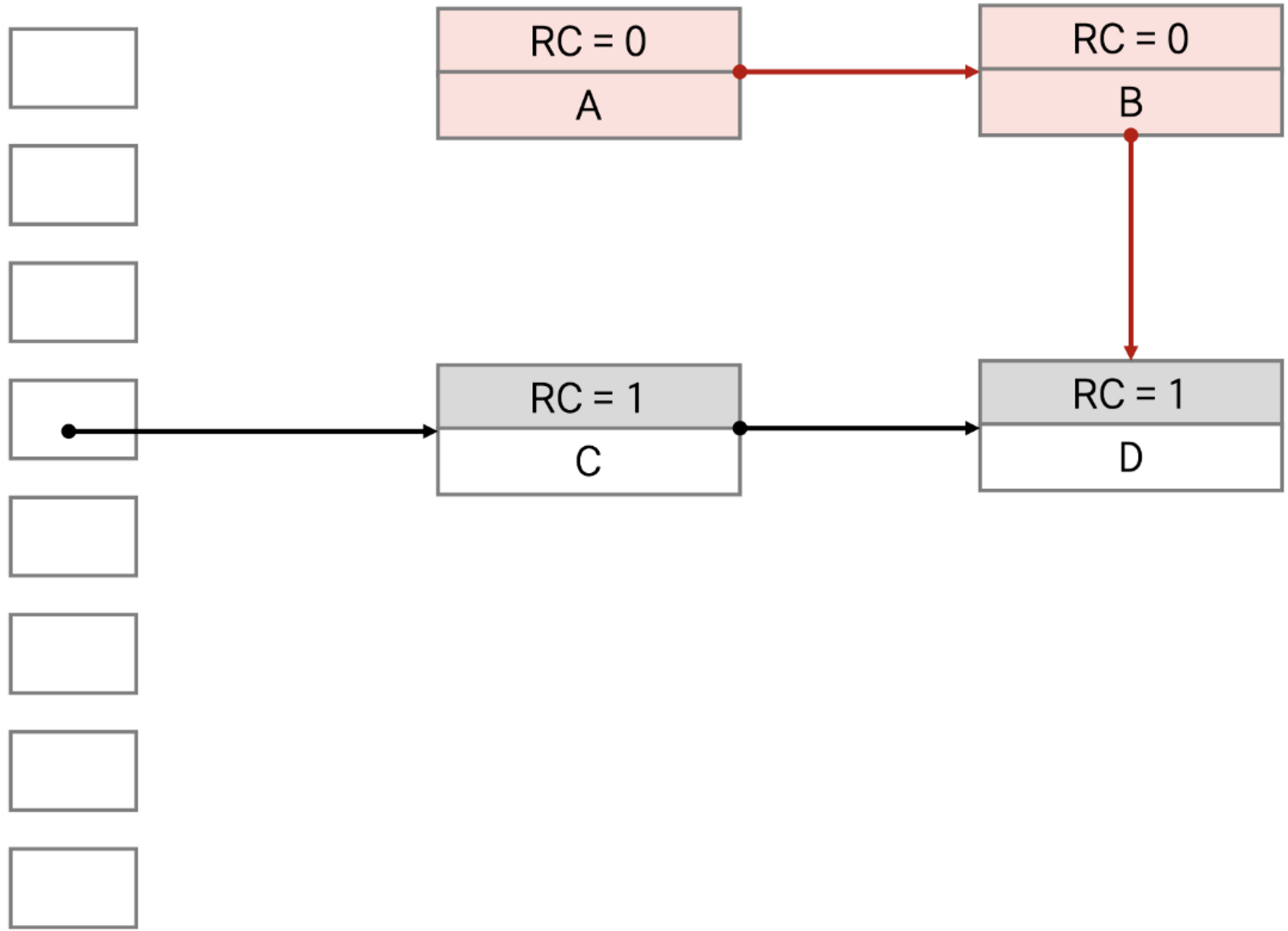
Reference Counting

- 레퍼런스 카운팅은 메모리 관리를 자동으로 하기 위한 패러다임 중 하나

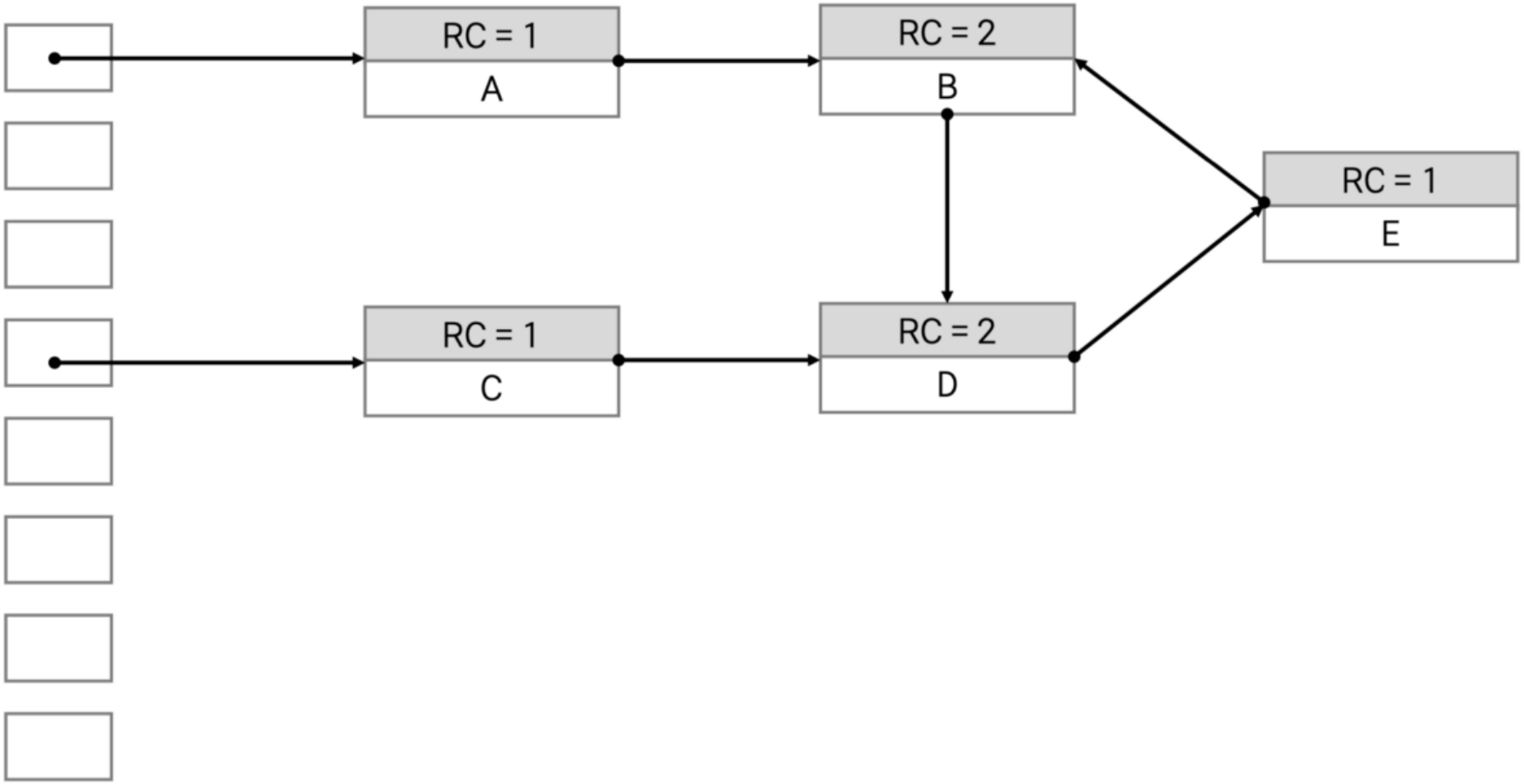
1. 모든 객체에 레퍼런스 카운트 (RC) 정보를 기입해 둡니다.
2. 변수가 참조하면 RC가 1 증가하고, 참조하지 않으면 1 감소 시킵니다.
3. RC를 감소시킬 때, 값이 0이 되면 메모리를 해제합니다.

- ex) Swift's RC

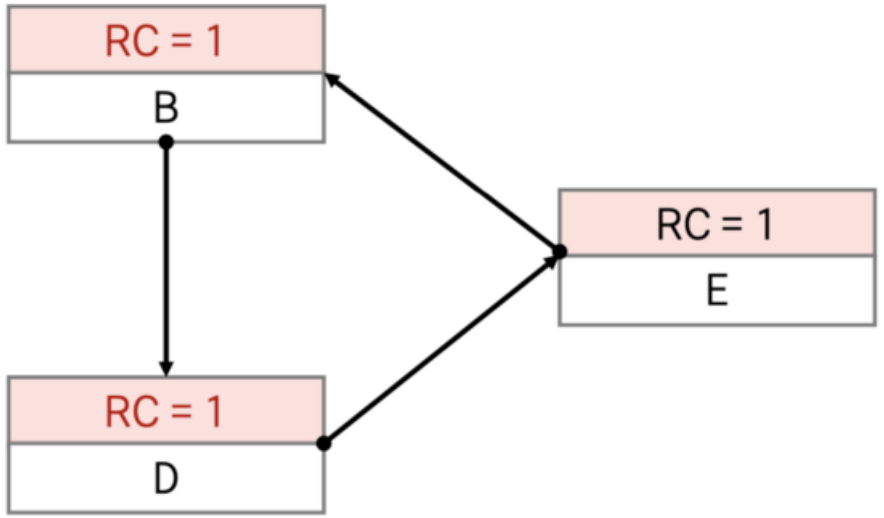

RC = 1
A
RC = 1
B
RC = 1
C
RC = 2
D


RC = 0
A
RC = 0
B
RC = 1
C
RC = 1
D

- 더블 링크드 리스트나 복잡한 그래프 구조에서는 순환 참조 (Circular Reference) 문제가 발생



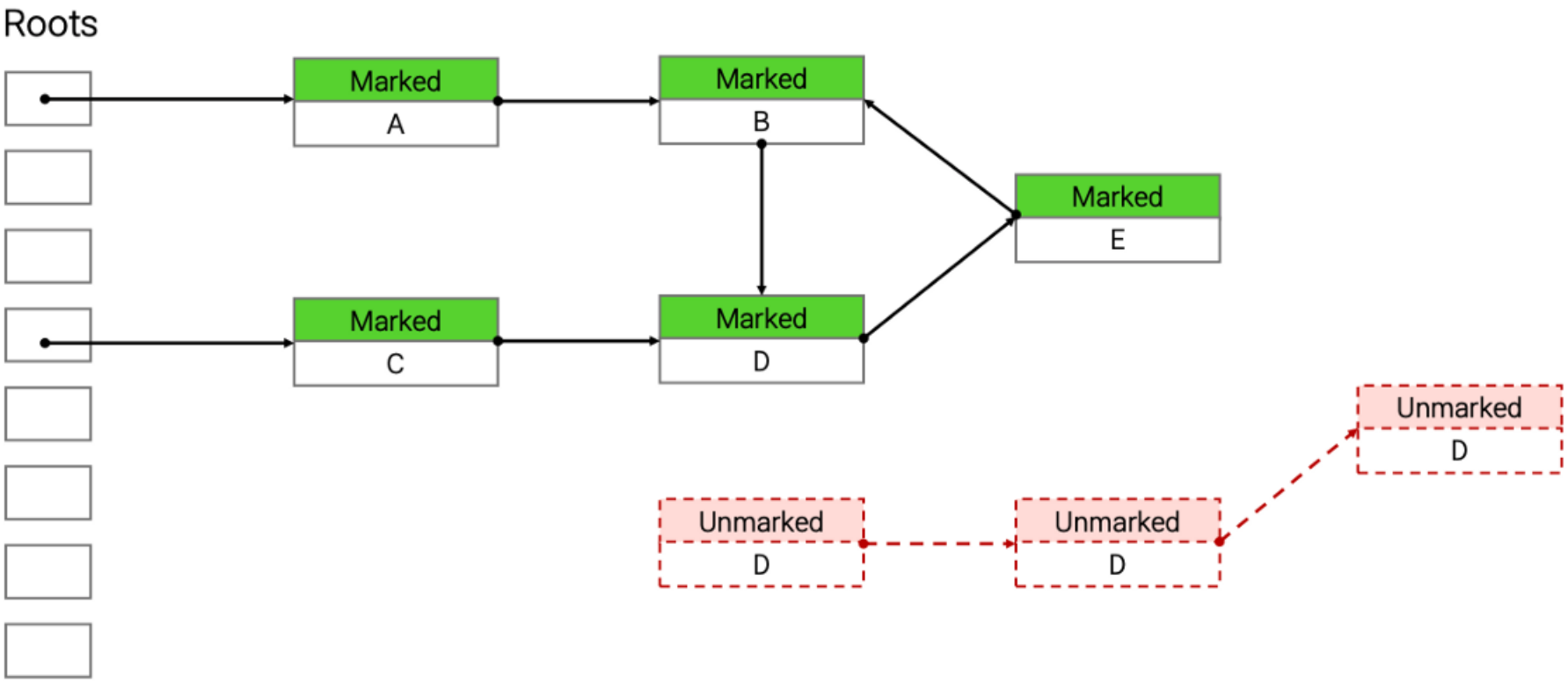
- A와 C의 참조를 없앤다면, A와 C는 각각 RC가 0이 되면서 메모리에서 해제
- 반면에 객체 B, D의 RC값은 1로 유지되면서 메모리에서 해제되지 않는 문제가 발생
- 힙에 할당된 객체를 참조하고 있는 변수가 없음에도, 메모리만 차지하는 객체가 존재하는 현상



Garbage Collector

- GC는 `mutator` 와 `collector` 두 가지 컴포넌트로 구성
- `mutator` 는 코드 실행 컨텍스트를 가지는 쓰레드
- 흔히 `collector` 를 제외한 모든 쓰레드는 `mutator`
- `collector` 는 실제 쓰지 않는 객체(=메모리)를 수거해가는 쓰레드

- `Mark Phase` | root에서 출발해서 모든 도달 가능한 객체를 Mark
- `Sweep Phase` | 모든 힙 메모리의 객체를 확인하면서 Mark 되지 않은 객체를 제거



STW: Stop The World

- 일단 쓰레기를 수집하는 동안 모든 `mutator` 쓰레드가 동작을 멈춰야 한다.
- 이유는? 중간에 힙 상태를 바꾸면서 메모리가 충돌되면 제대로 해제하지 못하기 때문이다.
- 어떤 GC 알고리즘을 사용하더라도 `STW` 는 발생한다. 대개의 경우 GC 튜닝이란 이 `STW` 시간을 줄이는 것이다.
- 쓰레기인지 아닌지 식별하기 위한 추가 정보가 객체에 붙어야 하기 때문에 공간 오버헤드도 발생
- Stop The World 때문에 성능은 더 느리겠지만 개발자가 참조 관계를 생각하지 않아도 되기 때문에 더 편리

2014 GC SLO

25% of the total CPU

Heap 2X live heap

10 ms STW pause every 50 ms

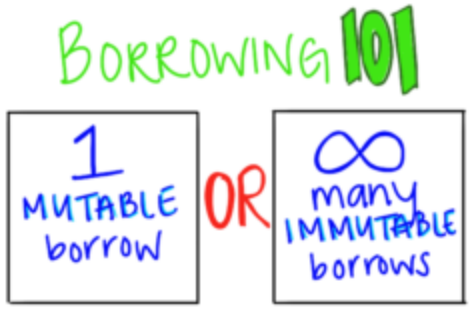
Goroutines allocation ∝ GC assists

- GC가 CPU의 25% 정도 점유하길 바라며, 50ms 마다 10ms의 STW pause를 가지도록 설계.
- 1초에 약 160ms는 메모리를 정리하는데 쓰겠다.

Ownership

- 소유권(Ownership)은 Rust의 메모리 관리 패러다임

1. 러스트에서 모든 변수는 소유자(owner)라고 불리는 변수를 가지고 있다.
2. 한 순간에 소유자는 단 하나이다.
3. 소유자가 scope를 벗어나면 해당 값은 제거된다.



- 변수 s는 스코프 A에서 생성되었다가 스코프를 벗어나면 러스트가 자동으로 변수 `s` 가 참조하는 객체의 `drop` 함수를 실행시켜서 메모리를 지운다.
- Rust는 `기초 자료형` 과 같이 크기가 고정되고 불변한 값은 보통 `스택` 에 저장
- 문자열처럼 길이가 변하는 값은 객체로 `힙` 메모리에 저장

```
fn main() {

    { // scope A
        let s = String::from("hello, world");
    }
    // 스코프를 벗어난다. s 객체의 drop 함수를 실행시켜서 메모리를 지운다.
}
```

스택 전용 데이터: 복사(copy)

- u32와 같은 모든 정수형 타입
- true, false 값만 가지는 bool 타입
- 문자타입, char
- f64와 같은 모든 부동 소수점 타입
- Copy 트레이트가 적용된 타입을 포함하는 튜플

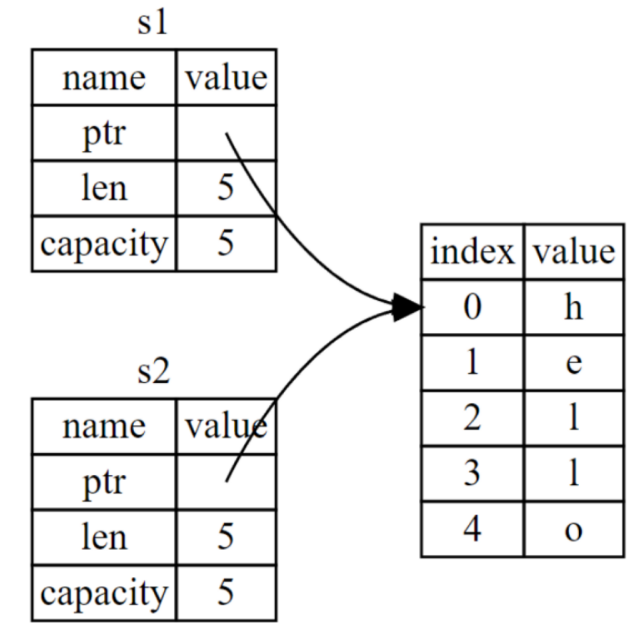
힙 메모리: 복제(clone)

```
let s1 = String::from("hello");
let s2 = s1.clone();

println!("s1 = {}, s2 = {}", s1, s2);
```

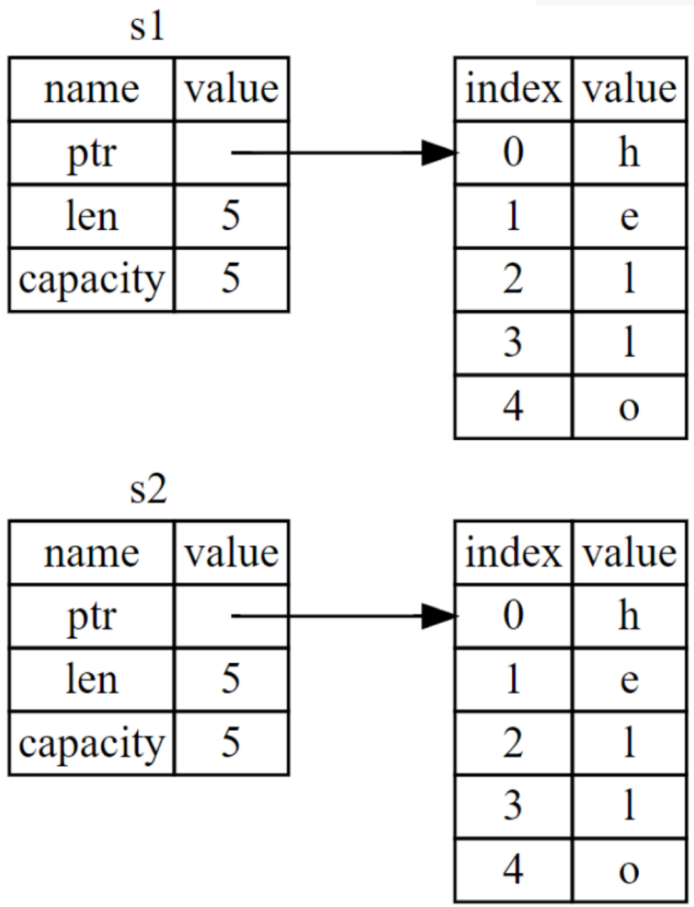
소유권의 이동

- 기초 자료형은 변수를 대입할 때 값을 복사
- 힙에 저장된 객체는 참조가 복사되어서 대입

```
let s1 = String::from("hello");
let s2 = s1;
```



- 변수를 대입할 때 참조를 복사하는게 아니라 값을 복사해버리면 동일한 객체가 여러개 존재하게 되서 매우 비효율적인 프로그램이 된다.



- 하기 함수는 `s1` 과 `s2` 변수 각각 `drop` 이 두 번 실행되어서 동일한 공간의 메모리 해제를 2번하게 되는 문제가 발생한다.
- 러스트의 해결방법은 s1을 다른 곳에 대입 시켰으면 변수 s1을 바로 `invalidate` 시킨다. 이를 소유권이 `이동 (Move)` 했다고 표현한다. 참조가 옮겨갈 뿐 `deep copy` 나 `shallow copy` 를 하지는 않는다.

```
{
    let s1 = String::from("hello");
    let s2 = s1; // 변수 s1이 들고 있단 객체의 소유자가 변한다.
    println!("{}", s1);  // value borrowed here after move
}
```

함수 파라미터에서의 소유권 이동

- `print_something` 함수는 `text` 라는 변수를 받는데, 이를 실제로 호출할 때 파라미터로 소유권을 넘겼기 때문에 변수 `s1` 을 쓰려고 하면 컴파일 에러가 발생한다.

```
fn main() {
    let s1 = String::from("Hello, World!");
    print_something(s1); // s1의 소유권이 이동함
    println!("{}", s1);  // 컴파일 에러 발생
}

fn print_something(text: String) {
    println!("{}", text)
}
```

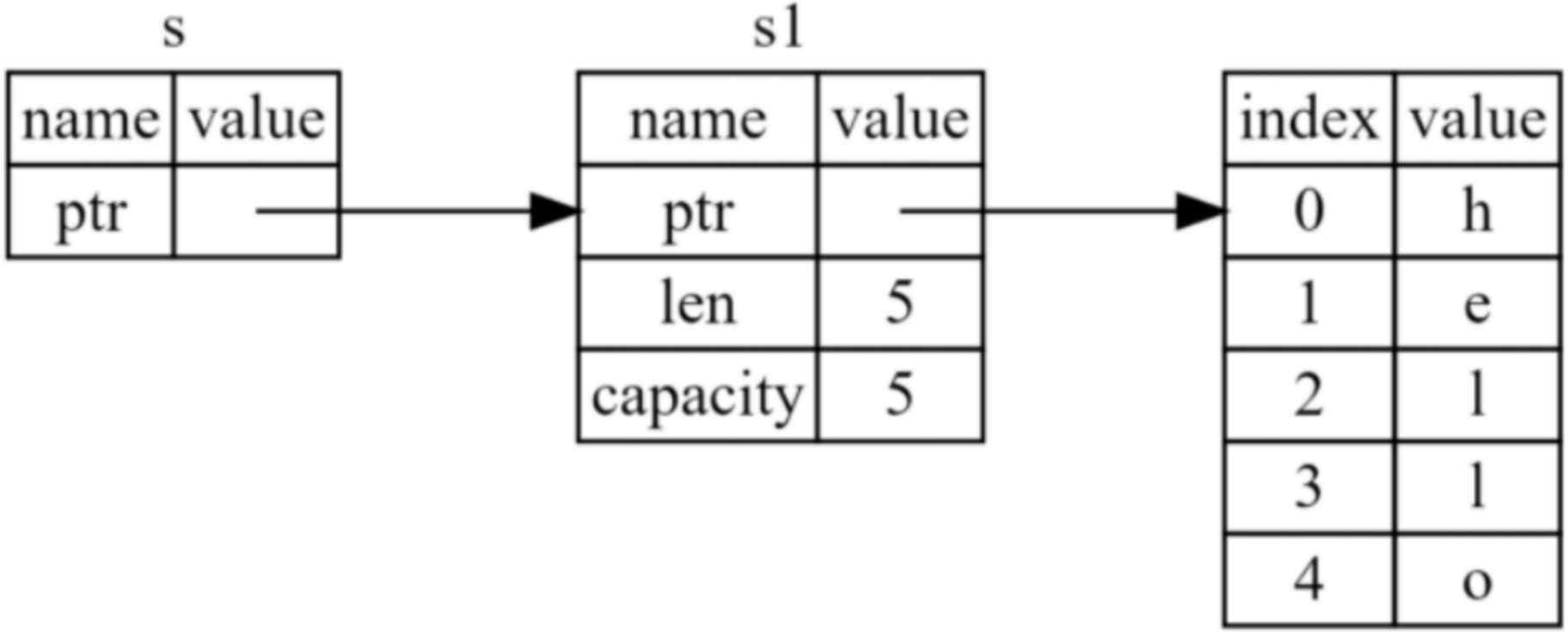
- 하기 코드를 참조하면, 함수에서 리턴 값을 이용해서 소유권을 다시 받아올 수도 있다.
- `takes` 함수는 파라미터로 받은 값을 그대로 리턴해서 소유권을 이전시킨다.
- `gives` 함수도 지역 변수로 생성한 객체의 소유권을 리턴해서 이전 시키기 때문에 에러가 발생하지 않는다.

```
fn main() {
    let s1 = gives();
    let s2 = String::from("hello");
    let s3 = takes(s2);
}

fn gives() -> String {
    let some_string = String::from("yours");
    some_string // return 구문은 생략해서 쓴다.
}

fn takes(a_string: String) -> String {
   a_string
}
```

- 레퍼런스(&, ampersand)를 생성해서 파라미터로 넘겨서 소유권을 이전시키지 않고도 사용할 수 있다.

```
fn main() {
    let s1 = String::from("hello");
    let len = len(&s1);
    println!("The length of '{}' is {}.", s1, len);
}

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

- `len` 함수는 `s` 변수의 타입을 `&String` 으로 선언해서 레퍼런스를 받겠다고 선언한다.
- 실제로 호출부에서도 `len(&s1)` 으로 변수의 레퍼런스를 생성해서 넘겨준다.
- 이 때는 소유권 이전이 발생하지 않는다.
- 실제 메모리 상에서는 `s1` 의 변수를 참조하는 객체를 생성해서 할당해주는 모습을 보인다.



- 이런 레퍼런스 생성이 가능한 이유는 변수 s의 `생명주기(Lifetime)`가 s1보다 짧기 때문에 가능하다.
- `len` 함수를 호출하면서 넘긴 파라미터의 생명주기가 s1의 `생명주기`보다 짧기 때문에 레퍼런스가 항상 값을 가지고 있다는 게 보장된다.

```
fn main() {
    let s1 = String::from("hello");                    +
    let len = len(&s1);                                |
    println!("The length of '{}' is {}.", s1, len);    |
}                                                      
                                                       + s1의 생명주기
fn len(s: &String) -> usize {
    s.len()
}
```

레퍼런스의 값을 수정하기

- 러스트는 레퍼런스로 빌려온 변수의 값을 수정하는 것을 제한한다.
- 멀티 쓰레드 환경에서는 항상 deadlock과 race condition 문제가 발생할 수 있다. 동시성 이슈 문제에서, 멀티 쓰레드를 현명하게 다루는게 더욱 중요해졌기 때문에 요즘 나오는 언어는 언어 자체 스펙에서 동시성 제어 기능을 지원한다. (예: 코루틴)
- 러스트는 하나의 변수를 변경할 수 있는 권한을 한 소유자로 제한 하기 때문에 데이터에 있어서는 race condition 문제를 제거할 수 있다. (Memory-safety over Thread-safety)

- &mut로 레퍼런스를 생성하면 값을 수정할 수 있는 레퍼런스가 만들어지고 이는 수정할 수 있다.

```
let mut x = String::from("hello");
let y = &mut x;

y.push_str(", world");

println!("x = {}", x); // x = hello, world
```

```
fn main() {
    let mut x = String::from("hello");
    let y = &mut x;

    println!("x = {}", x); // ???

    y.push_str(", world");
}
```

- 먼저 hello를 출력하고 y가 나중에 값을 바꾼다.

```
fn main() {
    let mut x = String::from("hello");
    let y = &mut x;

    println!("x = {}", x); // ???

    y.push_str(", world");
}
error[E0502]: cannot borrow `x` as immutable because it is also borrowed as mutable
 --> src/main.rs:5:24
  |
3 |     let y = &mut x;
  |             ------ mutable borrow occurs here
4 |
5 |     println!("x = {}", x); // ???
  |                        ^ immutable borrow occurs here
6 |
7 |     y.push_str(", world");
  |     -------------------- mutable borrow later used here
```

불변 참조자의 사용자는 사용중인 동안에 값이 바뀌리라 예상하지 않는다.

1. 그런데 가변 참조자인 `y` 는 문자열 값을 바꾸겠다고 선언한 바와 다를 바 없다. `y` 가 그 선언 대로 문자열 값을 바꾸면 메모리에서 새 공간에 할당한 뒤에 기존 메모리는 해제시켜야 한다. 그 이유는, 문자열의 크기가 달라졌기 때문이다.
2. 원래 변수 x가 가지고 있는 참조 객체가 새로운 공간에 할당되었다. 이 때 `println!` 함수로 먼저 사용한 값이 메모리 상에서는 이미 사라진 공간일 수도 있다.

> Moving enforces exclusion, allowing multiple threads to write to the same memory, but never at the same time. Since an owner is confined to a single thread, what happens if another thread borrows a variable?

> In Rust, you can have either one mutable borrow or as many immutable borrows as you want. You can never simultaneously have a mutable borrow and an immutable borrow (or multiple mutable borrows). When we talk about memory safety, this ensures that resources are freed properly, but when we talk about thread safety, it means that only one thread can ever modify a variable at a time.

> Furthermore, we know that no other threads will try to reference an out of date borrow —borrowing enforces either sharing or writing, but never both. 링크

Dangling Reference

> this function's return type contains a borrowed value, but there is no value for it to be borrowed from.

> s가 dangle안에서 만들어졌기 때문에, dangle의 코드가 끝이나면 s는 할당 해제됩니다. 하지만 우리는 이것의 참조자를 반환하려고 했습니다. 이는 곧 이 참조자가 어떤 무효화된 String을 가리키게 될 것이란 뜻이 아닙니까! 별로 안 좋죠. 러스트는 우리가 이런 짓을 못하게 합니다.

```
fn dangle() -> &String { // dangle은 String의 참조자를 반환합니다

    let s = String::from("hello"); // s는 새로운 String입니다

    &s // 우리는 String s의 참조자를 반환합니다.
} // 여기서 s는 스코프를 벗어나고 버려집니다. 이것의 메모리는 사라집니다.
  // 위험하군요!
```

- 여기서의 해법은 `String` 을 직접 반환하는 것입니다.
- 이 코드는 아무런 문제없이 동작합니다. 소유권이 밖으로 이동되었고, 아무것도 할당 해제되지 않습니다.

```
fn no_dangle() -> String {
    let s = String::from("hello");

    s
}
```

참조자의 규칙

어떠한 경우이든 간에 아래의 두 경우 중 하나만 가질 수 있다.

- 가. 하나의 가변 참조자
- 나. n개의 불변 참조자

참조자는 항상 유효해야만 한다.

슬라이스

```
fn first_word(s: &String) -> usize {
    let bytes = s.as_bytes();

    for (i, &item) in bytes.iter().enumerate() {
        if item == b' ' {
            return i;
        }
    }

    s.len()
}

fn main() {
    let mut s = String::from("hello world");

    let word = first_word(&s);

    s.clear();

    println!("the first word is: {}", word);
}
```

- 불변 참조자를 만들었을 경우, 가변 참조자를 만들 수 없다.

```
fn first_word_new(s: &String) -> &str {
    let bytes = s.as_bytes();

    for (i, &item) in bytes.iter().enumerate() {
        if item == b' ' {
            return &s[0..i];
        }
    }

    &s[..]
}

fn main() {
    let mut s = String::from("hello world");
    // previous borrow of `s` occurs here;
    // the immutable borrow prevents
    // subsequent moves or mutable borrows of `s` until the borrow ends

    let word = first_word_new(&s);

    s.clear(); // cannot borrow `s` as mutable because it is also borrowed as immutable [E0502]

    println!("the first word is: {}", word);
}
```

The lifetime parameter of Rust

Rust 함수의 lifetime parameter는 언제 써야 하고 언제 생략할 수 있나요?

Sigrid Jin (Jin Hyung Park) on '22. 4. 24.

소개

- 다른 언어와 마찬가지로, Rust의 오브젝트들도 lifetime을 갖는다.
- 하지만 다른 언어들과 다르게, Rust에서는 오브젝트에 대한 레퍼런스의 lifetime에 대해 컴파일 타임에 체크하고, 컴파일 에러를 발생시킨다.
- 또한 lifetime이 모호한 경우에도 컴파일 에러가 발생하는데, 이를 방지하기 위해 lifetime parameter라는 Rust만의 독특한 문법을 활용해야 한다.

Borrow Checker

- 하기 코드는 컴파일 에러가 발생한다.
- 그 이유는 무엇일까?

```
let p;

{
    let num = 10;
    p = &num;
}

println!("{}",p);
```

Borrow Checker (cont'd 1/n)

- `p`가 `num`을 가리키도록 되어 있는데, `println`이 호출되는 시점에 `num`의 `lifetime`이 만료되어 `p`가 `dangling reference`가 되었기 때문이다.
- 이런 상황에서 C와 같은 대부분의 언어는 컴파일 에러가 발생하지 않는다.
- Rust는 각 레퍼런스마다의 `lifetime`을 추적하며, `lifetime`이 만료된 시점에서 접근할 경우 컴파일 에러를 발생한다. Rust 컴파일러의 이러한 기능을 `Borrow checker`라고 부른다.
- Rust 컴파일러는 컴파일 에러 메시지에서, `Borrow checker`가 어느 시점에 `lifetime`이 소멸되었는지 표시해주므로, 디버깅할 때 편리하다.

Borrow Checker (cont'd 2/n)

- `Borrow checker` 덕분에 Rust에서는 각 레퍼런스의 `lifetime`을 컴파일 타임에 미리 알 수 있다. 하지만 단순 코드 상으로 `lifetime`을 추론해내는 것이 불가능한 경우가 있을 수 있다.
- 아래 함수에서는 리턴값이 `x의 레퍼런스` 일지, `y의 레퍼런스` 일지 알 수 없다.

```
fn longest(x: &str, y: &str) -> &str {
    if x.len() > y.len() {
        x
    };
    y
}

// 5 | fn longest(x: &str, y: &str) -> &str { expected lifetime parameter
```

Declare a lifetime parameter

- 명시법은 제너릭 타입과 유사한데, 앞에 `'` 만 붙이면 된다.
- `a` 라는 `lifetime` 을 두 매개변수와 리턴값에 대해 명시해주었다.

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

Warning!

- `lifetime parameter`는 실제 `lifetime`에 영향을 끼치지 않는다.
- `lifetime parameter`는 컴파일러에게 `x`와 `y` 그리고 `반환값`이 최소한 `a`만큼의 `lifetime`을 갖는다는 힌트를 줄 뿐이다.
- 즉, `리턴 값의 lifetime이 최소한 x와 y 중 lifetime이 작은 것만큼은 된다` 라는 의미!

Borrow Checker (cont'd 3/n)

- 아래 함수의 `result` 는 `longest` 함수 내의 지역 변수이고, 이를 리턴하는 것은 전형적인 `dangling pointer` 의 발생 사례이다.
- `lifetime paremeter` 가 오브젝트의 `lifetime` 을 변경하는 것이 아니므로, 위 함수는 컴파일 에러가 난다.

```
fn longest<'a>(x: &str, y: &str) -> &'a str {
    let result = String::from("really long string");
    result.as_str()
}
```

Borrow Checker (con'td 4/n)

- r의 라이프타임은 `'a`로, x의 라이프타임은 `'b`로 표현됐다.
- `'a` 라이프타임을 가진 r이 `'b` 라이프타임을 가진 오브젝트를 참조하고 있다.
- `'b` 라이프타임이 `'a` 라이프타임보다 작아서 컴파일러가 이 프로그램을 거부한다.
- 참조자가 오래 살지 못하기 때문이다.

```
{
    let r;           // -------+-- 'a
                     //        |
    {                //        |
        let x = 5;   // -+-----+-- 'b
        r = &x;      //  |     |
    }                // -+     |
                     //        |
    println!("r: {}", r); //  |
                     //        |
                     // -------+
}
```

- 아래 프로그램은 댕글링 참조자가 없어서 정상적으로 컴파일된다.
- x의 라이프타임인 'b가 'a보다 커서 r이 x를 참조할 수 있다.

```
{
    let x = 5;                // -----+-- 'b
                              //      |
    let r = &x;               // --+--+-- 'a
                              //   |  |
    println!("r: {}", r);     //   |  |
                              // --+  |
}                             // -----+
```

```
fn main() {
    let string1 = String::from("long string is long");
    let result;
    {
        let string2 = String::from("xyz");
        result = longest(string1.as_str(), string2.as_str());
    }
    println!("The longest string is {}", result);
}
```